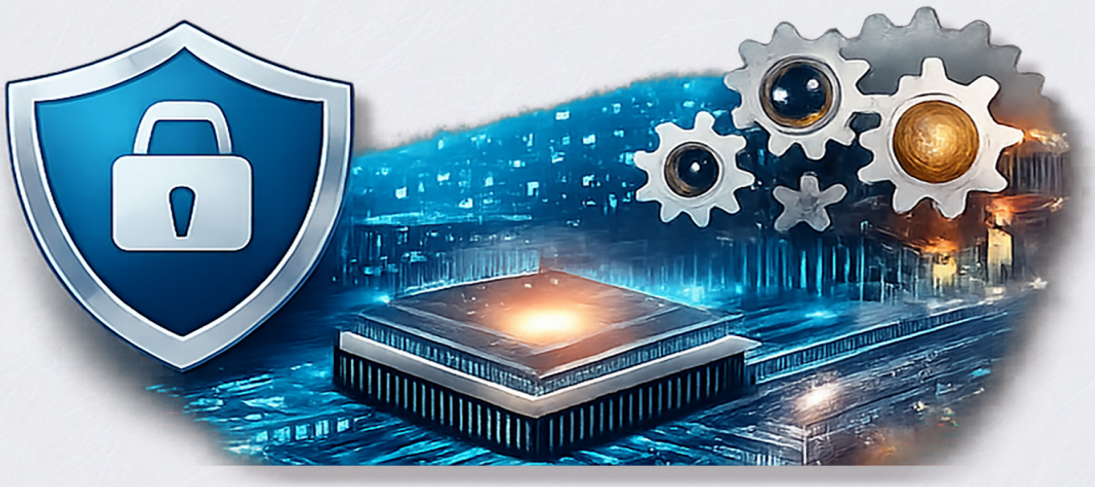
https://simplifycpp.org

The Modern C++ Safety Handbook (C++11 → C++26)

A Professional Guide to Memory Safety, Concurrency and Robust Systems Design



Prepared by Ayman alheraki

DRAFT

First Edition

Modern C++ Safety Handbook
(C++11 → C++26)

A Professional Guide to Memory Safety,
Concurrency, and Robust Systems Design

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared By Ayman Alheraki
simplifycpp.org

January 2025

Contents

Contents **2**

Author's Note **46**

Why this booklet was written 46

Who this reference is for 52

How it differs from traditional C++ books 56

Preface **62**

C++ as a systems programming language 62

Why C++ errors are rarely "simple bugs" 70

Engineering responsibility in C++ 75

The difference between legacy C++ and Modern C++ 83

I Memory Model and Object Lifetime 91

1 How C++ Sees Memory 93

1.1 Stack vs heap 93

1.1.1 First: the correct C++ terms (what the standard actually models) . . 93

1.1.2 What people mean by "stack" in C++ 94

1.1.3 What people mean by “heap” in C++ 95
1.1.4 Static storage and thread-local storage (not stack, not typical heap) . 96
1.1.5 Why stack vs heap is really about *lifetime* and *ownership* 97
1.1.6 The two most dangerous failure modes 98
1.1.7 Performance reality: when stack is faster, and when heap is necessary 99
1.1.8 Practical guidance: how to choose safely 100
1.1.9 Key takeaway 101
1.2 Static storage duration 101
1.2.1 What static storage duration means in standard C++ 101
1.2.2 Four common forms of static storage duration 102
1.2.3 Initialization and destruction: the important reality 103
1.2.4 Professional solution: the “construct on first use” idiom 104
1.2.5 Thread-safety and function-local static initialization (since C++11) 105
1.2.6 Destruction: static objects can die after `main` 106
1.2.7 Static storage duration and the One Definition Rule (ODR) 107
1.2.8 Common safety risks with static storage duration 107
1.2.9 Best practices for professional codebases 108
1.2.10 Key takeaway 109
1.3 Global and thread-local storage 109
1.3.1 Why this section matters 109
1.3.2 Global storage in C++: what it really means 109
1.3.3 The two global hazards: initialization order and shared mutable state 110
1.3.4 Prefer constant initialization for globals 112
1.3.5 Thread-local storage: a different storage duration 113
1.3.6 When thread-local is a good design choice 114
1.3.7 Thread-local pitfalls and engineering constraints 115
1.3.8 Global vs thread-local: choosing responsibly 116

1.3.9 Key takeaway . . . 117
1.4 Alignment and padding . . . 118
1.4.1 Why alignment and padding matter for safety and performance . . . 118
1.4.2 The standard model: object representation and alignment requirement 118
1.4.3 What padding is and why it exists . . . 119
1.4.4 Member order: a practical rule that reduces padding . . . 120
1.4.5 Alignment control: `alignas` and over-aligned types . . . 120
1.4.6 Misalignment: what can go wrong . . . 121
1.4.7 Alignment and strict aliasing: do not mix them up . . . 122
1.4.8 Atomic operations and alignment requirements . . . 123
1.4.9 False sharing: padding as a concurrency performance tool . . . 124
1.4.10 Practical checklist for safe layout work . . . 124
1.4.11 Key takeaway . . . 125
1.5 Cache awareness (high-level overview) . . . 125
1.5.1 Why cache awareness matters in C++ systems . . . 125
1.5.2 What a cache is, in simple terms . . . 126
1.5.3 Locality: the key to cache efficiency . . . 126
1.5.4 How data layout influences cache behavior . . . 127
1.5.5 Cache lines, alignment, and false sharing . . . 128
1.5.6 Prefetching and access patterns . . . 128
1.5.7 Working set size and cache levels . . . 129
1.5.8 Data structure design and cache-friendliness . . . 129
1.5.9 Cache awareness in concurrent systems . . . 129
1.5.10 Measurement and validation . . . 130
1.5.11 Practical summary for C++ engineers . . . 130
1.5.12 Key takeaway . . . 131

2 Object Lifetime Rules **132**

2.1 Construction and destruction semantics 132

2.1.1 Construction vs lifetime: what C++ actually guarantees 132

2.1.2 When lifetime begins and ends (the practical model) 133

2.1.3 Order of construction and destruction (bases, members, and the most important rule) 133

2.1.4 Virtual destructors: the correctness boundary for polymorphic deletion 135

2.1.5 Destructors and exceptions: why they are almost always `noexcept` 136

2.1.6 Copy, move, and destruction: what the special member functions mean 138

2.1.7 Temporaries and lifetime extension (a common source of dangling references) 139

2.1.8 Advanced: explicit lifetime control with placement `new` and `std::destroy_at` 140

2.1.9 Practical engineering rules for construction and destruction 141

2.1.10 Key takeaway 142

2.2 Temporary objects 142

2.2.1 What a temporary object is 142

2.2.2 The default rule: temporaries live until the end of the full-expression 143

2.2.3 Lifetime extension: when a temporary lives longer 143

2.2.4 The most important exception: binding to a reference parameter does *not* extend beyond the call 145

2.2.5 Temporaries and subobjects: do not assume “parts” live longer . . . 145

2.2.6 Temporaries in expressions: why chaining can hide lifetimes 146

2.2.7 Range-based `for` and temporaries: a C++23 correctness improvement 147

2.2.8 Temporary materialization (C++17 and later): what you must remember 149

2.2.9 Safe engineering rules for temporary objects 149

2.2.10 Key takeaway 150

2.3 Lifetime extension 150
2.3.1 What “lifetime extension” means 150
2.3.2 The default rule (no extension) 150
2.3.3 The main extension rule: binding a temporary to a reference variable 151
2.3.4 Lifetime extension to reference members (common but misunderstood) 152
2.3.5 What does *not* extend lifetime 153
2.3.6 C++17 and later: prvalues and temporary materialization 155
2.3.7 Range-based `for`: lifetime extension and the C++23 fix 156
2.3.8 Practical engineering rules for lifetime extension 157
2.3.9 Key takeaway 157
2.4 Common lifetime violations 158
2.4.1 Why lifetime violations are uniquely dangerous in C++ 158
2.4.2 Violation 1: returning a pointer or reference to a local object 158
2.4.3 Violation 2: use-after-free (dangling after `delete`) 159
2.4.4 Violation 3: double-delete and mismatched delete 160
2.4.5 Violation 4: dangling references/views to temporaries 161
2.4.6 Violation 5: iterator, pointer, and reference invalidation by containers 162
2.4.7 Violation 6: pointer/reference to a moved-from object’s internals . . 163
2.4.8 Violation 7: capturing references that outlive the referenced object . 164
2.4.9 Violation 8: storing pointers/references to short-lived buffers 165
2.4.10 Violation 9: manual lifetime management mistakes (placement new) 166
2.4.11 Common patterns that look safe but are not 167
2.4.12 Professional prevention strategy 167
2.4.13 Key takeaway 168

3 Undefined Behavior: The Silent Killer 169
3.1 What undefined behavior really means 169

3.1.1 A precise definition (in practical engineering terms) 169
3.1.2 UB is different from other kinds of “not fully specified” behavior . 170
3.1.3 Why the compiler “cares”: UB enables optimization 170
3.1.4 Common sources of undefined behavior (the ones that matter most in real systems) 171
3.1.5 Why UB does not reliably reproduce 174
3.1.6 The engineering goal: make UB unrepresentable by design 175
3.1.7 Verification: you must detect UB, not guess it 176
3.1.8 Key takeaway 176
3.2 Why the compiler is not your friend here 177
3.2.1 The uncomfortable truth 177
3.2.2 UB is an optimization permission, not a runtime event 177
3.2.3 Example 1: “My null check should protect me” 178
3.2.4 Example 2: signed overflow and “impossible” branches 178
3.2.5 Example 3: out-of-bounds makes the optimizer trust your indices . 179
3.2.6 Example 4: strict aliasing and “I can reinterpret anything” 180
3.2.7 Example 5: data races are UB, so the compiler can reorder aggressively 181
3.2.8 Why Debug builds often “work” and Release builds fail 182
3.2.9 What the compiler *can* do that feels “hostile” 182
3.2.10 Professional discipline: make the compiler your ally again 183
3.2.11 Key takeaway 183
3.3 Real-world consequences 183
3.3.1 Why undefined behavior is a production risk, not a theory topic . . 184
3.3.2 Consequence 1: “Heisenbugs” that vanish under debugging 184
3.3.3 Consequence 2: Release-only failures and “impossible” branches . 185
3.3.4 Consequence 3: silent data corruption (the worst kind of failure) . . 186

3.3.5 Consequence 4: security vulnerabilities (integrity, confidentiality, availability) . . . 187
3.3.6 Consequence 5: concurrency failures that cannot be reasoned about 187
3.3.7 Consequence 6: “tooling blind spots” and incomplete test coverage 188
3.3.8 How professionals reduce real-world UB impact . . . 189
3.3.9 Key takeaway . . . 190
3.4 How undefined behavior propagates . . . 190
3.4.1 The core idea: UB is not local . . . 190
3.4.2 Propagation path 1: the optimizer treats UB paths as impossible . . 191
3.4.3 Propagation path 2: UB contaminates values (“poisoning” program state) . . . 192
3.4.4 Propagation path 3: memory corruption spreads across objects and time 193
3.4.5 Propagation path 4: lifetime UB invalidates the type system . . . 194
3.4.6 Propagation path 5: aliasing UB enables load/store reordering . . . 195
3.4.7 Propagation path 6: data races break the memory model . . . 196
3.4.8 Why UB propagates further in modern builds . . . 198
3.4.9 The key safety implication: you cannot contain UB with local checks 198
3.4.10 How professionals limit UB propagation . . . 199
3.4.11 Key takeaway . . . 199

II Classic Memory Errors and How Professionals Avoid Them 201

4 Memory Leaks 203
4.1 Why leaks still happen in modern code . . . 203
4.1.1 The modern reality . . . 203
4.1.2 Leak vs intentional retention . . . 204
4.1.3 Reason 1: ownership is still unclear at API boundaries . . . 204

4.1.4 Reason 2: `std::shared_ptr` cycles 205
4.1.5 Reason 3: containers retain memory by design 207
4.1.6 Reason 4: caches, pools, and "one-time" allocations that become unbounded 208
4.1.7 Reason 5: early exits around non-RAII resources 208
4.1.8 Reason 6: exceptions + partial construction + raw ownership 210
4.1.9 Reason 7: background work that outlives its owners (threads, callbacks, async) 211
4.1.10 Reason 8: allocator mismatches and mixed allocation domains . . . 212
4.1.11 Reason 9: "it leaks only on shutdown" is still a leak (sometimes) . . 213
4.1.12 How professionals prevent leaks in modern C++ 213
4.1.13 Key takeaway 214
4.2 Exception paths and early returns 215
4.2.1 Why this still matters in Modern C++ 215
4.2.2 Early return leaks: the classic pattern 215
4.2.3 Exceptions: leaks when ownership is not established immediately . 216
4.2.4 Multiple resources: leaks and partial cleanup ordering 218
4.2.5 Exceptions during construction: partial objects are still cleaned correctly 219
4.2.6 Destructors must not throw (or you risk termination) 220
4.2.7 Modern tools that reduce leak risk on non-happy paths 221
4.2.8 Practical guidelines for leak-free error handling 222
4.2.9 Verification: you must test the non-happy paths 222
4.2.10 Key takeaway 223
4.3 Professional RAII-based design 223
4.3.1 What RAII really is (and why professionals rely on it) 223
4.3.2 The core RAII contract 224
4.3.3 Professional design rule 1: model ownership explicitly 225

4.3.4 Professional design rule 2: acquire and bind ownership immediately 225
4.3.5 Professional design rule 3: make copying and moving correct by construction . . . 227
4.3.6 Professional design rule 4: destructors must be noexcept and best-effort 229
4.3.7 Professional design rule 5: RAII for locks and invariants . . . 230
4.3.8 Professional design rule 6: RAII for "rollback on failure" . . . 230
4.3.9 Professional design rule 7: prefer standard RAII types and patterns . 232
4.3.10 Professional checklist: what to review in RAII code . . . 233
4.3.11 Key takeaway . . . 233

5 Dangling Pointers and References 234
5.1 Stack vs heap lifetime mistakes . . . 234
5.1.1 The fundamental confusion . . . 234
5.1.2 Stack (automatic storage): why it is easy to dangle . . . 235
5.1.3 Heap (dynamic storage): different risks, different dangling patterns 237
5.1.4 Stack vs heap: the most common misunderstanding . . . 238
5.1.5 Safe lifetime modeling: the professional approach . . . 239
5.1.6 Stack-specific traps that look correct . . . 240
5.1.7 Professional checklist: preventing stack/heap lifetime mistakes . . . 241
5.1.8 Key takeaway . . . 242
5.2 Returning references and pointers . . . 242
5.2.1 The core issue . . . 242
5.2.2 Returning references/pointers to local (automatic) objects . . . 243
5.2.3 Safe alternatives . . . 243
5.2.4 Returning pointers/references to heap objects with clear ownership . 244
5.2.5 Returning references/pointers to static objects . . . 245
5.2.6 Returning views into containers . . . 245

5.2.7 Returning iterators and container pointers 246
5.2.8 Lifetime extension rules (and their limits) 247
5.2.9 Professional checklist for safe returns 248
5.2.10 Key takeaway 248
5.3 Views vs ownership 248
5.3.1 The fundamental distinction 248
5.3.2 Why views exist (and why professionals use them) 249
5.3.3 The most common bug: returning a view to something you created . 250
5.3.4 Views as parameters: the correct and common usage 251
5.3.5 Views can be invalidated even when the owner is alive 252
5.3.6 Ownership types: strong safety, but not free 253
5.3.7 Designing APIs: when to return a view vs return an owner 254
5.3.8 Views, raw pointers, and “borrowed” references 255
5.3.9 Professional rules for avoiding dangling views 256
5.3.10 Key takeaway 256
5.4 Safer alternatives 256
5.4.1 What “safer” means in professional C++ 257
5.4.2 Alternative 1: return by value (owning results) 257
5.4.3 Alternative 2: `std::unique_ptr` for unique dynamic ownership 258
5.4.4 Alternative 3: `std::shared_ptr` + `std::weak_ptr` for shared lifetimes 259
5.4.5 Alternative 4: use `std::span` instead of pointer + length 260
5.4.6 Alternative 5: use `std::string_view` for parameters, but return `std::string` for results 261
5.4.7 Alternative 6: prefer values and references for short-lived usage, avoid storing borrows 262
5.4.8 Alternative 7: indices/keys instead of pointers and iterators 263

5.4.9 Alternative 8: RAII guards for lifecycle and cleanup 263
5.4.10 Alternative 9: bounds-checked access in safety-critical paths 264
5.4.11 How to choose the right alternative (practical decision rules) 264
5.4.12 Key takeaway 265

6 Uninitialized and Invalid Memory 266
6.1 Default initialization traps 266
6.1.1 Why default initialization is a common source of bugs 266
6.1.2 The three important forms: default, value, and zero initialization . . 267
6.1.3 Trap 1: local fundamental types are not initialized 267
6.1.4 Trap 2: “new T” vs “new T()” vs “new T{}” 268
6.1.5 Trap 3: arrays and containers do not always zero their elements . . 268
6.1.6 Trap 4: user-defined types can still leave members uninitialized . . 269
6.1.7 Trap 5: = default and implicitly-generated constructors are not magic 270
6.1.8 Trap 6: padding bytes and comparing objects bytewise 271
6.1.9 Trap 7: “invalid” values that are not merely uninitialized 272
6.1.10 Professional defaults: patterns that prevent initialization bugs 272
6.1.11 Quick reference: initialization syntax and outcome (common cases) 273
6.1.12 Key takeaway 274
6.2 Indeterminate values 274
6.2.1 What an indeterminate value is (precise meaning) 274
6.2.2 Why indeterminate values are dangerous 275
6.2.3 The most common source: uninitialized automatic variables 275
6.2.4 Indeterminate does not always mean “UB in every context” 276
6.2.5 Trap 1: class types can still contain indeterminate members 276
6.2.6 Trap 2: `new T` does not necessarily initialize `T` 277
6.2.7 Trap 3: padding bytes and bytewise operations 278

6.2.8 Trap 4: indeterminate values in control flow cause “unreproducible” failures . . . 279
6.2.9 Indeterminate vs unspecified vs implementation-defined vs undefined behavior . . . 279
6.2.10 Professional prevention strategy . . . 280
6.2.11 Key takeaway . . . 281
6.3 Zero-initialization vs value-initialization . . . 281
6.3.1 Why these terms matter . . . 281
6.3.2 Zero-initialization (what it means) . . . 281
6.3.3 Value-initialization (what it means) . . . 282
6.3.4 The practical difference (most important cases) . . . 283
6.3.5 The subtle class-type trap: user-provided default constructors . . . 284
6.3.6 How to force “all zeros” reliably (when you truly need it) . . . 285
6.3.7 Why professionals prefer braces `{}` . . . 285
6.3.8 Quick reference table (most common patterns) . . . 285
6.3.9 Professional rules to avoid initialization mistakes . . . 286
6.3.10 Key takeaway . . . 286

7 Buffer Overflows and Out-of-Bounds Access 287
7.1 Raw arrays vs containers . . . 287
7.1.1 Why this comparison matters . . . 287
7.1.2 Raw arrays: what they are and why they are risky . . . 288
7.1.3 Standard containers: why they are safer . . . 289
7.1.4 Bounds checking: the real difference in safety behavior . . . 290
7.1.5 The professional sweet spot: containers for ownership, `std::span` for parameters . . . 291
7.1.6 When raw arrays are still appropriate . . . 292

7.1.7 Common pitfalls and how containers help 293
7.1.8 Performance note: containers are not inherently slower 294
7.1.9 Professional checklist 295
7.1.10 Key takeaway 295
7.2 Safer indexing strategies 296
7.2.1 Why indexing is a high-risk operation 296
7.2.2 Strategy 1: prefer range-based loops over manual indexing 296
7.2.3 Strategy 2: use iterators and algorithms where possible 297
7.2.4 Strategy 3: use `at()` at trust boundaries 297
7.2.5 Strategy 4: validate indices explicitly and centralize the checks . . . 298
7.2.6 Strategy 5: avoid signed/unsigned traps (`int` vs `size_t`) 299
7.2.7 Strategy 6: use `std::span` to bind pointer and length together . . 300
7.2.8 Strategy 7: prefer "two-iterator" ranges over indexing for slicing . . 300
7.2.9 Strategy 8: use `std::ssize` and `std::ptrdiff_t` for signed counts 301
7.2.10 Strategy 9: treat index arithmetic as a proof obligation 302
7.2.11 Strategy 10: rely on tooling for verification in critical code 302
7.2.12 Professional checklist 303
7.2.13 Key takeaway 303
7.3 `std::span` as a modern solution 304
7.3.1 Why `std::span` exists 304
7.3.2 What `std::span` is (and what it is not) 304
7.3.3 Static extent vs dynamic extent 306
7.3.4 Span as an API boundary tool 306
7.3.5 Bounds safety: what span guarantees and what it does not 307
7.3.6 Safe slicing with `first`, `last`, and `subspan` 309
7.3.7 Interoperability: span with C and OS APIs 310

7.3.8 Byte views with `std::as_bytes` and `std::as_writable_bytes` 310
7.3.9 Lifetime and invalidation rules (what professionals watch for) . . . 311
7.3.10 Using `std::span` in a C++11–C++17 codebase 312
7.3.11 Professional checklist 312
7.3.12 Key takeaway 313

III Ownership, RAII, and Resource Management 314

8 RAII: The Core of Safe C++ 316
8.1 Philosophy and Guarantees 316
8.1.1 RAII is the Default Safety Model in C++ 316
8.1.2 What Counts as a "Resource" 317
8.1.3 The Core Guarantees RAII Relies On 317
8.1.4 RAII Guarantees You Actually Get 318
8.1.5 RAII and the "Commit or Roll Back" Mindset 320
8.1.6 The "Noexcept Destructor" Rule and Its Safety Consequences . . . 322
8.1.7 Zero-Overhead Philosophy 323
8.1.8 What RAII Does *Not* Guarantee 323
8.1.9 Professional Checklist: What "Good RAII" Looks Like 324
8.1.10 Summary 325
8.2 RAII beyond memory 326
8.2.1 The Core Idea: "Resource" Means Much More Than Heap Memory 326
8.2.2 File Ownership: Streams and "Handle RAII" 326
8.2.3 Lock Ownership: RAII for Concurrency Correctness 328
8.2.4 Thread Lifetime: RAII for Joining and Cooperative Stop 329
8.2.5 Temporary Files and Directories: RAII for Cleanup and Security . . 331

8.2.6 Memory-Mapped Regions: RAII for `mmap`/`munmap`-style APIs . . 333
8.2.7 State Restoration: RAII for "Must Revert" Changes 335
8.2.8 Error Handling Rule: "Destructors Release, Functions Report" . . . 336
8.2.9 Checklist: Applying RAII to Any Non-Memory Resource 337
8.2.10 Summary 337
8.3 Designing RAII types correctly 337
8.3.1 Design Goal: Make Resource Correctness Automatic 337
8.3.2 State, Invariants, and Ownership 338
8.3.3 The Rule of Zero, Rule of Five, and Rule of "One Resource" 339
8.3.4 Destructor Rules: Never Let Cleanup Throw 341
8.3.5 Construction: Ensure Strong Invariants and Exception Safety 342
8.3.6 Copy and Move Semantics: Choose the Right Ownership Model . . 343
8.3.7 Use Standard Building Blocks Whenever Possible 344
8.3.8 Provide a Minimal, Safe API Surface 345
8.3.9 Make It Hard to Misuse 346
8.3.10 Performance and Layout Considerations (Without Sacrificing Safety) 347
8.3.11 Validation Checklist: A Correct RAII Type Should Satisfy These Properties 348
8.3.12 Summary 348

9 Smart Pointers Done Right 350
9.1 `std::unique_ptr` 350
9.1.1 What `std::unique_ptr` Is (and Why It Exists) 350
9.1.2 The Safety Contract 350
9.1.3 Correct Creation: Prefer `std::make_unique` 351
9.1.4 Core Operations You Must Know 352
9.1.5 Moves and Ownership Transfer 354

9.1.6 Custom Deleters: Managing Non-`delete` Resources 355
9.1.7 Arrays: `unique_ptr<T[]>` Has Different Rules 356
9.1.8 Incomplete Types: Powerful, but You Must Respect the Rule 357
9.1.9 API Design: Passing and Returning `unique_ptr` Safely 358
9.1.10 Common Misuses and How to Avoid Them 360
9.1.11 Summary 361
9.2 `std::shared_ptr` 361
9.2.1 What `std::shared_ptr` Is (and What It Is Not) 361
9.2.2 The Control Block: Where the "Shared" Part Lives 362
9.2.3 Correct Creation: Prefer `std::make_shared` 362
9.2.4 Core Semantics You Must Understand 363
9.2.5 Threading: What Is Safe, What Is Not 365
9.2.6 The Biggest Pitfall: Cycles and Leaks 366
9.2.7 Owning `this`: `std::enable_shared_from_this` 368
9.2.8 Advanced Technique: The Aliasing Constructor 369
9.2.9 Custom Deleters: Shared Ownership of Non-`delete` Resources . 370
9.2.10 API Design Rules for `shared_ptr` 370
9.2.11 Performance and Predictability Considerations 371
9.2.12 Common Misuses and Their Fixes 372
9.2.13 Summary 372
9.3 `std::weak_ptr` 373
9.3.1 What `std::weak_ptr` Is 373
9.3.2 The Control Block and What `weak_ptr` Actually Points To 373
9.3.3 Primary Use Cases 374
9.3.4 Correct Usage: `lock()` First 377
9.3.5 Key Member Functions and Their Meaning 379
9.3.6 Threading and Safety Guarantees 379

9.3.7 Relationship to `enable_shared_from_this` 380
9.3.8 Design Guidance: Where `weak_ptr` Belongs 380
9.3.9 Common Mistakes and How to Avoid Them 381
9.3.10 Summary 382
9.4 Common misuse patterns 382
9.4.1 Why Smart Pointers Still Get Misused 382
9.4.2 Misuse 1: Constructing an owning smart pointer from a non-owned raw pointer 383
9.4.3 Misuse 2: Two `shared_ptr` control blocks for the same object . 383
9.4.4 Misuse 3: `shared_ptr(this)` instead of `enable_shared_from_this` 385
9.4.5 Misuse 4: Treating `shared_ptr` as "the default pointer type" . . 386
9.4.6 Misuse 5: Reference cycles with `shared_ptr` 386
9.4.7 Misuse 6: Letting `get()` escape beyond the owner 388
9.4.8 Misuse 7: `weak_ptr::expired()` as a "safety check" 388
9.4.9 Misuse 8: Using `use_count()` for correctness 389
9.4.10 Misuse 9: Misunderstanding thread safety 390
9.4.11 Misuse 10: Misusing the aliasing constructor 391
9.4.12 Misuse 11: Throwing cleanup through a deleter 392
9.4.13 Misuse 12: Passing smart pointers in APIs without meaning 392
9.4.14 Practical Checklist: Spotting Misuse Quickly 393
9.4.15 Summary 394
9.5 Performance considerations 394
9.5.1 A Practical Cost Model 394
9.5.2 `std::unique_ptr`: Usually "Zero-Overhead" in Practice 395
9.5.3 `std::shared_ptr`: Reference Counting Is the Cost Center . . . 396

9.5.4 `make_shared` and `allocate_shared`: Allocation and Locality Tradeoffs . . . 397
9.5.5 `std::weak_ptr`: Observation Has a Cost . . . 399
9.5.6 Atomic Shared Ownership: Avoid Accidental Contention . . . 400
9.5.7 Deleters and Object Size: Small Details That Matter . . . 401
9.5.8 A Decision Guide That Improves Performance and Clarity . . . 401
9.5.9 Summary . . . 402

10 Raw Pointers: When Are They Acceptable? 403
10.1 Observers vs owners . . . 403
10.1.1 The Central Distinction . . . 403
10.1.2 Why the Distinction Matters . . . 404
10.1.3 Owners: What Ownership Looks Like in Modern C++ . . . 404
10.1.4 Observers: When Raw Pointers Are Acceptable . . . 406
10.1.5 Recognizing Ownership at the Call Site . . . 408
10.1.6 The Two Classic Bugs (and Their Fixes) . . . 408
10.1.7 A Practical Rule Set for Chapter 10 . . . 410
10.1.8 Summary . . . 410
10.2 Documenting ownership clearly . . . 411
10.2.1 The Goal: Make Lifetime Obvious Without Reading the Implementation 411
10.2.2 Document Ownership by Choosing the Right Parameter Types . . . 411
10.2.3 Document Ownership by Choosing the Right Return Types . . . 413
10.2.4 Make Ownership Visible in Data Members . . . 415
10.2.5 Annotating Legacy Ownership When Raw Owning Pointers Exist . . . 416
10.2.6 Naming Conventions That Communicate Ownership . . . 417
10.2.7 Documenting Lifetime Assumptions Precisely . . . 417
10.2.8 Tooling: Enforcing Ownership Documentation . . . 418

10.2.9 Summary 419
10.3 Modern alternatives 419
10.3.1 What This Section Means by “Modern Alternatives” 419
10.3.2 Alternative 1: Prefer Values and RAII Objects Over Dynamic Allocation 420
10.3.3 Alternative 2: Use Smart Pointers to Represent Ownership (Not “Because Pointers”) 421
10.3.4 Alternative 3: Express Non-Null Contracts with `not_null` 422
10.3.5 Alternative 4: Use Views for Sequences Instead of “Pointer + Size” 422
10.3.6 Alternative 5: Use Ranges and Views for General Iteration (C++20+) 424
10.3.7 Alternative 6: Replace “Nullable Return Pointer” with `std::optional` or `std::expected` 424
10.3.8 Alternative 7: Store “References” Safely Using `std::reference_wrapper` 425
10.3.9 Alternative 8: Prefer Non-Owning Callable Views Over “Function Pointer + void*” 426
10.3.10 When Raw Pointers Still Win (and How to Keep Them Safe) 426
10.3.11 Summary 427

11 Resource Management Beyond Memory 428
11.1 Files 428
11.1.1 Files Are Resources, Not “Just I/O” 428
11.1.2 The RAII Principle Applied to Files 428
11.1.3 Using Standard C++ File Streams Safely 429
11.1.4 Bridging to C APIs: `FILE*` with RAII 431
11.1.5 POSIX File Descriptors: A Minimal `unique_fd` 432
11.1.6 Atomic Update Pattern: Write-Temp, Then Rename 434

11.1.7 Filesystem Facilities for File Management 436
11.1.8 Checklist: File Safety Rules That Prevent Common Bugs 436
11.1.9 Summary 437
11.2 Locks 438
11.2.1 Locks Are Resources (Just Like Files and Memory) 438
11.2.2 The Core RAII Lock Wrappers 438
11.2.3 Avoiding Deadlocks When Multiple Locks Are Involved 441
11.2.4 Shared Locks for Read-Mostly Data 442
11.2.5 Timed Locking and Try-Locking 443
11.2.6 Common Lock Misuse Patterns (and How RAII Prevents Them) . . 444
11.2.7 Lock Ownership as an Explicit State (Lock Tags) 445
11.2.8 Forward-Looking Note for C++26 and Beyond 446
11.2.9 Summary 446
11.3 Sockets 447
11.3.1 Sockets Are Resources (Descriptors and Kernel State) 447
11.3.2 Standard C++ Reality: No Portable Sockets in `std::` Yet 448
11.3.3 Core RAII Pattern: A Move-Only Socket Owner 448
11.3.4 Shutdown vs Close: Correct Semantics and Correct Intent 451
11.3.5 Avoiding Common Socket Resource Bugs 452
11.3.6 Close-on-exec Hygiene (POSIX) 455
11.3.7 Graceful Shutdown Strategy 455
11.3.8 Error Handling: Platform Error Codes vs C++ Error Types 456
11.3.9 Recommended Practice: Library-Based Networking When Portability Matters 456
11.3.10 Summary 457
11.4 OS handles 457
11.4.1 What an “OS handle” really is 457

11.4.2 The RAII contract for handles 458
11.4.3 POSIX: file descriptors (`int`) 459
11.4.4 Windows: `HANDLE` and `CloseHandle` 461
11.4.5 Encoding ownership with `std::unique_ptr` (when appropriate) 463
11.4.6 Ownership transfer and sharing rules 464
11.4.7 Common OS-handle failure modes (and how RAII prevents them) . 464
11.4.8 Checklist: Handle safety rules 465
11.5 Transaction-style RAII 466
11.5.1 What "transaction-style RAII" means 466
11.5.2 Why you need it beyond "release on scope exit" 466
11.5.3 The three scope-guard forms: always, on-fail, on-success 467
11.5.4 A practical approach across C++11 → C++26 467
11.5.5 Pattern 1: "rollback unless committed" 469
11.5.6 Pattern 2: Strong guarantee via "stage, then commit" (copy-and-swap family) 470
11.5.7 Pattern 3: External transactions (filesystem) with compensating actions 471
11.5.8 Design rules for correct transaction-style RAII 472
11.5.9 Summary 474

IV Concurrency, Threads, and Memory Safety 475

12 The C++ Memory Model (Simplified) 477
12.1 Data Races 477
12.1.1 What a Data Race Is (and Why It Matters) 477
12.1.2 The Standard Definition (Simplified but Accurate) 477
12.1.3 Conflicting Access: "Same Memory Location" in Practice 478
12.1.4 Happens-Before: The Only Valid Way to "Make It Safe" 478

12.1.5 Minimal Example: A Real Data Race (Undefined Behavior) 479
12.1.6 Fix Strategy 1: Use a Mutex (Establish Happens-Before) 480
12.1.7 Fix Strategy 2: Use Atomics (Avoid Non-Atomic Conflicts) 480
12.1.8 A Professional Pattern: Publish Data Safely 481
12.1.9 What `volatile` Does *Not* Do 483
12.1.10 Data Race vs Race Condition 483
12.1.11 Detection and Verification in Real Projects 483
12.1.12 Professional Checklist 484
12.2 Happens-before 484
12.2.1 Why "happens-before" exists 484
12.2.2 The building blocks 485
12.2.3 Why happens-before matters: data races and undefined behavior . . 486
12.2.4 How to create happens-before edges in real code 487
12.2.5 A simplified mental model (that stays correct) 492
12.2.6 Summary 492
12.3 Visibility vs ordering 493
12.3.1 Two different questions (that are easy to mix up) 493
12.3.2 Why ordering and visibility can diverge 494
12.3.3 A concrete mental model that stays correct 494
12.3.4 Example 1: Relaxed atomic gives atomicity, not safe publication . . 495
12.3.5 Example 2: Ordering without ownership of a particular atomic (fences) 497
12.3.6 Example 3: Mutex ordering implies visibility for protected data . . 497
12.3.7 What to remember about `memory_order_seq_cst` 498
12.3.8 A safety checklist for visibility vs ordering 499
12.3.9 Summary 499

13 Race Conditions and Data Corruption **500**

13.1 Why “it works on my machine” fails 500

13.1.1 The uncomfortable truth: concurrency bugs are often *allowed* to look correct 500

13.1.2 Four reasons your machine can hide a race that production reveals . 501

13.1.3 Canonical example: the increment that “usually works” 502

13.1.4 Canonical example: the “publish a pointer” bug 504

13.1.5 Data-race-free programs behave *much* better 505

13.1.6 How to stop trusting luck 505

13.1.7 Summary 506

13.2 Common concurrency bugs 507

13.2.1 Why these bugs are so common 507

13.2.2 Bug 1: Data races on non-atomic shared state 507

13.2.3 Bug 2: “Atomic flag” used with `memory_order_relaxed` to publish data 508

13.2.4 Bug 3: Check-then-act races (TOCTOU) under concurrency 510

13.2.5 Bug 4: Lost wakeups and spurious wakeups with condition variables 511

13.2.6 Bug 5: Deadlocks from inconsistent lock ordering 512

13.2.7 Bug 6: Livelock and starvation 513

13.2.8 Bug 7: Holding locks while calling unknown code 514

13.2.9 Bug 8: Misusing `volatile` for synchronization 514

13.2.10 Bug 9: Thread lifetime bugs (dangling references, detached threads) 515

13.2.11 Bug 10: Non-atomic object teardown while another thread uses it . 516

13.2.12 Bug 11: False sharing (performance bug that looks like a concurrency bug) 516

13.2.13 Bug 12: Misinterpreting tool results (or not using tools at all) . . . 517

13.2.14 A practical checklist 517

13.2.15 Summary 517
13.3 Diagnosing race conditions 518
13.3.1 What makes race conditions hard to diagnose 518
13.3.2 Start with the one question that matters 518
13.3.3 A professional workflow: from symptom to root cause 519
13.3.4 Recognizing the most common signatures in reports 521
13.3.5 Do not sabotage your own diagnosis 523
13.3.6 A practical checklist for diagnosing races 523
13.3.7 Summary 524

14 Mutexes and Locks the Right Way **525**
14.1 `std::mutex` 525
14.1.1 What `std::mutex` is for 525
14.1.2 Key properties and lifetime rules 525
14.1.3 The synchronization guarantee (why mutexes also solve visibility) . 526
14.1.4 Correct usage: RAII locking is the default 526
14.1.5 Preconditions and undefined behavior traps 528
14.1.6 Avoiding deadlock with multiple mutexes 529
14.1.7 When `std::mutex` is not the right mutex 529
14.1.8 Safety rules to adopt as project policy 530
14.1.9 Summary 530
14.2 `std::lock_guard` 531
14.2.1 What `std::lock_guard` is 531
14.2.2 Why `std::lock_guard` is the “right default” 531
14.2.3 Basic usage pattern 532
14.2.4 Construction modes and their meaning 533
14.2.5 Common misuse patterns (and why they are dangerous) 534

14.2.6 Choosing between `lock_guard`, `unique_lock`, and `scoped_lock` . . . 535
14.2.7 Practical rules for professional codebases . . . 536
14.2.8 Summary . . . 536
14.3 `std::unique_lock` . . . 537
14.3.1 What `std::unique_lock` is (and why it exists) . . . 537
14.3.2 Core ownership model . . . 537
14.3.3 Construction strategies (how you choose the initial state) . . . 538
14.3.4 The main feature of `unique_lock`: controlled unlock/relock . . . 540
14.3.5 The reason `condition_variable` uses `unique_lock` . . . 541
14.3.6 Timed locking (when supported) . . . 542
14.3.7 Choosing between `lock_guard`, `unique_lock`, and `scoped_lock` . . . 543
14.3.8 Common misuse patterns (and how to avoid them) . . . 544
14.3.9 Project policy: safe default rules . . . 544
14.3.10 Summary . . . 545
14.4 Avoiding deadlocks with RAII . . . 545
14.4.1 Deadlock in one sentence . . . 545
14.4.2 Why RAII helps: it prevents accidental lock leaks . . . 545
14.4.3 Rule 1: Never manually pair `lock()` and `unlock()` in application code . . . 546
14.4.4 Rule 2: If you lock more than one mutex, do not “hand-roll” ordering 547
14.4.5 Technique A (C++11+): `std::lock` + `adopt_lock` . . . 547
14.4.6 Technique B (C++17+): `std::scoped_lock` for multiple mutexes 548
14.4.7 Technique C: `std::unique_lock` + `std::lock` (for complex flows) . . . 549
14.4.8 Rule 3: Don’t call unknown code while holding a lock . . . 550

14.4.9 Rule 4: Keep lock scope small and intentional 551
14.4.10 A compact deadlock-prevention checklist 551
14.4.11 Summary 552

15 Atomics: When Mutexes Are Too Much **553**
15.1 When atomics are appropriate 553
15.1.1 The decision you are really making 553
15.1.2 Appropriate uses of atomics (high-confidence patterns) 554
15.1.3 When atomics are *not* the right tool 556
15.1.4 A practical selection guide (safe defaults) 558
15.1.5 Memory-order choices: what is appropriate in practice 558
15.1.6 How to validate that your atomic design is correct 560
15.1.7 Summary 560
15.2 Common misuse of atomics 560
15.2.1 Why atomic misuse is so dangerous 561
15.2.2 Misuse #1: Using `volatile` as if it were atomic or synchronization 561
15.2.3 Misuse #2: Treating `memory_order_relaxed` as if it "makes other writes visible" 562
15.2.4 Misuse #3: Mixing atomic and non-atomic access to the same object 563
15.2.5 Misuse #4: Using atomics to protect a multi-variable invariant . . . 564
15.2.6 Misuse #5: Incorrect `compare_exchange` usage 565
15.2.7 Misuse #6: Assuming "lock-free" is guaranteed (or always faster) . 567
15.2.8 Misuse #7: Using condition variables with an atomic "condition" incorrectly 567
15.2.9 Misuse #8: Treating fences as "magic synchronization" 568
15.2.10 Misuse #9: Forgetting what atomics do *not* guarantee 568
15.2.11 A professional "atomic misuse" checklist 569

15.3 Performance vs correctness 570
15.3.1 The rule that prevents most mistakes 570
15.3.2 Why “atomics are faster” is not a reliable rule 570
15.3.3 The performance model: what you are paying for 571
15.3.4 Correctness constraints that shape performance choices 571
15.3.5 A safe way to think about memory orders (performance through minimal correctness) 572
15.3.6 When a mutex beats atomics for performance 574
15.3.7 When atomics beat a mutex for performance 574
15.3.8 Correctness pitfalls that look like “performance optimizations” . . . 575
15.3.9 How to measure safely without fooling yourself 575
15.3.10 Practical guidance (what to put in a team coding standard) 576
15.3.11 Summary 576

V Polymorphism, Templates, and Type Safety 578

16 Virtual Functions and Object Destruction 580
16.1 Vtable Basics 580
16.1.1 What a “vtable” is (and what it is not) 580
16.1.2 The key mental model 581
16.1.3 What the standard *does* guarantee about virtual dispatch 581
16.1.4 Object layout in practice: common ABI patterns 583
16.1.5 Virtual destructors and safe polymorphic deletion 584
16.1.6 “Pure virtual” and vtable-related undefined behavior 585
16.1.7 Performance reality: cost is often indirect 586
16.1.8 Security and robustness implications (why vtable basics matter) . . 587
16.1.9 Practical guidance: what to rely on, what to avoid 588

16.2 Virtual Destructors 589
16.2.1 Why virtual destructors exist 589
16.2.2 The core safety rule: deleting through a base without a virtual destructor 590
16.2.3 Correct design: virtual destructor in a polymorphic base 591
16.2.4 When you should *not* make the destructor public and virtual 592
16.2.5 Pure virtual destructors (abstract bases) and the "definition required" rule 593
16.2.6 Implementation reality: what "virtual destructor" usually means in the ABI 594
16.2.7 Interaction with `operator delete` (including sized and aligned deallocation) 594
16.2.8 Cost model and when it matters 595
16.2.9 Professional guidelines (safe defaults) 595
16.3 Object Slicing 596
16.3.1 Definition: what "object slicing" means 597
16.3.2 The root cause: value semantics with a polymorphic base 597
16.3.3 Canonical example: passing by value slices 598
16.3.4 Slicing in containers: `std::vector<Base>` is a red flag 599
16.3.5 Why slicing is a safety problem (not just "wrong output") 599
16.3.6 How to prevent slicing by design 600
16.3.7 When value semantics are desirable: alternatives to inheritance . . . 602
16.3.8 Advanced pattern: polymorphic copy via `clone()` 603
16.3.9 Common misconceptions and clarifications 603
16.3.10 Professional checklist 604
16.4 Polymorphic Ownership 605
16.4.1 What "polymorphic ownership" means 605
16.4.2 The hard rule: polymorphic deletion requires a virtual destructor . . 606

16.4.3 Preferred model: single ownership with `std::unique_ptr<Base>` 606
16.4.4 Borrowing vs owning: express intent in the type 607
16.4.5 Shared ownership: `std::shared_ptr<Base>` when it is truly needed 608
16.4.6 Custom destruction: polymorphic ownership across ABI or allocator boundaries 610
16.4.7 Preventing accidental slicing and unsafe copying 611
16.4.8 Non-owning access patterns: safer APIs 612
16.4.9 Checklist: safe polymorphic ownership in production C++ 613

17 Templates: Power with Responsibility **615**
17.1 Compile-time Complexity 615
17.1.1 Why compile-time complexity matters in a safety handbook 615
17.1.2 What "compile-time complexity" means for templates 616
17.1.3 The instantiation model: why templates multiply work 616
17.1.4 Primary drivers of template compile-time blowups 617
17.1.5 How to measure: make compile-time visible 620
17.1.6 Engineering patterns to control compile-time complexity 621
17.1.7 Safety-focused guidance: balancing compile time and robustness . . 624
17.2 Error Readability 625
17.2.1 Why template errors become unreadable 625
17.2.2 The most effective tool: Concepts and constraints (C++20+) 626
17.2.3 Fail early with `static_assert` and clear messages 628
17.2.4 Reduce overload ambiguity: keep candidate sets small 629
17.2.5 Make requirements readable: use `requires` expressions 630
17.2.6 Prefer simple type-level APIs: alias templates and traits 631
17.2.7 Control instantiation depth: isolate heavy templates 631

17.2.8 Improve errors by improving APIs: “value semantics” alternatives . 632
17.2.9 Tooling and build settings that help (portable guidance) 632
17.2.10 Professional checklist: writing templates that fail well 632
17.3 Safer Template Design 633
17.3.1 Why templates need “safety design” 633
17.3.2 Design principle 1: constrain templates at the boundary (C++20+) . 634
17.3.3 Design principle 2: fail early, fail clearly 636
17.3.4 Design principle 3: separate owning from non-owning interfaces . . 636
17.3.5 Design principle 4: avoid forwarding-reference traps 638
17.3.6 Design principle 5: avoid returning dangling references 639
17.3.7 Design principle 6: avoid implicit narrowing and surprising conversions 639
17.3.8 Design principle 7: choose the right generic abstraction 640
17.3.9 Design principle 8: control specialization and customization points 641
17.3.10 Design principle 9: reduce compile-time cost without sacrificing safety 642
17.3.11 Design principle 10: auditability and maintainability are safety properties 643
17.3.12 Practical “safe template” patterns (ready to reuse) 643
17.4 Concepts (C++20+) 645
17.4.1 What Concepts are 645
17.4.2 The building blocks 646
17.4.3 Requires-expressions: describing what must be valid 647
17.4.4 How constraints affect overload resolution (why this improves safety) 649
17.4.5 Standard library concepts you should use 650
17.4.6 Concepts vs SFINAE: what changes and what does not 650
17.4.7 Safety-oriented best practices 651
17.4.8 Common pitfalls 652
17.4.9 Practical checklist for Concepts adoption 653

18 Type Safety and Strong Types **655**

18.1 Avoiding Primitive Obsession 655

18.1.1 What “primitive obsession” means 655

18.1.2 Why primitives are dangerous in safety-critical code 656

18.1.3 The goal: make invalid states and invalid calls harder 657

18.1.4 Technique 1: Prefer standard strong types 657

18.1.5 Technique 2: Use `enum class` instead of magic integers/booleans 658

18.1.6 Technique 3: Strong typedefs (distinct types with the same representation) 659

18.1.7 Technique 4: Enforce invariants at construction 660

18.1.8 Technique 5: Make units explicit (and hard to mix) 662

18.1.9 Technique 6: Strong error types instead of integers 663

18.1.10 Where to apply strong types first (highest payoff) 663

18.1.11 Interop strategy: strong types outside, primitives inside 664

18.1.12 Performance and memory impact 664

18.1.13 Common pitfalls and how to avoid them 664

18.1.14 Professional checklist: eliminating primitive obsession 665

18.2 `enum class` 666

18.2.1 What `enum class` is 666

18.2.2 The safety problems with unscoped `enum` 667

18.2.3 How `enum class` solves these problems 667

18.2.4 Choosing the underlying type 668

18.2.5 Comparisons and ordering 669

18.2.6 Switch statements and exhaustive handling 670

18.2.7 Bitmask enums (flags) with `enum class` 671

18.2.8 Interoperability with C and legacy APIs 672

18.2.9 Common mistakes and professional recommendations 673

18.2.10 Checklist: adopting `enum class` in safety code 674
18.3 Strong Typedefs 675
18.3.1 What a “strong typedef” is 675
18.3.2 Why plain `using` is not enough 675
18.3.3 Design goals for strong typedefs in safety code 676
18.3.4 Technique 1: Minimal strong typedef wrapper (“tagged type”) . . . 676
18.3.5 Choosing which operations to support (do not overexpose) 678
18.3.6 Technique 2: Add comparisons (C++11+) and optional three-way (C++20+) 678
18.3.7 Technique 3: Strong typedefs for units and quantities 680
18.3.8 Technique 4: Enforcing invariants (valid-by-construction strong types) 681
18.3.9 Explicit conversion at boundaries (and only at boundaries) 682
18.3.10 Interoperability patterns 682
18.3.11 Where strong typedefs deliver the biggest safety gains 683
18.3.12 Common pitfalls and safe recommendations 684
18.3.13 Professional checklist 685
18.4 Safer APIs 686
18.4.1 Why API design is a safety topic 686
18.4.2 Principle 1: Express ownership in types 687
18.4.3 Principle 2: Express borrowing and nullability clearly 688
18.4.4 Principle 3: Make bounds explicit 688
18.4.5 Principle 4: Prefer strong types over primitives 689
18.4.6 Principle 5: Make failure explicit and consistent 691
18.4.7 Principle 6: Use const-correctness and value categories wisely . . . 692
18.4.8 Principle 7: Clarify lifetime and threading contracts 693
18.4.9 Principle 8: Ensure APIs are hard to misuse 693
18.4.10 Design patterns for safer APIs (common recipes) 694

18.4.11 Checklist: auditing an API for safety 695

VI Modern C++ Safety Tools and Language Features 697

19 Sanitizers and Runtime Analysis 699

19.1 AddressSanitizer 699
19.1.1 What AddressSanitizer is 699
19.1.2 How ASan works (mental model) 700
19.1.3 When to use ASan 700
19.1.4 Building with ASan (Clang / GCC) 701
19.1.5 Building with ASan (MSVC on Windows) 702
19.1.6 Running ASan-instrumented programs 702
19.1.7 Suppressions and ignore lists 703
19.1.8 A small example: ASan catching use-after-free 704
19.1.9 Limitations and correct expectations 705
19.1.10 Professional workflow (recommended) 705
19.2 UndefinedBehaviorSanitizer 706
19.2.1 What UndefinedBehaviorSanitizer is 706
19.2.2 What UBSan can detect (practical categories) 707
19.2.3 How UBSan works (a useful mental model) 708
19.2.4 Building with UBSan (Clang / GCC) 709
19.2.5 Running UBSan: controlling behavior with `UBSAN_OPTIONS` . . 710
19.2.6 Suppressions and targeted disabling 710
19.2.7 A minimal example: shift-out-of-bounds 711
19.2.8 How UBSan fits with other sanitizers 712
19.2.9 Platform notes (important for planning) 713
19.2.10 Professional workflow (recommended) 713

19.3 ThreadSanitizer 714
19.3.1 What ThreadSanitizer is 714
19.3.2 What TSan detects in practice 715
19.3.3 How TSan works (a useful mental model) 716
19.3.4 Supported platforms (important planning note) 716
19.3.5 Building with TSan (Clang / GCC) 717
19.3.6 Running TSan: `TSAN_OPTIONS` 718
19.3.7 A minimal example: a real data race 718
19.3.8 Fix patterns (what TSan reports usually mean) 719
19.3.9 Common false alarms and how to reason about them 721
19.3.10 Performance and usage guidance 722
19.3.11 Professional workflow (recommended) 722
19.4 When and how to use them 723
19.4.1 Why sanitizers belong in every safety workflow 723
19.4.2 Which sanitizer to use (mapping defect class to tool) 724
19.4.3 When to run sanitizers 724
19.4.4 How to build sanitized configurations correctly 725
19.4.5 Runtime configuration: make reports actionable 727
19.4.6 How to interpret reports (and avoid common mistakes) 728
19.4.7 Suppressions and ignore lists: use them like a scalpel 729
19.4.8 Where sanitizers fit in a modern CI pipeline 730
19.4.9 How to integrate sanitizers with CMake (typical pattern) 730
19.4.10 Practical “how-to” recipes 731
19.4.11 Common pitfalls (and how to avoid them) 732
19.4.12 Bottom line recommendations 732

20 Static Analysis Tools **734**

20.1 What static analysis can (and cannot) catch 734

20.1.1 What static analysis is (in one sentence) 734

20.1.2 Two major families of static analysis 734

20.1.3 What static analysis can catch well 736

20.1.4 What static analysis usually cannot catch (or catches unreliably) . . 737

20.1.5 Concrete examples: what the tool can and cannot infer 739

20.1.6 Why static analysis sometimes “misses obvious bugs” 741

20.1.7 Practical guidance: how to get the best results 741

20.1.8 Summary: the correct expectations 742

20.2 Professional workflows 743

20.2.1 Goal: make static analysis routine, not heroic 743

20.2.2 Baseline workflow: five stages 744

20.2.3 Workflow A: clang-tidy in daily development 746

20.2.4 Workflow B: Clang Static Analyzer for deeper bug finding 747

20.2.5 Workflow C: MSVC Code Analysis and Core Guidelines checkers . 749

20.2.6 Workflow D: CodeQL for security and variant analysis 749

20.2.7 Putting it together: a practical CI blueprint 750

20.2.8 How to keep the workflow stable and low-noise 751

20.2.9 Minimal templates you can adapt 752

21 Modern Language Features for Safety **754**

21.1 `std::span` 754

21.1.1 What `std::span` is 754

21.1.2 Why `std::span` is a safety feature 755

21.1.3 Core concepts: extent, size, and representation 755

21.1.4 Construction: where spans come from 756

21.1.5 The most important rule: a span does not extend lifetime 757
21.1.6 Indexing and bounds: what `span` does and does not do 758
21.1.7 Slicing safely: `subspan`, `first`, `last` 759
21.1.8 Working with bytes: `as_bytes` and `as_writable_bytes` . . 759
21.1.9 API patterns that become safer with `span` 760
21.1.10 Common mistakes and professional rules 761
21.1.11 Checklist: adopting `std::span` safely 762
21.2 `std::optional` 763
21.2.1 What `std::optional` is 763
21.2.2 Why `std::optional` improves safety 763
21.2.3 Core operations and correct usage 764
21.2.4 Access patterns: safe, safer, and dangerous 765
21.2.5 Modifying state: `reset` and `emplace` 767
21.2.6 C++23 monadic operations: safer composition without nested `if` . 767
21.2.7 Design guidance: choosing `optional` vs alternatives 769
21.2.8 Common pitfalls and professional rules 770
21.2.9 Checklist: safe adoption of `std::optional` 771
21.3 `std::expected` 772
21.3.1 What `std::expected` is 772
21.3.2 Why `std::expected` is a safety feature 773
21.3.3 Key associated types and tags 773
21.3.4 Core operations (the minimal set you must master) 774
21.3.5 Construction patterns (safe and idiomatic) 775
21.3.6 Monadic operations (C++23): composing safely without nested `if` 776
21.3.7 `expected<void, E>`: success without a value 778
21.3.8 Choosing the error type `E` (professional guidance) 778
21.3.9 Common misuse patterns and how to avoid them 779

21.3.10 Practical API patterns (recommended) . . . 780
21.3.11 Checklist: adopting `std::expected` safely . . . 781
21.4 `constexpr` for correctness . . . 782
21.4.1 Why `constexpr` matters for safety . . . 782
21.4.2 Mental model: compile-time evaluation vs runtime evaluation . . . 783
21.4.3 How `constexpr` improves correctness . . . 784
21.4.4 Modern `constexpr`: what changed from C++11 to C++23 . . . 787
21.4.5 Correctness patterns you can standardize in a codebase . . . 788
21.4.6 What `constexpr` cannot guarantee . . . 790
21.4.7 Costs and pitfalls (professional guidance) . . . 790
21.4.8 Checklist: `constexpr` as a correctness tool . . . 791
21.5 Safer initialization patterns . . . 792
21.5.1 Why initialization is a safety topic . . . 792
21.5.2 Rule 1: Always initialize (and do it where the variable is defined) . . . 793
21.5.3 Rule 2: Prefer braced initialization `{}` when it improves correctness 794
21.5.4 Rule 3: Use constructors to enforce invariants . . . 795
21.5.5 Rule 4: Prefer RAII and factory functions for owning objects . . . 796
21.5.6 Rule 5: Represent "may be missing" explicitly . . . 797
21.5.7 Rule 6: Make static and global initialization safe . . . 798
21.5.8 Rule 7: Prefer simple, explicit lazy initialization patterns . . . 800
21.5.9 Rule 8: Use designated initializers for aggregates when it reduces mistakes (C++20) . . . 800
21.5.10 A compact professional checklist . . . 801

VII Professional Engineering Practices 803

22 Designing Safe APIs 805

22.1 Expressing ownership in interfaces 805
22.1.1 Why ownership belongs in the type, not in comments 805
22.1.2 Ownership vocabulary: four common intent categories 806
22.1.3 Rule set: parameter types that communicate ownership 807
22.1.4 Return types: how to communicate ownership back to the caller . . 809
22.1.5 Factory functions: build-then-publish is safer than partially-initialized objects 810
22.1.6 Where raw pointers still appear (and how to keep them safe) 811
22.1.7 Polymorphism and ownership: the destructor rule 811
22.1.8 A compact decision table you can apply consistently 812
22.1.9 Checklist: professional ownership clarity 813
22.2 Minimizing misuse 814
22.2.1 Why “misuse resistance” is an API requirement 814
22.2.2 Principle 1: Put the contract in the type system 814
22.2.3 Principle 2: Make invalid states unrepresentable 816
22.2.4 Principle 3: Remove ambiguous or dangerous call patterns 817
22.2.5 Principle 4: Make ignoring results difficult 818
22.2.6 Principle 5: Enforce non-null and bounds expectations 819
22.2.7 Principle 6: Make preconditions explicit and checkable 820
22.2.8 Principle 7: Reduce error-prone conversions 821
22.2.9 A practical review checklist for “misuse-minimized” APIs 821
22.3 Fail-fast design 822
22.3.1 What “fail-fast” means in C++ safety engineering 822
22.3.2 A decision rule: bug vs recoverable failure 823
22.3.3 Fail-fast tools in modern C++ 824
22.3.4 Design patterns that minimize misuse and amplify fail-fast 827
22.3.5 Fail-fast and debugging: making termination useful 829

22.3.6 Contracts in the language: direction (C++26 work) 829
22.3.7 Checklist: a practical fail-fast policy for safe APIs 830

23 Error Handling Strategies 832
23.1 Exceptions vs. error codes 832
23.1.1 The real question: what kind of failure is this? 832
23.1.2 What exceptions provide (and what they cost) 833
23.1.3 What error codes provide (and what they cost) 834
23.1.4 A modern compromise: result types (`expected`) 835
23.1.5 Safety tradeoffs in practice 836
23.1.6 Choosing between exceptions and error codes: a decision guide . . 837
23.1.7 How to avoid the two biggest failure modes 838
23.1.8 A practical hybrid strategy used in many professional codebases . . 839
23.2 RAII and exceptions 840
23.2.1 Why RAII and exceptions fit together 840
23.2.2 The three practical exception-safety guarantees 841
23.2.3 Stack unwinding: what actually happens 841
23.2.4 The destructor rule: destructors must not throw 841
23.2.5 RAII building blocks you should standardize on 844
23.2.6 Designing RAII types for exception safety 846
23.2.7 Writing exception-safe operations: commit/rollback 847
23.2.8 `noexcept` boundaries and termination 848
23.2.9 Checklist: RAII + exceptions done professionally 849
23.3 Designing failure-safe code 850
23.3.1 What “failure-safe” means in C++ 850
23.3.2 The minimum bar: the Basic exception-safety guarantee 851
23.3.3 Rule 1: Use RAII everywhere to prevent leaks 852

23.3.4 Rule 2: Keep invariants true at all times . . . 853
23.3.5 Rule 3: Never let destructors throw . . . 855
23.3.6 Rule 4: Build-then-publish for initialization safety . . . 856
23.3.7 Rule 5: Make failure propagation hard to ignore . . . 857
23.3.8 Rule 6: Decide where to fail fast . . . 858
23.3.9 A short implementation checklist . . . 858

24 Testing for Memory Safety **860**
24.1 Unit testing limitations . . . 860
24.1.1 Why unit tests matter, and why they are not enough . . . 860
24.1.2 Limitation 1: Unit tests sample executions, not the full state space . 861
24.1.3 Limitation 2: Undefined behavior can pass tests . . . 861
24.1.4 Limitation 3: Memory corruption is often delayed and misattributed 862
24.1.5 Limitation 4: Concurrency bugs evade unit tests . . . 863
24.1.6 Limitation 5: Sanitizer coverage depends on how you build and run tests . . . 864
24.1.7 Limitation 6: Tests can accidentally introduce UB themselves . . . 864
24.1.8 Limitation 7: Unit tests validate behavior, not contracts . . . 865
24.1.9 What to do professionally: treat unit tests as one layer . . . 865
24.1.10 A concise checklist: when unit tests are likely to miss memory bugs 866
24.2 Stress and fuzz testing . . . 867
24.2.1 Why memory safety needs more than unit tests . . . 867
24.2.2 Fuzz testing fundamentals (coverage-guided fuzzing) . . . 868
24.2.3 Writing a good fuzz target (harness) . . . 869
24.2.4 Engine choices and where each fits . . . 870
24.2.5 Fuzzing + sanitizers: the standard best practice . . . 871
24.2.6 Stress testing fundamentals . . . 871

24.2.7 Practical stress tests for memory safety and concurrency 872
24.2.8 Combining fuzzing and stress testing 874
24.2.9 What these techniques cannot guarantee 875
24.2.10 A compact professional workflow 875
24.3 Concurrency testing 876
24.3.1 Why concurrency bugs are uniquely hard to test 876
24.3.2 What you must test for 877
24.3.3 A professional concurrency testing stack 877
24.3.4 Testing patterns for common concurrency hazards 881
24.3.5 Interpreting results correctly 883
24.3.6 Checklist: a practical concurrency-testing policy 883

VIII Modern C++ up to C++26 885

25 The Evolution of Safety in C++ 887
25.1 From C++11 to C++26 887
25.1.1 What changed, and why it matters for safety 887
25.1.2 C++11: the safety foundation 888
25.1.3 C++14: small changes that reduce error surface 889
25.1.4 C++17: vocabulary types and safer interfaces 890
25.1.5 C++20: constraints and structured concurrency tools 891
25.1.6 C++23: safer error propagation and better diagnostics 892
25.1.7 C++26: contracts, hardening, and safety-oriented library additions . 893
25.1.8 Cross-version safety trend: from “discipline” to “expressed intent” . 895
25.1.9 What to standardize on today (C++11 $\rightarrow$ C++26 mindset) 896
25.2 Problems that were solved 896
25.2.1 Overview: safety progress as problem elimination 897

25.2.2 Problem 1: “Who owns this pointer?” (ownership ambiguity) . . . 897
25.2.3 Problem 2: “Error ignored” bugs (unchecked results) 898
25.2.4 Problem 3: “Sentinel values” and unsafe out-parameters for failures 899
25.2.5 Problem 4: “Pointer without size” APIs (bounds mistakes) 900
25.2.6 Problem 5: “Forgot to join” threads and unsafe thread lifetime . . . 901
25.2.7 Problem 6: “Template misuse is discovered too late” 902
25.2.8 Problem 7: “Contracts live only in comments” 903
25.2.9 What remains unsolved (and why that matters) 904
25.3 Problems that remain 905
25.3.1 Context: modern C++ is safer, not automatically safe 905
25.3.2 1) Undefined behavior still exists (and can invalidate testing) 905
25.3.3 2) Lifetime safety is still a human responsibility 906
25.3.4 3) Bounds safety is still not guaranteed 907
25.3.5 4) Data races are still UB, and testing cannot enumerate interleavings 907
25.3.6 5) Atomics remain easy to misuse 908
25.3.7 6) Contracts help express intent, but they do not make C++ memory-safe909
25.3.8 7) Unsafe escape hatches remain necessary (and risky) 910
25.3.9 8) Legacy code and ecosystem constraints do not disappear 910
25.3.10 Practical conclusion: the remaining problems define the safety workflow911

26 Writing Future-Proof C++ Code 912
26.1 Avoiding Deprecated Idioms 912
26.1.1 Why deprecation matters in safety-focused C++ 912
26.1.2 What “deprecated” means in C++ (and what it does not) 913
26.1.3 C++26 removals you should avoid *now* 913
26.1.4 C++23 deprecations still common in codebases 918

26.1.5 Safety-driven “deprecated idioms” even when the Standard does not deprecate them . . . 919
26.1.6 Migration patterns you can apply mechanically . . . 922
26.1.7 Enforcement: keep deprecated usage out of the codebase . . . 923
26.1.8 Summary checklist . . . 924
26.2 Designing for evolution . . . 924
26.2.1 What it means to design for evolution . . . 924
26.2.2 Architectural levers that keep change safe . . . 925
26.2.3 Compatibility strategy: source, ABI, and behavioral stability . . . 927
26.2.4 Design contracts that can evolve safely . . . 928
26.2.5 Error handling that survives change . . . 930
26.2.6 Concurrency design that can adopt new standards incrementally . . . 931
26.2.7 Build and module boundaries that tolerate toolchain evolution . . . 932
26.2.8 Deprecation policy as an evolution tool . . . 933
26.2.9 Practical checklist for evolution-ready C++ APIs . . . 933
26.3 Long-term maintainability . . . 934
26.3.1 Maintainability is a safety feature . . . 934
26.3.2 Design principles that keep code maintainable for a decade . . . 935
26.3.3 Resource and lifetime discipline (the foundation of sustainable code) 937
26.3.4 Error handling that remains readable under change . . . 938
26.3.5 Concurrency maintainability . . . 939
26.3.6 Contracts and profiles: designing code that can be checked more deeply over time . . . 940
26.3.7 Tooling and process: how maintainability is sustained . . . 941
26.3.8 Preparing for language evolution up to C++26 . . . 942
26.3.9 Long-term maintainability checklist . . . 943

Conclusion **944**

C++ is safe when used professionally 944

Discipline beats restrictions 950

Modern C++ as an engineering contract 958

Appendices **967**

Appendix A — Common Memory Bugs Checklist 967

26.4 Appendix B — RAII Design Checklist 980

Appendix B — RAII Design Checklist 980

26.5 Appendix C — Smart Pointer Decision Guide 989

Appendix C — Smart Pointer Decision Guide 989

26.6 Appendix D — Debugging Memory Issues in Practice 999

Appendix D — Debugging Memory Issues in Practice 999

26.7 Appendix E — Glossary of Modern C++ Safety Terms 1009

Appendix E — Glossary of Modern C++ Safety Terms 1009

References **1019**

Author's Note

Why this booklet was written

The problem this book addresses

Modern C++ is a high-performance systems language that is used to build operating systems components, browsers, game engines, trading systems, databases, embedded firmware, robotics stacks, and large-scale server infrastructure. In these environments, failures are rarely "small": a single memory bug can become a security vulnerability, a rare crash in production, silent data corruption, or a non-reproducible heisenbug that consumes weeks of engineering time.
The reality is simple:

- C++ gives you powerful control over memory, object lifetimes, and concurrency.
- That same control makes it easy to produce undefined behavior, data races, lifetime errors, and resource leaks if discipline and structure are missing.

This booklet was written to be a practical engineering guide that turns "safety" from a vague intention into a concrete, repeatable workflow.

Why “Modern C++ Safety” must be treated as a system

Safety in C++ is not one feature, one library, or one rule. It is a system:

- **Language rules** (object lifetimes, aliasing, initialization, evaluation order)
- **Library design** (RAII, ownership types, value semantics, error channels)
- **Tooling** (sanitizers, static analysis, warnings as errors, fuzzing)
- **Concurrency discipline** (ownership across threads, synchronization, invariants)
- **Project architecture** (APIs that prevent misuse, clear contracts, test strategy)

If any part is missing, the whole program becomes fragile. This booklet treats safety as an engineering system that must be designed, enforced, and verified.

Why this book spans C++11 to C++26

C++11 was a turning point: it introduced the modern foundation for safe resource management and concurrency (move semantics, smart pointers, threading primitives, atomics, and a vastly improved standard library).
From C++11 through C++23, the language and library steadily expanded the tools that enable safer design: better type utilities, improved containers, constexpr evolution, stronger library abstractions, and more expressive generic programming. As of today, the current ISO C++ standard is C++23, formally published as ISO/IEC 14882:2024(E). The C++26 work is in progress within WG21, and major compilers continue to implement adopted features.
This handbook covers:

- what is stable and standardized (C++11 through C++23),
- what is practically relevant from ongoing C++26 evolution,

- and how to build robust systems *today* using current toolchains.

A professional definition of “memory safety” in C++

In this book, “memory safety” means the disciplined avoidance and detection of:

- out-of-bounds access (stack/heap/global)
- use-after-free and use-after-scope
- double-delete and invalid free
- uninitialized reads
- violating object lifetime rules (dangling references/pointers, invalid views)
- undefined behavior that silently breaks correctness or security

C++ does not automatically prevent these categories. Instead, it gives you the tools to *design them out* and *detect them early*.

Why concurrency is included as a first-class safety topic

Many severe production failures are not caused by single-threaded bugs, but by concurrency hazards:

- data races
- broken invariants due to weak synchronization
- deadlocks, livelocks, and starvation
- unsafe publication and lifetime issues across threads

Concurrency safety is inseparable from memory safety because thread boundaries change object lifetimes, ownership assumptions, and the validity of references. This handbook treats concurrency as part of the same safety model, not as an optional advanced chapter.

Why tooling is emphasized: safety must be verified, not assumed

Professional C++ safety is not achieved by "being careful." It is achieved by **making bugs expensive to write and cheap to detect**.
That is why this booklet treats the following as mandatory engineering tools:

- **AddressSanitizer (ASan)** for detecting memory errors at runtime
- **UndefinedBehaviorSanitizer (UBSan)** for catching common undefined behavior
- **ThreadSanitizer (TSan)** for detecting data races
- **Static analysis** and compiler warnings configured as engineering gates

These tools exist across major ecosystems (Clang/LLVM, GCC toolchains, and MSVC), and their practical usage has matured significantly in recent years, including stronger platform support and clearer integration guidance.

A small example: why "it works on my machine" is not a safety argument

The following bug may appear to work, until it suddenly does not:

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

int main() {
    std::vector<int> v{1,2,3};
    int* p = &v[0];          // pointer into the vector's storage
```

```
    v.push_back(4);          // may reallocate; p may now dangle
    ↪ (undefined behavior)
    std::cout << *p << "\n"; // may print 1, may crash, may corrupt data
}
```

A safer design uses stable ownership and stable references only when the container guarantees it, or it avoids keeping raw pointers across operations that can invalidate them:

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

int main() {
    std::vector<int> v{1,2,3};
    std::size_t i = 0;       // keep an index instead of a raw pointer
    v.push_back(4);          // reallocation is fine; index remains valid
    std::cout << v[i] << "\n";
}
```

This is the mindset of the handbook: **design APIs and code so that common operations cannot silently create undefined behavior**.

What you should expect from this handbook

This booklet is written for practicing engineers who want repeatable results. You should expect:

- Clear explanations of high-impact failure modes (memory, lifetime, concurrency).
- Practical rules that are compatible with modern industry guidance.
- Concrete examples of both unsafe code and robust alternatives.
- Tool-driven verification practices suitable for CI and production workflows.
- Emphasis on maintainable designs, not just isolated “safe snippets.”

What this handbook does not try to be

To remain useful and actionable, this booklet is *not*:

- a complete language reference,
- a compiler manual,
- a formal verification textbook,
- or a collection of style preferences.

Its purpose is narrower and more practical: **help you build C++ systems that remain correct under real-world pressure.**

Authoritative foundations used in this booklet

The guidance in this handbook is aligned with widely recognized, authoritative foundations, including:

- The ISO C++ standard status and published standard information from Standard C++ (isocpp.org), which states the current standard is C++23, formally published as ISO/IEC 14882:2024(E).
- The *C++ Core Guidelines* (Bjarne Stroustrup, Herb Sutter, and contributors), maintained by the ISO C++ community.
- Official compiler and tool documentation for sanitizers (Clang/LLVM documentation) and Microsoft Visual C++ AddressSanitizer documentation.

Who this reference is for

Intended professional audience

This reference is written for engineers who work with C++ in environments where correctness, performance, and long-term maintainability matter. It assumes the reader treats C++ as a systems engineering language, not as a scripting or rapid-prototyping tool.
The primary audience includes:

- Professional C++ developers working on production systems
- Engineers responsible for performance-critical or safety-critical code
- Developers maintaining large, long-lived C++ codebases
- Engineers working close to hardware, operating systems, or concurrency

This handbook is not limited to a specific industry. Its guidance applies equally to desktop software, servers, embedded systems, game engines, financial systems, networking stacks, and infrastructure software.

Required background knowledge

This booklet assumes that the reader already:

- understands basic C++ syntax and compilation
- is familiar with functions, classes, templates, and the standard library
- has written real C++ programs beyond toy examples

It does *not* assume expert-level mastery of every language feature. Concepts are introduced incrementally, but the focus is on engineering judgment rather than language memorization. If you are completely new to C++, this handbook will be difficult without prior study. It is designed as a professional reference, not an introductory tutorial.

For engineers transitioning to Modern C++

Many experienced developers learned C++ before C++11 and continue to work in legacy codebases. This reference is especially written for engineers who:

- were trained in pre-C++11 C++ (manual memory management, raw pointers)
- inherited large legacy systems with unsafe patterns
- want to modernize code safely without rewriting everything

The handbook explains *why* modern techniques exist and how they map to real failure modes observed in older designs.

For engineers responsible for correctness and reliability

This reference is particularly relevant if you are accountable for:

- production crashes and rare, non-reproducible bugs
- security vulnerabilities related to memory or concurrency
- long-term maintenance costs and technical debt
- code review standards and engineering guidelines

The material is written to support decision-making under responsibility, not just code writing in isolation.

For engineers working with concurrency and parallelism

If your software uses threads, tasks, asynchronous execution, or shared state, this handbook is directly applicable.
It addresses:

- ownership transfer across threads
- safe publication of objects
- data race prevention strategies
- lifetime management in concurrent systems

Concurrency topics are treated as part of safety engineering, not as an advanced afterthought.

For teams and organizations

This reference is suitable not only for individuals, but also for teams and organizations that want shared standards.
It can be used to:

- establish internal safety guidelines
- define code review checklists
- standardize tooling expectations (sanitizers, warnings, analysis)
- train new team members in disciplined C++ practices

The guidance emphasizes repeatable processes that scale beyond a single developer.

Who this reference is *not* for

To avoid misuse, it is important to state clearly who this book is not aimed at.
This reference is *not* intended for:

- absolute beginners learning their first programming language
- developers seeking quick syntax tricks or shortcuts
- projects where correctness and reliability are optional
- environments where C++ is used casually without testing or tooling

C++ safety requires discipline. This handbook assumes the reader is willing to adopt that discipline.

How to use this reference effectively

This book is designed to be used as:

- a reference during design and code review
- a checklist when introducing new APIs or subsystems
- a guide when diagnosing subtle memory or concurrency failures
- a foundation for building internal engineering standards

It is not meant to be read once and forgotten. Its value increases when it is revisited as systems evolve and complexity grows.

A final note on professional mindset

This reference assumes one core principle:

> C++ safety is not achieved by avoiding power, but by using power with structure, clarity, and verification.

If that principle resonates with you, this handbook was written for you.

How it differs from traditional C++ books

Not a language tour, but a safety-focused engineering reference

Traditional C++ books often aim to explain the language feature by feature: syntax, keywords, templates, containers, and algorithms. While that approach is valuable for learning the language, it does not directly address how real systems fail.
This handbook takes a different approach. It does not ask:

> "What features does C++ have?"

Instead, it asks:

> "How do real C++ systems break, and how do we prevent that?"

Language features are introduced only when they serve a clear safety, correctness, or robustness purpose.

Problem-driven rather than feature-driven

Most traditional books are structured around language constructs:

- classes and inheritance
- templates and metaprogramming
- containers and algorithms
- syntax rules and idioms

This handbook is structured around *failure modes*:

- lifetime and ownership errors
- invalid memory access and undefined behavior
- concurrency hazards and data races
- broken invariants under optimization or reordering
- fragile APIs that invite misuse

Features are discussed only in the context of preventing or detecting these failures.

Safety as an explicit design goal, not an afterthought

In many C++ books, safety advice appears as side notes:

> "Be careful with pointers."
> "Remember to delete what you allocate."
> "Watch out for concurrency issues."

This handbook treats safety as a first-class design objective. Every rule, example, and recommendation is evaluated by a single question:

> Does this design reduce the probability of catastrophic failure?

If the answer is unclear, the design is considered incomplete.

Emphasis on lifetime, ownership, and invariants

Traditional C++ texts often emphasize object-oriented structure or generic abstraction, but spend relatively little time on:

- precise object lifetime boundaries
- ownership transfer and responsibility
- invariants that must hold across threads and APIs

This handbook places these concepts at the center. It treats ownership, lifetime, and invariants as the *true architecture* of a C++ system, with syntax serving only as an implementation detail.

Concurrency integrated, not isolated

Many books isolate concurrency into a single advanced chapter near the end. As a result, readers subconsciously treat multithreading as optional or exceptional.
This reference integrates concurrency into the safety model from the beginning. Memory safety and concurrency safety are treated as inseparable, because:

- object lifetimes change across threads
- unsynchronized access invalidates assumptions
- data races silently destroy correctness

Concurrency is presented as a normal condition of modern systems, not as a specialized niche topic.

Tooling as a core discipline, not optional advice

Traditional books often mention debugging or analysis tools briefly, if at all. This handbook assumes that modern C++ safety is impossible without tooling.
It treats the following as standard engineering practice:

- runtime sanitizers for memory and concurrency
- static analysis integrated into builds
- compiler warnings configured as hard errors
- continuous verification through testing and CI

The message is explicit: *safety must be verified, not trusted*.

Focused on production reality, not idealized examples

Classic C++ books often rely on small, self-contained examples that work in isolation. Real systems are not small, isolated, or forgiving.
This handbook emphasizes:

- long-lived codebases
- partial refactoring under constraints
- backward compatibility and incremental improvement
- diagnosing rare, non-reproducible failures

Examples are chosen to reflect realistic usage patterns and failure scenarios, not just conceptual clarity.

Compatibility with industry guidance, not personal style

Many traditional books reflect the personal style or preferences of the author. This handbook deliberately avoids subjective style debates.
Its guidance aligns with:

- widely accepted industry safety principles
- modern ISO C++ evolution and intent
- established best practices from large-scale systems engineering

The goal is consistency and predictability, not aesthetic preference.

Designed to be used repeatedly, not read once

Traditional books are often read sequentially from start to finish. This handbook is designed as a reference that grows more valuable over time.
It is meant to be:

- consulted during design discussions
- referenced during code reviews
- used when diagnosing subtle failures
- revisited as systems evolve

Its structure supports selective reading and repeated use under pressure.

A different promise to the reader

Most C++ books promise knowledge of the language. This handbook promises something narrower, but more demanding:

> If you apply the practices in this book consistently, your C++ systems will fail less often, fail more predictably, and be easier to reason about when they do fail.

That difference defines the purpose of this reference.

`Ayman Alheraki`

Preface

C++ as a systems programming language

What “systems programming” means in this handbook

In this handbook, *systems programming* means building software where you must control the fundamental costs and behaviors of execution:

- memory layout, object lifetime, and ownership
- CPU usage, latency, and predictable performance
- direct interaction with the operating system, hardware, and ABIs
- concurrency under a defined memory model
- long-lived robustness under real production pressure

C++ was designed for this domain: it provides low-level access comparable to C, while also providing high-level abstractions intended to compile down efficiently.

Two pillars: hardware access and zero-overhead abstractions

C++ is unusual because it deliberately combines two goals:

1. **Direct mapping to the machine and OS:** pointers, explicit object lifetime, predictable data layouts, and efficient calling conventions.

2. **Abstractions that can be as efficient as hand-written low-level code:** RAII, templates, generic programming, and library types that can be optimized away.

This combination is why C++ remains a primary choice when performance, integration, and control are non-negotiable. It is also why C++ safety must be engineered: the language does not prevent all misuse by default.

Why C++ is still used for critical infrastructure

C++ remains common in critical infrastructure because it can satisfy requirements that many languages cannot meet simultaneously:

- **Performance and predictability:** finely controlled allocation and data locality.
- **Resource ownership as a first-class concept:** deterministic cleanup via RAII.
- **Interoperability:** stable C interfaces, ABIs, and integration with existing ecosystems.
- **Portability:** from embedded devices to large servers with shared code patterns.

However, these strengths are inseparable from risk. In C++, misuse often becomes **undefined behavior**: programs may appear to work, then fail unpredictably under optimization, different builds, different input, or different machines.

Modern C++ (C++11 and later) changed the safety baseline

C++11 introduced language and library facilities that fundamentally improved the safety baseline for production code:

- move semantics (enabling efficient ownership transfer)
- smart pointers and stronger RAII patterns
- a standardized concurrency model with atomics and threads

Later standards expanded the toolkit with stronger library abstractions (for example, `std::span` for safe views, and `std::jthread` for safer thread lifetime management), and better support for building robust systems using library types rather than raw primitives. This handbook uses "Modern C++" to mean C++11 through today's published standard, and it also discusses practical, compiler-supported evolution toward C++26 where it matters.

The central engineering trade-off: power without automatic safety

In many environments, safety comes from a managed runtime or strict restrictions. C++ takes a different route: it provides power and trusts engineering discipline.
This has two consequences:

- You can build extremely fast and precise systems.
- You must design code so that dangerous states are difficult or impossible to represent.

The C++ Core Guidelines emphasize that many rules are architectural: they shape interfaces, resource management, memory management, and concurrency. In other words, safety is not a "coding trick"; it is a design property.

RAII as the primary safety mechanism for resources

The most important safety technique in idiomatic C++ is RAII: bind every resource (memory, file handles, sockets, locks, transactions) to an object, and release it in the destructor.

Example: replacing manual cleanup with RAII

```
#include <cstdio>

void unsafe_log(const char* path) {
    std::FILE* f = std::fopen(path, "a");
    if (!f) return;

    // Many exits are possible in real code:
    // exceptions, early returns, error branches...
    std::fputs("start\n", f);

    // If we return early here, fclose is skipped -> resource leak.
    if (std::ferror(f)) return;

    std::fputs("done\n", f);
    std::fclose(f);
}
```

```
#include <cstdio>
#include <memory>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept { if (f) std::fclose(f);
    ↪ }
};

void safe_log(const char* path) {
    std::unique_ptr<std::FILE, FileCloser> f(std::fopen(path, "a"));
    if (!f) return;

    std::fputs("start\n", f.get());
```

```
    if (std::ferror(f.get())) return; // safe: file is closed
    ↪ automatically
    std::fputs("done\n", f.get());
}
```

RAII is not only about convenience. It is a reliability mechanism: it makes cleanup deterministic, even when control flow becomes complex.

Concurrency is part of systems programming, not an optional chapter

Modern systems are concurrent by default: servers, GUIs, pipelines, and high-performance software all rely on multiple threads or tasks. C++ provides concurrency primitives, but it also has a strict rule:

> A data race in a C++ program results in undefined behavior.

That statement is not a warning; it is a fundamental contract that compilers rely on for optimization. Therefore, concurrency safety is a core part of C++ systems design.

Example: a data race is not “just a bug”

```
#include <thread>
#include <iostream>

int counter = 0;

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

```
int main() {
    std::thread t1(worker), t2(worker);
    t1.join(); t2.join();
    std::cout << counter << "\n"; // may print anything
}
```

```
#include <thread>
#include <atomic>
#include <iostream>

std::atomic<int> counter{0};

void worker() {
    for (int i = 0; i < 1'000'000; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::thread t1(worker), t2(worker);
    t1.join(); t2.join();
    std::cout << counter.load(std::memory_order_relaxed) << "\n";
}
```

The safe version uses atomics to eliminate the data race. The memory order shown is appropriate for a simple counter where you only need atomicity, not ordering constraints.

Bounds and views: safer access without losing performance

Many memory bugs come from indexing errors and invalid pointer arithmetic. Modern C++ encourages representing memory as *ranges* and *views* with explicit size.

Example: prefer size-aware views

```
#include <span>
#include <cstddef>

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

int main() {
    int a[] = {1,2,3,4};
    (void)sum(a); // size is carried by std::span
}
```

Using `std::span` does not automatically prevent every out-of-bounds error, but it makes size explicit and enables safer API contracts than raw pointers.

Tool-assisted safety is mandatory in real systems

Because C++ allows undefined behavior, professional teams verify safety with tools. This handbook treats the following as standard practice for serious C++ systems work:

- **AddressSanitizer (ASan):** detects many memory errors such as out-of-bounds and use-after-free.
- **UndefinedBehaviorSanitizer (UBSan):** detects many common undefined behaviors at runtime.
- **ThreadSanitizer (TSan):** detects data races and some concurrency bugs.

- **Core Guidelines-aligned review and analysis:** rules that shape architecture and interfaces.

The purpose of this book is to connect language rules, library design, and tooling into one coherent engineering approach.

A brief note on standardization and version naming

The current ISO C++ standard is informally called **C++23**. It is formally published as **ISO/IEC 14882:2024(E)** because publication occurred in 2024 even though technical work concluded in 2023. Work on **C++26** is in progress.
This handbook uses the C++11 $\rightarrow$ C++26 framing to cover:

- stable standardized foundations (C++11 through C++23),
- and carefully marked, practically relevant evolution toward C++26.

What this section implies for the rest of the handbook

Because C++ is a systems language:

- safety must be designed into interfaces (ownership, lifetime, invariants),
- concurrency must be treated as a first-class correctness constraint,
- and verification must be continuous (sanitizers, analysis, tests, reviews).

The chapters that follow treat memory safety, concurrency, and robust design as one unified discipline: professional systems engineering in Modern C++.

Why C++ errors are rarely “simple bugs”

In C++, many mistakes change the meaning of the entire program

In many languages, an error typically produces a predictable failure: an exception, a crash with a stack trace, or at worst an incorrect value in a local place.
In C++, a large class of mistakes triggers **undefined behavior**. Undefined behavior means the language standard imposes *no requirements* on what happens next: the program may appear to work, may crash, may corrupt data, or may behave differently under a different compiler, optimization level, or input.
This is why many C++ failures are not “simple bugs” that reproduce reliably. They are often **violations of the rules the compiler assumes you obey**.

Undefined behavior is not a runtime error; it is a broken contract

Undefined behavior is frequently misunderstood as “something bad that happens at runtime”. In practice, it is a design contract: compilers assume undefined behavior *never happens*, and they optimize based on that assumption.
As a result, when undefined behavior exists, the compiler is allowed to generate code that looks surprising to a human reader who expected the program to behave like a defensive interpreter.

Example: out-of-bounds access is not just “reading the wrong element”

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

int main() {
    std::vector<int> v{10, 20, 30};
```

```
    std::cout << v[3] << "\n"; // out-of-bounds access: undefined
    ↪ behavior
}
```

This is not guaranteed to print “garbage” in a consistent way. It may crash, it may print a value that changes between runs, or it may corrupt unrelated state.

Many C++ bugs are temporal: the program breaks later, far away

C++ programs have explicit object lifetimes. If an object is destroyed and you continue to use a pointer or reference to it, the program becomes ill-formed at runtime (undefined behavior). The failure often appears *later* than the mistake and in a different subsystem, because the memory may be reused for something else.

Example: use-after-free can look correct until it is not

```
#include <iostream>

int* make_ptr() {
    int* p = new int(42);
    delete p;
    return p; // dangling pointer
}

int main() {
    int* p = make_ptr();
    std::cout << *p << "\n";   // use-after-free: undefined behavior
}
```

The output is not meaningful even if you happen to see `42` once. Optimization level, allocator behavior, and surrounding code can change the result.

C++ concurrency mistakes are not “racey” results; they are undefined behavior

The C++ memory model defines a strict rule: **a data race makes program behavior undefined**.
This matters because developers sometimes expect:

> “It might just print the wrong number sometimes.”

In C++, it is worse: the program has no defined meaning once a data race occurs.

Example: incrementing a shared counter without synchronization

```
#include <thread>
#include <iostream>

int counter = 0;

void work() {
    for (int i = 0; i < 1'000'000; ++i) {
        ++counter; // data race: undefined behavior
    }
}

int main() {
    std::thread t1(work), t2(work);
    t1.join(); t2.join();
    std::cout << counter << "\n";
}
```

Correct version: make the shared state atomic (or protect it with a mutex)

```
#include <thread>
```

```cpp
#include <atomic>
#include <iostream>

std::atomic<int> counter{0};

void work() {
    for (int i = 0; i < 1'000'000; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::thread t1(work), t2(work);
    t1.join(); t2.join();
    std::cout << counter.load(std::memory_order_relaxed) << "\n";
}
```

The relaxed ordering is appropriate for a simple counter where you only need atomicity, not cross-thread ordering of other data.

C++ errors interact with optimization, inlining, and reordering

When code is compiled with optimization, transformations such as inlining, constant propagation, vectorization, and reordering are applied aggressively.
If the compiler can assume certain behaviors are impossible (because the standard says they are undefined), it may remove checks, reorder operations, or eliminate entire branches. This can make a bug disappear in debug builds and appear in release builds, or do the opposite.
This is why professional safety work in C++ must be validated under realistic builds, not only under “debug” assumptions.

Some failures are architectural: APIs that allow misuse

In many C++ systems, the deepest bugs are not a single bad line. They are **APIs that make it easy to represent unsafe states**:

- returning raw pointers with unclear ownership
- exposing internal references that outlive their source
- allowing concurrent access without clear synchronization contracts
- mixing resource management and business logic without RAII boundaries

Such problems produce failures that look random because they depend on timing, allocation patterns, and the behavior of other components.

Why tooling is essential: you must detect the category, not guess the symptom

Because many failures stem from undefined behavior, guessing by symptoms is unreliable. Professional C++ engineering uses tooling to detect the actual category of failure:

- **AddressSanitizer** for many memory errors (out-of-bounds, use-after-free)
- **UndefinedBehaviorSanitizer** for common undefined behaviors
- **ThreadSanitizer** for data races

These tools instrument code to expose the real rule violation near the source, instead of forcing you to diagnose the final crash site after corruption has spread.

A practical conclusion for the rest of this handbook

C++ errors are rarely “simple bugs” because they often:

- violate rules the compiler depends on (undefined behavior),
- break object lifetime and ownership boundaries (temporal bugs),
- break concurrency rules (data races are undefined behavior),
- and spread across the program before visible symptoms appear.

Therefore, this handbook focuses on three professional habits:

1. **Design for safety:** make ownership, lifetime, and synchronization explicit.
2. **Prefer safe abstractions:** RAII, value semantics, size-aware views.
3. **Verify continuously:** sanitizers, warnings, static analysis, and tests in CI.

Engineering responsibility in C++

C++ gives power, therefore it assigns responsibility

C++ is designed for systems programming: it allows direct control over memory, object lifetime, performance, and concurrency. That control is not neutral. It means the programmer is responsible for maintaining the assumptions that the compiler, the standard library, and the hardware execution model depend on.
In practical terms:

- If you violate object lifetime rules, the program can execute with undefined behavior.

- If you introduce a data race, the program can execute with undefined behavior.
- If you produce an invalid view of memory (bounds errors, dangling pointers), the program can execute with undefined behavior.

These are not “edge cases”. They are the core engineering constraints of the language.

Responsibility means designing code that resists misuse

Professional responsibility in C++ is not mainly about writing careful lines of code. It is about designing interfaces that make incorrect usage hard or impossible.
A responsible C++ design makes the following explicit:

- **Ownership:** who frees this resource, and when?
- **Lifetime:** how long is this object valid?
- **Threading:** who may access this object concurrently, and under what synchronization?
- **Failure behavior:** how does the API report errors, and what cleanup is guaranteed?

Make ownership visible in the type system

When ownership is unclear, bugs become inevitable. The goal is to encode responsibility directly into types and APIs.

Example: unclear ownership (fragile API)

```
struct Image { /* ... */ };

Image* load_image(const char* path); // Who deletes? Caller? Callee?
↪  Shared?
```

```
void use() {
    Image* img = load_image("a.png");
    // If an exception or early return occurs, img may leak.
    // Different callers may choose different deletion rules.
    delete img; // Maybe correct, maybe wrong, maybe double-delete
     ↪  elsewhere.
}
```

Example: ownership made explicit (robust API)

```
#include <memory>

struct Image { /* ... */ };

std::unique_ptr<Image> load_image(const char* path);

void use() {
    auto img = load_image("a.png");
    // Ownership is explicit and single: no leaks, no ambiguity.
}
```

If the intent is shared ownership, use `std::shared_ptr` deliberately, but treat it as a design decision with cost and semantic consequences.

RAII is the engineering foundation for reliable cleanup

RAII (Resource Acquisition Is Initialization) makes cleanup deterministic by binding resource lifetime to object lifetime. This is a core responsibility technique in C++: it prevents resource leaks and enforces cleanup even when control flow becomes complex.

Example: lock management must be automatic

```cpp
#include <mutex>

std::mutex m;

void unsafe() {
    m.lock();
    // If an exception is thrown here, the mutex may remain locked
    ↪ forever.
    m.unlock();
}
```

```cpp
#include <mutex>

std::mutex m;

void safe() {
    std::lock_guard<std::mutex> lock(m); // unlocks automatically
    // Safe even with multiple returns or exceptions.
}
```

In C++, correctness includes the optimizer

C++ compilers are allowed to perform aggressive optimizations. Those optimizations are based on the rules of the language, including the assumption that undefined behavior does not occur. Therefore, an engineer must think in terms of *valid programs*, not only "what seems to work in a debug build".

Responsible practice includes:

- testing under realistic optimization settings
- treating warnings as serious signals

- eliminating undefined behavior rather than trying to “work around” symptoms

Concurrency responsibility: data races are a correctness failure

Concurrency is not only about performance. It is about preserving invariants under simultaneous execution. The C++ concurrency model has a non-negotiable rule: unsynchronized concurrent access to shared state (a data race) makes the program’s behavior undefined.

Example: accidental shared state

```
#include <thread>

int shared_counter = 0;

void work() {
    for (int i = 0; i < 100000; ++i) {
        ++shared_counter; // data race if called concurrently
    }
}

int main() {
    std::thread a(work), b(work);
    a.join(); b.join();
}
```

Example: explicit synchronization

```
#include <thread>
#include <atomic>

std::atomic<int> shared_counter{0};
```

```
void work() {
    for (int i = 0; i < 100000; ++i) {
        shared_counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::thread a(work), b(work);
    a.join(); b.join();
}
```

The correct choice of synchronization depends on the invariants you need. Atomics solve atomicity; mutexes solve larger invariants across multiple variables.

Bounds and views: responsibility includes memory shape

Many severe C++ failures arise from losing track of memory bounds: passing a pointer without its size, slicing arrays, or iterating past the end. Responsible design prefers size-aware interfaces that carry bounds explicitly.

Example: size-less pointer interface (easy to misuse)

```
#include <cstddef>

int sum(const int* p) {            // How many elements? Unknown.
    int s = 0;
    for (std::size_t i = 0; i < 100; ++i) { // Assumption baked into
    ↪  code.
        s += p[i];                 // Potential out-of-bounds.
    }
    return s;
```

```
}
```

Example: size-aware view (explicit contract)

```
#include <span>

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

This does not magically prevent every mistake, but it makes the correct usage natural and misuse easier to detect in review and testing.

Responsibility is enforced through process, not heroics

Professional C++ safety is not built on individual caution. It is built on repeatable engineering process. A responsible team standardizes:

- **Code review rules:** ownership clarity, lifetime rules, synchronization contracts.
- **Compiler configuration:** high warnings, treated as errors for production builds.
- **Static analysis:** regular checks aligned with modern C++ safety guidance.
- **Sanitizers:** memory and concurrency instrumentation in CI and test pipelines.
- **Testing strategy:** targeted unit tests, stress tests, and fuzzing for unsafe surfaces.

This process turns safety from a personal intention into an organizational capability.

A practical checklist for responsible C++ engineering

When designing or reviewing C++ code, insist on the following questions being answered:

1. Who owns each resource, and how is that ownership represented in types?
2. Are object lifetimes explicit, and can references or pointers dangle?
3. Are bounds explicit when memory is passed across an API boundary?
4. Is concurrent access defined, and is the synchronization contract documented and enforced?
5. Are failure paths safe: no leaks, no partial invariants, no throwing destructors?
6. Is verification in place: warnings, analysis, sanitizers, and tests under realistic builds?

If these questions cannot be answered confidently, the design is not yet safe enough.

Conclusion: C++ responsibility is a professional discipline

C++ rewards engineers who treat correctness as a design constraint, not as an afterthought. The responsible approach is consistent:

- design APIs that encode ownership and lifetime
- prefer RAII and type-driven safety
- make concurrency analyzable and race-free
- verify continuously with modern tooling

The rest of this handbook builds these responsibilities into concrete rules, patterns, and workflows you can apply in real production systems.

The difference between legacy C++ and Modern C++

What this handbook means by “legacy” and “modern”

In this handbook:

- **Legacy C++** generally refers to C++98/C++03-era style and codebases that primarily rely on raw pointers, manual resource management, and ad-hoc conventions.
- **Modern C++** refers to C++11 and later, where the language and standard library provide first-class tools for expressing ownership, lifetime, and concurrency safely.

The difference is not cosmetic. It is an engineering shift: Modern C++ makes many safe designs *idiomatic and efficient*, so safety becomes the default choice rather than an optional discipline.

The central shift: from manual discipline to type-driven safety

Legacy C++ often depends on human discipline:

- “Remember to `delete` what you `new`.”
- “Remember to unlock mutexes.”
- “Remember which function owns the pointer.”

Modern C++ prefers **type-driven safety**:

- ownership is represented by types (e.g., `std::unique_ptr`)
- cleanup is automatic via RAII
- concurrency is expressed with standard primitives under a defined memory model

This is not about adding wrappers for comfort; it is about designing code so that incorrect states are difficult to represent and mistakes are easier to detect.

Memory and ownership: raw pointers versus ownership types

Legacy pattern: ownership hidden in comments

```
struct Widget { /* ... */ };

Widget* make_widget(); // "caller must delete" (often undocumented or
↪  forgotten)

void f() {
    Widget* p = make_widget();
    // Many exits are possible: early return, error, exception...
    delete p; // might be forgotten, duplicated, or placed incorrectly
}
```

This pattern fails in real code because control flow is complex. Leaks and double-deletes are not rare accidents; they are natural outcomes of unclear ownership.

Modern pattern: ownership expressed in the return type

```
#include <memory>

struct Widget { /* ... */ };

std::unique_ptr<Widget> make_widget();

void f() {
    auto p = make_widget(); // ownership is explicit and deterministic
} // automatic cleanup
```

Modern C++ makes the safe solution both simpler and typically as efficient as the manual version.

Resource management: manual cleanup versus RAII everywhere

Legacy code often contains paired operations (`open/close`, `lock/unlock`, `init/destroy`) that can be broken by exceptions, early returns, or error branches.

Legacy pattern: cleanup can be skipped

```
#include <mutex>

std::mutex m;

void legacy() {
    m.lock();
    // If an exception occurs here, the mutex may remain locked.
    m.unlock();
}
```

Modern pattern: RAII makes cleanup unconditional

```
#include <mutex>

std::mutex m;

void modern() {
    std::lock_guard<std::mutex> g(m);
    // Always unlocks, regardless of how the function exits.
}
```

RAII is the professional baseline in Modern C++: it prevents entire classes of failures by design.

Move semantics: making safe ownership fast

Legacy C++ often avoided "safe" designs because copying resources was expensive or unclear. C++11 introduced move semantics, allowing efficient transfer of ownership and resources without manual pointer tricks.

Legacy pain: expensive copies or fragile pointer swapping

```
#include <vector>

// Pre-C++11 style often needed manual workarounds to avoid expensive
↪  copies.
std::vector<int> make_vec_legacy() {
    std::vector<int> v;
    // ... fill ...
    return v; // historically could be expensive or rely on optimization
    ↪  assumptions
}
```

Modern reality: value semantics with efficient moves

```
#include <vector>

std::vector<int> make_vec() {
    std::vector<int> v;
    // ... fill ...
    return v; // efficient: moves are available (and NRVO often applies)
}
```

Move semantics enable safe, simple value-based designs to remain efficient.

Bounds and views: pointers with missing size versus size-aware interfaces

A major legacy weakness is passing memory as "pointer only" without size information. Modern C++ increasingly favors size-aware views.

Legacy pattern: pointer without length

```
#include <cstddef>

int sum(const int* p) {        // how many elements?
    int s = 0;
    for (std::size_t i = 0; i < 100; ++i) {
        s += p[i];             // may go out of bounds
    }
    return s;
}
```

Modern pattern: `std::span` carries size with the pointer

```
#include <span>

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

`std::span` is a standard, non-owning view over contiguous memory, available since C++20.

Concurrency: from ad-hoc threads to standard, safer building blocks

Legacy code often used platform-specific threading APIs or early library extensions. Modern C++ has standardized concurrency primitives and a defined memory model.

Modern baseline: standard threads and atomics

```
#include <thread>
#include <atomic>

std::atomic<int> counter{0};

void work() {
    for (int i = 0; i < 100000; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::thread a(work), b(work);
    a.join(); b.join();
}
```

Modern improvement: `std::jthread` for safer thread lifetime

One classic legacy failure mode is forgetting to join or detach a thread. C++20 introduced `std::jthread`, which automatically joins on destruction and supports cooperative stopping.

```
#include <thread>
#include <stop_token>
#include <chrono>
```

```
void worker(std::stop_token st) {
    while (!st.stop_requested()) {
        // do work
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}

int main() {
    std::jthread t(worker);
    // when t is destroyed, it requests stop and joins
}
```

Error handling style: implicit assumptions versus explicit contracts

Legacy codebases often:

- mix error codes, exceptions, and global flags inconsistently
- rely on informal conventions for cleanup and rollback

Modern C++ engineering favors:

- RAII-based rollback for partial construction and failure paths
- explicit invariants and strong exception safety where exceptions are used
- clear error channels (exceptions or explicit return objects) chosen deliberately per subsystem

The key point is not “always exceptions” or “never exceptions”. It is **consistency** and **automatic cleanup**, so failure paths do not create hidden leaks or broken state.

What “modernizing” means in practice

Modernizing a legacy C++ codebase usually does not mean rewriting everything. It means systematically changing the defaults:

- replace owning raw pointers with RAII ownership types
- make interfaces size-aware when passing buffers or arrays
- express concurrency contracts explicitly and enforce them
- reduce undefined behavior by design, and verify with tooling in real builds

A brief standard-status note

The current ISO C++ standard is informally called **C++23**, formally published as ISO/IEC 14882:2024(E). Work toward **C++26** is in progress.
For this handbook, “Modern C++” means using the stable C++11–C++23 foundations today, while tracking practical C++26 direction where it changes real-world safety engineering.

Part I

Memory Model and Object Lifetime

Chapter 1

How C++ Sees Memory

1.1 Stack vs heap

1.1.1 First: the correct C++ terms (what the standard actually models)

In everyday speech, programmers say “stack” and “heap”. In C++, the more precise model is based on **storage duration** and **lifetime**:

- **Automatic storage duration**: objects whose lifetime is tied to a scope (typically implemented using a call stack).
- **Dynamic storage duration**: objects created by `new` (managed by the allocator; typically implemented using a heap).
- **Static storage duration**: objects that exist for the entire program lifetime.
- **Thread storage duration**: objects that exist for the lifetime of a thread (`thread_local`).

Important: the C++ standard does *not* require an implementation to use a literal "stack" or a literal "heap". Those are common implementation strategies, not language guarantees. However, the concepts are still useful because they match what real systems do.

1.1.2 What people mean by "stack" in C++

When programmers say "stack memory", they almost always mean:

- memory used for **automatic objects** (local variables),
- plus the **call frames** (return address, saved registers, etc.).

Automatic objects: created and destroyed by scope

```
#include <cstddef>

int f(int x) {
    int a = x + 1;          // automatic object (usually on the stack)
    std::size_t n = 10;     // automatic object
    return a * 2;           // a and n are destroyed when f returns
}
```

Key properties of stack/automatic objects:

- **Lifetime is deterministic**: constructed when the scope is entered and destroyed when it is exited.
- **Allocation is fast**: typically a simple adjustment of a stack pointer (implementation detail).
- **Capacity is limited**: large objects and deep recursion can overflow the stack.

Stack overflow: a real engineering constraint

Large local arrays and deep recursion are common sources of stack overflow.

```
#include <array>

void risky() {
    std::array<char, 10'000'000> big; // may overflow the thread stack
    (void)big;
}
```

A safer approach is to allocate large buffers dynamically (or use a pool / arena if needed).

1.1.3 What people mean by "heap" in C++

When programmers say "heap memory", they usually mean memory used for **dynamic allocation**. In C++ terms, this is **dynamic storage duration**, commonly called the **free store**.

Dynamic objects: created by `new`, destroyed by `delete`

```
#include <iostream>

int main() {
    int* p = new int(42);   // dynamic storage duration (typically
    ↪ heap/free store)
    std::cout << *p << "\n";
    delete p;               // ends lifetime, releases storage to the
    ↪ allocator
}
```

Key properties of heap/dynamic allocation:

- **Lifetime is not tied to scope**: it lasts until you explicitly destroy the object (or use RAII types that do it for you).

- **Allocation is more expensive**: allocators must manage free lists, metadata, alignment, and sometimes synchronization.
- **Memory can fragment**: repeated allocations of varied sizes can lead to fragmentation over time.

The professional rule: never express ownership with a raw pointer

Raw pointers are good for *observation*, not *ownership*. Use RAII.

```
#include <memory>

int main() {
    auto p = std::make_unique<int>(42); // owns dynamic memory safely
    // no delete needed; lifetime ends automatically
}
```

1.1.4 Static storage and thread-local storage (not stack, not typical heap)

Not all objects are stack or heap.

Static storage duration

Objects with static storage duration exist for the entire program lifetime.

```
#include <string>

int& global_counter() {
    static int counter = 0;  // static storage duration
    return counter;
}
```

Thread storage duration (`thread_local`)

A `thread_local` object exists for the lifetime of a thread, not the whole program.

```
#include <cstdint>

thread_local std::uint64_t tls_id = 0; // one instance per thread
```

These categories matter because many lifetime bugs come from confusing them.

1.1.5 Why stack vs heap is really about *lifetime* and *ownership*

The engineering question is rarely "Where is it stored?" It is usually:

- **Who owns it?**
- **How long must it live?**
- **Is the lifetime obvious and enforced by types?**

If you need an object to survive beyond a scope, you must not return a pointer or reference to an automatic object.

Classic bug: returning the address of a stack object

```
int* bad() {
    int x = 7;      // automatic object
    return &x;      // dangling pointer after return (use-after-scope)
}
```

Correct alternatives

Return by value (often optimized efficiently), or return an owning type.

```
int good_value() {
    int x = 7;
    return x; // safe
}
```

```
#include <memory>

std::unique_ptr<int> good_owner() {
    return std::make_unique<int>(7); // safe ownership
}
```

1.1.6 The two most dangerous failure modes

Stack lifetime bug: use-after-return / use-after-scope

This happens when code keeps a pointer or reference to an automatic object after the scope ends.

```
#include <string_view>
#include <string>

std::string_view bad_view() {
    std::string s = "hello"; // automatic object
    return s;                // view dangles after return
}
```

Heap lifetime bug: use-after-free

This happens when code uses a pointer after `delete` (or after an owning RAII object releases it).

```cpp
#include <iostream>

int main() {
    int* p = new int(1);
    delete p;
    std::cout << *p << "\n"; // use-after-free (undefined behavior)
}
```

In both cases, the program has undefined behavior. The symptom can appear far away from the cause.

1.1.7 Performance reality: when stack is faster, and when heap is necessary

Why stack allocation is typically faster

Automatic objects usually require no general-purpose allocator work. This often yields:

- very low overhead for creation/destruction,
- excellent cache locality for short-lived temporaries,
- deterministic cleanup at scope exit.

When dynamic allocation is justified

Dynamic allocation is appropriate when:

- the lifetime must extend beyond the current scope,
- the size is not known at compile time or can be very large,
- objects must be shared or transferred across threads or subsystems,

- you are building container-like structures (graphs, trees, polymorphic ownership).

Avoid unnecessary heap allocations

Modern C++ offers many ways to avoid frequent allocations while keeping safety:

- prefer value semantics and return-by-value where appropriate,
- use standard containers that manage memory safely,
- reserve capacity in vectors when size is known,
- consider object pools / arenas for hot paths (with clear ownership rules).

1.1.8 Practical guidance: how to choose safely

Use these rules as a baseline:

1. Prefer **automatic storage** for small, local, short-lived objects.
2. Prefer **value semantics** and return-by-value over returning pointers/references.
3. When lifetime must outlive the scope, prefer **RAII ownership**: `std::unique_ptr`, `std::shared_ptr` (only when truly shared).
4. Avoid passing raw pointers without size; prefer **size-aware views** such as `std::span`.
5. Treat lifetime bugs as **undefined behavior**, not as “minor mistakes”.
6. Verify with tooling (sanitizers and analysis) under realistic builds, not only debug builds.

1.1.9 Key takeaway

"Stack vs heap" is a useful mental shortcut, but professional C++ design focuses on:

- explicit lifetime boundaries,
- explicit ownership representation,
- and verification of memory safety with modern tools.

Once lifetime and ownership are correct, the exact region of memory becomes a secondary detail.

1.2 Static storage duration

1.2.1 What static storage duration means in standard C++

In C++, every object has a **storage duration** category that determines how long its storage is reserved. **Static storage duration** means:

- the object's storage is reserved for the **entire program execution**, and
- the object can be accessed by name (or through references/pointers) throughout the program, subject to initialization order and visibility rules.

This category includes:

- objects declared at namespace scope (global variables),
- objects declared with the `static` specifier at namespace scope,
- `static` data members (one per class, not one per object),

- local variables declared `static` inside a function.

Static storage duration is about *where the storage is reserved in time*, not about a specific memory segment name. Most implementations place such objects in data/bss-like regions, but the standard describes the lifetime rules, not the physical layout.

1.2.2 Four common forms of static storage duration

1) Namespace-scope objects

```
#include <cstdint>

std::uint64_t g_counter = 0; // static storage duration
```

2) Namespace-scope `static` (internal linkage)

```
static int file_local_flag = 1; // static storage duration, internal
↪  linkage
```

3) Static data members

```
#include <atomic>

struct Metrics {
    static std::atomic<int> requests;
};

std::atomic<int> Metrics::requests{0}; // definition (static storage
↪  duration)
```

4) Function-local `static`

```
int& next_id() {
```

```
    static int id = 0;  // static storage duration (lives for entire
    ↪  program)
    return ++id;
}
```

This last form is extremely important in Modern C++ because it provides lazy initialization with well-defined thread-safety rules (discussed below).

1.2.3 Initialization and destruction: the important reality

Static storage duration does *not* mean "initialized before `main` in a simple way". Initialization and destruction have precise rules and several pitfalls.

Zero-initialization and constant initialization

Before any other initialization, objects with static storage duration are **zero-initialized** (when applicable). After that, an object may be:

- **constant-initialized** (computed at compile time), or
- **dynamically initialized** (requires runtime execution).

A major engineering objective is to prefer constant initialization when possible. It reduces complexity, avoids order issues, and often improves startup time.

Dynamic initialization order across translation units is not reliable

Across different translation units (different `.cpp` files), the order of dynamic initialization of non-local static objects is **unspecified**. This is a common source of production failures known as the *static initialization order fiasco*.

Example: classic initialization order failure

Assume the following are in different translation units:

```
// a.cpp
#include <string>
extern std::string config_path;
std::string log_prefix = "LOG:" + config_path; // may run before
↪  config_path is initialized
```

```
// b.cpp
#include <string>
std::string config_path = "/etc/app.conf"; // dynamic initialization
```

Even though this code compiles, `log_prefix` may be initialized before `config_path`, leading to undefined behavior or incorrect results depending on how the initialization is evaluated.

1.2.4 Professional solution: the “construct on first use” idiom

To avoid cross-translation-unit initialization order problems, use a function-local static:

```
#include <string>

const std::string& config_path() {
    static const std::string path = "/etc/app.conf";
    return path;
}

const std::string& log_prefix() {
    static const std::string prefix = "LOG:" + config_path();
    return prefix;
}
```

Why this works

- Initialization occurs when the function is first called.
- Since C++11, initialization of function-local statics is **thread-safe**: if multiple threads enter concurrently, only one performs initialization, and the others wait until it completes.

This does not remove the need for careful design (for example, avoid expensive or failure-prone initialization in hot paths), but it removes the global-order hazard.

1.2.5 Thread-safety and function-local static initialization (since C++11)

A critical Modern C++ improvement is that initialization of a block-scope variable with static storage duration is performed exactly once, even under concurrency.

Example: safe lazy singleton-like state

```
#include <mutex>

std::mutex& global_mutex() {
    static std::mutex m; // initialized once, thread-safe since C++11
    return m;
}
```

What is and is not guaranteed

- Guaranteed: the initialization is performed once, without data races on the initialization itself.
- Not guaranteed: that all uses of the object are automatically thread-safe. The object may still require synchronization during normal access.

1.2.6 Destruction: static objects can die after `main`

Objects with static storage duration are destroyed during program termination. This includes:

- leaving `main`,
- calling `std::exit`,
- normal termination paths that trigger static destruction.

The destruction order of non-local static objects across translation units is also problematic, and it can cause the *static deinitialization order problem*: a destructor may access another static object that has already been destroyed.

Example: deinitialization hazard

```
#include <iostream>

struct Logger {
    ~Logger() {
        // This is dangerous if std::cout is not usable at this stage on
        ↪  some platforms,
        // or if it depends on other destroyed statics in the program.
        std::cout << "shutdown\n";
    }
};

Logger g_logger; // static storage duration
```

Professional guidance

- Avoid complex work in destructors of non-local statics.

- Prefer explicit start/stop lifetimes for subsystems.
- If you need global objects, consider keeping them trivial or constant-initialized.

1.2.7 Static storage duration and the One Definition Rule (ODR)

Because static storage objects often have global visibility, they interact strongly with the One Definition Rule:

- A variable with external linkage typically must be defined exactly once in the program.
- Violating this leads to linker errors or, worse, ODR violations with unpredictable behavior.

Modern tool: `inline` variables (since C++17)

C++17 introduced `inline` variables to allow a single definition across translation units in a header when needed.

```
// header.hpp
#pragma once
#include <atomic>

inline std::atomic<int> g_requests{0}; // one program-wide object,
↪  header-friendly
```

1.2.8 Common safety risks with static storage duration

1) Hidden shared state

Non-local statics introduce global shared state that can:

- make testing harder,

- break modular reasoning,
- create unexpected coupling,
- cause data races if accessed concurrently.

2) Initialization order problems

Dynamic initialization across translation units can break correctness.

3) Destruction order problems

Shutdown-time access to destroyed objects can break correctness.

4) Concurrency hazards

Static objects are often shared across threads. That is not automatically safe.

1.2.9 Best practices for professional codebases

1. Prefer **local variables** and explicit dependency injection over globals.
2. Prefer **constant initialization** for static objects where possible.
3. Avoid non-local statics with complex initialization across translation units.
4. Use **function-local statics** for lazy initialization when global access is needed.
5. Avoid doing important work in destructors of non-local statics; use explicit shutdown.
6. Treat every non-const static object as shared state: define synchronization and invariants.
7. Use `inline` variables (C++17) carefully for header-level state when truly needed.

1.2.10 Key takeaway

Static storage duration is powerful, but it creates program-wide state and lifetime effects. In Modern C++, the safest approach is usually:

- minimize global state,
- prefer constant initialization,
- use function-local statics when global access is required,
- and keep initialization and shutdown behavior explicit and testable.

1.3 Global and thread-local storage

1.3.1 Why this section matters

In real C++ systems, a large percentage of the hardest bugs come from **long-lived state**:

- global objects that are initialized too early or destroyed too late,
- shared global state accessed concurrently without a clear contract,
- thread-local objects that behave differently across threads and at thread exit.

To engineer safe systems, you must understand what C++ actually guarantees about **global** and **thread-local** storage, and what remains an implementation detail.

1.3.2 Global storage in C++: what it really means

In everyday usage, “global” usually means an object declared at namespace scope. In standard C++ terms, most global objects have **static storage duration**: their storage exists for the entire program execution.

Common forms of global state

```
#include <cstdint>

// 1) Namespace-scope object (external linkage by default)
std::uint64_t g_requests = 0;

// 2) Namespace-scope object with internal linkage (file-local)
static int g_debug_flag = 0;

// 3) Static data member: one object shared by all instances
struct Metrics {
    static std::uint64_t total;
};
std::uint64_t Metrics::total = 0;
```

What is guaranteed, and what is not

- Guaranteed: the storage exists for the entire program execution.
- Guaranteed: objects are initialized and later destroyed as part of program start/termination, but the **order** is subtle.
- Not guaranteed: any particular memory segment name (“data”, “bss”) or physical layout.
- Not guaranteed: that accessing global objects is thread-safe (it is not, unless you enforce it).

1.3.3 The two global hazards: initialization order and shared mutable state

Hazard 1: initialization order across translation units

If two non-local globals in different `.cpp` files depend on each other and require **dynamic initialization**, the order is not reliably controllable. This is the classic *static initialization order fiasco*.

```
// a.cpp
#include <string>
extern std::string g_config;
std::string g_banner = "CFG=" + g_config; // depends on g_config

// b.cpp
#include <string>
std::string g_config = "prod";            // dynamic initialization
```

This can fail because `g_banner` may initialize before `g_config`.

Professional solution: “construct on first use”

Use function-local statics to make initialization occur on first call:

```
#include <string>

const std::string& config() {
    static const std::string cfg = "prod";
    return cfg;
}

const std::string& banner() {
    static const std::string b = "CFG=" + config();
    return b;
}
```

Since C++11, initialization of function-local statics is thread-safe: it is performed once, even under concurrent entry.

Hazard 2: global mutable state is shared state

A non-const global is effectively a shared variable across the entire program. If multiple threads can access it, you must define:

- which operations are allowed concurrently,
- what synchronization protects it,
- what invariants must hold.

```
#include <atomic>

// If all you need is an atomic counter:
inline std::atomic<std::uint64_t> g_requests{0}; // C++17 inline variable
```

If the global represents a multi-field invariant, atomics are not enough; use a mutex (or redesign the ownership model).

1.3.4 Prefer constant initialization for globals

A major goal is to avoid dynamic initialization for non-local objects.

`constinit` (C++20): enforce constant initialization

If a variable can be constant-initialized, `constinit` forces that property. If it cannot be constant-initialized, the program is ill-formed.

```cpp
#include <cstdint>

constinit std::uint32_t g_mask = 0xFFu; // guaranteed constant
↪  initialization
```

This is a powerful tool for preventing initialization-order problems, because constant initialization does not depend on runtime execution order.

1.3.5 Thread-local storage: a different storage duration

C++ provides `thread_local` objects with **thread storage duration**:

- each thread gets its own instance of the variable,
- the variable exists for the lifetime of the thread,
- initialization and destruction occur per thread.

Basic examples

```cpp
#include <cstdint>

// One instance per thread:
thread_local std::uint64_t tls_id = 0;

// Common use: per-thread counters or caches:
thread_local std::uint64_t tls_requests = 0;
```

Thread-local initialization: the safe mental model

The safest way to reason about `thread_local` is:

- each thread constructs its own instance before that thread uses it,

- each thread destroys its own instance when that thread exits.

This avoids the most common misunderstanding: a `thread_local` object is not "a global with thread safety". It is **per-thread state**.

Thread-local is not automatically race-free

Thread-local variables eliminate sharing of that variable itself. But the code inside initialization and normal use can still touch shared state.

```
#include <string>
#include <mutex>

std::mutex g_m;
std::string g_shared;

thread_local std::string tls_copy = []{
    // Still accesses shared data: needs synchronization
    std::lock_guard<std::mutex> lock(g_m);
    return g_shared;
}();
```

The per-thread instance is safe from concurrent modification by other threads, but the initialization path must still be correct.

1.3.6 When thread-local is a good design choice

Thread-local storage is useful when you need **per-thread state** without passing it through every API:

- per-thread caches or scratch buffers to reduce allocations,

- per-thread statistics counters,
- per-thread random number engines,
- thread-affine context objects (logging context, request id, tracing span).

A thread-local design is often preferable to a global shared design when sharing is not required and would only introduce locking.

1.3.7 Thread-local pitfalls and engineering constraints

Pitfall 1: memory overhead scales with thread count

A `thread_local` object is replicated per thread. If it is large, or if you create many threads, memory usage can grow dramatically.

```
#include <array>

// One 1 MiB buffer per thread:
thread_local std::array<char, 1 << 20> tls_buffer;
```

Use per-thread buffers intentionally, and prefer smaller buffers or dynamic strategies when thread counts can be high.

Pitfall 2: destructor timing at thread exit

A `thread_local` object with a non-trivial destructor is destroyed when the thread exits. This can be surprising if:

- the destructor performs logging or touches other global objects,
- the thread is exiting during program shutdown,

- ordering interacts with other shutdown activities.

Professional guidance:

- Keep thread-local destructors simple.
- Avoid relying on complex global subsystems during thread-local destruction.
- Prefer explicit lifecycle management for critical subsystems.

Pitfall 3: "thread-local singleton" is not a program singleton

A function returning a reference to a `thread_local` static returns a *different* object in each thread:

```
#include <string>

std::string& context() {
    thread_local std::string ctx;
    return ctx; // different per thread
}
```

This can be exactly what you want (thread context), but it is not a global shared instance.

1.3.8 Global vs thread-local: choosing responsibly

Decision rules

Use these rules as a practical baseline:

1. Prefer **local state** and dependency injection over globals.
2. If the state must be program-wide, prefer **constant initialization** and minimize mutability.

3. If the state must be shared and mutable, define synchronization and invariants explicitly.

4. If the state is naturally per-thread, prefer **thread_local** to avoid unnecessary sharing.

5. Keep global and thread-local destructors simple; avoid complex shutdown dependencies.

A simple comparison

- **Global (static storage duration):** one object for the whole program; easy to couple modules; concurrency must be engineered explicitly.

- **Thread-local (thread storage duration):** one object per thread; reduces locking for per-thread state; memory usage and destruction timing must be managed deliberately.

1.3.9 Key takeaway

Global and thread-local storage are powerful because they extend lifetime and simplify access. They are also dangerous because they create hidden dependencies and long-lived state. Professional C++ safety relies on:

- minimizing global state,

- preferring constant initialization and explicit lifecycles,

- treating thread-local as per-thread design (not as automatic thread safety),

- and making concurrency contracts visible and enforceable.

1.4 Alignment and padding

1.4.1 Why alignment and padding matter for safety and performance

Alignment and padding are not “micro-optimizations”. They affect:

- **Correctness**: misaligned access can be undefined behavior for certain types and platforms.
- **Portability**: layout assumptions can break between compilers, ABIs, and architectures.
- **Performance**: alignment impacts cache efficiency, vectorization, and atomic operations.
- **Security**: incorrect layout assumptions can cause out-of-bounds reads/writes and data disclosure.

Modern C++ engineers must understand how object layout works and how to express layout requirements explicitly and safely.

1.4.2 The standard model: object representation and alignment requirement

Every object type in C++ has an **alignment requirement**: an object of that type must be placed at an address that is a multiple of its alignment. The alignment of a type `T` is available as `alignof(T)`.

Observing alignment in code

```
#include <cstddef>
#include <iostream>
```

```cpp
struct S {
    int    a;
    double b;
};

int main() {
    std::cout << "alignof(int)    = " << alignof(int) << "\n";
    std::cout << "alignof(double) = " << alignof(double) << "\n";
    std::cout << "alignof(S)      = " << alignof(S) << "\n";
    std::cout << "sizeof(S)       = " << sizeof(S) << "\n";
}
```

1.4.3 What padding is and why it exists

Padding is unused space inserted by the implementation to satisfy alignment. Padding can appear:

- **between members** (internal padding),
- **at the end of a struct** (tail padding) so that arrays of the struct remain aligned.

A typical padding pattern

```cpp
struct P {
    char   c;   // offset 0
    // padding here to align the next member
    int    i;   // typically aligned to 4
    char   d;   // may cause tail padding
};
```

The exact size and member offsets are implementation-defined and depend on the ABI and target, but the *reason* is consistent: alignment must be preserved.

1.4.4 Member order: a practical rule that reduces padding

A widely used rule for memory efficiency is to order members by decreasing alignment. This often reduces internal padding.

Example: padding difference by member order

```
#include <cstddef>

struct Bad {
    char   c;
    double d;
    int    i;
};

struct Good {
    double d;
    int    i;
    char   c;
};

static_assert(sizeof(Good) <= sizeof(Bad), "Reordering often reduces
↪  padding");
```

This is not a universal guarantee, but it is a reliable first heuristic in practice.

1.4.5 Alignment control: `alignas` and over-aligned types

C++ allows you to request a stricter alignment using `alignas`. This is useful for:

- SIMD types and vectorization,
- cache-line alignment to reduce false sharing,

- hardware interfaces and memory-mapped structures (with great care).

Example: requesting stronger alignment

```cpp
#include <cstddef>

struct alignas(32) Vec256 {
    double x[4]; // example payload
};

static_assert(alignof(Vec256) == 32);
```

Over-aligned allocation (modern C++)

If a type has alignment greater than the default new-alignment, allocations must satisfy it. Modern C++ provides aligned allocation support so that `new T` produces correctly aligned storage for `T` (including over-aligned `T`) on conforming implementations.

1.4.6 Misalignment: what can go wrong

The core problem

If you treat an address as if it contains an object of type `T` when it does not meet `alignof(T)`, you may produce undefined behavior.
This often happens when:

- casting arbitrary bytes to a pointer to a stronger-aligned type,
- using packed structs incorrectly,
- parsing binary formats by pointer reinterpretation.

Unsafe example: reinterpreting unaligned bytes

```
#include <cstdint>

std::uint64_t read_u64_unsafe(const std::uint8_t* p) {
    // If p is not aligned to alignof(std::uint64_t), this may be
    ↪  undefined behavior.
    return *reinterpret_cast<const std::uint64_t*>(p);
}
```

Safe example: copy bytes into a properly aligned object

Use `std::memcpy` (or `std::bit_cast` where appropriate) to avoid alignment and aliasing issues.

```
#include <cstdint>
#include <cstring>

std::uint64_t read_u64_safe(const std::uint8_t* p) {
    std::uint64_t v = 0;
    std::memcpy(&v, p, sizeof(v));
    return v;
}
```

This is the standard approach for parsing binary data portably and safely.

1.4.7 Alignment and strict aliasing: do not mix them up

Alignment and aliasing are separate constraints:

- **Alignment**: address must satisfy `alignof(T)`.
- **Aliasing**: accessing an object through an incompatible type can be undefined behavior.

The safe pattern for raw bytes is:

- store bytes in `std::byte` or `unsigned char`,
- move bytes into typed objects using `std::memcpy` when you need typed access.

1.4.8 Atomic operations and alignment requirements

Atomic types may have stronger alignment requirements on some platforms. Also, an atomic operation that is not naturally aligned can be slow, can trap, or can be unsupported.

Example: do not place atomics in packed layouts

```
#include <atomic>

#pragma pack(push, 1)
struct Packed {
    char c;
    std::atomic<int> a; // dangerous: packing may break alignment
    ↪  requirements
};
#pragma pack(pop)
```

Packing is not a standard C++ feature; it is a compiler extension. When it is used, it can create misaligned members. Misaligned atomics are especially dangerous.
Professional rule: avoid packing objects that contain types with non-trivial alignment requirements. Prefer explicit serialization/deserialization instead of relying on packed binary layouts.

1.4.9 False sharing: padding as a concurrency performance tool

In multithreaded code, two independent variables that reside on the same cache line can cause performance collapse due to **false sharing**: multiple cores repeatedly invalidate the same cache line even though they are updating different variables.
A common mitigation is to separate hot, independently modified variables by alignment or padding.

Example: aligning counters to reduce false sharing

```
#include <atomic>
#include <new>

struct alignas(std::hardware_destructive_interference_size) Counter {
    std::atomic<std::uint64_t> value{0};
};
```

`std::hardware_destructive_interference_size` (C++17) is a library constant intended to help avoid destructive interference (false sharing) when placing objects in memory.

1.4.10 Practical checklist for safe layout work

1. Use `alignof(T)` to understand type requirements; never guess.
2. Use `alignas(N)` only when you can justify the requirement (SIMD, cache, hardware interface).
3. Do not parse binary formats by casting bytes to typed pointers; use `std::memcpy`.
4. Avoid packed layouts for normal program data. If you need a binary format, serialize explicitly.

5. Be careful with atomics: ensure proper alignment and avoid placing them in packed structs.

6. For performance under concurrency, consider false sharing and use alignment/padding intentionally.

7. Never assume member offsets or struct sizes are portable across compilers and platforms unless the type is explicitly specified to be standard-layout and you still control the ABI assumptions.

1.4.11 Key takeaway

Alignment is a correctness constraint, padding is the mechanism used to satisfy it. A professional Modern C++ codebase treats layout assumptions as **explicit engineering decisions**:

- express alignment requirements with `alignof` and `alignas`,
- avoid undefined behavior by using `std::memcpy` for byte-to-type conversions,
- and consider cache effects and false sharing when writing concurrent systems.

1.5 Cache awareness (high-level overview)

1.5.1 Why cache awareness matters in C++ systems

Modern processors are designed with multiple levels of cache between the CPU and main memory. Caches dramatically improve performance when data locality is good, but they can also become a bottleneck when access patterns are poor. In high-performance systems —

often written in C++ — understanding how memory access interacts with caches is a key engineering concern.
This section provides a high-level overview of cache behavior and practical guidance that supports safe, efficient C++ design. The focus is on concepts that affect correctness, performance, and predictability without assuming deep hardware expertise.

1.5.2 What a cache is, in simple terms

A cache is a small, fast memory located close to the CPU that stores copies of data from main memory. When the processor needs a value, it first checks the cache:

- If the value is in the cache (a **cache hit**), access is fast.
- If it is not (a **cache miss**), the value must be fetched from main memory, which is slower.

Caches operate on blocks of memory called **cache lines**. A cache line typically ranges from 32 to 128 bytes on modern CPUs. When one byte in a cache line is accessed, the entire line is fetched into the cache.

1.5.3 Locality: the key to cache efficiency

There are two primary forms of locality:

- **Spatial locality**: accessing memory near the recently accessed location.
- **Temporal locality**: accessing the same memory location multiple times within a short period.

C++ data structures and algorithms that exploit these localities tend to perform much better in practice because they reduce the number of expensive main memory accesses.

1.5.4 How data layout influences cache behavior

The way objects are laid out in memory influences how often adjacent data is brought into cache. This interacts with alignment and padding discussed earlier.

Array versus pointer-linked structures

Arrays and contiguous containers such as `std::vector` have strong spatial locality:

```
#include <vector>

int sum(const std::vector<int>& v) {
    int s = 0;
    for (int x : v) {
        s += x; // sequential access benefits from spatial locality
    }
    return s;
}
```

In contrast, pointer-linked structures such as linked lists may scatter nodes in memory:

```
struct Node {
    int value;
    Node* next;
};

int sum(Node* head) {
    int s = 0;
    for (Node* p = head; p; p = p->next) {
        s += p->value; // may cause many cache misses
    }
    return s;
}
```

Unless the list nodes happen to be near each other in memory, spatial locality will be weaker.

1.5.5 Cache lines, alignment, and false sharing

Cache line concept

Operations on data within the same cache line often complete without extra memory traffic. This makes contiguous and properly aligned data easier for the hardware to handle efficiently.

False sharing in concurrent code

In multithreaded systems, separate threads updating distinct variables that reside on the same cache line can cause performance degradation. This phenomenon is called **false sharing**.

```
#include <atomic>

struct Counters {
    std::atomic<long> a;
    std::atomic<long> b;
};

Counters counters;
```

If `a` and `b` lie on the same cache line and two threads update them independently, cache coherence traffic between cores will increase, slowing both updates. Aligning or padding such hot variables can reduce contention.

1.5.6 Prefetching and access patterns

Modern CPUs use prefetching heuristics to predict and load data into cache before it is requested. Sequential access patterns are easy for prefetchers to detect and benefit from preloading.

Strided or random access patterns are harder to anticipate, which can reduce cache effectiveness and increase memory latency.

1.5.7 Working set size and cache levels

Large data sets that exceed cache capacity lead to frequent evictions and misses. Programmers should be aware of the working set size — the subset of data actively used during a phase of execution — and structure loops and algorithms to operate on smaller, cache-friendly chunks when possible.

1.5.8 Data structure design and cache-friendliness

When designing data structures, consider:

- Using contiguous storage (e.g., `std::vector`) instead of node-based structures when possible and appropriate.
- Storing frequently accessed members together to improve spatial locality.
- Avoiding unnecessary pointer indirection in hot loops.
- Carefully balancing size and alignment so that structures fit within cache lines when access patterns benefit from it.

1.5.9 Cache awareness in concurrent systems

Concurrent algorithms can interact with the memory hierarchy more intensely:

- Atomics and locks can serialize cache line ownership changes.
- Contention on shared data structures can increase cache misses.

- Thread-local data can reduce contention and improve temporal locality per thread.

Understanding how threads interact with shared data and the cache system leads to more scalable concurrent designs.

1.5.10 Measurement and validation

Cache effects are not always intuitive. Performance testing under realistic workloads is essential:

- Use performance counters or profilers that report cache miss rates.
- Test with data sizes that reflect real use cases.
- Compare different data layouts and access patterns empirically.

Empirical observation is the only reliable way to quantify the real-world impact of cache behavior.

1.5.11 Practical summary for C++ engineers

- Favor contiguous storage and predictable access patterns to improve spatial locality.
- Structure algorithms to reuse data recently accessed to benefit from temporal locality.
- When working with concurrent code, minimize false sharing by isolating hot variables.
- Measure and profile cache behavior on target hardware, rather than rely on assumptions.

1.5.12 Key takeaway

Cache awareness is not about memorizing hardware specifics; it is about understanding how memory access patterns interact with the memory hierarchy. Modern C++ design that considers data layout, locality, and access patterns can achieve significant improvements in performance and predictability without sacrificing safety or correctness.

Chapter 2

Object Lifetime Rules

2.1 Construction and destruction semantics

2.1.1 Construction vs lifetime: what C++ actually guarantees

In C++, an object has **storage** (a region of memory) and a **lifetime** (the period during which the program is allowed to treat that storage as a valid object of a given type).

- **Construction** is the process that initializes an object (runs constructors for the object, its bases, and its members).
- **Destruction** is the process that ends an object (runs the destructor for the object, then destroys members and bases).
- **Lifetime rules** define precisely when it is valid to read/write the object as that type.

A common engineering mistake is to treat "has storage" as "is alive". In standard C++, storage can exist while the object is not alive (for example, before construction, after destruction, or during manual lifetime management).

2.1.2 When lifetime begins and ends (the practical model)

Automatic and static objects

For most everyday objects:

- Lifetime begins when initialization completes for the object.
- Lifetime ends when the destructor begins for the object (from the outside observer perspective), with subobjects still being destroyed in a well-defined order.

Dynamic objects

For `new T(...)`:

- storage is obtained first,
- then `T` is constructed in that storage,
- lifetime ends when `delete` runs the destructor, then releases storage.

Manual lifetime control (advanced)

With placement `new` and explicit destruction (`std::destroy_at`):

- you can begin a lifetime in pre-existing storage,
- you can end a lifetime without releasing the storage,
- but you must not access the object as type `T` outside its lifetime.

2.1.3 Order of construction and destruction (bases, members, and the most important rule)

C++ enforces a strict, predictable order that is essential for correctness:

Construction order

1. Base classes are constructed first (in the order they appear in the base-specifier list).
2. Data members are constructed next (in the order they are declared in the class).
3. The constructor body runs last.

Destruction order

Destruction happens in the reverse order:

1. The destructor body runs first.
2. Data members are destroyed next (reverse declaration order).
3. Base classes are destroyed last (reverse base order).

Professional implication

Member initialization order follows declaration order, not initializer-list order. This is a frequent source of subtle bugs.

```
#include <string>
#include <string_view>

struct Bad {
    std::string storage;
    std::string_view view;

    Bad(const char* s)
      : view(storage)        // looks like it uses storage...
      , storage(s)           // ...but storage is constructed first
       ↪  anyway
```

```
    {}
};
```

The code above is still wrong because `view(storage)` sees `storage` before it holds the final content (and may observe an empty string). The correct design is to initialize in a way that matches the actual dependency:

```
#include <string>
#include <string_view>

struct Good {
    std::string storage;
    std::string_view view;

    Good(const char* s)
      : storage(s)
      , view(storage)
    {}
};
```

2.1.4 Virtual destructors: the correctness boundary for polymorphic deletion

If you delete an object through a base-class pointer, the base class must have a **virtual destructor** unless deletion through the base is forbidden by design.

Bug: non-virtual destructor in a polymorphic base

```
struct Base {
    ~Base() = default;        // not virtual
};
```

```
struct Derived : Base {
    ~Derived() { /* releases resources */ }
};

void f() {
    Base* p = new Derived;
    delete p;                // undefined behavior: Derived::~Derived may
     ↪  not run
}
```

Fix: virtual destructor

```
struct Base {
    virtual ~Base() = default;
};

struct Derived : Base {
    ~Derived() override { /* releases resources */ }
};
```

2.1.5 Destructors and exceptions: why they are almost always `noexcept`

In Modern C++, destructors are implicitly **non-throwing** by default: they are `noexcept` unless you explicitly declare otherwise.

Throwing an exception out of a destructor is dangerous because destructors commonly run during stack unwinding (when another exception is already active). In that situation, a second exception escaping a destructor will typically cause program termination.

Professional rule

- Destructors should not emit failures by throwing.
- If cleanup can fail (for example I/O flush, network close), provide an explicit **close()/flush()/commit()** operation that reports errors, and keep the destructor best-effort (or terminate intentionally if failure is truly unrecoverable).

Pattern: explicit close, non-throwing destructor

```
#include <cstdio>
#include <system_error>

class File {
    std::FILE* f = nullptr;

public:
    explicit File(const char* path) {
        f = std::fopen(path, "wb");
        if (!f) throw std::system_error(errno, std::generic_category());
    }

    // Report close errors explicitly.
    void close() {
        if (!f) return;
        if (std::fclose(f) != 0) {
            f = nullptr;
            throw std::system_error(errno, std::generic_category());
        }
        f = nullptr;
    }
```

```
    ~File() noexcept {
        // Best-effort cleanup; do not throw.
        if (f) std::fclose(f);
    }
};
```

2.1.6 Copy, move, and destruction: what the special member functions mean

Construction and destruction semantics determine how objects behave in containers, returns, error paths, and concurrency.

Copy construction

Creates a new object from an existing one (two independent lifetimes).

Move construction

Transfers state from a source object into a new object. After the move:

- the destination is fully valid,
- the source remains valid but is typically in an unspecified (but safe-to-destroy) state.

Destruction

Always runs exactly once per lifetime for normal objects, and must be correct even when:

- the object is partially constructed (a constructor throws),
- the object is moved-from,
- destruction occurs during stack unwinding.

Example: RAII remains correct under exceptions

```
#include <memory>
#include <vector>

struct Resource {
    Resource();  // might throw
    ~Resource() noexcept;
};

void g() {
    std::vector<std::unique_ptr<Resource>> v;
    v.push_back(std::make_unique<Resource>()); // if Resource() throws,
    ↪  no leak occurs
} // v destroys owned resources safely
```

2.1.7 Temporaries and lifetime extension (a common source of dangling references)

Temporary objects usually live until the end of the **full-expression**. In a few specific cases, a temporary's lifetime is extended, most notably when it is bound directly to a reference in certain initialization contexts.

Bug pattern: returning a view to a temporary

```
#include <string>
#include <string_view>

std::string_view bad() {
    return std::string("hello"); // temporary destroyed at end of return
    ↪  statement
}
```

Correct alternatives

```
#include <string>

std::string good() {
    return std::string("hello"); // return by value is safe
}
```

```
#include <string_view>

std::string_view ok_literal() {
    return "hello"; // string literal has static storage duration
}
```

2.1.8 Advanced: explicit lifetime control with placement `new` and `std::destroy_at`

Sometimes you must manage lifetime manually (custom allocators, arenas, variant-like storage).

Correct lifetime start/end in raw storage

```
#include <new>
#include <memory>
#include <cstddef>
#include <iostream>

struct X {
    int v;
    explicit X(int n) : v(n) {}
    ~X() noexcept { std::cout << "X destroyed\n"; }
};
```

```
int main() {
    alignas(X) std::byte buffer[sizeof(X)];

    X* p = ::new (static_cast<void*>(buffer)) X(42); // begin lifetime
    std::cout << p->v << "\n";

    std::destroy_at(p);                                // end lifetime
    ↪ (does not free storage)
}
```

Professional constraints for manual lifetime management

- Only treat storage as `T` while `T`'s lifetime is active.
- End the lifetime exactly once (avoid double destruction).
- Reusing storage for a different object type requires correct alignment and careful rules.
- Prefer standard library facilities (`std::destroy_at`, allocator traits, containers) rather than ad-hoc destructor calls.

2.1.9 Practical engineering rules for construction and destruction

1. Express ownership using RAII types; destructors must be safe under all exits.
2. Never assume member initialization order follows the initializer list; it follows declaration order.
3. Use virtual destructors for polymorphic bases that can be deleted via base pointers.
4. Keep destructors `noexcept` and non-failing; report failure via explicit operations.

5. Treat temporaries and views carefully; avoid returning references/views to short-lived objects.

6. Use placement `new` and `std::destroy_at` only when you truly need manual lifetime control.

2.1.10 Key takeaway

C++ construction and destruction are not only “automatic cleanup” mechanics. They define the boundaries of object lifetime, ownership, and correctness. Professional C++ safety requires designing APIs and types so that lifetime is explicit, misuse is difficult, and cleanup remains correct under exceptions, moves, and concurrency.

2.2 Temporary objects

2.2.1 What a temporary object is

A **temporary object** is an unnamed object created by the evaluation of an expression. In everyday C++ code, temporaries appear when you write things like:

- a prvalue such as `T{}`, `T(...)` or `std::string("hi")`
- the result of most arithmetic and user-defined operator expressions
- the value returned by a function that returns by value
- implicit conversions that create a new object (e.g., converting to a different type)

Since C++17, the language describes many expressions as producing **prvalues** that are not necessarily materialized into a distinct temporary object unless a real object is needed (a

temporary materialization conversion). The practical safety rule remains: **when a temporary object is created, its lifetime is usually very short.**

2.2.2 The default rule: temporaries live until the end of the full-expression

By default, a temporary object is destroyed at the end of the **full-expression** that created it. A **full-expression** is typically:

- the entire statement ending with `;`,
- the controlling expression of `if`, `while`, `for`,
- or the full expression inside a `return` statement.

Example: temporary ends at the semicolon

```
#include <string>

int main() {
    std::string("tmp").size(); // the temporary std::string is destroyed
    ↪  at ';'
}
```

This default destruction timing is the root cause of many dangling references and dangling views.

2.2.3 Lifetime extension: when a temporary lives longer

C++ has a specific and very important exception: in some cases, binding a temporary to a reference **extends the lifetime** of the temporary.

The common safe case: bind to a reference variable

When you initialize a reference variable directly from a temporary, the temporary's lifetime is extended to match the lifetime of the reference.

```
#include <string>

int main() {
    const std::string& r = std::string("hello"); // lifetime extended to
    ↪  end of scope
    // r is valid here
}
```

This is a real language rule, and it is widely used. However, it is also widely misunderstood. The extension applies only in specific forms of initialization, and there are important exceptions.

Lifetime extension applies to the temporary, not to unrelated derived data

A common mistake is to store a *view* into a temporary (like `std::string_view`) and assume the view keeps the temporary alive. It does not.

```
#include <string>
#include <string_view>

int main() {
    std::string_view sv = std::string("hello"); // dangling immediately
    ↪  after ';'
    (void)sv;
}
```

The `std::string` temporary is destroyed at the semicolon. The view does not own memory, so `sv` becomes dangling.

2.2.4 The most important exception: binding to a reference parameter does *not* extend beyond the call

Passing a temporary to a function that takes a reference parameter extends the temporary only until the end of the full-expression containing the call (typically, the end of the statement), not to the lifetime of any reference that the function returns.

Danger: returning a reference to a temporary passed as an argument

```
#include <string>

const std::string& bad_ref(const std::string& s) {
    return s; // returns a reference to the argument
}

int main() {
    const std::string& r = bad_ref(std::string("tmp"));
    // The temporary "tmp" is destroyed at end of this statement -> r
    ↪  dangles.
    (void)r;
}
```

Professional rule:

> Never return (or store) a reference/pointer to an object you do not own unless its lifetime is clearly guaranteed to outlive the reference.

2.2.5 Temporaries and subobjects: do not assume “parts” live longer

When a temporary is created, references to its subobjects do not automatically remain valid unless the language rule specifically extends the lifetime of the whole temporary in that context.

Example: view into a temporary is a classic dangling bug

```
#include <string>
#include <string_view>

std::string_view bad() {
    return std::string("hello"); // temporary destroyed at end of return
    ↪ statement
}
```

Correct alternatives:

```
#include <string>

std::string good() {
    return std::string("hello"); // return by value is safe
}
```

```
#include <string_view>

std::string_view ok_literal() {
    return "hello"; // string literal has static storage duration
}
```

2.2.6 Temporaries in expressions: why chaining can hide lifetimes

Operator overloading and fluent APIs can easily hide the creation of temporaries, especially when returning proxy objects or views.

Example: chaining creates temporaries you may not see

```
#include <string>
#include <string_view>
```

```
std::string_view dangerous(std::string_view a, std::string_view b) {
    std::string_view v = std::string(a) + std::string(b); // creates
    ↪ temporary std::string
    return v; // returns a view into a destroyed temporary
}
```

Professional guidance:

- If a function returns a view (`string_view`, `span`, iterator pair), it must return a view into storage that outlives the view.
- If the function needs to create new storage, return an owning type (e.g., `std::string`).

2.2.7 Range-based `for` and temporaries: a C++23 correctness improvement

Range-based `for` loops deserve special attention because the range expression can be a temporary.

The safe mental model

The loop is specified roughly as:

- evaluate the range initializer once,
- bind it to an internal variable,
- then iterate using `begin` and `end`.

If the range initializer is a temporary, its lifetime is extended to cover the loop. However, **before C++23**, not all temporaries inside the range initializer were guaranteed to be extended, which could lead to subtle dangling behavior when chaining expressions.

Example pattern that was problematic before C++23

```
// If foo() returns a temporary object by value and items() returns a
↪  reference-like view,
// this pattern could be dangerous before C++23 depending on the exact
↪  types involved.
for (auto& x : foo().items()) {
    (void)x;
}
```

Since C++23, the lifetime rules for range-based `for` were strengthened so that more temporary cases in the range initializer are safe until the end of the loop. Professional guidance remains:

- Prefer writing the range expression in a way that clearly keeps the owning object alive.
- When in doubt, name the owner explicitly.

Robust form: name the owning object

```
auto owner = foo();                // owns storage, lifetime is clear
for (auto& x : owner.items()) {    // items() can safely refer into owner
    (void)x;
}
```

2.2.8 Temporary materialization (C++17 and later): what you must remember

Since C++17, prvalues do not always create a distinct temporary object unless needed. This change improved performance and simplified some cases (for example, guaranteed copy elision in certain returns), but it does *not* change the safety rule:

> If you bind a reference/view/pointer to something that does not outlive it, you get a dangling reference/view/pointer.

So, even with C++17+ semantics, avoid returning views to ephemeral objects, and prefer returning values (or owning types) when storage must be created.

2.2.9 Safe engineering rules for temporary objects

Use these rules consistently in professional C++ code:

1. Assume a temporary dies at the end of the statement unless you *know* a lifetime extension rule applies.
2. Lifetime extension is reliable for direct initialization of a reference variable from a temporary, but **not** for reference parameters beyond the end of the call.
3. Never return `string_view`, `span`, iterators, pointers, or references to a temporary. Return an owning type (e.g., `std::string`, container, smart pointer) instead.
4. Be suspicious of expression chaining that hides temporaries, especially when views are involved.
5. In range-based `for`, if the range depends on a temporary owner, name the owner explicitly.

6. Prefer designs that make lifetime obvious in types and APIs, and validate with sanitizers and analysis tools.

2.2.10 Key takeaway

Temporary objects are a normal and essential part of C++ expression semantics. They are also one of the most common sources of dangling references and undefined behavior. Professional C++ safety means:

- understanding when temporaries are destroyed,
- using lifetime extension rules intentionally (not accidentally),
- and designing APIs so that returned views always refer to storage with a clear, longer lifetime.

2.3 Lifetime extension

2.3.1 What “lifetime extension” means

Most temporary objects in C++ are destroyed at the end of the **full-expression** (typically the end of the statement). **Lifetime extension** is a special rule that, in specific situations, keeps a temporary alive longer than usual.
This rule is extremely important for safety because many dangling-reference bugs come from *assuming* an extension happens when it does not.

2.3.2 The default rule (no extension)

If you create a temporary in an expression, assume it is destroyed at the end of the full-expression unless you are in one of the explicitly allowed extension cases.

Example: temporary destroyed at the semicolon

```
#include <string>
#include <string_view>

int main() {
    std::string_view v = std::string("hello"); // dangling after this
    ↪  statement
    (void)v;
}
```

The temporary `std::string` is destroyed at the semicolon. The view does not own the data.

2.3.3 The main extension rule: binding a temporary to a reference variable

A temporary's lifetime is extended when it is **directly bound** to a reference that is being initialized.

Binding to a const lvalue reference

```
#include <string>

int main() {
    const std::string& r = std::string("hello"); // temporary lives until
    ↪  end of scope
    // r is valid here
}
```

Binding to an rvalue reference (since C++11)

```
#include <string>

int main() {
    std::string&& rr = std::string("hello"); // temporary lives until end
    ↪  of scope
    // rr is valid here
}
```

Important constraint: the binding must be direct

The extension applies to the temporary that is directly bound in the initialization. It does *not* mean that any derived pointer/view/reference keeps the object alive.

2.3.4 Lifetime extension to reference members (common but misunderstood)

A temporary can also have its lifetime extended when it is bound directly to a **reference data member** during initialization of an object.

Example: reference member bound to a temporary

```
#include <string>

struct Holder {
    const std::string& ref;
    explicit Holder(const std::string& r) : ref(r) {}
};

int main() {
```

```
    Holder h{std::string("hello")}; // the temporary's lifetime is
    ↪  extended to match h
    // h.ref is valid as long as h is alive
}
```

Professional warning

Even though this is well-defined, it can still be a poor design because:

- it stores a reference (non-owning) as a data member,
- it invites accidental dangling if constructed differently,
- it makes lifetime coupling implicit and harder to reason about.

In safety-critical code, prefer ownership (`std::string`, `std::vector`, smart pointers) instead of reference members unless you have a strong reason and strict usage constraints.

2.3.5 What does *not* extend lifetime

1) Passing a temporary to a function parameter does not extend beyond the call

A temporary passed as an argument can live until the end of the full-expression containing the call, but **not longer**. This becomes dangerous if a function returns a reference (or stores a reference) to its argument.

```
#include <string>

const std::string& bad(const std::string& s) {
    return s; // returns a reference to the argument
}
```

```
int main() {
    const std::string& r = bad(std::string("tmp"));
    // The temporary is destroyed at the end of this statement -> r
    ↪ dangles.
    (void)r;
}
```

2) “Passing it on” does not extend lifetime

In general, you cannot extend a temporary’s lifetime by binding another reference to a reference that already refers to the temporary.

```
#include <string>

const std::string& first() {
    static const std::string& r = std::string("x"); // lifetime extended
    ↪ (static ref)
    return r;
}

int main() {
    const std::string& a = first(); // does not extend anything; it just
    ↪ refers to the same object
    (void)a;
}
```

The safe mental model is simple:

> Lifetime extension happens only at the point where the temporary is directly bound during initialization. It cannot be “forwarded” to create more extension.

3) Views do not extend lifetime (string_view, span, iterators)

Non-owning views are designed to be lightweight and fast. They do not own storage and do not control lifetime. Returning them or storing them requires a strong lifetime guarantee.

```
#include <string>
#include <string_view>

std::string_view bad_view() {
    return std::string("hello"); // view into destroyed temporary
}
```

Correct alternatives:

```
#include <string>

std::string good_owner() {
    return std::string("hello"); // return owning value
}
```

```
#include <string_view>

std::string_view good_static() {
    return "hello"; // string literal has static storage duration
}
```

2.3.6 C++17 and later: prvalues and temporary materialization

Since C++17, many prvalues act like an “unmaterialized recipe” until an actual object is needed (for example, when binding a reference, accessing a member, or calling a member function). This improves performance and removes some unnecessary temporaries. However, it does not change the safety rule:

> If you bind a reference/view/pointer to something that does not outlive it, you get a dangling reference/view/pointer.

So, write code that makes the owner and lifetime explicit, rather than relying on subtle materialization details.

2.3.7 Range-based `for`: lifetime extension and the C++23 fix

Range-based `for` loops are common places where temporaries appear.

The safe and clear pattern: name the owner

If the iteration depends on an object that provides a view into its own storage, name that owner explicitly:

```
auto owner = make_owner();          // owns storage
for (auto& x : owner.view()) {      // view refers into owner
    (void)x;
}
```

C++23: stronger lifetime rules for the range initializer

C++23 strengthened the lifetime rules for temporaries inside the `for-range-initializer` so that certain previously dangerous patterns become well-defined for the duration of the loop.
Even with this improvement, professional guidance remains:

- Do not rely on subtle lifetime rules to make code correct.
- Prefer making the owner explicit when the loop iterates through a view.

2.3.8 Practical engineering rules for lifetime extension

Use these rules consistently in safety-oriented C++:

1. Assume temporaries die at the end of the statement unless you *know* a lifetime extension rule applies.
2. Lifetime extension is reliable for direct initialization of a reference variable from a temporary.
3. Passing a temporary to a function does not make it live beyond the full-expression containing the call. Never return a reference/pointer/view to such an argument unless the argument is guaranteed to outlive it.
4. Views (`string_view`, `span`, iterators) do not extend lifetimes. They require an owner that outlives them.
5. In range-based `for`, prefer naming the owner explicitly when the range is derived from an owning temporary.
6. Prefer designs where ownership is explicit in types (`std::string`, containers, `unique_ptr`), and treat reference-based APIs as advanced tools requiring strict contracts.

2.3.9 Key takeaway

Lifetime extension is a narrow, explicit set of rules. It can prevent unnecessary copies and enable clean code, but it is also a common source of dangerous misunderstandings.
A safe C++ design philosophy is:

- keep owners explicit,

- return owning types when you create storage,
- return views only when the referenced storage clearly outlives the view.

2.4 Common lifetime violations

2.4.1 Why lifetime violations are uniquely dangerous in C++

In C++, an object is only safe to use while its **lifetime is active**. Many failures happen because code continues to use a pointer, reference, iterator, or view after the object it refers to has been destroyed, moved, or invalidated.
These violations typically cause **undefined behavior**: the program may appear to work, may crash, may corrupt unrelated data, or may fail only in optimized builds.

2.4.2 Violation 1: returning a pointer or reference to a local object

Local (automatic) objects are destroyed when the scope ends. Returning their address creates an immediate dangling pointer/reference.

Bad: returning address of a local

```
int* bad_ptr() {
    int x = 7;
    return &x; // dangling after return
}

const int& bad_ref() {
    int x = 7;
    return x;  // dangling after return
}
```

Professional fixes

```
int good_value() {
    int x = 7;
    return x; // return by value (safe)
}
```

```
#include <memory>

std::unique_ptr<int> good_owner() {
    return std::make_unique<int>(7); // explicit ownership (safe)
}
```

2.4.3 Violation 2: use-after-free (dangling after `delete`)

After `delete` (or `delete[]`), the object lifetime ends. The pointer value may still look non-null, but it is invalid to dereference or to use as if it points to a live object.

Bad: use-after-free

```
#include <iostream>

int main() {
    int* p = new int(42);
    delete p;
    std::cout << *p << "\n"; // undefined behavior
}
```

Professional fix: express ownership with RAII

```
#include <memory>
#include <iostream>
```

```
int main() {
    auto p = std::make_unique<int>(42);
    std::cout << *p << "\n"; // safe
} // automatic destruction
```

2.4.4 Violation 3: double-delete and mismatched delete

Deleting the same allocation twice, or using `delete` on memory allocated by `new[]` (or vice versa), is undefined behavior.

Bad: double delete / mismatch

```
int main() {
    int* p = new int(1);
    delete p;
    delete p;         // double delete (undefined behavior)

    int* a = new int[4];
    delete a;         // mismatch: should be delete[] (undefined behavior)
}
```

Professional fixes

```
#include <memory>

int main() {
    auto p = std::make_unique<int>(1);
    auto a = std::make_unique<int[]>(4); // correct array ownership
}
```

2.4.5 Violation 4: dangling references/views to temporaries

A temporary object usually dies at the end of the full-expression (typically the end of the statement). Non-owning views do not keep temporaries alive.

Bad: `std::string_view` from a temporary string

```
#include <string>
#include <string_view>

int main() {
    std::string_view sv = std::string("hello"); // dangles after ';'
    (void)sv;
}
```

Professional fixes

```
#include <string>

std::string good() {
    return std::string("hello"); // return an owner
}
```

```
#include <string_view>

std::string_view good_literal() {
    return "hello"; // string literal has static storage duration
}
```

2.4.6 Violation 5: iterator, pointer, and reference invalidation by containers

Standard containers have well-defined invalidation rules. A common lifetime mistake is to keep an iterator/pointer/reference into a container, then perform an operation that invalidates it.

Bad: `std::vector` reallocation invalidates pointers/references/iterators

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

int main() {
    std::vector<int> v{1,2,3};
    int* p = &v[0];          // pointer into v's storage
    v.push_back(4);          // may reallocate -> p may dangle
    std::cout << *p << "\n"; // undefined behavior if reallocated
}
```

Professional fixes

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

int main() {
    std::vector<int> v{1,2,3};
    std::size_t i = 0;       // index remains valid across reallocation
    v.push_back(4);
    std::cout << v[i] << "\n";
}
```

```cpp
#include <vector>

int main() {
    std::vector<int> v;
    v.reserve(1024);          // reduce reallocations when size is known
    // now pointers/refs are stable until capacity is exceeded
}
```

2.4.7 Violation 6: pointer/reference to a moved-from object's internals

After a move, the moved-from object remains valid but its internal state is unspecified. Pointers, iterators, and views into the moved-from object can become invalid.

Bad: view into moved-from string

```cpp
#include <string>
#include <string_view>

int main() {
    std::string s = "hello";
    std::string_view v = s;         // view into s
    std::string t = std::move(s);   // s is moved-from; v may now dangle
    (void)t; (void)v;
}
```

Professional rule

Do not keep non-owning references/views into an object across operations that can change its storage ownership (moves, swaps, reassignments, container mutations).

2.4.8 Violation 7: capturing references that outlive the referenced object

This is one of the most frequent production bugs in asynchronous and concurrent code: a lambda captures by reference, then runs after the referenced object is destroyed.

Bad: capturing reference for async use

```
#include <thread>
#include <string>

void launch() {
    std::string msg = "hi";

    std::thread t([&] { // captures msg by reference
        // if this runs after launch() returns, msg is destroyed ->
        ↪  dangling reference
        (void)msg.size();
    });

    t.detach(); // makes the bug more likely
}
```

Professional fix: capture by value or share ownership

```
#include <thread>
#include <string>

void launch() {
    std::string msg = "hi";
    std::thread t([m = std::move(msg)] { // capture by value (own the
    ↪  data)
        (void)m.size();
```

```
    });
    t.detach();
}
```

```
#include <thread>
#include <memory>
#include <string>

void launch() {
    auto msg = std::make_shared<std::string>("hi");
    std::thread t([msg] { // shared ownership
        (void)msg->size();
    });
    t.detach();
}
```

2.4.9 Violation 8: storing pointers/references to short-lived buffers

It is common to store `c_str()` pointers or pointers into local arrays and then use them later.

Bad: storing pointer to a temporary buffer

```
#include <string>

const char* store_cstr_bad() {
    std::string s = "hello";
    return s.c_str(); // pointer becomes dangling after return
}
```

Professional fix: return an owner

```
#include <string>
```

```
std::string store_cstr_good() {
    return "hello"; // caller can use c_str() while the string lives
}
```

2.4.10 Violation 9: manual lifetime management mistakes (placement new)

Advanced code sometimes constructs objects into raw storage. The storage may exist, but the object lifetime begins only after construction and ends at explicit destruction.

Bad: using storage as if an object is alive

```
#include <new>
#include <cstddef>

struct X { int v; };

int main() {
    alignas(X) std::byte buf[sizeof(X)];
    X* p = reinterpret_cast<X*>(buf);
    p->v = 1; // undefined behavior: X lifetime has not begun
}
```

Correct: begin and end lifetime explicitly

```
#include <new>
#include <memory>
#include <cstddef>

struct X { int v; explicit X(int n) : v(n) {} };

int main() {
```

```
    alignas(X) std::byte buf[sizeof(X)];
    X* p = ::new (static_cast<void*>(buf)) X(1); // begins lifetime
    (void)p->v;
    std::destroy_at(p);                          // ends lifetime
}
```

2.4.11 Common patterns that look safe but are not

"It is non-null, so it must be valid"

Non-null is not a validity guarantee. A dangling pointer can be non-null.

"It worked in Debug"

Lifetime violations often disappear in Debug builds and appear in Release builds because optimization changes timing, inlining, reordering, and allocation patterns.

"The view/iterator is cheap, so it is safe"

Views and iterators are cheap because they do not own. Their safety depends entirely on the lifetime of the underlying storage.

2.4.12 Professional prevention strategy

Design rules

1. Express ownership in types: `std::unique_ptr`, `std::shared_ptr`, values, RAII wrappers.

2. Return values (owners) when you create storage; return views only when storage clearly outlives the view.

3. Avoid reference data members and long-lived raw pointers unless lifetime is enforced by a stronger owner.
4. Treat container invalidation rules as part of your API contract.
5. In concurrent/asynchronous code, prefer capturing by value and explicit ownership transfer.

Verification rules (mandatory for serious codebases)

1. Run AddressSanitizer in tests to detect use-after-free, stack-use-after-scope, and bounds violations.
2. Use UndefinedBehaviorSanitizer to catch common undefined behaviors early.
3. Enable strong warnings and static analysis, and treat high-confidence lifetime warnings as defects.

2.4.13 Key takeaway

Most lifetime defects are not "small bugs". They are violations of the language rules that define whether an object is alive. The safest Modern C++ approach is:

- make ownership explicit,
- make lifetimes obvious,
- and verify aggressively with tooling.

Chapter 3

Undefined Behavior: The Silent Killer

3.1 What undefined behavior really means

3.1.1 A precise definition (in practical engineering terms)

In C++, **undefined behavior** (UB) means:

> The C++ standard imposes *no requirements* on what happens if the program executes a construct whose behavior is undefined.

That statement has a specific and non-obvious consequence:

- UB is **not** a "runtime error category" like an exception.
- UB is a **broken language contract**: once UB is reachable, the compiler is free to assume it never happens and optimize accordingly.
- Therefore, UB can produce **any observable result**: correct output, wrong output, crashes, silent data corruption, or different behavior under different builds.

The safest professional model is:

> A program that has reachable UB is not a valid C++ program in the sense required by the optimizer.

3.1.2 UB is different from other kinds of "not fully specified" behavior

C++ has several categories that people often confuse:

- **Implementation-defined behavior:** the implementation chooses a behavior and documents it (examples: size of `int`, signed integer representation, some ABI details). Your program can rely on it *for that implementation*.
- **Unspecified behavior:** the implementation may choose among multiple permitted behaviors, and does not need to document which. Both outcomes are correct. Your program must not depend on which one occurs.
- **Indeterminate values:** a value that is not specified; using it in most computations is incorrect and may lead to UB. This often occurs with uninitialized objects.
- **Undefined behavior:** no requirements at all; the program has stepped outside the rules.

Professional takeaway:

> Implementation-defined and unspecified behavior are part of the language model. Undefined behavior is leaving the language model.

3.1.3 Why the compiler "cares": UB enables optimization

Modern C++ compilers are built around aggressive transformations: inlining, vectorization, reordering, elimination of redundant loads/stores, and more.

Many of these transformations are correct only if the program obeys the standard rules. Because UB is assumed not to happen, the compiler may:

- remove checks that you thought were protecting you,
- reorder operations in ways that break your mental model,
- assume pointers are non-null or non-aliasing when UB would be required otherwise,
- transform code so that a bug disappears in Debug and appears in Release (or vice versa).

This is why UB is called “silent”: the program can fail far away from the mistake.

3.1.4 Common sources of undefined behavior (the ones that matter most in real systems)

1) Out-of-bounds access

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

int main() {
    std::vector<int> v{1,2,3};
    std::cout << v[3] << "\n"; // UB: out-of-bounds
}
```

Note: bounds-checked access (`v.at(i)`) throws an exception on error, but `operator[]` does not.

2) Dangling pointers and references (use-after-scope / use-after-free)

```
int* bad() {
    int x = 42;
    return &x; // UB: pointer dangles after return
}
```

```
#include <iostream>

int main() {
    int* p = new int(7);
    delete p;
    std::cout << *p << "\n"; // UB: use-after-free
}
```

3) Data races in multi-threaded code

In C++, a data race is UB. This is a correctness rule, not “sometimes wrong output”.

```
#include <thread>

int x = 0;

void inc() {
    for (int i = 0; i < 100000; ++i) {
        ++x; // UB if run concurrently: data race
    }
}

int main() {
    std::thread a(inc), b(inc);
    a.join(); b.join();
}
```

4) Signed integer overflow

Signed overflow is UB. Unsigned overflow is well-defined modulo arithmetic.

```
#include <limits>

int main() {
    int x = std::numeric_limits<int>::max();
    x = x + 1; // UB: signed overflow
}
```

If you need wraparound behavior, use unsigned types explicitly (or use checked arithmetic utilities).

5) Invalid shift operations

Shifting by a negative amount or by an amount greater than or equal to the bit width is UB.

```
#include <cstdint>

int main() {
    std::uint32_t x = 1u;
    auto y = x << 32; // UB: shift count equals width
    (void)y;
}
```

6) Strict aliasing violations (accessing an object through an incompatible type)

A common unsafe pattern is type-punning through pointer casts.

```
#include <cstdint>

float pun_bad(std::uint32_t u) {
```

```
    return *reinterpret_cast<float*>(&u); // UB: violates aliasing rules
}
```

A safe, portable approach is to copy bytes:

```
#include <bit>
#include <cstdint>

float pun_good(std::uint32_t u) {
    return std::bit_cast<float>(u); // C++20, requires same size
}
```

7) Uninitialized reads (indeterminate values) leading to UB

Reading an uninitialized value (other than certain allowed byte-like cases) is incorrect and often becomes UB.

```
#include <iostream>

int main() {
    int x;
    std::cout << x << "\n"; // uses indeterminate value; not a valid
    ↪ program
}
```

3.1.5 Why UB does not reliably reproduce

UB often depends on factors that change between runs:

- optimization level (`-O0` vs `-O2`/`-O3`),
- inlining decisions,

- allocator layout and timing,
- thread scheduling,
- different compilers or versions,
- different target CPUs and ABIs.

So a program can:

- "work" for years and fail after a compiler upgrade,
- pass tests but fail in production,
- crash only under heavy load or only on one platform.

3.1.6 The engineering goal: make UB unrepresentable by design

A professional C++ safety mindset treats UB as a class of defects to eliminate structurally. Key strategies:

- **Ownership and RAII:** use `std::unique_ptr`, `std::shared_ptr` deliberately, and RAII wrappers for non-memory resources.
- **Bounds-aware APIs:** prefer `std::span` and container/range-based interfaces to raw pointers.
- **Concurrency discipline:** avoid shared mutable state; use atomics/mutexes with clear invariants.
- **Avoid risky casts:** prefer `std::bit_cast`, `std::memcpy`, and explicit conversions.
- **Prefer values over references when returning newly created data:** return owners, not views into temporaries.

3.1.7 Verification: you must detect UB, not guess it

Because UB is silent and non-deterministic, debugging by symptoms is unreliable. Serious C++ projects verify with tools:

- **AddressSanitizer (ASan):** detects many memory errors (out-of-bounds, use-after-free, etc.).
- **UndefinedBehaviorSanitizer (UBSan):** detects many UB categories (overflow, invalid shifts, etc.).
- **ThreadSanitizer (TSan):** detects data races.
- **Static analysis and warnings:** catches risky patterns before runtime.

Sanitizers do not prove absence of UB, but they make UB visible early, close to the root cause.

3.1.8 Key takeaway

Undefined behavior is not "a bug that crashes." It is the absence of a defined contract. Once UB is reachable, the optimizer is allowed to produce code with *any* outcome.
Modern C++ safety engineering is therefore built on:

- designs that make ownership and lifetime explicit,
- APIs that carry size and invariants,
- disciplined concurrency rules,
- and continuous verification with sanitizers and analysis.

3.2 Why the compiler is not your friend here

3.2.1 The uncomfortable truth

When your program has **undefined behavior** (UB), the compiler is not an investigator trying to preserve your intent. It is an optimizer trying to produce the fastest correct machine code for *well-defined* C++ programs.
The C++ model is built on a contract:

> If your program follows the rules, the compiler must preserve its meaning. If your program violates the rules (UB), the compiler is free to assume that path never happens.

This is why UB is so dangerous: the compiler may **transform your code** in ways that make bugs appear, disappear, or move to unrelated locations.

3.2.2 UB is an optimization permission, not a runtime event

Many developers imagine UB as “the CPU will do something random at runtime.” The more important reality is:

- UB gives the compiler permission to assume certain states are impossible.
- If a check is only useful when UB is possible, the compiler can remove the check.
- If a branch can only be taken when UB occurs, the compiler can treat it as unreachable.

So the compiler is not your friend here because it is allowed to **stop defending you** once the code is outside the language rules.

3.2.3 Example 1: “My null check should protect me”

In valid C++, dereferencing a null pointer is UB. That means the compiler may assume a pointer used for dereference is not null, and it may remove checks that depend on it.

```
int f(int* p) {
    *p = 1;            // UB if p is null
    if (p == nullptr) // the compiler may treat this as always false
        return 0;
    return 1;
}
```

Professional takeaway:

> Do not write code that relies on “detecting UB after it happens”. Prevent the UB in the first place.

3.2.4 Example 2: signed overflow and “impossible” branches

In C++, signed integer overflow is UB. The optimizer may assume it never occurs, and it can simplify logic accordingly.

```
#include <limits>

int g(int x) {
    if (x > 0 && x + 1 < x) { // relies on overflow to become true
        return 1;
    }
    return 0;
}
```

Because `x + 1 < x` can only be true if signed overflow happens, the compiler is free to treat the condition as impossible and remove the branch.

Professional guidance:

- If you need wraparound arithmetic, use unsigned integers intentionally (well-defined modulo).
- If you need checked arithmetic, use explicit checked operations (or library utilities), not “hope and test” logic.

3.2.5 Example 3: out-of-bounds makes the optimizer trust your indices

Accessing outside array bounds is UB. That enables loop and bounds optimizations, including assuming that an index is always within the valid range on well-defined paths.

```
int sum10(const int* a, int n) {
    int s = 0;
    for (int i = 0; i < n; ++i) {
        s += a[i]; // UB if i ever goes out of bounds of the actual array
    }
    return s;
}
```

If the caller passes `n` larger than the actual array length, the program is already invalid. The optimizer is allowed to assume this does not occur and may reorder, vectorize, or eliminate checks you might expect.
Professional fix:

- Carry bounds with the pointer (e.g., `std::span`).
- Validate external sizes at API boundaries.

3.2.6 Example 4: strict aliasing and “I can reinterpret anything”

Type-punning through incompatible pointer types can violate aliasing rules (UB). Compilers use aliasing rules to keep values in registers, reorder loads/stores, and eliminate what look like redundant operations.

```
#include <cstdint>

float pun_bad(std::uint32_t u) {
    return *reinterpret_cast<float*>(&u); // UB: aliasing violation
}
```

Safe alternatives:

```
#include <bit>
#include <cstdint>

float pun_good(std::uint32_t u) {
    return std::bit_cast<float>(u); // C++20, requires same size
}
```

Or:

```
#include <cstdint>
#include <cstring>

float pun_memcpy(std::uint32_t u) {
    float f;
    std::memcpy(&f, &u, sizeof(f));
    return f;
}
```

3.2.7 Example 5: data races are UB, so the compiler can reorder aggressively

In C++, a data race is UB. That means you cannot reason about racy code by imagining "it will probably interleave like this." The compiler can reorder memory operations, and the CPU can also reorder at runtime.

```
#include <thread>
#include <iostream>

int flag = 0;
int data = 0;

void producer() {
    data = 123;  // unsynchronized
    flag = 1;    // unsynchronized
}

void consumer() {
    while (flag == 0) {}   // data race
    std::cout << data;     // also data race; may print 0, 123, or worse
}

int main() {
    std::thread a(producer), b(consumer);
    a.join(); b.join();
}
```

Professional fix: use atomics (with correct ordering) or mutexes to define synchronization.

3.2.8 Why Debug builds often “work” and Release builds fail

UB is frequently masked in Debug builds because:

- less inlining and fewer optimizations preserve accidental timing/layout,
- debug allocators and guard pages catch some memory mistakes,
- extra checks change control flow and memory layout.

Release builds often:

- inline aggressively and remove redundancy,
- vectorize loops and reorder memory operations,
- assume UB never happens and optimize based on that assumption.

So UB bugs are famous for appearing only in production.

3.2.9 What the compiler *can* do that feels “hostile”

Once UB is reachable, the optimizer may legally:

- remove conditional checks that depend on UB being possible,
- assume pointers are valid and properly aligned when required by the rules,
- treat certain branches as unreachable and delete code,
- reorder operations in ways that break your informal reasoning,
- propagate “impossible” assumptions far beyond the original mistake.

This is not malicious behavior. It is the direct consequence of compiling under the C++ abstract machine rules.

3.2.10 Professional discipline: make the compiler your ally again

You make the compiler your ally by staying inside the rules:

1. **Make ownership explicit:** RAII, `std::unique_ptr`, `std::shared_ptr` when truly needed.
2. **Make bounds explicit:** `std::span`, container interfaces, validated sizes at boundaries.
3. **Make concurrency defined:** atomics/mutexes with a documented synchronization contract.
4. **Avoid risky casts:** prefer `std::bit_cast` or `std::memcpy` for byte-level work.
5. **Verify continuously:** run sanitizers and static analysis in CI under realistic configurations.

3.2.11 Key takeaway

The compiler is not your friend in the presence of undefined behavior because it is allowed to assume UB never occurs and optimize accordingly. If UB is reachable, you cannot rely on "reasonable" execution.
Your job in Modern C++ safety engineering is to prevent UB structurally, so the compiler is forced to preserve your intent and can optimize safely within a well-defined program.

3.3 Real-world consequences

3.3.1 Why undefined behavior is a production risk, not a theory topic

Undefined behavior (UB) is not merely “unexpected output.” In real systems it becomes a **risk multiplier** because it can manifest as:

- **availability failures**: crashes, hangs, watchdog resets, service restarts
- **integrity failures**: silent data corruption, broken invariants, corrupted files or databases
- **confidentiality failures**: information leaks through out-of-bounds reads or use-after-free reuse
- **security vulnerabilities**: exploitable memory corruption paths

The most dangerous property is that UB breaks the optimization contract: after UB becomes reachable, the compiler is allowed to produce code with *any* outcome, and the failure can appear far away from the cause.

3.3.2 Consequence 1: “Heisenbugs” that vanish under debugging

UB often disappears when you:

- enable debug checks,
- add logging,
- change optimization level,
- change allocator behavior,
- or change timing (threads, I/O).

That is why teams spend days chasing bugs that “cannot be reproduced”.

Example: uninitialized read that changes when you add a print

```
#include <iostream>

int f(bool choose) {
    int x;                  // uninitialized
    if (choose) x = 10;
    return x + 1;           // UB if choose == false (indeterminate value
    ↪ used)
}

int main() {
    std::cout << f(false) << "\n"; // may print anything, may differ by
    ↪ build
}
```

Adding a log statement can change stack layout and make the symptom disappear, while the UB remains.

3.3.3 Consequence 2: Release-only failures and “impossible” branches

Optimizers exploit language rules aggressively. If a branch can only be taken when UB happens, the compiler may treat that branch as unreachable and remove it.

Example: logic that relies on signed overflow

```
#include <limits>

bool suspicious(int x) {
    // The second condition can only be true if signed overflow occurs.
    if (x > 0 && x + 1 < x) {
        return true;
```

```
    }
    return false;
}
```

In practice, different compilers and optimization levels can transform such code in ways that surprise developers. This is why UB bugs frequently appear after a compiler upgrade or after switching from `-O0` to `-O2`.

3.3.4 Consequence 3: silent data corruption (the worst kind of failure)

Crashes are painful but visible. UB frequently produces **silent corruption**: the program continues running with broken state.
Common corruption patterns:

- writing past the end of a buffer and corrupting an adjacent object
- use-after-free writing into memory that has been reused for a different object
- violating invariants that later cause incorrect decisions (wrong routing, wrong permissions, wrong totals)

Example: one out-of-bounds write, many delayed symptoms

```
#include <array>

struct Record {
    std::array<char, 8> tag;
    int value;
};

void write_tag(Record& r, const char* src) {
```

```
    // If src is longer than 8, this can overwrite r.value or other
    ↪ adjacent memory.
    for (int i = 0; src[i] != '\0'; ++i) {
        r.tag[i] = src[i]; // UB when i >= 8 (out-of-bounds)
    }
}
```

The bug might not crash immediately. It can corrupt `value`, corrupt a vptr, or corrupt a container header, and the failure can occur much later.

3.3.5 Consequence 4: security vulnerabilities (integrity, confidentiality, availability)

A large portion of serious security issues in native code arise from memory safety violations that are UB in C++:

- out-of-bounds reads (information disclosure)
- out-of-bounds writes (memory corruption)
- use-after-free (memory corruption, potential control-flow hijack)
- type confusion and lifetime violations

From a security engineering perspective, UB defects are particularly dangerous because attackers can often shape inputs and heap layout to make a rare bug reliably exploitable. They can turn a crash into a controlled write, or turn a stale pointer into a deliberate leak.

3.3.6 Consequence 5: concurrency failures that cannot be reasoned about

In C++, a **data race is undefined behavior**. That means racy code is not merely "non-deterministic"; it is outside the language contract. The compiler can reorder memory

operations, and the CPU can also reorder at runtime.

Example: racy publication of data

```
#include <thread>
#include <iostream>

int flag = 0;
int data = 0;

void producer() {
    data = 123; // unsynchronized write
    flag = 1;   // unsynchronized write
}

void consumer() {
    while (flag == 0) {}   // unsynchronized read (data race)
    std::cout << data;     // unsynchronized read (data race)
}

int main() {
    std::thread a(producer), b(consumer);
    a.join(); b.join();
}
```

This can print `0`, `123`, hang, or do something worse, depending on optimization and hardware. The correct fix is not "add sleeps." The correct fix is a defined synchronization design: atomics with a correct ordering contract, or a mutex.

3.3.7 Consequence 6: "tooling blind spots" and incomplete test coverage

Tests tend to cover intended behavior, not UB. UB can remain dormant until:

- a new input triggers an edge case,
- the allocator layout changes (different libc, different platform, different build),
- concurrency timing changes under load,
- an optimization transforms control flow.

This is why serious C++ codebases treat UB detection as a continuous engineering activity, not an occasional debugging step.

3.3.8 How professionals reduce real-world UB impact

1) Design to make UB difficult to write

- Express ownership with RAII (`unique_ptr`, values, containers).
- Use bounds-aware interfaces (`std::span`, container ranges) rather than raw pointer + length.
- Avoid storing non-owning views unless the owner lifetime is obvious and enforced.
- Avoid shared mutable global state; define explicit concurrency contracts.

2) Use sanitizers as standard practice

Sanitizers are engineered specifically to make UB visible:

- AddressSanitizer: detects many out-of-bounds and use-after-free patterns.
- UndefinedBehaviorSanitizer: detects many UB categories (overflow, invalid shifts, misalignment, etc.).
- ThreadSanitizer: detects data races.

They do not prove absence of UB, but they reliably surface classes of bugs that otherwise reach production.

3) Treat UB findings as correctness defects, not “warnings”

A UB report is a broken contract. In safety-focused C++ engineering:

- UB is fixed at the root cause (ownership, bounds, lifetime, synchronization).
- “Workarounds” that silence symptoms without restoring correctness are rejected.

3.3.9 Key takeaway

Undefined behavior has real-world consequences that are often worse than a crash: silent corruption, security vulnerabilities, and failures that appear only after optimization, hardware changes, or increased load.
Modern C++ safety engineering is the discipline of keeping the program inside the defined rules so that compilers, CPUs, and libraries can optimize aggressively without turning your defects into chaos.

3.4 How undefined behavior propagates

3.4.1 The core idea: UB is not local

Undefined behavior (UB) is not like an exception that occurs at one point and stays contained. UB is a **broken language contract**. Once the program executes a UB-triggering operation, the C++ abstract machine provides **no guarantees** about what happens next.
In practice, UB propagates in two main ways:

- **Runtime propagation**: memory corruption or invalid reads/writes poison program state.

- **Compile-time propagation**: the optimizer assumes UB never happens and transforms code under that assumption, potentially deleting checks and reordering operations.

3.4.2 Propagation path 1: the optimizer treats UB paths as impossible

Many UB categories allow the compiler to assume that certain states cannot occur. That assumption can spread far beyond the original bug because modern optimization is global: inlining, constant propagation, dead-code elimination, vectorization, and interprocedural analysis.

Example: UB can eliminate your safety check

Dereferencing a null pointer is UB. Therefore, if code dereferences a pointer, the compiler may assume the pointer is non-null and simplify later checks:

```
int f(int* p) {
    *p = 1;                  // UB if p is null
    if (p == nullptr) {      // may be optimized away as unreachable
        return 0;
    }
    return 1;
}
```

Once UB is reachable, a check that would only be true in UB scenarios can be removed. This is a key reason UB feels “hostile”: the compiler is permitted to stop defending you.

Example: signed overflow turns branches into “dead code”

Signed integer overflow is UB. If a branch condition relies on overflow being possible, the compiler may treat that branch as impossible:

```
#include <limits>

bool overflow_branch(int x) {
    // Only true if signed overflow occurs.
    if (x > 0 && x + 1 < x) {
        return true;
    }
    return false;
}
```

The optimizer can legally transform this into a constant `false`, because for a valid C++ program the condition is never true.

3.4.3 Propagation path 2: UB contaminates values (“poisoning” program state)

Some UB originates as a bad value:

- an indeterminate value (uninitialized read),
- a pointer that is invalid or incorrectly aligned,
- a reference that dangles,
- an integer computation that overflows (signed),
- a shift count that is invalid.

Once a value becomes invalid under the language rules, computations using it are no longer meaningful. The compiler may treat results derived from UB as unconstrained and optimize aggressively.

Example: uninitialized value spreads silently

```
#include <cstddef>

int compute(bool choose) {
    int x;                  // uninitialized
    if (choose) x = 10;
    int y = x + 1;          // UB if choose == false
    return y * 2;           // y propagates the UB further
}
```

Even if the symptom appears later (wrong return value, crash in unrelated code), the root cause is earlier: the program used a value that never had a defined meaning.

3.4.4 Propagation path 3: memory corruption spreads across objects and time

The most destructive UB categories are memory safety violations: out-of-bounds writes, use-after-free writes, and type/lifetime violations that write into storage that does not currently hold the expected object.
Memory corruption has two properties that make it propagate:

- **Spatial propagation**: one overflow can damage adjacent objects in memory.
- **Temporal propagation**: corruption may not be observed until much later.

Example: out-of-bounds write corrupts a neighbor

```
#include <array>
#include <cstdint>

struct Packet {
    std::array<std::uint8_t, 8> tag;
    std::uint32_t length;
};

void set_tag(Packet& p, const std::uint8_t* src, std::size_t n) {
    // UB if n > 8: writes beyond tag, may corrupt length
    for (std::size_t i = 0; i < n; ++i) {
        p.tag[i] = src[i];
    }
}
```

This might:

- corrupt `length` (logic failure),
- corrupt padding (later used in comparisons or hashing),
- corrupt an adjacent object if the struct is inside a container.

The failure often appears in a different subsystem than where the write occurred.

3.4.5 Propagation path 4: lifetime UB invalidates the type system

C++ lifetime rules define when a region of storage may be treated as an object of type `T`. If you read or write through a pointer/reference to an object whose lifetime has ended (or has not begun), you have UB.

A key reason this propagates is that the compiler uses the type system to optimize. Once you violate lifetime/type rules, the optimizer may legally assume conditions that no longer hold.

Example: use-after-free creates “type confusion”

```
#include <new>
#include <cstddef>

struct A { int x; };
struct B { double y; };

int main() {
    void* mem = ::operator new(sizeof(B));

    A* a = new (mem) A{1};
    a->~A();                  // end lifetime of A

    B* b = new (mem) B{3.0}; // begin lifetime of B in same storage

    // UB: treating storage as A after B's lifetime began
    int v = a->x;
    (void)v;

    b->~B();
    ::operator delete(mem);
}
```

Here the storage is real, but the *object* you are pretending exists is not. The bug may manifest as incorrect values, crashes, or optimizer-driven miscompilations.

3.4.6 Propagation path 5: aliasing UB enables load/store reordering

C++ aliasing rules restrict how you may access an object through different types. Compilers use these rules for optimizations such as keeping values in registers and reordering memory operations.

If you violate aliasing rules, you can observe seemingly impossible behavior: updates appear to be ignored, reads appear stale, and values appear to change “by magic”.

Example: illegal type-punning through pointer cast

```
#include <cstdint>

float pun_bad(std::uint32_t u) {
    return *reinterpret_cast<float*>(&u); // UB: aliasing violation
}
```

Safe patterns:

```
#include <bit>
#include <cstdint>

float pun_good(std::uint32_t u) {
    return std::bit_cast<float>(u); // C++20, same-size bit
    ↪  reinterpretation
}
```

```
#include <cstdint>
#include <cstring>

float pun_memcpy(std::uint32_t u) {
    float f;
    std::memcpy(&f, &u, sizeof(f));
    return f;
}
```

3.4.7 Propagation path 6: data races break the memory model

In C++, a **data race** is UB. This is stronger than “the result is unpredictable”. It means the program is outside the language rules, so neither you nor the compiler can reason about the

behavior reliably.

Data races propagate because:

- the compiler may reorder operations,
- the CPU may reorder operations,
- caches and coherence interact with timing,
- the observed failure depends on load and scheduling.

Example: racy publication

```
#include <thread>
#include <iostream>

int flag = 0;
int data = 0;

void producer() {
    data = 123; // unsynchronized
    flag = 1;   // unsynchronized
}

void consumer() {
    while (flag == 0) {}     // data race
    std::cout << data << "\n"; // data race
}

int main() {
    std::thread a(producer), b(consumer);
    a.join(); b.join();
}
```

Correct fixes require a defined synchronization contract (atomics with correct ordering, or a mutex).

3.4.8 Why UB propagates further in modern builds

Modern build configurations amplify UB propagation:

- **Inlining**: UB inside a small function becomes visible to the caller and enables broader simplifications.
- **LTO (Link-Time Optimization)**: the optimizer sees across translation units and can propagate "impossible" assumptions globally.
- **Vectorization and unrolling**: one UB in a loop can transform the entire loop body.
- **Profile-guided optimization**: hot paths are rewritten more aggressively.

This is why UB often appears "after" a compiler upgrade or after enabling `-O2/-O3` or LTO, even though the bug existed all along.

3.4.9 The key safety implication: you cannot contain UB with local checks

Once UB is executed, local reasoning fails. For example:

- adding `assert` after a UB operation does not make the program safe,
- catching exceptions does not help (UB is not an exception),
- logging can hide the symptom by changing layout and timing.

The only reliable strategy is to prevent UB from being reachable.

3.4.10 How professionals limit UB propagation

Design-level prevention

1. Make ownership explicit (RAII, `std::unique_ptr`, containers).
2. Make bounds explicit (`std::span`, safe container access at boundaries).
3. Avoid dangling views (return owners when creating storage; return views only when storage outlives them).
4. Use defined concurrency (atomics/mutexes with documented invariants).
5. Avoid undefined arithmetic and invalid shifts; use checked or unsigned arithmetic intentionally.

Tooling-level detection

- AddressSanitizer: catches many spatial/temporal memory safety errors.
- UndefinedBehaviorSanitizer: catches many UB categories (overflow, invalid shift, misalignment, etc.).
- ThreadSanitizer: catches data races.
- Static analysis and warnings: catch risky patterns before runtime.

3.4.11 Key takeaway

Undefined behavior propagates because it breaks the rules that both the compiler and your reasoning depend on. It spreads through:

- optimizer assumptions (removing checks, deleting branches, reordering),

- contaminated values (indeterminate/invalid state),
- memory corruption (spatial and temporal spread),
- lifetime/type and aliasing violations (breaking the type system),
- concurrency data races (breaking the memory model).

Therefore, UB must be treated as a **root-cause defect**. The professional goal is not to “handle UB” but to design code so UB is not reachable, and to continuously verify that property with modern tooling.

Part II

Classic Memory Errors and How Professionals Avoid Them

Chapter 4

Memory Leaks

4.1 Why leaks still happen in modern code

4.1.1 The modern reality

Modern C++ (C++11 and later) provides strong tools that *reduce* leaks: RAII, smart pointers, standard containers, and safer library abstractions. Yet leaks still happen in production code for one simple reason:

> Modern C++ makes leaks harder to write accidentally, but it cannot prevent leaks that are created by design mistakes, lifetime mismatches, ownership confusion, or hidden resource retention.

This section explains the most common real-world leak mechanisms that remain in modern code, even when teams “use RAII everywhere”.

4.1.2 Leak vs intentional retention

Not every unreleased allocation is a bug. Some memory is intentionally kept until process exit (caches, singletons, pools). The professional question is:

- Is the retention **bounded and justified**?
- Is it **observable and measured**?
- Is it **safe under reload/restart patterns** (services, plugins, tests)?

If the retention is unbounded or grows with workload, it becomes a leak in practice, even if the code "still has a pointer to it".

4.1.3 Reason 1: ownership is still unclear at API boundaries

The biggest source of leaks in modern systems is not raw `new` in leaf code; it is mismatched ownership between components.

Classic boundary: returning raw pointers

```
struct Widget { /* ... */ };

Widget* make_widget(); // Who owns the result? Who deletes it?

void use() {
    Widget* p = make_widget();
    // many exits possible...
    // delete p; might be forgotten or duplicated
}
```

Professional fix: ownership in the type

```
#include <memory>

struct Widget { /* ... */ };

std::unique_ptr<Widget> make_widget();

void use() {
    auto p = make_widget(); // ownership is explicit
}
```

If ownership must be shared, use `std::shared_ptr` deliberately and document the model. If ownership is not transferred, use references or `std::span`/`string_view` (with a clear owner).

4.1.4 Reason 2: `std::shared_ptr` cycles

`std::shared_ptr` uses reference counting. If two objects keep strong references to each other, their counters never reach zero, so the objects are never destroyed.

Leak: strong cycle

```
#include <memory>

struct B;

struct A {
    std::shared_ptr<B> b;
};

struct B {
```

```
    std::shared_ptr<A> a;
};

int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a; // cycle -> leak
}
```

Professional fix: break cycles with `std::weak_ptr`

```
#include <memory>

struct B;

struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::weak_ptr<A> a; // non-owning link
};
```

Professional guidance:

- Default to `std::unique_ptr` for ownership.
- Use `std::shared_ptr` only when shared lifetime is truly required.
- When shared ownership creates graphs, expect cycles and design with `std::weak_ptr`.

4.1.5 Reason 3: containers retain memory by design

Standard containers manage memory safely, but they may retain capacity:

- `clear()` destroys elements but does not necessarily free capacity.
- repeated growth can raise capacity, which may remain high for the process lifetime.

Not a leak, but can look like one

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

void handle_requests() {
    static std::vector<std::string> cache; // long-lived by design

    // cache grows with workload, maybe never shrinks
    cache.push_back("data");
    cache.clear(); // elements destroyed, capacity may remain
}
```

Professional approach

- If you need bounded memory, implement explicit limits (LRU, maximum capacity, eviction policy).
- Use `shrink_to_fit()` only as a *non-binding* request (do not depend on it for correctness).
- Prefer per-request/local containers instead of statics when possible.

4.1.6 Reason 4: caches, pools, and "one-time" allocations that become unbounded

Modern code often introduces:

- memoization tables,
- caches keyed by request/user,
- thread-local buffers,
- arenas/pools with monotonic growth.

These are safe in the sense that they are owned, but they can still grow without bound.

Example: cache without eviction

```
#include <unordered_map>
#include <string>

std::unordered_map<std::string, std::string> g_cache;

const std::string& lookup(const std::string& key) {
    return g_cache[key]; // grows forever if keys grow forever
}
```

Professional fix: define **an eviction policy** and **a memory budget**.

4.1.7 Reason 5: early exits around non-RAII resources

Even in modern C++ codebases, many leaks come from integrating C APIs or OS handles:

- `FILE*`, sockets, file descriptors

- mutexes/locks used manually
- memory from `malloc`, `posix_memalign`, platform allocators

Leak: missing cleanup on error path

```cpp
#include <cstdio>

void write_bad(const char* path) {
    std::FILE* f = std::fopen(path, "wb");
    if (!f) return;

    if (std::fputc('A', f) == EOF) return; // leak: fclose not called
    std::fclose(f);
}
```

Professional fix: RAII wrapper

```cpp
#include <cstdio>
#include <memory>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f);
    }
};

void write_good(const char* path) {
    std::unique_ptr<std::FILE, FileCloser> f(std::fopen(path, "wb"));
    if (!f) return;

    std::fputc('A', f.get()); // cleanup is automatic
}
```

4.1.8 Reason 6: exceptions + partial construction + raw ownership

If you manually own resources and an exception is thrown in the middle of a function, the cleanup code may never run. This is exactly what RAII prevents, but leaks still appear when:

- raw pointers are used for ownership in legacy or mixed code,
- resources are acquired before being wrapped in RAII,
- multiple resources are acquired with ad-hoc cleanup ordering.

Leak: resource acquired before RAII takes over

```
#include <stdexcept>

struct R { /* ... */ };

R* acquire();
void release(R*);

void bad() {
    R* r = acquire();
    if (!r) throw std::runtime_error("fail");
    // If something throws here, release(r) is skipped
    release(r);
}
```

Professional fix: wrap immediately

```
#include <memory>
#include <stdexcept>

struct R { /* ... */ };
```

```
R* acquire();
void release(R*);

struct RDeleter {
    void operator()(R* p) const noexcept { if (p) release(p); }
};

void good() {
    std::unique_ptr<R, RDeleter> r(acquire());
    if (!r) throw std::runtime_error("fail");
    // safe under exceptions
}
```

4.1.9 Reason 7: background work that outlives its owners (threads, callbacks, async)

Leaks often occur when a subsystem starts background work and forgets to stop it, or captures shared ownership unintentionally.

Leak pattern: never-ending background thread keeps state alive

```
#include <thread>
#include <memory>

struct Service {
    std::shared_ptr<int> state = std::make_shared<int>(42);

    void start() {
        std::thread([s = state] {
            for (;;) { /* work forever */ } // state never released
```

```
        }).detach();
    }
};
```

Professional fixes:

- use `std::jthread` (C++20) and cooperative stop with `std::stop_token`
- avoid detached threads for long-lived services unless there is a strict lifecycle manager
- ensure callbacks do not accidentally capture shared ownership forever

4.1.10 Reason 8: allocator mismatches and mixed allocation domains

Leaks happen when allocation and deallocation belong to different domains:

- `new` with `free` (invalid)
- `malloc` with `delete` (invalid)
- library allocators that require library-specific release functions

These are sometimes detected as crashes, but they can also become leaks if the program avoids deallocation due to fear of mismatches.
Professional approach:

- always pair allocation/deallocation within the same abstraction boundary
- wrap foreign allocations in RAII with the correct deleter immediately

4.1.11 Reason 9: “it leaks only on shutdown” is still a leak (sometimes)

Many teams ignore shutdown leaks. That can be acceptable for short-lived CLI tools, but it is often unacceptable for:

- long-running services that reload configuration or plugins
- test suites that run thousands of cases in one process
- embedded systems that must run for months

A safe rule:

> If the process lifetime is long or the component is reloadable, shutdown leaks become runtime leaks.

4.1.12 How professionals prevent leaks in modern C++

Design rules

1. Express ownership in types: values, containers, `std::unique_ptr` by default.
2. Use `std::shared_ptr` only when shared ownership is required; design against cycles.
3. Wrap foreign resources immediately in RAII (custom deleters).
4. Avoid global mutable caches without budgets and eviction.
5. Prefer structured lifetimes for background work (`std::jthread` + stop).

Verification rules

Use leak detection as a routine engineering practice:

- run AddressSanitizer with leak detection (or LeakSanitizer where available) in CI tests
- use platform tooling (Valgrind on Linux, MSVC CRT leak detection on Windows) when appropriate
- track memory growth under load tests (leaks are often workload-dependent)

4.1.13 Key takeaway

Modern C++ significantly reduces leaks when ownership is explicit and RAII is used consistently. Leaks still happen when:

- ownership crosses boundaries without being expressed in types,
- reference-count cycles exist,
- containers and caches retain memory unboundedly,
- non-RAII resources escape cleanup on error paths,
- background tasks outlive their owners,
- allocation domains are mixed incorrectly.

Professional leak prevention is therefore a mix of **good design** (explicit ownership and bounded retention) and **continuous verification** (sanitizers and realistic load testing).

4.2 Exception paths and early returns

4.2.1 Why this still matters in Modern C++

Even with Modern C++ practices, leaks still happen most often on **non-happy paths**:

- early returns from error handling,
- exceptions thrown mid-function,
- partially completed operations that skip cleanup,
- complex functions with multiple exit points.

The engineering goal is simple:

> Acquire resources in a way that guarantees cleanup on every exit path.

This is exactly what RAII provides. When leaks appear in modern code, it is usually because RAII was not applied consistently at the point where resources were acquired.

4.2.2 Early return leaks: the classic pattern

Bad: manual cleanup skipped on return

```
#include <cstdlib>

int process(bool ok1, bool ok2) {
    int* p = static_cast<int*>(std::malloc(1024 * sizeof(int)));
    if (!p) return -1;

    if (!ok1) return -2;   // leak: p not freed
    if (!ok2) return -3;   // leak: p not freed
```

```
    std::free(p);
    return 0;
}
```

Professional fix: RAII with a smart pointer + correct deleter

```
#include <cstdlib>
#include <memory>

struct FreeDeleter {
    void operator()(void* p) const noexcept { std::free(p); }
};

int process(bool ok1, bool ok2) {
    std::unique_ptr<void, FreeDeleter> p(std::malloc(1024 *
    ↪  sizeof(int)));
    if (!p) return -1;

    if (!ok1) return -2;    // safe: cleanup happens
    if (!ok2) return -3;    // safe: cleanup happens

    return 0;               // safe: cleanup happens
}
```

Here, cleanup is correct for *every* return path because ownership is bound to scope.

4.2.3 Exceptions: leaks when ownership is not established immediately

RAII prevents leaks under exceptions *only after the resource is owned by an RAII object.* A common leak pattern is acquiring a resource and doing more work before wrapping it.

Bad: resource acquired, exception thrown before ownership is safe

```
#include <stdexcept>

struct Resource { /* ... */ };
Resource* acquire();
void release(Resource*);

void work_that_may_throw();

void bad() {
    Resource* r = acquire();
    if (!r) throw std::runtime_error("acquire failed");

    work_that_may_throw();   // if this throws, r leaks

    release(r);
}
```

Professional fix: wrap immediately

```
#include <memory>
#include <stdexcept>

struct Resource { /* ... */ };
Resource* acquire();
void release(Resource*);

struct RDeleter {
    void operator()(Resource* p) const noexcept { if (p) release(p); }
};

void work_that_may_throw();
```

```
void good() {
    std::unique_ptr<Resource, RDeleter> r(acquire());
    if (!r) throw std::runtime_error("acquire failed");

    work_that_may_throw();   // safe: r will be released if an exception
    ↪  occurs
}
```

Professional rule:

> Wrap the resource in RAII immediately after acquisition, before any operation that can fail.

4.2.4 Multiple resources: leaks and partial cleanup ordering

When a function acquires multiple resources, manual cleanup becomes fragile: one missed `free`/`close`/`unlock` on one path becomes a leak.

Bad: ad-hoc cleanup with multiple exit points

```
#include <cstdio>

int bad(bool ok) {
    std::FILE* f = std::fopen("out.bin", "wb");
    if (!f) return -1;

    char* buf = new char[4096];
    if (!ok) return -2;      // leaks: file not closed, buf not freed

    delete[] buf;
    std::fclose(f);
```

```
    return 0;
}
```

Professional fix: compose RAII objects

```
#include <cstdio>
#include <memory>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept { if (f) std::fclose(f);
    ↪ }
};

int good(bool ok) {
    std::unique_ptr<std::FILE, FileCloser> f(std::fopen("out.bin",
    ↪ "wb"));
    if (!f) return -1;

    auto buf = std::make_unique<char[]>(4096);

    if (!ok) return -2;     // safe: both resources cleaned automatically
    return 0;
}
```

4.2.5 Exceptions during construction: partial objects are still cleaned correctly

A key safety feature of C++ is that if a constructor throws:

- fully constructed subobjects are destroyed automatically,
- members that have already been constructed are destroyed,

- the destructor of the partially constructed object is not run (because the object never finished construction).

RAII members therefore remain safe under exceptions in constructors.

Example: member RAII prevents leaks when a constructor throws

```
#include <memory>
#include <stdexcept>

struct Owner {
    std::unique_ptr<int[]> buf;
    Owner() : buf(std::make_unique<int[]>(1024)) {
        throw std::runtime_error("fail during construction");
    }
};

int main() {
    try {
        Owner o;
    } catch (...) {
        // buf was released automatically
    }
}
```

4.2.6 Destructors must not throw (or you risk termination)

Leak prevention depends on destructors executing reliably. If a destructor throws during stack unwinding (while another exception is active), the program will typically terminate. Professional rule:

- Destructors should be `noexcept` and should not report failure by throwing.

- For operations that can fail, provide explicit methods (e.g., `close()`, `flush()`, `commit()`) and keep the destructor best-effort.

4.2.7 Modern tools that reduce leak risk on non-happy paths

`std::unique_ptr` with custom deleters

Use this for resources that have non-`delete` cleanup:

- `std::free`,
- OS handles,
- C library types that require a special close/release function.

`std::make_unique` and `std::make_shared`

Prefer factory functions that construct and bind ownership in one expression.

`std::lock_guard` and `std::unique_lock`

Locks are resources. RAII for locks prevents deadlocks and ensures unlock on exceptions/returns.

```
#include <mutex>

std::mutex m;

void safe(bool ok) {
    std::lock_guard<std::mutex> lock(m); // unlocks automatically
    if (!ok) return;                     // safe
}
```

`std::jthread` and cooperative stopping (C++20)

Threads are long-lived resources. A detached thread can keep state alive indefinitely. `std::jthread` joins on destruction and supports cooperative stop with `std::stop_token`.

```
#include <thread>

void run(std::stop_token st);

void start_and_stop() {
    std::jthread t(run); // joined automatically on scope exit
}
```

4.2.8 Practical guidelines for leak-free error handling

1. Treat every resource (memory, file, socket, lock, thread) as something that must be owned by an RAII object.
2. Establish ownership **immediately** after acquisition.
3. Keep functions small and reduce the number of exit points in complex code, but do not rely on "one return" as a leak prevention strategy. Rely on RAII.
4. Prefer returning values and RAII-managed objects, not raw owning pointers.
5. In mixed C/C++ code, wrap C resources with `unique_ptr` + custom deleter or a dedicated RAII class.

4.2.9 Verification: you must test the non-happy paths

Leaks often appear only under failures. Professional verification includes:

- tests that force errors (allocation failure simulation, I/O failures, invalid input),
- running AddressSanitizer with leak detection (or LeakSanitizer where available),
- long-running stress tests to observe memory growth trends.

4.2.10 Key takeaway

Exception paths and early returns are the primary habitat of leaks in real-world C++. Modern C++ prevents these leaks when you:

- wrap resources immediately in RAII objects,
- use smart pointers and containers to represent ownership,
- keep destructors non-throwing,
- and test failure paths with sanitizers and stress tests.

4.3 Professional RAII-based design

4.3.1 What RAII really is (and why professionals rely on it)

RAII stands for **Resource Acquisition Is Initialization**. It is the design rule that makes C++ resource management predictable:

> A resource is acquired during object initialization, and released in the object's destructor.

RAII works because C++ guarantees that destructors for fully constructed objects are executed when leaving scope, whether that happens by:

- normal control flow,
- early return,
- exception propagation.

So RAII is not only about memory. In professional systems, “resource” includes:

- heap memory,
- file descriptors / `FILE*`,
- sockets and network connections,
- locks (mutexes, rwlocks),
- threads and background tasks,
- mapped memory regions,
- temporary configuration changes (restore-on-exit),
- transaction scopes (commit/rollback discipline).

4.3.2 The core RAII contract

A robust RAII type must make the following contract true:

1. **Construction establishes invariants.** After construction succeeds, the object owns a valid resource or represents “no resource” safely.
2. **Destruction releases exactly once.** No leaks and no double release.
3. **Destruction does not throw.** Destructors should be `noexcept` in practice.

4. **Ownership is explicit and enforceable.** Copying/moving follows a clear rule (unique ownership or shared ownership).

4.3.3 Professional design rule 1: model ownership explicitly

Unique ownership by default

Use `std::unique_ptr` (or value members) whenever possible. Unique ownership is simpler, faster, and easier to reason about than shared ownership.

```
#include <memory>

struct Image { /* ... */ };

std::unique_ptr<Image> load_image(); // clear owner
```

Shared ownership is a deliberate choice

Use `std::shared_ptr` only when multiple owners must extend lifetime together. Design for cycles using `std::weak_ptr` where needed.

```
#include <memory>

struct Node {
    std::shared_ptr<Node> next;  // can create cycles if used for graphs
    std::weak_ptr<Node>   prev;  // break cycles when needed
};
```

4.3.4 Professional design rule 2: acquire and bind ownership immediately

The most common RAII failure is a time window where a resource exists but is not yet owned by an RAII wrapper.

Bad: resource acquired, then work happens before wrapping

```
#include <cstdio>

void bad() {
    std::FILE* f = std::fopen("out.bin", "wb");
    if (!f) return;

    // If something throws or returns here, f leaks.
    // ...

    std::fclose(f);
}
```

Good: wrap immediately

```
#include <cstdio>
#include <memory>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f);
    }
};

void good() {
    std::unique_ptr<std::FILE, FileCloser> f(std::fopen("out.bin",
    ↪  "wb"));
    if (!f) return;

    // Any exit path is safe now.
}
```

4.3.5 Professional design rule 3: make copying and moving correct by construction

A resource owner must define what happens on copy and move.

Rule of zero (preferred)

If your class owns resources through RAII members (`std::unique_ptr`, containers, locks), you can often write no special member functions and let the compiler generate correct ones.

```
#include <memory>
#include <vector>

struct Buffer {
    std::vector<int> v;           // RAII
    std::unique_ptr<int> p;       // RAII
    // Rule of zero: no custom destructor/copy/move needed
};
```

Rule of five (when you manage ownership manually)

If you implement a destructor manually and own a resource directly, you typically must define: copy ctor, copy assign, move ctor, move assign, destructor. Professionals avoid this unless necessary.

Make unique ownership non-copyable

```
#include <cstdio>

class FileHandle {
```

```
    std::FILE* f = nullptr;

public:
    explicit FileHandle(std::FILE* fp) noexcept : f(fp) {}

    FileHandle(const FileHandle&)            = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    FileHandle(FileHandle&& other) noexcept : f(other.f) {
        other.f = nullptr;
    }

    FileHandle& operator=(FileHandle&& other) noexcept {
        if (this != &other) {
            reset();
            f = other.f;
            other.f = nullptr;
        }
        return *this;
    }

    void reset() noexcept {
        if (f) std::fclose(f);
        f = nullptr;
    }

    ~FileHandle() noexcept { reset(); }

    std::FILE* get() const noexcept { return f; }
};
```

This design makes ownership rules explicit:

- no accidental copying (prevents double-close),
- move transfers ownership (prevents leaks),
- destructor is safe and non-throwing.

4.3.6 Professional design rule 4: destructors must be noexcept and best-effort

Destructors frequently execute during stack unwinding. If a destructor throws during unwinding, the program will typically terminate.

Correct pattern: report failure via explicit operation

If release can fail (flush, commit, close with error), do not throw from the destructor. Provide an explicit operation that can report errors, and keep destruction best-effort.

```
#include <cstdio>
#include <system_error>

class FileWriter {
    std::FILE* f = nullptr;

public:
    explicit FileWriter(const char* path) {
        f = std::fopen(path, "wb");
        if (!f) throw std::system_error(errno, std::generic_category());
    }

    void close() { // can throw
        if (!f) return;
        if (std::fclose(f) != 0) {
```

```
            f = nullptr;
            throw std::system_error(errno, std::generic_category());
        }
        f = nullptr;
    }

    ~FileWriter() noexcept {
        if (f) std::fclose(f); // best-effort, no throw
    }
};
```

4.3.7 Professional design rule 5: RAII for locks and invariants

Locks are resources. Manual `lock()/unlock()` is fragile under exceptions and early returns. Use scope-based locking.

```
#include <mutex>

std::mutex m;
int shared = 0;

void update() {
    std::lock_guard<std::mutex> lock(m); // unlocks automatically
    ++shared;
} // lock released here on all paths
```

For more control, use `std::unique_lock` (deferred locking, timed locking, etc.).

4.3.8 Professional design rule 6: RAII for "rollback on failure"

Some resources need a transactional discipline: either fully apply changes, or roll back. RAII can represent "rollback unless committed".

Example: scope guard for rollback

```
#include <utility>

class ScopeExit {
    void (*fn_)(void*) = nullptr;
    void* ctx_ = nullptr;

public:
    ScopeExit(void (*fn)(void*), void* ctx) noexcept : fn_(fn), ctx_(ctx)
    ↪ {}

    ScopeExit(const ScopeExit&)            = delete;
    ScopeExit& operator=(const ScopeExit&) = delete;

    ScopeExit(ScopeExit&& other) noexcept : fn_(other.fn_),
    ↪ ctx_(other.ctx_) {
        other.fn_ = nullptr;
        other.ctx_ = nullptr;
    }

    ~ScopeExit() noexcept {
        if (fn_) fn_(ctx_);
    }

    void release() noexcept { fn_ = nullptr; }
};

struct Tx {
    bool committed = false;
};

void rollback_fn(void* p) noexcept {
```

```
    auto* tx = static_cast<Tx*>(p);
    if (!tx->committed) {
        // rollback actions here (must be noexcept)
    }
}

void do_work() {
    Tx tx{};
    ScopeExit rollback(&rollback_fn, &tx);

    // perform steps...
    // if an exception happens, rollback runs

    tx.committed = true;
    rollback.release(); // cancel rollback
}
```

This technique is widely used to guarantee cleanup and invariant restoration when functions have multiple failure points.

4.3.9 Professional design rule 7: prefer standard RAII types and patterns

In modern code, prefer standard facilities:

- `std::unique_ptr`, `std::shared_ptr` for dynamic lifetime
- standard containers for owned collections
- `std::lock_guard`, `std::unique_lock` for mutexes
- `std::jthread` (C++20) for threads that must be joined

Write custom RAII wrappers when the resource is not covered by the standard library (OS handles, third-party objects), and keep them small and mechanical.

4.3.10 Professional checklist: what to review in RAII code

1. Is ownership visible in the type?
2. Can the resource leak on early return or exception?
3. Can the resource be released twice?
4. Are copy/move operations correct and intentional?
5. Is the destructor `noexcept` and safe under partial construction and moves?
6. Are invariants restored on failure (rollback/cleanup)?
7. Are shared-ownership graphs protected against cycles?

4.3.11 Key takeaway

Professional RAII-based design is not just “use smart pointers”. It is a complete ownership and lifetime discipline:

- acquire resources and bind them to RAII immediately,
- encode ownership rules in types (unique by default),
- make destructors non-throwing and reliable,
- use RAII for non-memory resources (locks, threads, state changes),
- and use rollback-style guards when partial failure must restore invariants.

With this discipline, leaks become rare, localized, and detectable early by tooling.

Chapter 5

Dangling Pointers and References

5.1 Stack vs heap lifetime mistakes

5.1.1 The fundamental confusion

Many dangling pointer and dangling reference bugs come from mixing two different concepts:

- **Where storage comes from**: stack (automatic storage), heap (dynamic storage), static storage, etc.
- **How long an object lives**: the object lifetime rules of C++.

In practice, most mistakes occur when code assumes:

> "If the memory is still there, the object must still be valid."

That is false in C++. Storage can remain allocated while the object lifetime has ended, and stack storage can be reused immediately after leaving scope.

5.1.2 Stack (automatic storage): why it is easy to dangle

Automatic objects are destroyed when their scope ends. Their addresses may still look valid, but they become invalid to use because the object lifetime has ended and the stack space can be reused.

Mistake 1: returning pointer/reference to a stack object

```
int* bad_ptr() {
    int x = 42;      // stack object
    return &x;       // dangling after return
}

const int& bad_ref() {
    int x = 42;      // stack object
    return x;        // dangling after return
}
```

Professional fix: return by value or return an owning object

```
int good_value() {
    int x = 42;
    return x;        // safe: returns a value
}
```

```
#include <memory>

std::unique_ptr<int> good_owner() {
    return std::make_unique<int>(42); // explicit heap ownership
}
```

Mistake 2: storing addresses of stack objects beyond the scope

```
#include <vector>

std::vector<const int*> g;

void bad_store() {
    int x = 7;
    g.push_back(&x); // stores pointer to stack object
} // x destroyed here -> g contains dangling pointer
```

Professional fix: store values, or store ownership (e.g., `std::unique_ptr`), or store indices/keys to a stable owning container.

Mistake 3: capturing references in callbacks or detached threads

This is a real-world source of serious production bugs.

```
#include <thread>
#include <string>

void start_async() {
    std::string msg = "hello";

    std::thread t([&] {         // captures msg by reference
        (void)msg.size();       // UB if runs after start_async returns
    });
    t.detach();                 // makes it likely
} // msg destroyed here
```

Professional fixes:

- capture by value (move into the lambda),

- or use shared ownership when necessary.

```
#include <thread>
#include <string>

void start_async() {
    std::string msg = "hello";
    std::thread t([m = std::move(msg)] {
        (void)m.size(); // safe: lambda owns the string
    });
    t.detach();
}
```

5.1.3 Heap (dynamic storage): different risks, different dangling patterns

Heap storage is obtained via `new`, `new[]`, allocators, `malloc`, or library APIs. Heap objects can outlive the creating scope, but they can also be destroyed while pointers to them remain.

Mistake 4: use-after-free (dangling after delete)

```
#include <iostream>

int main() {
    int* p = new int(5);
    delete p;           // lifetime ends here
    std::cout << *p;    // dangling pointer use (undefined behavior)
}
```

Professional fix: represent ownership with RAII and avoid raw owning pointers.

```
#include <memory>
```

```cpp
#include <iostream>

int main() {
    auto p = std::make_unique<int>(5);
    std::cout << *p;   // safe
} // cleanup automatic
```

Mistake 5: returning non-owning views into heap objects that are freed elsewhere

This is common when a function returns a pointer/reference to internal storage that is freed later.

```cpp
#include <string_view>
#include <memory>
#include <string>

std::string_view bad(std::unique_ptr<std::string> s) {
    return *s; // returns view into a string that will be destroyed on
    ↪  return
}
```

Professional fix: return an owning object when lifetime must extend beyond the call.

```cpp
#include <string>
#include <memory>

std::string good(std::unique_ptr<std::string> s) {
    return *s; // returns a new owning string
}
```

5.1.4 Stack vs heap: the most common misunderstanding

Developers sometimes move a bug from stack to heap thinking it is “fixed”:

> "If I allocate it on the heap, it will live longer."

Heap allocation does not automatically solve lifetime. It only changes who must manage it. If ownership is unclear, heap objects create different bugs:

- leaks (never destroyed),
- use-after-free (destroyed too early),
- double delete (destroyed twice),
- shared ownership cycles (never destroyed).

The real fix is **ownership correctness**, not heap usage.

5.1.5 Safe lifetime modeling: the professional approach

1) Own with values when possible

If an object does not need dynamic lifetime, store it by value. This is the safest and simplest model.

```
#include <string>

struct User {
    std::string name; // owned, no dangling risk
};
```

2) Use `std::unique_ptr` for unique heap ownership

```
#include <memory>

struct Session { /* ... */ };
```

```
struct Server {
    std::unique_ptr<Session> session; // clear ownership
};
```

3) Use `std::shared_ptr` only for true shared lifetime

And break cycles with `std::weak_ptr` when the object graph can become cyclic.

4) Use non-owning references only with an obvious owner

Non-owning references are acceptable when:

- the owner is clearly longer-lived than the reference,
- the API contract documents and enforces that relationship,
- the reference is not stored beyond the owner's lifetime.

Typical safe non-owning tools:

- references (`T&`, `const T&`) for short-lived usage,
- `std::span` for contiguous ranges with explicit size,
- `std::string_view` for read-only string views when the backing storage outlives the view.

5.1.6 Stack-specific traps that look correct

Trap: reference to a temporary does not always extend lifetime the way you think

Lifetime extension happens only in specific cases (direct binding to a reference during initialization). A view (`string_view`) never extends lifetime.

```
#include <string>
#include <string_view>

std::string_view trap() {
    return std::string("tmp"); // view dangles
}
```

Trap: pointer to `std::vector` element becomes invalid after growth

Not stack-specific, but very common in stack-local containers that reallocate.

```
#include <vector>

void trap_vector() {
    std::vector<int> v{1,2,3};
    int* p = &v[0];
    v.push_back(4); // may reallocate -> p may dangle
    (void)*p;
}
```

Professional fix: store an index, reserve capacity, or use a container with stable references if needed.

5.1.7 Professional checklist: preventing stack/heap lifetime mistakes

1. Never return or store pointers/references to stack objects beyond their scope.
2. Do not detach background work that captures references to local variables.
3. Never dereference after `delete`; represent ownership with RAII.
4. Treat views and iterators as **non-owning**: they require a guaranteed owner lifetime.

5. Know container invalidation rules; do not keep pointers/iterators across invalidating operations.

6. Prefer values and `std::unique_ptr`; use `std::shared_ptr` only when necessary.

5.1.8 Key takeaway

Stack and heap are storage strategies, not lifetime guarantees. Stack objects die when the scope ends; heap objects die when their owner releases them. Dangling bugs happen when code uses pointers, references, or views outside the owner's lifetime.
Professional C++ avoids these mistakes by making ownership explicit, using RAII for all resources, and treating non-owning references as short-lived, contract-bound tools.

5.2 Returning references and pointers

5.2.1 The core issue

In C++, functions frequently return references or pointers because they are flexible and efficient. However, if the object referred to does not outlive the caller's context, the result becomes a **dangling reference** or a **dangling pointer**, leading to undefined behavior.
The safe rule is simple:

> Only return a reference or pointer if the object's lifetime is guaranteed to extend beyond the call.

Anything else is unsafe.

5.2.2 Returning references/pointers to local (automatic) objects

Automatic objects (locals) are destroyed when their scope ends. Returning a reference or pointer to them produces a dangling result.

Incorrect pattern: returning a local reference

```
const int& bad_ref() {
    int x = 42;       // automatic object
    return x;         // dangling after exit
}

int* bad_ptr() {
    int y = 7;
    return &y;        // dangling after exit
}
```

Why this is unsafe

After the function returns, the underlying object no longer exists. Using the returned reference or pointer is undefined behavior: the compiler and optimizer do not guarantee any particular result.

5.2.3 Safe alternatives

Instead of returning a reference or pointer to a local object:

Return by value

For small and copyable types, returning by value is safe and often optimized (copy elision, return value optimization, guaranteed copy elision in C++17 and later).

```cpp
int good_value() {
    int x = 42;
    return x; // safe: returns a copy
}
```

Return an owning object

For dynamically allocated or resource-managing types, return a value of a smart pointer or container.

```cpp
#include <memory>
#include <string>

std::unique_ptr<std::string> make_string() {
    return std::make_unique<std::string>("hello");
}
```

5.2.4 Returning pointers/references to heap objects with clear ownership

Returning a pointer or reference to a heap object can be safe *only if* ownership and lifetime are clear and the caller is responsible for managing the resource.

Example: returning a raw pointer to a heap object with documented ownership

```cpp
#include <memory>

std::unique_ptr<int> make_owner() {
    return std::make_unique<int>(42);
}

int* get_ptr(std::unique_ptr<int>& owner) {
    return owner.get(); // safe as long as owner outlives the pointer
}
```

Professional guideline

A raw pointer (or reference) returned from a function should not imply ownership unless explicitly documented. The caller must know:

- who owns the object,
- when it will be destroyed,
- whether the pointer/reference can become invalid after mutation.

5.2.5 Returning references/pointers to static objects

Objects with static storage duration (global variables, function-local statics) outlive all function calls, so returning pointers or references to them is safe in terms of lifetime.

```
const std::string& global_string() {
    static const std::string value = "persistent";
    return value; // safe: lifetime is the entire program
}
```

Professional caution

Although returning references to static objects is lifetime-safe, global state has other hazards: shared mutability, hidden coupling, concurrency risks, and test isolation complications. Use static references sparingly and only when justified.

5.2.6 Returning views into containers

Types such as `std::string_view` and `std::span` represent non-owning views. They are safe to return only when the underlying storage is guaranteed to outlive the view.

Unsafe: returning a view into a temporary

```
#include <string>
#include <string_view>

std::string_view bad_view() {
    return std::string("temp"); // view into destroyed temporary
}
```

Safe: returning a view into a caller-provided buffer

```
#include <string_view>

std::string_view prefix(std::string_view s) {
    return s.substr(0, 3); // view into caller's storage
}
```

5.2.7 Returning iterators and container pointers

Iterators and pointers into containers are safe only if the container is not modified in ways that invalidate them. Container operations that reallocate or remove elements can invalidate pointers and iterators.

Unsafe pattern

```
#include <vector>

auto bad_iter(std::vector<int> v) {
    return v.begin(); // iterator into temporary container
}
```

The container `v` is destroyed at function exit; the iterator becomes dangling.

Safe alternatives

- accept the container by reference (ensure caller owns it),
- return a container by value (so the caller receives an owning container),
- return an index or key instead of a pointer/iterator.

```
#include <vector>

// returns an index instead of iterator
std::size_t safe_index(const std::vector<int>& v, int value) {
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == value) return i;
    }
    return v.size(); // not found
}
```

5.2.8 Lifetime extension rules (and their limits)

C++ has a specific lifetime extension rule for temporaries bound to **const lvalue references**:

```
const std::string& f() {
    static const std::string s = "hello";
    return s; // safe: static storage, lifetime outlives caller
}
```

Bind-to-reference lifetime extension does *not* apply when:

- the reference is a function parameter,
- the reference is stored beyond the initialization expression,
- the reference is to a subobject (e.g., `string_view` into a temporary).

5.2.9 Professional checklist for safe returns

1. Do not return references/pointers to local (automatic) objects.
2. When returning pointers/references, know and document who owns the object.
3. Prefer returning values (owners) when possible.
4. If returning a view, ensure the storage it refers to outlives the view.
5. Avoid returning raw owning pointers; prefer smart pointers for dynamic storage.
6. Prefer indices or keys over pointers/iterators when appropriate to avoid invalidation.
7. Treat container invalidation rules as part of the API contract.

5.2.10 Key takeaway

Returning references and pointers is safe only when the referred object's lifetime is guaranteed to outlive the use of that reference or pointer. Undefined behavior results when the lifetime ends earlier. Modern C++ safety engineering emphasizes clear ownership models, return-by-value, and explicit lifetime contracts to prevent dangling references and pointers.

5.3 Views vs ownership

5.3.1 The fundamental distinction

Modern C++ uses two very different categories of types:

- **Owning types** manage lifetime. They allocate and free resources automatically. Examples: `std::string`, `std::vector<T>`, `std::unique_ptr<T>`, `std::shared_ptr<T>`.

- **View types** do not manage lifetime. They are non-owning references into storage owned elsewhere. Examples: `T*` and `T&` (non-owning by default), `std::string_view`, `std::span<T>`, iterators, `std::basic_string_view`.

This distinction is one of the most important lifetime safety rules in modern C++:

> A view is safe only as long as its underlying owner remains alive and unchanged in ways that invalidate the view.

5.3.2 Why views exist (and why professionals use them)

Views are useful because they are:

- cheap to copy (usually pointer + length),
- fast to pass into functions (avoids allocation and deep copies),
- expressive for APIs (explicitly communicates "I will not own this").

Owning types, by contrast, are responsible for:

- allocation and deallocation,
- deep copies and moves,
- stable lifetime boundaries,
- resource release in destructors (RAII).

The correct professional model is:

> Use views for *parameters* and short-lived access. Use owners for *storage* and anything that must outlive the current scope.

5.3.3 The most common bug: returning a view to something you created

If a function creates storage (temporary string, local vector, local array) and returns a view into it, the view will dangle immediately after the function returns.

Bad: returning `std::string_view` to a temporary

```
#include <string>
#include <string_view>

std::string_view bad() {
    return std::string("hello"); // view refers to destroyed temporary
}
```

Bad: returning `std::span` to a local array

```
#include <span>

std::span<int> bad_span() {
    int a[4] = {1,2,3,4};
    return a; // span refers to stack memory that dies on return
}
```

Professional fix: return an owner

If you create storage, return an owning type (or move it out).

```
#include <string>

std::string good() {
    return std::string("hello"); // safe owner returned by value
}
```

```cpp
#include <vector>

std::vector<int> good_vec() {
    return {1,2,3,4}; // safe owner returned by value
}
```

5.3.4 Views as parameters: the correct and common usage

Views are ideal for parameters because they express non-ownership and avoid unnecessary allocations.

`std::string_view` for read-only string parameters

```cpp
#include <string_view>
#include <cstddef>

std::size_t count_slashes(std::string_view s) {
    std::size_t n = 0;
    for (char c : s) if (c == '/') ++n;
    return n;
}
```

This can accept:

- string literals,
- `std::string`,
- other character ranges that provide a contiguous string interface.

`std::span<T>` for contiguous ranges

```cpp
#include <span>
```

```cpp
#include <cstddef>

std::size_t sum(std::span<const int> s) {
    std::size_t total = 0;
    for (int x : s) total += static_cast<std::size_t>(x);
    return total;
}
```

This can accept:

- arrays,
- `std::vector<int>`,
- `std::array<int, N>`,
- and other contiguous buffers.

5.3.5 Views can be invalidated even when the owner is alive

Views are not only sensitive to *lifetime*. They are also sensitive to *mutation*.

Invalidation by reallocation (vector)

```cpp
#include <vector>
#include <span>

int main() {
    std::vector<int> v{1,2,3};
    std::span<int> s(v);      // view into v's storage

    v.push_back(4);           // may reallocate -> s may become dangling
    (void)s[0];               // undefined behavior if reallocated
}
```

Invalidation by string modification

```
#include <string>
#include <string_view>

int main() {
    std::string str = "hello";
    std::string_view v = str;

    str += " world";          // may reallocate -> v may become dangling
    (void)v.size();
}
```

Professional rule:

> Any operation that can reallocate or invalidate iterators can invalidate views.

5.3.6 Ownership types: strong safety, but not free

Owning types provide strong safety because they control lifetime, but they may be more expensive:

- allocations,
- deep copies (unless moved),
- larger objects, slower passes if not optimized.

Modern C++ mitigates much of this cost:

- move semantics (C++11),
- guaranteed copy elision in key cases (C++17),

- small-string optimization (implementation detail, but common),
- efficient containers and allocators.

Professional perspective:

> Choose ownership for correctness first, then optimize with moves and views where safe.

5.3.7 Designing APIs: when to return a view vs return an owner

Return a view only when storage is owned elsewhere and guaranteed stable

Safe return of a view usually requires one of these patterns:

- The view refers to **static storage** (string literals, global immutable data).
- The view refers to a **long-lived object** owned by the caller (input parameter).
- The view refers to internal storage of an object that clearly outlives the view (a member function returning a view, with the object itself as the owner).

Example: member function returning a view tied to `*this`

```
#include <string>
#include <string_view>

class User {
    std::string name_;

public:
    explicit User(std::string n) : name_(std::move(n)) {}
```

```
    std::string_view name() const noexcept {
        return name_; // safe as long as *this is alive and name_ not
        ↪  invalidated
    }
};
```

Return an owner when you create or transform data

If the function builds a new string/buffer/collection, returning an owning type is correct.

```
#include <string>
#include <string_view>

std::string make_prefixed(std::string_view s) {
    std::string out;
    out.reserve(3 + s.size());
    out += "ID:";
    out.append(s);
    return out; // owns new storage
}
```

5.3.8 Views, raw pointers, and “borrowed” references

Raw pointers (`T*`) and references (`T&`) are **views** in the lifetime sense unless explicitly documented as owning (which is discouraged in modern code).
Professional guideline:

- Use raw pointers as **non-owning observers** (nullable).
- Use references as **non-null observers** (must refer to a live object).
- Use smart pointers to represent ownership.

5.3.9 Professional rules for avoiding dangling views

1. Treat `std::string_view`, `std::span`, iterators, and raw pointers as non-owning.
2. Never return a view to local or temporary storage.
3. Do not store a view unless the owner is guaranteed to outlive it.
4. Assume mutation can invalidate views (reallocation, erase, insert, resize, move).
5. Prefer returning owning types when building new data.
6. If you must return a view, document the lifetime contract explicitly and keep it mechanically enforceable.

5.3.10 Key takeaway

Views are powerful tools for performance and clean APIs, but they are **not** lifetime management. Ownership types keep data alive; views only point into it.
A professional C++ codebase succeeds by:

- using views for parameters and short-lived inspection,
- using owners for storage and returned results,
- and never confusing “cheap to copy” with “safe to keep”.

5.4 Safer alternatives

5.4.1 What “safer” means in professional C++

In the context of dangling pointers and references, “safer alternatives” are techniques and types that:

- make ownership explicit,
- make lifetime constraints visible in types,
- reduce reliance on raw pointers and implicit contracts,
- prevent or detect misuse early (at compile time or reliably at runtime).

The goal is not to ban pointers and references. The goal is to use them in ways where their validity is obvious and enforceable.

5.4.2 Alternative 1: return by value (owning results)

When a function produces a new object, return it by value. Modern C++ makes this efficient via move semantics and copy elision.

Instead of returning a pointer to internal or temporary storage

```
#include <string_view>
#include <string>

std::string_view bad_make() {
    return std::string("tmp"); // dangling view
}
```

Return an owner

```
#include <string>

std::string good_make() {
    return std::string("tmp"); // safe: owning return
}
```

Professional guidance:

> If you create storage, return an owning type.

5.4.3 Alternative 2: `std::unique_ptr` for unique dynamic ownership

If the lifetime must outlive the current scope and dynamic allocation is required, represent ownership explicitly with `std::unique_ptr`.

Instead of raw owning pointers

```
struct Widget {};

Widget* bad_factory() {
    return new Widget(); // who deletes it?
}
```

Return explicit ownership

```
#include <memory>

struct Widget {};

std::unique_ptr<Widget> good_factory() {
    return std::make_unique<Widget>();
}
```

Custom deleters for non-`delete` resources

```
#include <memory>
#include <cstdio>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f);
    }
};

using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file(const char* path) {
    return unique_file(std::fopen(path, "rb"));
}
```

5.4.4 Alternative 3: `std::shared_ptr` + `std::weak_ptr` for shared lifetimes

Use shared ownership only when multiple components must keep an object alive together.

Safe shared lifetime

```
#include <memory>

struct Session { /* ... */ };

std::shared_ptr<Session> make_session() {
    return std::make_shared<Session>();
}
```

Avoiding cycles with `std::weak_ptr`

```
#include <memory>

struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev; // breaks cycles
};
```

Professional rule:

> Default to `unique_ptr`. Use `shared_ptr` only when the lifetime model truly requires it, and design against cycles.

5.4.5 Alternative 4: use `std::span` instead of pointer + length

Many bugs come from passing `T*` and a separate size that can disagree. `std::span` ties the pointer to its extent and improves clarity.

Instead of raw pointer + size

```
#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];
    return s;
}
```

Prefer `std::span`

```
#include <span>
```

```
int sum_good(std::span<const int> s) {
    int total = 0;
    for (int x : s) total += x;
    return total;
}
```

This improves safety by making the “range” a single value and encouraging bounds-aware design.

5.4.6 Alternative 5: use `std::string_view` for parameters, but return `std::string` for results

`std::string_view` is excellent for read-only parameters because it avoids allocations. But it is dangerous as a return type if the function creates the string data.

Good pattern: view in, owner out

```
#include <string>
#include <string_view>

std::string normalize(std::string_view input) {
    std::string out;
    out.reserve(input.size());
    for (char c : input) {
        if (c != ' ') out.push_back(c);
    }
    return out; // owning result
}
```

5.4.7 Alternative 6: prefer values and references for short-lived usage, avoid storing borrows

References (`T&`) and raw pointers (`T*`) are fine as **short-lived borrows**. Problems begin when you store them (especially in member variables) and lose track of the owner.

Instead of storing a raw pointer with unclear lifetime

```
struct Config { /* ... */ };

struct Bad {
    const Config* cfg; // who owns cfg? how long does it live?
};
```

Prefer ownership, or enforce a lifetime contract

```
#include <memory>

struct Config { /* ... */ };

struct GoodOwner {
    std::shared_ptr<const Config> cfg; // explicit shared ownership
};
```

```
#include <functional>

struct Config { /* ... */ };

struct GoodBorrow {
    std::reference_wrapper<const Config> cfg; // explicit non-null borrow
    // still requires owner to outlive this object, but the intent is
    ↪  clearer
};
```

5.4.8 Alternative 7: indices/keys instead of pointers and iterators

Pointers and iterators into containers are often invalidated by mutation. Indices and keys can be safer when the container may reallocate.

Instead of returning an iterator into a container you might mutate

```
#include <vector>

std::size_t find_index(const std::vector<int>& v, int value) {
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == value) return i;
    }
    return v.size();
}
```

This avoids iterator invalidation hazards, especially across API boundaries.

5.4.9 Alternative 8: RAII guards for lifecycle and cleanup

Dangling issues often appear when background work outlives its owner. Use RAII to bind the lifecycle of work to a scope or object.

`std::jthread` (C++20) instead of detached threads

```
#include <thread>

void worker(std::stop_token st);

void run() {
    std::jthread t(worker); // joins on destruction
    // ...
```

```
} // safe shutdown of the thread here
```

This prevents the common bug where a detached thread accesses destroyed state.

5.4.10 Alternative 9: bounds-checked access in safety-critical paths

Unchecked indexing can produce out-of-bounds UB. For critical paths, prefer checked access or explicit validation at boundaries.

`at()` for checked access

```
#include <vector>

int safe_get(const std::vector<int>& v, std::size_t i) {
    return v.at(i); // throws on out-of-range
}
```

Use this strategically: it trades performance for correctness and diagnosability in unsafe contexts (e.g., external input parsing).

5.4.11 How to choose the right alternative (practical decision rules)

1. **If you create data:** return an owning type (value, `std::string`, container).
2. **If you need unique dynamic lifetime:** use `std::unique_ptr`.
3. **If lifetime must be shared:** use `std::shared_ptr` and design against cycles.
4. **If you accept a range:** use `std::span` (or iterators) and validate boundaries.
5. **If you accept a string:** use `std::string_view` for input; avoid returning it unless storage is guaranteed.

6. **If container mutation is expected:** prefer indices/keys over iterators/pointers.

7. **If background work exists:** bind lifetimes with RAII (`std::jthread`, guards), avoid detached access to locals.

5.4.12 Key takeaway

Dangling pointers and references are primarily a design problem: unclear ownership and hidden lifetimes. Modern C++ provides safer alternatives that make lifetimes explicit: `std::unique_ptr`, `std::shared_ptr`/`std::weak_ptr`, value returns, `std::span`, and disciplined view usage.
Use views for short-lived access. Use owners for storage and returned results. Bind lifetimes to scopes with RAII. This is the professional path to eliminating dangling bugs.

Chapter 6

Uninitialized and Invalid Memory

6.1 Default initialization traps

6.1.1 Why default initialization is a common source of bugs

Modern C++ provides strong abstractions, but it still allows you to create objects whose values are **not initialized**. This is one of the most common sources of subtle defects because:

- code may “work” in Debug builds and fail in Release builds,
- the bug can be silent (wrong results) rather than a crash,
- the language rules distinguish multiple forms of initialization that look similar in code,
- reading an indeterminate value is not just “garbage”; in many cases it is **undefined behavior**.

The professional rule:

> Never assume “default” means “zero”. It depends on the type and on the initialization form.

6.1.2 The three important forms: default, value, and zero initialization

C++ uses different initialization rules depending on syntax:

- **Default-initialization** (e.g., `T x;`) may leave fundamental types uninitialized.
- **Value-initialization** (e.g., `T x{};` or `T()` in many contexts) typically yields zero for fundamental types.
- **Zero-initialization** is part of what happens for objects with static storage duration and in some value-initialization cases.

Professionally, you must recognize these patterns instantly.

6.1.3 Trap 1: local fundamental types are not initialized

Bad: `int x;` is uninitialized

```
#include <iostream>

int main() {
    int x;                  // default-initialized -> indeterminate value
    std::cout << x << "\n"; // using x is not valid; can be undefined
    ↪  behavior
}
```

Correct: use brace initialization for zero

```
int main() {
```

```
    int x{};  // value-initialized -> zero
    int y = 0;
}
```

6.1.4 Trap 2: “new T” vs “new T()” vs “new T{}”

Dynamic allocation does not automatically initialize fundamental types.

Bad: `new int` leaves the int uninitialized

```
int* p = new int;    // *p is indeterminate
int  x = *p;         // using it is not valid
delete p;
```

Good: `new int()` or `new int{}` initializes

```
int* p1 = new int();  // value-initialized -> zero
int* p2 = new int{};  // value-initialized -> zero
delete p1;
delete p2;
```

Professional rule:

> Prefer `std::make_unique` and brace initialization; avoid raw `new`.

6.1.5 Trap 3: arrays and containers do not always zero their elements

Local arrays of fundamental types

```
int main() {
    int a[4];     // elements are uninitialized
    int b[4]{};   // elements are zero-initialized
}
```

`std::array` behaves like a struct

```
#include <array>

int main() {
    std::array<int, 4> a;    // elements are uninitialized
    std::array<int, 4> b{};  // elements are zero-initialized
}
```

`std::vector` default-constructs elements when you size it

```
#include <vector>

int main() {
    std::vector<int> v1(4);    // 4 elements value-initialized -> zero
    std::vector<int> v2;       // empty
    v2.resize(4);              // new elements value-initialized -> zero
}
```

Key distinction:

- `std::vector<int> v(4);` creates 4 elements and initializes them.
- `std::array<int,4> a;` creates 4 elements but does not initialize them unless you use `{}`.

6.1.6 Trap 4: user-defined types can still leave members uninitialized

C++ does not automatically initialize members unless you tell it to. If a class has a user-provided default constructor that does not initialize every member, some members can remain indeterminate.

Bad: default constructor forgets initialization

```
struct S {
    int x;
    S() {} // x is uninitialized
};

int main() {
    S s;
    (void)s.x; // using s.x is not valid
}
```

Professional fix: in-class member initializers or delegating constructors

```
struct S {
    int x = 0; // always initialized
};

struct T {
    int x;
    T() : x(0) {}
};
```

Professional rule:

> Prefer in-class member initializers for safety and maintainability.

6.1.7 Trap 5: = default and implicitly-generated constructors are not magic

`= default` requests the compiler-generated special member function. That function will perform the default behavior for each member:

- fundamental members remain uninitialized unless they have in-class initializers,
- class-type members call their default constructors.

```
struct S {
    int x;                // uninitialized unless explicitly initialized
    S() = default;        // does not zero x
};
```

6.1.8 Trap 6: padding bytes and comparing objects bytewise

Even when your fields are initialized, **padding bytes** in structs may be indeterminate. Reading padding as bytes (for hashing, serialization, memcmp) can produce unstable results and can be incorrect.

Bad: `memcmp` on a type with padding

```
#include <cstring>

struct P {
    char c;
    int  x;  // likely has padding after c
};

bool equal_bad(const P& a, const P& b) {
    return std::memcmp(&a, &b, sizeof(P)) == 0; // not a valid general
    ↪  equality test
}
```

Professional fixes:

- define logical equality field-by-field,

- for serialization/hashing, write explicit encoders for fields,
- if you truly need stable byte representation, use carefully designed standard-layout types and initialize all bytes explicitly (often via dedicated serialization buffers).

6.1.9 Trap 7: "invalid" values that are not merely uninitialized

Some memory reads are invalid even if "something is stored" there:

- reading an object through a pointer/reference after its lifetime ends,
- reading from a union member that is not active (except in allowed cases),
- accessing storage that does not currently contain an object of that type.

This chapter focuses on initialization, but professionals must remember:

> Correct initialization does not fix lifetime violations.

6.1.10 Professional defaults: patterns that prevent initialization bugs

Prefer brace initialization

```
int x{};                  // zero
double d{};               // 0.0
std::array<int, 4> a{};   // all zero
```

Use in-class member initializers

```
struct Config {
    int timeout_ms = 1000;
    bool enabled = true;
};
```

Prefer standard containers and RAII

Containers default-initialize their elements by well-defined rules and reduce raw memory handling.

Enable warnings and treat them seriously

Compilers commonly warn about uninitialized reads under optimization (especially with `-Wall -Wextra` style flags). These warnings should be treated as defects, not noise.

Use sanitizers and static analysis

- MemorySanitizer (MSan) can detect uses of uninitialized memory (where supported).
- UndefinedBehaviorSanitizer (UBSan) can catch some related undefined behaviors.
- Static analysis can flag missing member initialization and suspicious paths.

6.1.11 Quick reference: initialization syntax and outcome (common cases)

- `int x;` → indeterminate
- `int x{};` → zero
- `int* p = new int;` → `*p` indeterminate
- `int* p = new int{};` → `*p` zero
- `std::array<int,4> a;` → indeterminate elements
- `std::array<int,4> a{};` → zero elements
- `std::vector<int> v(4);` → 4 zeros

6.1.12 Key takeaway

Default initialization traps persist because C++ offers multiple initialization forms that look similar but have different semantics. Professional C++ safety requires:

- always initializing fundamental data,
- using brace initialization consistently,
- using in-class member initializers,
- avoiding bytewise operations on padded objects,
- and verifying with warnings, sanitizers, and analysis.

6.2 Indeterminate values

6.2.1 What an indeterminate value is (precise meaning)

In C++, an object has an **indeterminate value** when its stored bits do not represent a value that the program is allowed to read as a meaningful value of that type.
This happens most commonly when:

- an object of fundamental type (e.g., `int`, `double`, pointer types) was not initialized,
- storage was obtained but no object lifetime began for that type,
- an object's lifetime ended and the storage was later reused,
- padding bytes are present and were never initialized.

Professional mindset:

> An indeterminate value is not a “random number.” It is a value you are not allowed to use.

6.2.2 Why indeterminate values are dangerous

Using an indeterminate value typically results in undefined behavior (UB) or, at minimum, a program whose behavior is not reliable.
The danger is not only wrong output:

- the optimizer may assume certain cases never occur,
- the bug may disappear with logging or debugging,
- the symptom may appear far from the cause.

6.2.3 The most common source: uninitialized automatic variables

Example: reading an uninitialized variable

```
#include <iostream>

int main() {
    int x;                  // indeterminate
    if (x == 0) {           // using x is not valid
        std::cout << "zero\n";
    }
}
```

Professional fix

```
int main() {
    int x{};                // value-initialized -> 0
```

```
    if (x == 0) { /* ... */ }
}
```

6.2.4 Indeterminate does not always mean “UB in every context”

C++ makes important distinctions:

Reading object representation vs reading a typed value

There are limited cases where you can inspect raw bytes (object representation) without interpreting them as a typed value, typically via `unsigned char` or `std::byte`. However, using indeterminate bytes as if they were a valid `int`, `double`, pointer, or most other typed values is not valid.
This distinction matters in low-level code, but for safety-focused application code, the practical rule remains:

> Do not depend on uninitialized storage. Initialize what you use.

6.2.5 Trap 1: class types can still contain indeterminate members

User-defined types can carry indeterminate values when constructors do not initialize all members.

Example: constructor forgets member initialization

```
struct S {
    int x;
    S() {} // x indeterminate
};

int main() {
```

```
    S s;
    (void)s.x; // using x is not valid
}
```

Professional fix: in-class member initializers

```
struct S {
    int x = 0;
};
```

6.2.6 Trap 2: `new T` does not necessarily initialize `T`

Dynamic allocation can create indeterminate values when you allocate without value-initializing.

Example

```
int* p = new int;     // *p indeterminate
int  x = *p;          // using it is not valid
delete p;
```

Fix

```
int* p = new int{};   // *p == 0
delete p;
```

Professional rule:

> Avoid raw `new`. Prefer `std::make_unique<T>()` and brace initialization.

6.2.7 Trap 3: padding bytes and bytewise operations

Even when fields are initialized, padding bytes may remain indeterminate. This becomes a correctness and safety problem when you:

- compare objects with `memcmp`,
- hash objects by raw bytes,
- serialize objects by dumping `sizeof(T)` bytes.

Example: memcmp is not a general equality test

```
#include <cstring>

struct P {
    char c;
    int  x; // likely padding after c
};

bool equal_bad(const P& a, const P& b) {
    return std::memcmp(&a, &b, sizeof(P)) == 0; // can differ due to
    ↪  padding
}
```

Professional fixes:

- compare fields explicitly,
- define `operator==` by logical members,
- serialize/have stable hashes using explicit field encoding.

6.2.8 Trap 4: indeterminate values in control flow cause “unreproducible” failures

Even a single indeterminate value used in a branch can create unpredictable execution paths.

```
#include <vector>

int main() {
    int n;                  // indeterminate
    std::vector<int> v;
    if (n > 100) {          // invalid use
        v.resize(static_cast<std::size_t>(n)); // may allocate huge
        ↪ memory
    }
}
```

This can lead to massive allocations, exceptions, or later crashes, depending on whatever bits happened to be in `n`.

6.2.9 Indeterminate vs unspecified vs implementation-defined vs undefined behavior

These terms must not be confused:

- **Indeterminate value**: a value state you are not allowed to use as a meaningful typed value.
- **Unspecified behavior**: the implementation chooses among permitted behaviors.
- **Implementation-defined behavior**: the implementation documents what it does.
- **Undefined behavior**: no requirements; the program has violated the rules.

Indeterminate values often *lead to* undefined behavior when used.

6.2.10 Professional prevention strategy

1) Use consistent initialization style

- Prefer brace initialization: `T x{};`
- Use in-class member initializers for data members.
- Prefer constructors that fully initialize invariants.

2) Avoid raw storage unless you manage lifetime explicitly

If you must use raw storage (placement new, object pools), ensure you:

- begin lifetime with placement new,
- end lifetime with `std::destroy_at`,
- never read as `T` unless a live `T` is present.

3) Turn on warnings and treat them as defects

Most compilers can warn about uninitialized reads in optimized builds. These warnings should be treated as correctness bugs.

4) Use analysis and sanitizers

- **MemorySanitizer (MSan)**: detects uses of uninitialized memory (where supported).
- **Static analysis**: finds missing member initialization and suspicious paths.
- **UBSan/ASan**: catch related undefined behaviors and memory defects.

6.2.11 Key takeaway

Indeterminate values are not “harmless garbage.” They are values that the program is not allowed to treat as meaningful typed values. Using them undermines program correctness and can lead to undefined behavior.
Professional Modern C++ avoids indeterminate values by consistent initialization, RAII ownership, and continuous verification with compiler warnings, static analysis, and sanitizers.

6.3 Zero-initialization vs value-initialization

6.3.1 Why these terms matter

In safety-critical C++ code, many bugs start with a simple wrong assumption:

> “This variable will be zero by default.”

Whether you get zeros depends on **which initialization rules apply**. Two key concepts that look similar but are not the same are:

- **Zero-initialization**
- **Value-initialization**

Understanding the difference is essential for writing code that never reads indeterminate values.

6.3.2 Zero-initialization (what it means)

Zero-initialization means the object is initialized as follows:

- for scalar types (integers, floating point, pointers): set to zero (or null pointer for pointers),

- for arrays: each element is zero-initialized,
- for class/struct objects: each subobject is zero-initialized (as part of the overall rules).

Zero-initialization happens automatically in some important cases:

- objects with **static storage duration** (globals, namespace-scope variables, and function-local `static` objects) are zero-initialized before any dynamic initialization,
- as a component step of other initialization forms (notably value-initialization in many cases).

6.3.3 Value-initialization (what it means)

Value-initialization is an initialization form triggered by syntax such as:

- `T{}` (brace empty initialization),
- `T()` in many contexts,
- `new T()` and `new T{}`.

What value-initialization *does* depends on whether `T` is a class type or a non-class type:

- If `T` is a **non-class scalar type** (e.g., `int`), value-initialization results in **zero-initialization**.
- If `T` is a **class type**, value-initialization performs:
  - zero-initialization first, and then
  - calls the selected constructor (typically the default constructor).

Professional warning:

> Value-initialization does *not* guarantee that user-written constructors initialize every member.

6.3.4 The practical difference (most important cases)

Case 1: automatic local scalars

```
int a;    // default-initialized -> indeterminate
int b{};  // value-initialized -> zero
```

Case 2: globals and function-local statics

```
int g; // static storage -> zero-initialized

void f() {
    static int s; // static storage -> zero-initialized
}
```

Case 3: dynamic allocation

```
int* p = new int;   // default-initialized -> *p indeterminate
int* q = new int{}; // value-initialized -> *q == 0
delete p;
delete q;
```

Case 4: aggregates and arrays

```
int a1[4];    // elements indeterminate
int a2[4]{};  // elements zero-initialized

struct S { int x; double y; };
S s1;    // default-initialized -> x,y indeterminate (as automatic
↪  object)
S s2{};  // value-initialized -> zero-initializes members (aggregate
↪  initialization)
```

Case 5: `std::array` behaves like an aggregate

```
#include <array>

std::array<int, 4> x;   // elements indeterminate (automatic)
std::array<int, 4> y{}; // elements zero-initialized
```

6.3.5 The subtle class-type trap: user-provided default constructors

For class types, value-initialization includes a constructor call. If the constructor does not initialize everything, members can remain indeterminate.

Example: user-provided default constructor defeats your expectations

```
struct Bad {
    int x;
    Bad() {} // does not initialize x
};

int main() {
    Bad b{};      // value-initialization calls Bad::Bad()
    (void)b.x;    // x is indeterminate -> using it is not valid
}
```

Professional fix: in-class member initializers

```
struct Good {
    int x = 0;
};
```

6.3.6 How to force “all zeros” reliably (when you truly need it)

If you want guaranteed zero-initialization for members:

- use in-class member initializers,
- design constructors to fully initialize invariants,
- for aggregates, use `{}` initialization when appropriate.

```
struct Config {
    int timeout_ms = 0;
    bool enabled = false;
};
```

6.3.7 Why professionals prefer braces `{}`

Brace initialization is preferred in safety-focused code because:

- `T{}` clearly expresses “initialize”,
- it avoids many narrowing conversions,
- it provides consistent behavior across scalars, aggregates, and standard library types.

6.3.8 Quick reference table (most common patterns)

- `int x;` → indeterminate (automatic storage)
- `int x{};` → zero
- `static int x;` → zero (static storage)
- `new int;` → indeterminate

- `new int{};` → zero
- `std::array<int,4> a;` → indeterminate elements (automatic)
- `std::array<int,4> a{};` → zero elements

6.3.9 Professional rules to avoid initialization mistakes

1. Never rely on default initialization of fundamental types.
2. Prefer `T x{};` for local scalars and aggregates.
3. Use in-class member initializers for class members.
4. Avoid user-provided default constructors that do not fully initialize members.
5. For dynamic allocation, prefer `std::make_unique<T>()` and `std::make_shared<T>()`, and ensure the constructed object is fully initialized.
6. Treat compiler warnings about uninitialized use as correctness defects.

6.3.10 Key takeaway

Zero-initialization is a specific rule (all bits to zero in the sense of the language model) that happens for static storage and as a component step in other forms.
Value-initialization is a syntax-driven form (`T{}`, `T()`, `new T{}`) that often *results in* zero-initialization for scalars, but for class types it also calls constructors. If those constructors fail to initialize members, indeterminate values can still exist.
For safety-focused Modern C++ code, the professional solution is consistent brace initialization and constructors that fully initialize invariants.

Chapter 7

Buffer Overflows and Out-of-Bounds Access

7.1 Raw arrays vs containers

7.1.1 Why this comparison matters

Buffer overflows and out-of-bounds access are among the most destructive classes of defects in C++. They can cause:

- crashes and data corruption,
- undefined behavior that the compiler may optimize around,
- security vulnerabilities (information disclosure, memory corruption).

A major design decision that influences these risks is how you represent contiguous storage:

- **raw arrays** and raw pointers (e.g., `T a[N]`, `T*` + length),

- **standard containers** and views (e.g., `std::array`, `std::vector`, `std::span`).

Professional guideline:

> Use raw arrays only when you truly need their low-level form. Use containers and spans by default because they carry size and support safer interfaces.

7.1.2 Raw arrays: what they are and why they are risky

Key properties

- Fixed-size raw arrays (`T a[N]`) do not know their size once they decay to a pointer.
- Raw pointers (`T*`) do not carry bounds information.
- Many C-style APIs expect `T*` and a separate length, which can disagree.

These properties make it easy to write code that compiles and runs but accesses memory out of bounds.

Typical overflow: wrong index

```
void bad() {
    int a[4] = {1,2,3,4};
    a[4] = 9; // out-of-bounds write (UB): valid indices are 0..3
}
```

Typical overflow: pointer + length mismatch

```
#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]; // UB if n is larger than the real array length
    }
    return s;
}
```

Here, `n` is a separate value. If it is wrong, the compiler cannot protect you.

Raw arrays and decay: losing size information

```
#include <cstddef>

void takes_ptr(const int* p);

void demo() {
    int a[10]{};
    takes_ptr(a); // a decays to int*, size is lost
}
```

Once it decays, the callee cannot know the correct bounds unless you pass them explicitly.

7.1.3 Standard containers: why they are safer

Standard containers are RAII-managed and have explicit size:

- `std::array<T,N>`: fixed size known at compile time, stores size in the type.
- `std::vector<T>`: dynamic size, stores its size and capacity.
- `std::string`: manages contiguous characters with length tracking.

These do not automatically prevent out-of-bounds (unchecked operator[] still exists), but they make safe patterns easier and unsafe patterns more obvious.

`std::array` keeps size in the type

```
#include <array>

void good_array() {
    std::array<int, 4> a{1,2,3,4};
    // a.size() is always correct
}
```

`std::vector` owns dynamic storage safely

```
#include <vector>

void good_vector() {
    std::vector<int> v = {1,2,3,4};
    v.push_back(5); // grows safely (may reallocate, but ownership
    ↪  remains correct)
}
```

7.1.4 Bounds checking: the real difference in safety behavior

The crucial detail is how you access elements:

Unchecked access exists everywhere

- `a[i]` on raw arrays is unchecked.
- `v[i]` on `std::vector` is also unchecked.

So containers are not “magically safe” if you still write incorrect indices.

Checked access is available in containers

Containers provide `at()`, which checks bounds and typically throws `std::out_of_range`.

```
#include <vector>

int safe_get(const std::vector<int>& v, std::size_t i) {
    return v.at(i); // throws if i >= v.size()
}
```

This is extremely valuable at trust boundaries (parsing, network input, file formats), where correctness matters more than micro-optimization.
Professional guidance:

> Use unchecked access only when indices are proven correct by construction. Use checked access at trust boundaries.

7.1.5 The professional sweet spot: containers for ownership, `std::span` for parameters

A powerful modern pattern is:

- store data in an owning container (`std::vector`, `std::array`, `std::string`),
- pass data using `std::span` to represent a contiguous view with a size.

Prefer `std::span` over pointer + length

```
#include <span>
#include <cstddef>
```

```
int sum_good(std::span<const int> s) {
    int total = 0;
    for (int x : s) total += x;
    return total;
}
```

Now the size and pointer travel together in a single value.

Works with raw arrays and containers

```
#include <array>
#include <vector>
#include <span>

int sum_good(std::span<const int> s);

void demo() {
    int raw[4] = {1,2,3,4};
    std::array<int,4> a = {1,2,3,4};
    std::vector<int> v = {1,2,3,4};

    (void)sum_good(raw);
    (void)sum_good(a);
    (void)sum_good(v);
}
```

7.1.6 When raw arrays are still appropriate

Raw arrays remain valid in specific contexts:

- ABI and C interop where the API demands `T*` buffers,
- low-level embedded/freestanding code with restricted library availability,

- fixed-layout data structures where you need exact layout and zero overhead,
- performance-critical hot loops where you control and prove bounds carefully.

Even in these cases, professionals reduce risk by:

- wrapping raw buffers in `std::span` at function boundaries,
- centralizing boundary checks,
- avoiding manual "length" variables scattered through code.

7.1.7 Common pitfalls and how containers help

Pitfall 1: manual size tracking goes out of sync

```
#include <cstddef>

struct Buf {
    char* p;
    std::size_t n;
};

void bad(Buf b) {
    // if n does not match real allocation, overflow occurs
    for (std::size_t i = 0; i < b.n; ++i) b.p[i] = 0;
}
```

Container fix: store the buffer in `std::vector<char>` or `std::string` so the size is not a second, separately-maintained truth.

Pitfall 2: `sizeof` confusion

```cpp
#include <cstddef>

void bad_size(const int* p) {
    // sizeof(p) is size of pointer, not size of array
    std::size_t n = sizeof(p) / sizeof(p[0]);
    (void)n;
}
```

Using `std::array` avoids this trap because `a.size()` is correct.

Pitfall 3: buffer copying without limits

```cpp
#include <cstring>

void bad_copy(char* dst, const char* src) {
    std::strcpy(dst, src); // unsafe: no bound check
}
```

Professional alternatives:

- use sized operations with explicit limits (and validate lengths),
- use `std::string` / `std::vector<char>` and append safely,
- represent buffers as `std::span<char>` and copy only within the span length.

7.1.8 Performance note: containers are not inherently slower

A common misconception is that containers are slow. In reality:

- `std::array` is essentially a raw array with a safer interface.

- `std::vector` provides contiguous storage and is highly optimized; access through `operator[]` is typically the same as raw pointer indexing.
- the overhead comes from dynamic growth and allocation decisions, not from basic indexing.

Professional approach:

> Use containers for safety and clarity. Optimize capacity management (`reserve`), allocation strategy, and access patterns when performance is critical.

7.1.9 Professional checklist

1. Store buffers in `std::array` or `std::vector` rather than raw arrays/pointers.
2. Pass buffers as `std::span` (or `std::string_view` for strings) instead of pointer + length.
3. Use `at()` or explicit checks at trust boundaries.
4. Avoid C-style unbounded copy functions; prefer sized operations and validated lengths.
5. Know container invalidation rules when taking pointers/iterators/views into containers.

7.1.10 Key takeaway

Raw arrays are powerful but fragile: bounds are not carried with the data, so mistakes are easy and often catastrophic. Standard containers provide ownership and size, enabling safer APIs and simpler reasoning. The most professional modern pattern is:

> Containers own; spans view.

7.2 Safer indexing strategies

7.2.1 Why indexing is a high-risk operation

Most buffer overflows in C++ are not caused by exotic features. They are caused by ordinary indexing that is slightly wrong:

- off-by-one errors,
- negative indices converted to huge unsigned values,
- mismatched sizes after conversions,
- stale indices after container mutation,
- assumptions about layout and length that are not enforced.

Professional rule:

> Indexing must be either bounds-checked or bounds-proven by construction.

This section presents strategies that make bounds explicit, enforceable, and testable.

7.2.2 Strategy 1: prefer range-based loops over manual indexing

If you do not need the index, do not create it. Range-based loops remove the most common overflow class: wrong index arithmetic.

Instead of

```
#include <vector>

int sum_bad(const std::vector<int>& v) {
```

```
    int s = 0;
    for (std::size_t i = 0; i <= v.size(); ++i) { // off-by-one
        s += v[i];
    }
    return s;
}
```

Prefer

```
#include <vector>

int sum_good(const std::vector<int>& v) {
    int s = 0;
    for (int x : v) s += x;
    return s;
}
```

7.2.3 Strategy 2: use iterators and algorithms where possible

Standard algorithms express intent and avoid error-prone bounds logic.

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

int sum_algo(const std::vector<int>& v) {
    return std::accumulate(v.begin(), v.end(), 0);
}
```

7.2.4 Strategy 3: use `at()` at trust boundaries

`operator[]` is unchecked for `std::vector`, `std::array`, and `std::string`. When indices come from external input (files, network, user input), prefer checked access.

Example: checked access

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

int safe_read(const std::vector<int>& v, std::size_t i) {
    return v.at(i); // throws std::out_of_range if i >= v.size()
}
```

Professional use:

- parsing and validation layers,
- protocol decoding,
- file format readers,
- any code where invalid indices are plausible.

7.2.5 Strategy 4: validate indices explicitly and centralize the checks

If exceptions are not desired, validate and handle errors explicitly.

Example: explicit check, no exceptions

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

std::optional<int> try_read(const std::vector<int>& v, std::size_t i) {
    if (i >= v.size()) return std::nullopt;
    return v[i];
}
```

Professional rule:

> Prefer one clear boundary check over repeated scattered checks.

7.2.6 Strategy 5: avoid signed/unsigned traps (`int` vs `size_t`)

A classic overflow bug comes from mixing signed indices with unsigned sizes.

Trap: negative index becomes a huge unsigned value

```
#include <vector>

int bad(const std::vector<int>& v, int i) {
    // If i is negative, i is converted to a huge size_t in the
    ↪  comparison.
    if (i < v.size()) {
        return v[i]; // UB if i is negative (indexing with negative int)
    }
    return 0;
}
```

Professional fixes

- Prefer `std::size_t` (or `std::vector<int>::size_type`) for indices that must be non-negative.
- If indices are inherently signed (e.g., offsets, deltas), validate sign first.

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

int good(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];
}
```

7.2.7 Strategy 6: use `std::span` to bind pointer and length together

Many overflow bugs come from passing `T*` and a separate length. `std::span` makes the size part of the type.

```
#include <span>
#include <cstddef>
#include <optional>

std::optional<int> span_read(std::span<const int> s, std::size_t i) {
    if (i >= s.size()) return std::nullopt;
    return s[i];
}
```

7.2.8 Strategy 7: prefer "two-iterator" ranges over indexing for slicing

Instead of manual index arithmetic when creating subranges, use iterator/range slicing patterns.

Example: safe slice with iterators

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

std::vector<int> slice_copy(const std::vector<int>& v, std::size_t from,
↪   std::size_t to) {
    if (from > to) return {};
```

```
    if (to > v.size()) return {};
    return std::vector<int>(v.begin() +
    ↪  static_cast<std::ptrdiff_t>(from),
                            v.begin() + static_cast<std::ptrdiff_t>(to));
}
```

This makes bounds explicit and avoids off-by-one confusion.

7.2.9 Strategy 8: use `std::ssize` and `std::ptrdiff_t` for signed counts

When you need signed counts (differences between indices), use a signed type intentionally. C++20 provides `std::ssize(container)` to obtain a signed size.

```
#include <vector>
#include <iterator> // std::ssize

int distance_from_end(const std::vector<int>& v, int i) {
    // signed size avoids accidental wrap-around when working with
    ↪  negative values
    auto n = std::ssize(v);
    return n - i;
}
```

Professional note:

- Use unsigned for quantities that cannot be negative.
- Use signed for offsets and differences.
- Convert only after validating ranges.

7.2.10 Strategy 9: treat index arithmetic as a proof obligation

Any manual indexing arithmetic is a local "proof" that you are staying in bounds. Professional code makes that proof obvious by:

- naming invariants,
- validating preconditions once,
- avoiding repeated ad-hoc arithmetic.

Example: precondition validation + safe loop bounds

```
#include <span>
#include <cstddef>
#include <optional>

std::optional<int> dot(std::span<const int> a, std::span<const int> b) {
    if (a.size() != b.size()) return std::nullopt;

    int s = 0;
    for (std::size_t i = 0; i < a.size(); ++i) {
        s += a[i] * b[i];
    }
    return s;
}
```

Here the single equality check establishes the invariant that both spans are valid for all indices.

7.2.11 Strategy 10: rely on tooling for verification in critical code

Even with best practices, indexing mistakes happen. Professionals verify with:

- AddressSanitizer (catches many out-of-bounds reads/writes),
- UndefinedBehaviorSanitizer (catches many UB categories),
- fuzzing at input boundaries,
- static analysis and compiler warnings.

The key is to make these tools effective by writing code that has clear boundaries and invariants.

7.2.12 Professional checklist

1. Avoid indexing when a range-based loop or algorithm expresses intent.
2. Use `at()` or explicit checks when indices can be invalid (trust boundaries).
3. Do not mix signed indices with unsigned sizes without careful validation.
4. Pass contiguous ranges as `std::span` (not pointer + length).
5. Validate preconditions once and use that invariant to justify safe indexing.
6. Keep slicing logic bounds-checked and avoid off-by-one mistakes.
7. Verify with sanitizers and fuzz tests for any code that processes external input.

7.2.13 Key takeaway

Safer indexing is not one trick. It is a discipline:

- reduce indexing with ranges and algorithms,
- make bounds explicit (`span`, preconditions),

- use checked access when inputs are untrusted,
- avoid signed/unsigned traps,
- and continuously verify with sanitizers and tests.

7.3 `std::span` as a modern solution

7.3.1 Why `std::span` exists

A large fraction of buffer overflows in C++ comes from APIs that pass a raw pointer plus a separate length (or worse: no length at all). The pointer and the length can silently disagree, and the callee has no reliable way to validate the bounds.
`std::span` (standardized in C++20) is the modern, standard way to express:

> "a non-owning view of a contiguous sequence of `T` with a known length."

This solves the most common API-level safety problem:

- **raw pointer alone** carries no bounds,
- **pointer + length** can get out of sync,
- **`std::span`** keeps pointer and length together.

7.3.2 What `std::span` is (and what it is not)

Span is a view, not an owner

A span does **not** allocate, does **not** free, and does **not** extend lifetime. It only points to storage owned elsewhere.

```cpp
#include <span>
#include <vector>

void f(std::span<int> s) {
    // s does not own memory; it only views it.
}

void demo() {
    std::vector<int> v{1,2,3};
    f(v); // span views v's contiguous storage
} // if a span outlived v, it would dangle
```

Professional rule:

> Containers own; spans view. Never store a span unless the owner is guaranteed to outlive it.

Contiguous only

A span can only refer to contiguous memory. That includes:

- C arrays (`T[N]`),
- `std::array<T,N>`,
- `std::vector<T>` (and other contiguous containers),
- raw contiguous buffers (`T*` + length).

It cannot represent non-contiguous structures (linked lists, trees).

7.3.3 Static extent vs dynamic extent

`std::span<T, Extent>` has an **extent**:

- **static extent**: `Extent` is a compile-time constant (e.g., `span<int, 16>`)
- **dynamic extent**: `Extent` is `std::dynamic_extent` and size is stored at runtime

Most APIs should use dynamic extent because it accepts any length:

```
#include <span>

void process(std::span<const int> s); // flexible API
```

Static extent is useful when a fixed length is part of the contract:

```
#include <span>
#include <cstdint>

// Exactly 16 bytes (e.g., UUID).
void parse_uuid(std::span<const std::uint8_t, 16> bytes);
```

7.3.4 Span as an API boundary tool

Replacing pointer + size parameters

```
#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];
    return s;
}
```

Better:

```
#include <span>

int sum_good(std::span<const int> s) {
    int total = 0;
    for (int x : s) total += x;
    return total;
}
```

This does not automatically bounds-check indexing, but it prevents a large class of mistakes: the caller can no longer accidentally pass a pointer from one buffer and a length from another.

Span integrates with common call sites

```
#include <span>
#include <array>
#include <vector>

int sum_good(std::span<const int> s);

void demo() {
    int raw[3] = {1,2,3};
    std::array<int,3> a = {1,2,3};
    std::vector<int> v = {1,2,3};

    (void)sum_good(raw);
    (void)sum_good(a);
    (void)sum_good(v);
}
```

7.3.5 Bounds safety: what span guarantees and what it does not

Span always knows its size

A span always provides:

- `size()`, `empty()`,
- iterators,
- `first()`, `last()`, `subspan()` for slicing.

Span indexing is traditionally unchecked

In C++20/C++23, `span::operator[]` is unchecked. Safe code must still validate indices, or structure loops so bounds are proven.

```
#include <span>
#include <cstddef>
#include <optional>

std::optional<int> read_checked(std::span<const int> s, std::size_t i) {
    if (i >= s.size()) return std::nullopt;
    return s[i]; // now bounds-proven
}
```

C++26 adds `span::at()`

In C++26, `std::span` gains `at(pos)` which performs bounds checking and throws `std::out_of_range` when out of bounds.

```
#include <span>
#include <cstddef>

int read_at(std::span<const int> s, std::size_t i) {
```

```
    return s.at(i); // bounds checked (C++26)
}
```

Professional guidance:

- use checked access (`at` in C++26, or explicit checks) at trust boundaries,
- use unchecked access only when indices are proven correct by invariant.

7.3.6 Safe slicing with `first`, `last`, and `subspan`

Slicing is a frequent source of off-by-one errors when done with pointer arithmetic. Span provides slicing utilities that keep the size consistent.

```
#include <span>
#include <cstddef>
#include <optional>

std::optional<std::span<const std::byte>> payload(std::span<const
↪  std::byte> packet) {
    // Example format: [4-byte header][payload...]
    constexpr std::size_t header = 4;
    if (packet.size() < header) return std::nullopt;

    return packet.subspan(header); // view of the remaining bytes
}
```

This pattern is safe because it checks the precondition once, then uses `subspan` to create a correctly-sized view.

7.3.7 Interoperability: span with C and OS APIs

Many system APIs still require `T*` and a length. Span is compatible:

- `s.data()` gives `T*`,
- `s.size()` gives the element count.

```
#include <span>
#include <cstddef>

extern "C" int c_api_write(const void* data, std::size_t bytes);

int write_bytes(std::span<const std::byte> s) {
    return c_api_write(s.data(), s.size());
}
```

Span improves safety at the C++ boundary while still interoperating cleanly.

7.3.8 Byte views with `std::as_bytes` and `std::as_writable_bytes`

Span provides a safe, explicit way to view object representations as bytes.

```
#include <span>
#include <cstddef>
#include <cstdint>

struct Header {
    std::uint32_t magic;
    std::uint16_t version;
    std::uint16_t flags;
```

```
};

std::span<const std::byte> header_bytes(std::span<const Header> h) {
    return std::as_bytes(h); // view bytes of the header array
}
```

Professional cautions:

- a byte view is still non-owning and follows the original lifetime,
- do not assume object byte layout is portable across compilers/ABIs for serialization formats; define wire formats explicitly when portability matters.

7.3.9 Lifetime and invalidation rules (what professionals watch for)

A span becomes invalid when the underlying storage becomes invalid, for example:

- the owning container is destroyed,
- the owning container reallocates (e.g., `std::vector` growth),
- the span views a temporary or local array that goes out of scope.

Common trap:

```
#include <span>
#include <vector>

std::span<int> bad_span() {
    std::vector<int> v = {1,2,3};
    return v; // dangling span: v is destroyed on return
}
```

Professional fix: return an owner, not a view:

```cpp
#include <vector>

std::vector<int> good_owner() {
    return {1,2,3};
}
```

7.3.10 Using `std::span` in a C++11–C++17 codebase

This handbook targets C++11 through C++26. `std::span` is standard starting in C++20. For earlier standards, the professional approach is:

- adopt a well-known span type from a widely used guideline-oriented library,
- or use a high-quality span backport,
- keep the API shape (`span`-based) so the migration to `std::span` is mechanical.

The key design insight is stable across all versions:

> Represent contiguous ranges as one object (pointer + size), not as loosely coupled parameters.

7.3.11 Professional checklist

1. Use `std::span` for parameters that represent contiguous buffers.
2. Store data in owning containers (`std::vector`, `std::array`, `std::string`); do not store spans unless lifetime is guaranteed.
3. Validate sizes once, then slice with `subspan`/`first`/`last`.
4. For indexing: use explicit bounds checks (and `span::at` in C++26 when available).

5. For byte-level processing, prefer `std::as_bytes`/`std::as_writable_bytes` instead of ad-hoc casts.

7.3.12 Key takeaway

`std::span` is a modern, standard solution to a classic C++ safety problem: it turns "pointer + length" into a single, explicit, non-owning range object. It improves API correctness, reduces overflow risk at boundaries, and composes naturally with containers, algorithms, and slicing.
It does not replace ownership or lifetime management. It complements them:

> Owners manage lifetime; spans communicate safe bounds.

Part III

Ownership, RAII, and Resource Management

Chapter 8

RAII: The Core of Safe C++

8.1 Philosophy and Guarantees

8.1.1 RAII is the Default Safety Model in C++

Resource Acquisition Is Initialization (RAII) is not a “pattern” you optionally apply. It is the default *safety model* of C++: the language gives objects well-defined construction and destruction points, and RAII binds every important resource to those points so that cleanup happens automatically.
In RAII, a resource is treated as a **class invariant**:

- After a successful constructor finishes, the object *owns* a resource (or holds it in a valid empty state).
- When the object is destroyed, it *releases* that resource.
- No other code path is responsible for cleanup.

This is the central idea behind safe C++ codebases: **If the lifetime is correct, the resource management is correct.**

8.1.2 What Counts as a "Resource"

In this chapter, a *resource* means anything that must be acquired and later released:

- dynamic memory (heap allocations)
- file descriptors, file handles, memory-mapped regions
- sockets, pipes, OS handles
- mutex locks, read/write locks, semaphores
- thread joins, stop tokens, background tasks that must be finished
- temporary state changes that must be reverted ("restore on exit")

The point of RAII is to turn these into **objects with deterministic cleanup**.

8.1.3 The Core Guarantees RAII Relies On

RAII works because C++ provides strong lifetime and unwinding guarantees.

Deterministic destruction at scope exit

For an object with automatic storage duration (a normal local variable), C++ guarantees destruction when control leaves the scope:

- by normal return
- by `break`/`continue`/`goto` leaving the scope

- by exception propagation (stack unwinding) that leaves the scope

This makes cleanup **path-independent**: you do not have to remember all exit paths.

Reverse-order cleanup

When multiple objects are created in a scope, they are destroyed in the reverse order of construction. This is crucial when resources depend on each other.

Constructor failure cleans up fully-constructed subobjects

If a constructor throws, C++ guarantees that any fully-constructed base classes and data members are destroyed automatically, in reverse construction order. This means "half-built" objects do not leak the resources that were successfully acquired before the failure.

Destruction of temporaries is deterministic at full-expression boundaries

Temporary objects are destroyed at well-defined points (typically at the end of the full-expression), which lets RAII protect short critical sections and single calls.

8.1.4 RAII Guarantees You Actually Get

It is important to separate what RAII *guarantees* from what it *suggests*.

Leak freedom (under a precise condition)

RAII gives you:

> No resource leaks *as long as the owning objects are destroyed*.

This condition is not a weakness; it is the engineering contract: **do not leak owning objects**. In practice, you enforce that contract with local variables and standard smart pointers.

Exception safety by construction

RAII is one of the main reasons idiomatic C++ can be strongly exception-safe:

- Resources acquired in constructors are released in destructors.
- Unwinding calls destructors on the way out.
- Cleanup code is centralized and automatic.

This turns many “cleanup problems” into normal object design.

Local reasoning and composability

RAII enables local reasoning:

- If you see a local RAII object, you know when the resource is acquired and released.
- Composing multiple RAII objects composes cleanup automatically.

This is the opposite of “manual cleanup,” which scales poorly as control flow grows.

Concurrency safety for ownership-based synchronization

RAII is foundational for safe synchronization:

- a lock is acquired in a constructor
- the lock is released in a destructor

This makes it hard to forget unlocking on early returns or exceptions.

```cpp
#include <mutex>

void update_shared_state(std::mutex& m, int& x) {
    std::lock_guard<std::mutex> lock(m); // lock acquired here
    ++x;                                 // protected work
}                                        // lock released here (always)
```

8.1.5 RAII and the "Commit or Roll Back" Mindset

A professional RAII design is guided by a simple rule:

> Either construction succeeds and the object is valid, or construction fails and everything acquired so far is cleaned up.

That leads to two best practices:

- perform acquisition in constructors (or factory functions that return a fully formed RAII object)
- avoid "two-phase initialization" that requires a separate `init()` call

Example: resource acquisition must be exception-safe

```cpp
#include <cstdio>
#include <stdexcept>

class File {
    std::FILE* f = nullptr;

public:
    explicit File(const char* path, const char* mode) :
    ↪  f(std::fopen(path, mode)) {
```

```
        if (!f) throw std::runtime_error("fopen failed");
    }

    File(const File&) = delete;
    File& operator=(const File&) = delete;

    File(File&& other) noexcept : f(other.f) { other.f = nullptr; }
    File& operator=(File&& other) noexcept {
        if (this != &other) {
            if (f) std::fclose(f);
            f = other.f;
            other.f = nullptr;
        }
        return *this;
    }

    ~File() noexcept {
        if (f) std::fclose(f);
    }

    std::FILE* get() const noexcept { return f; }
};
```

Key points:

- If `fopen` fails, the constructor throws and no `File` object is created.
- If the constructor succeeds, cleanup is guaranteed by `~File()` on scope exit.
- The destructor is `noexcept` to keep failure handling predictable.

8.1.6 The "Noexcept Destructor" Rule and Its Safety Consequences

For RAII to be reliable, destruction must be safe under all exit paths. In particular:

- A destructor should not emit exceptions as part of normal cleanup.
- If an exception escapes a destructor during stack unwinding, the program calls `std::terminate`.

Therefore, RAII cleanup code is typically designed to be:

- **non-throwing** (preferred), or
- **error-reporting via side channels** (logging, error codes stored elsewhere), or
- **explicitly handled by a separate *close/commit* function** that can fail, while the destructor remains best-effort and non-throwing.

Example: explicit close + non-throwing destructor

```
#include <cstdio>
#include <system_error>

class OutputFile {
    std::FILE* f = nullptr;

public:
    explicit OutputFile(const char* path) : f(std::fopen(path, "wb")) {
        if (!f) throw std::system_error(errno, std::generic_category(),
        ↪  "fopen");
    }

    // May fail; caller must handle it explicitly.
```

```
    void flush_or_throw() {
        if (std::fflush(f) != 0) {
            throw std::system_error(errno, std::generic_category(),
             ↪  "fflush");
        }
    }

    ~OutputFile() noexcept {
        if (f) std::fclose(f); // best-effort cleanup; do not throw
    }
};
```

The philosophy is simple: **operations that can fail should fail in normal functions, not in destructors.**

8.1.7 Zero-Overhead Philosophy

RAII matches the C++ "pay only for what you use" mindset:

- There is no garbage collector.
- There is no runtime "finally mechanism" required by the language.
- Cleanup code is emitted where needed, and only for objects you actually create.

In optimized builds, well-designed RAII wrappers often compile down to the same code you would write manually—but with dramatically stronger correctness properties.

8.1.8 What RAII Does *Not* Guarantee

RAII is powerful, but it is not magic. Common misconceptions cause real bugs.

RAII does not run if you bypass normal destruction

Destructors may not run in scenarios such as:

- abnormal termination (e.g., `std::terminate`, `abort`, `_Exit`)
- intentionally skipping unwinding (some termination paths)
- leaked objects (because their destructors are never reached)

In safety-critical code, treat "must run cleanup" as a design requirement and avoid termination paths that bypass destructors.

RAII cannot fix incorrect ownership graphs

RAII relies on correct ownership:

- raw owning pointers can be leaked, double-deleted, or freed too early
- cyclic graphs built from `std::shared_ptr` leak unless cycles are broken (e.g., with `std::weak_ptr`)

RAII does not prevent data races

RAII can guarantee that locks are released, but it cannot guarantee that the lock was the right lock, or that your concurrency design avoids races and deadlocks. RAII is a tool in the concurrency story, not the whole story.

8.1.9 Professional Checklist: What "Good RAII" Looks Like

A strong RAII type typically satisfies these properties:

- **Single clear owner** of the resource (or explicitly shared ownership).
- **Acquisition in the constructor** (or in a factory that returns a fully formed RAII object).
- **Release in the destructor** and the destructor is **`noexcept`**.
- **Non-copyable if unique ownership**; movable if transfer is meaningful.
- **Invariant is always true** after successful construction.
- **No manual `new/delete`** in user code; use standard smart pointers.

8.1.10 Summary

RAII is the core of safe C++ because it turns resource management into object lifetime management, and C++ makes object lifetimes deterministic.
The key guarantees are:

- deterministic destruction at scope exit
- reverse-order cleanup
- automatic cleanup of fully-constructed subobjects if construction fails
- predictable cleanup during exception propagation

Used consistently, RAII transforms cleanup from “a control-flow problem” into “a type design problem”, which is the most scalable way to build robust systems in Modern C++.

8.2 RAII beyond memory

8.2.1 The Core Idea: "Resource" Means Much More Than Heap Memory

In professional C++ systems, the most damaging leaks are often *not* memory leaks. They are leaks of operating-system and runtime resources that are scarce, global, and failure-prone. A **resource** is anything that must be acquired and later released (explicitly or implicitly), such as:

- file descriptors / file handles
- sockets, pipes, named handles
- mutexes, read/write locks, semaphores
- threads that must be joined (or stopped safely)
- memory-mapped regions
- temporary directories and files
- "state changes" that must be reverted (e.g., flags, modes, masks)

RAII turns every one of these into the same engineering model: **acquire in construction, release in destruction, never rely on remembering cleanup.**

8.2.2 File Ownership: Streams and "Handle RAII"

`std::fstream` as a built-in RAII type

Standard file streams are RAII: the file is owned by the stream object and is closed when the stream is destroyed. This is ideal for most portable file I/O.

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

std::string read_first_line(const std::string& path) {
    std::ifstream in(path);          // opens here
    std::string line;
    std::getline(in, line);
    return line;                     // closes here (scope exit), even on
    ↪  exception
}
```

C handles with `std::unique_ptr` + custom deleter

Sometimes you must work with C APIs (`FILE*`, `DIR*`, `sqlite3*`, OS handles). You can still get strong RAII by using `std::unique_ptr` with a deleter.

```
#include <cstdio>
#include <memory>
#include <stdexcept>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f);
    }
};

using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file(const char* path, const char* mode) {
    if (auto* f = std::fopen(path, mode)) {
        return unique_file{f};
    }
```

```
    throw std::runtime_error("fopen failed");
}

void write_hello(const char* path) {
    auto f = open_file(path, "wb");
    std::fputs("hello\n", f.get());
} // fclose happens automatically
```

8.2.3 Lock Ownership: RAII for Concurrency Correctness

RAII is the safest way to manage locks because it is **path-independent**: it releases the lock on every exit path (return, exception, early exit).

`std::lock_guard`: simplest correct lock

```
#include <mutex>

void increment(std::mutex& m, int& x) {
    std::lock_guard<std::mutex> g(m);  // lock acquired
    ++x;
}                                      // lock released (always)
```

`std::unique_lock`: flexible lock ownership

Use `std::unique_lock` when you need deferred locking, timed locking, unlocking before scope end, or using `std::condition_variable`.

```
#include <condition_variable>
#include <mutex>

void wait_for_ready(std::mutex& m, std::condition_variable& cv, bool&
↪  ready) {
```

```
    std::unique_lock<std::mutex> lk(m);       // lock acquired
    cv.wait(lk, [&]{ return ready; });        // wait temporarily releases
    ↪ and re-acquires lock
}                                             // lock released here
```

`std::scoped_lock`: multiple mutexes safely

When you must lock multiple mutexes together, prefer `std::scoped_lock` (C++17+) to reduce deadlock risk and to keep ownership scoped.

```
#include <mutex>

struct Pair {
    std::mutex a;
    std::mutex b;
    int x = 0;
    int y = 0;
};

void safe_update(Pair& p) {
    std::scoped_lock lock(p.a, p.b); // locks both for this scope
    ++p.x;
    ++p.y;
}
```

8.2.4 Thread Lifetime: RAII for Joining and Cooperative Stop

Threads are not “fire-and-forget” resources. A joinable thread is an owned responsibility: you must define how it finishes.

`std::thread` requires an explicit policy

If a `std::thread` is still joinable when its destructor runs, the program terminates. This is a lifetime rule, not a style preference.

```
#include <thread>

void example() {
    std::thread t([]{ /* work */ });

    // Professional policy: always join or detach intentionally.
    t.join();
}
```

`std::jthread`: RAII join + stop token (C++20)

`std::jthread` is an RAII thread: it automatically joins in its destructor. It also integrates cooperative stopping through `std::stop_token`.

```
#include <chrono>
#include <thread>

void worker(std::stop_token st) {
    while (!st.stop_requested()) {
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        // do periodic work
    }
}

void run_service() {
    std::jthread t(worker); // starts thread, will request stop + join on
    ↪ destruction
} // safe shutdown on scope exit
```

RAII callbacks for stop: `std::stop_callback`

Sometimes you want a cleanup action to run when stop is requested.
`std::stop_callback` is an RAII object that registers a callback for a given `std::stop_token` and unregisters it on destruction.

```
#include <thread>
#include <iostream>

void example(std::stop_token st) {
    std::stop_callback cb(st, []{
        std::cerr << "Stop requested\n";
    });

    // work...
}
```

8.2.5 Temporary Files and Directories: RAII for Cleanup and Security

Temporary artifacts are resources too: they consume disk space, can expose sensitive data, and can break later runs.
A professional approach is to wrap the whole lifecycle in a single type.

A small RAII temporary directory

```
#include <filesystem>
#include <stdexcept>

class TempDir {
    std::filesystem::path dir_;

public:
```

```
    TempDir() {
        auto base = std::filesystem::temp_directory_path();
        dir_ = base / std::filesystem::path("safety_tmp_%%%%-%%%%-%%%%");
        dir_ = std::filesystem::unique_path(dir_);
        std::filesystem::create_directories(dir_);
    }

    TempDir(const TempDir&) = delete;
    TempDir& operator=(const TempDir&) = delete;

    TempDir(TempDir&&) = delete;
    TempDir& operator=(TempDir&&) = delete;

    ~TempDir() noexcept {
        std::error_code ec;
        std::filesystem::remove_all(dir_, ec); // best-effort; no throw
        ↪  in destructor
    }

    const std::filesystem::path& path() const noexcept { return dir_; }
};
```

Usage:

```
#include <fstream>

void pipeline_stage() {
    TempDir tmp;
    std::ofstream out(tmp.path() / "stage.txt");
    out << "intermediate data\n";
} // directory is removed automatically
```

Design notes (professional and intentional):

- Destructors should not throw; cleanup is best-effort.
- If deletion failure matters, provide an explicit `cleanup()` that reports errors.

8.2.6 Memory-Mapped Regions: RAII for `mmap`/`munmap`-style APIs

Many OS APIs return a raw handle or pointer that must be released with a matching call. This is exactly what RAII is for: put the matching release in the destructor.

A generic "handle + deleter" wrapper

The simplest universal RAII design is: store the handle, store the cleanup function, release in the destructor, make it movable, and forbid copying.

```
#include <utility>

template <class Handle, class Deleter>
class unique_handle {
    Handle h{};
    Deleter d{};
    bool engaged = false;

public:
    unique_handle() = default;

    unique_handle(Handle handle, Deleter deleter)
        : h(handle), d(deleter), engaged(true) {}

    unique_handle(const unique_handle&) = delete;
    unique_handle& operator=(const unique_handle&) = delete;

    unique_handle(unique_handle&& other) noexcept
```

```
        : h(std::exchange(other.h, Handle{})),
          d(std::move(other.d)),
          engaged(std::exchange(other.engaged, false)) {}

    unique_handle& operator=(unique_handle&& other) noexcept {
        if (this != &other) {
            reset();
            h = std::exchange(other.h, Handle{});
            d = std::move(other.d);
            engaged = std::exchange(other.engaged, false);
        }
        return *this;
    }

    ~unique_handle() noexcept { reset(); }

    void reset() noexcept {
        if (engaged) {
            d(h);
            engaged = false;
        }
    }

    Handle get() const noexcept { return h; }
    explicit operator bool() const noexcept { return engaged; }
};
```

This pattern can wrap:

- sockets: `close(socket)`
- OS handles: `CloseHandle(h)` / `close(fd)`
- mappings: `munmap(ptr, size)` (often store size alongside)

- library objects: `sqlite3_close(db)`, etc.

8.2.7 State Restoration: RAII for "Must Revert" Changes

Not all resources are "things you close." Many are **temporary state changes** that must be undone reliably:

- temporarily disabling logging, checks, or interrupts
- switching a flag for a critical section
- swapping a pointer to a global service
- changing a mode that must be restored

The RAII rule is the same: capture the old state, install the new state, restore on destruction.

A simple state guard

```
#include <utility>

template <class T>
class restore_on_exit {
    T* target = nullptr;
    T old_value;

public:
    restore_on_exit(T& ref, T new_value)
        : target(&ref), old_value(std::exchange(ref,
        ↪  std::move(new_value))) {}

    restore_on_exit(const restore_on_exit&) = delete;
```

```
    restore_on_exit& operator=(const restore_on_exit&) = delete;

    ~restore_on_exit() noexcept {
        *target = std::move(old_value);
    }
};
```

Usage:

```
#include <string>

void temporarily_change(std::string& mode) {
    restore_on_exit<std::string> guard(mode, "SAFE_MODE");
    // mode == "SAFE_MODE" inside this scope
} // restored automatically
```

8.2.8 Error Handling Rule: "Destructors Release, Functions Report"

RAII guarantees release. It does *not* guarantee that release succeeds. Professional code separates two concerns:

- **Destructor**: best-effort cleanup, never throws.
- **Explicit operation**: if failure must be handled, provide a named function that can report it (return `bool`, `std::error_code`, `expected<T,E>`-style, or throw).

Example pattern:

```
class Output {
    // holds a resource
public:
    void close_or_throw();  // explicit: may fail, caller must handle
    ~Output() noexcept;     // implicit: best-effort, never throws
};
```

8.2.9 Checklist: Applying RAII to Any Non-Memory Resource

When you need RAII for a new resource type, follow this checklist:

1. Define the **owning type**: who releases the resource?
2. Acquire in construction (or a factory that returns a fully initialized RAII object).
3. Release in destruction (**non-throwing**).
4. Delete copy operations if ownership is unique; support move if transfer is meaningful.
5. Keep ownership **obvious at the call site**: locals, `std::unique_ptr`, and dedicated RAII wrappers.
6. If release can fail meaningfully, expose an explicit operation to report failures.

8.2.10 Summary

RAII is not "about `new`/`delete`." It is the universal C++ technique for making **all critical resources** safe under real control flow: early returns, exceptions, and complex branching. Modern C++ standard library types already embody this philosophy: file streams own file handles, lock objects own mutex ownership, and `std::jthread` owns a join policy. When you meet a raw handle API, you bring it into the same model by wrapping the handle and the release function into a move-only RAII type with a non-throwing destructor.

8.3 Designing RAII types correctly

8.3.1 Design Goal: Make Resource Correctness Automatic

A well-designed RAII type makes the correct behavior the easiest behavior:

- it acquires (or attaches to) a resource during construction
- it releases the resource during destruction
- it is hard (or impossible) to misuse
- it composes safely with other RAII objects

Think of an RAII type as a **resource handle**: an object that represents ownership and enforces cleanup through its lifetime.

8.3.2 State, Invariants, and Ownership

Define the invariant precisely

Every RAII type must have a simple, testable invariant that is always true after successful construction.
Typical invariants:

- **Engaged**: owns a valid resource handle and will release it in the destructor.
- **Empty**: owns nothing and destructor does nothing.

Use a representation that makes the invariant easy:

- store the handle plus a clear "empty" sentinel (e.g., `nullptr`, `-1`, or a known invalid handle value)
- keep the invariant enforced by all constructors, assignments, and modifiers

Be explicit about ownership

An RAII type is an **owner**. If the type does not own, it should not look like it owns. Practical guidance:

- **Owning types** release resources in their destructor.
- **Non-owning views** never release resources (they observe/borrow only).

Confusing these two roles is a common source of double-close, double-unlock, and use-after-close bugs.

8.3.3 The Rule of Zero, Rule of Five, and Rule of "One Resource"

Prefer the Rule of Zero when possible

If you can express the whole resource ownership using standard RAII members (`std::unique_ptr`, `std::vector`, `std::fstream`, `std::lock_guard`, `std::jthread`, etc.), do that and write **no** custom destructor/copy/move at all.

If you manage a raw handle, implement move-only ownership

Unique ownership is the default for resources like file descriptors, sockets, mutex ownership, and OS handles. That naturally maps to:

- deleted copy constructor and copy assignment
- move constructor and move assignment that transfer ownership

```
class UniqueFd {
    int fd_ = -1; // invariant: -1 means empty
```

```
public:
    UniqueFd() noexcept = default;

    explicit UniqueFd(int fd) noexcept : fd_(fd) {}

    UniqueFd(const UniqueFd&) = delete;
    UniqueFd& operator=(const UniqueFd&) = delete;

    UniqueFd(UniqueFd&& other) noexcept : fd_(other.fd_) {
        other.fd_ = -1;
    }

    UniqueFd& operator=(UniqueFd&& other) noexcept {
        if (this != &other) {
            reset();
            fd_ = other.fd_;
            other.fd_ = -1;
        }
        return *this;
    }

    ~UniqueFd() noexcept { reset(); }

    int get() const noexcept { return fd_; }
    explicit operator bool() const noexcept { return fd_ != -1; }

    void reset(int newfd = -1) noexcept {
        if (fd_ != -1) {
            // close(fd_);   // platform call (POSIX close, or a wrapper)
        }
        fd_ = newfd;
    }
```

```
    int release() noexcept {
        int tmp = fd_;
        fd_ = -1;
        return tmp;
    }
};
```

Design notes:

- The type is **move-only**, like `std::unique_ptr`.
- `reset()` releases the current resource and optionally adopts a new one.
- `release()` transfers out ownership without releasing.

8.3.4 Destructor Rules: Never Let Cleanup Throw

Destructors must not throw

RAII types are used precisely because they run during scope exit and unwinding. If a destructor throws during stack unwinding, the runtime will call `std::terminate`. Therefore:

- design the destructor to be **non-throwing**
- mark it **`noexcept`** (explicitly for clarity)

```
class Resource {
public:
    ~Resource() noexcept {
        // release resource (must not throw)
    }
};
```

When release can fail: separate “release” from “report errors”

Many releases are best-effort and should not throw (unlock, free, close-handle). But some releases can fail in a way you care about (e.g., flushing buffered output).
A robust pattern:

- destructor: best-effort cleanup, non-throwing
- explicit member function: performs the operation that can fail and reports the failure

```
#include <system_error>

class OutputSink {
public:
    void close_or_throw();   // explicit: may throw or return an error
    ~OutputSink() noexcept;  // implicit: best-effort, never throws
};
```

This prevents silent loss of important failures while keeping RAII reliable.

8.3.5 Construction: Ensure Strong Invariants and Exception Safety

Acquire in the constructor (or in a factory)

If the type represents ownership, it should not exist in a half-initialized state. Prefer:

- constructors that fully establish the invariant, or
- a factory function that returns a fully constructed RAII object (or reports failure)

If construction fails, nothing should leak

If your constructor acquires multiple resources, structure it so that each acquired resource is immediately owned by an RAII subobject. That way, if a later step fails, earlier steps are cleaned up automatically.

```
#include <memory>

struct A { /* acquire/release in ctor/dtor */ };
struct B { /* acquire/release in ctor/dtor */ };

struct Composite {
    A a;
    B b;
    Composite() : a(/*...*/), b(/*...*/) {} // if b throws, a is
    ↪  destroyed automatically
};
```

8.3.6 Copy and Move Semantics: Choose the Right Ownership Model

Unique ownership: delete copy, implement move

Most OS and synchronization resources are unique owners. The type should be move-only.

Shared ownership: use `std::shared_ptr` intentionally

If the resource truly has shared lifetime, use `std::shared_ptr` (or a dedicated shared handle type). Be aware of:

- cycles cause leaks unless broken (usually with `std::weak_ptr`)
- shared ownership can hide expensive lifetime coupling

Default guideline:

- prefer **unique ownership** unless sharing is required by design

8.3.7 Use Standard Building Blocks Whenever Possible

`std::unique_ptr` with custom deleter

For many C APIs, the simplest correct RAII wrapper is `std::unique_ptr` with a custom deleter.

```
#include <memory>
#include <cstdio>
#include <stdexcept>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f);
    }
};

using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file(const char* path, const char* mode) {
    if (auto* f = std::fopen(path, mode)) return unique_file{f};
    throw std::runtime_error("fopen failed");
}
```

This inherits well-tested move-only semantics from the standard library.

Prefer `std::jthread` over manual join management

For thread ownership where cooperative stop is needed, `std::jthread` provides RAII joining and integrates with `std::stop_token`. Prefer it when available and appropriate.

8.3.8 Provide a Minimal, Safe API Surface

A good RAII type exposes only what helps correct use.

Essential operations

For unique-ownership handles, these operations are usually enough:

- `get()` to observe the underlying handle without transferring ownership
- `reset()` to release current and optionally adopt a new handle
- `release()` to transfer ownership out
- `swap()` to exchange ownership efficiently
- `operator bool()` to test engaged/empty state

Avoid dangerous implicit conversions

Avoid implicit conversion to raw handle/pointer that makes it easy to store the resource elsewhere and outlive the owner. Prefer explicit accessors.

```
int native_handle() const noexcept;      // explicit
explicit operator bool() const noexcept; // safe truthiness
```

8.3.9 Make It Hard to Misuse

Prevent double-release and use-after-release

Enforce these by design:

- move transfers ownership and empties the source
- `release()` empties the object
- `reset()` always leaves the object in a valid state

Prefer "scoped" helpers for temporary ownership

For temporary actions (locks, state changes, restore-on-exit), prefer small scope-bound types. Example: a scope exit guard for one-off cleanup.

```
#include <utility>

template <class F>
class scope_exit {
    F f;
    bool active = true;

public:
    explicit scope_exit(F fn)
    ↪  noexcept(std::is_nothrow_move_constructible_v<F>)
        : f(std::move(fn)) {}

    scope_exit(const scope_exit&) = delete;
    scope_exit& operator=(const scope_exit&) = delete;
```

```
    scope_exit(scope_exit&& other)
    ↪  noexcept(std::is_nothrow_move_constructible_v<F>)
        : f(std::move(other.f)), active(std::exchange(other.active,
        ↪  false)) {}

    ~scope_exit() noexcept {
        if (active) f();
    }

    void release() noexcept { active = false; }
};
```

Usage:

```
#include <cstdio>

void example(std::FILE* f) {
    scope_exit cleanup([&] noexcept { if (f) std::fclose(f); });
    // work...
    // cleanup.release(); // optionally disable
}
```

8.3.10 Performance and Layout Considerations (Without Sacrificing Safety)

Good RAII types can be zero-overhead:

- store only the handle plus (if needed) a deleter
- make operations `noexcept` where appropriate to enable optimizations
- prefer move operations that are constant-time

If a deleter has no state (an empty function object), the compiler can often store it with no size overhead.

8.3.11 Validation Checklist: A Correct RAII Type Should Satisfy These Properties

Use this checklist before adopting a new RAII wrapper:

1. **Invariant**: after construction, the object is either engaged or empty, never "half-valid".
2. **Destructor**: non-throwing and safe on both engaged and empty states.
3. **Ownership**: copy is either correct (shared) or prohibited (unique). Unique is move-only.
4. **Move**: moved-from object becomes empty, and operations remain safe.
5. **API**: supports `get/reset/release/swap/operator bool` and avoids unsafe implicit conversions.
6. **Failure reporting**: operations that can fail report failures explicitly, not from the destructor.
7. **Composition**: can be used as a member field so constructor failure cleans up earlier members.

8.3.12 Summary

Designing RAII types correctly means designing **ownership and invariants** first, and then choosing the smallest set of operations that preserves them.
The two non-negotiable rules are:

- **the destructor must not throw**
- **ownership must be explicit and enforced by copy/move semantics**

When in doubt, prefer standard RAII types (especially `std::unique_ptr` and `std::jthread`) and adapt raw-handle APIs using move-only wrappers with non-throwing cleanup.

Chapter 9

Smart Pointers Done Right

9.1 `std::unique_ptr`

9.1.1 What `std::unique_ptr` Is (and Why It Exists)

`std::unique_ptr<T, D>` is the standard C++ smart pointer for **exclusive ownership**: at any time, *exactly one* `unique_ptr` owns a given object (or array), and that owner releases the resource automatically when it is destroyed.
Its design is intentionally strict:

- non-copyable (prevents accidental double-delete)
- movable (supports safe ownership transfer)
- zero-overhead in typical use (a raw pointer plus a deleter object)

9.1.2 The Safety Contract

Primary guarantee: deterministic destruction

When a `unique_ptr` is destroyed, it calls its deleter on the stored pointer if the pointer is non-null. With the default deleter, this means it performs `delete p;` (or `delete[] p;` for the array specialization).

Ownership is explicit in the type

A parameter of type `std::unique_ptr<T>` clearly communicates: *this function takes ownership*. This is a key guideline in professional APIs.

9.1.3 Correct Creation: Prefer `std::make_unique`

Why `make_unique` is the default choice

Prefer `std::make_unique` because it:

- avoids repeating the type name
- is exception-safe in expressions that create multiple objects
- is the idiomatic way to create a `unique_ptr`

```
#include <memory>
#include <string>

struct Session {
    std::string user;
    explicit Session(std::string u) : user(std::move(u)) {}
};

std::unique_ptr<Session> open_session() {
    return std::make_unique<Session>("alice");
}
```

`make_unique` for arrays

Use the array form when you truly need a dynamic array:
`std::make_unique<T[]>(n)` creates an array owned by
`std::unique_ptr<T[]>`.

```
#include <memory>

std::unique_ptr<int[]> make_buffer(std::size_t n) {
    auto p = std::make_unique<int[]>(n);
    for (std::size_t i = 0; i < n; ++i) p[i] = 0;
    return p;
}
```

9.1.4 Core Operations You Must Know

`get()`, `operator bool()`, and non-owning access

`get()` returns the raw pointer *without transferring ownership*. Use it to call APIs that only observe the pointer.

```
#include <memory>

struct Node { int value; };

void observe(const Node*); // does not store or delete

void example() {
    auto p = std::make_unique<Node>(Node{42});
    if (p) {
        observe(p.get()); // non-owning observation
    }
} // Node deleted here
```

Rule: if you pass `p.get()` to something, that something must not outlive `p` unless you have a stronger lifetime proof (or you transfer ownership).

`release()`: transfer responsibility without deleting

`release()` returns the raw pointer and makes the `unique_ptr` empty. After `release()`, **someone else must eventually delete/close the resource**.

```
#include <memory>

struct X { int v; };

X* hand_to_legacy() {
    auto p = std::make_unique<X>(X{7});
    return p.release(); // caller must delete
}
```

Use `release()` rarely. It is a controlled escape hatch for legacy interfaces.

`reset()`: delete current, then adopt a new pointer

```
#include <memory>

struct X { };

void example() {
    std::unique_ptr<X> p = std::make_unique<X>();
    p.reset(new X{}); // deletes old X, owns the new one
    p.reset();        // deletes current X, becomes empty
}
```

Professional guidance:

- prefer `p = std::make_unique<T>(...);` over `reset(new T(...))`
- use `reset` when you already have a pointer from an API you must adopt

`swap()`: constant-time ownership exchange

```
#include <memory>
#include <utility>

struct A { };

void example() {
    auto p = std::make_unique<A>();
    auto q = std::make_unique<A>();
    p.swap(q); // constant-time
}
```

9.1.5 Moves and Ownership Transfer

`unique_ptr` is move-only. Moving transfers ownership and makes the source empty.

```
#include <memory>

struct Widget { };

std::unique_ptr<Widget> make_widget() {
    auto p = std::make_unique<Widget>();
    return p; // moved out (or elided)
}

void take_ownership(std::unique_ptr<Widget> w) {
    // w owns the Widget in this scope
}
```

```
void example() {
    auto p = make_widget();
    take_ownership(std::move(p)); // explicit transfer
    // p is now empty
}
```

Key rule: **use `std::move` only when you are transferring ownership**.

9.1.6 Custom Deleters: Managing Non-`delete` Resources

Why the deleter exists

Many resources are not freed with `delete`:

- `FILE*` needs `fclose`
- a library handle needs a matching `close`/`free`/`destroy`
- OS handles often need platform-specific release

`unique_ptr<T, D>` stores a deleter object and calls it on destruction.

Example: `FILE*` with `fclose`

```
#include <cstdio>
#include <memory>
#include <stdexcept>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f);
    }
```

```
};

using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file(const char* path, const char* mode) {
    if (auto* f = std::fopen(path, mode)) return unique_file{f};
    throw std::runtime_error("fopen failed");
}
```

Professional notes:

- the deleter should be `noexcept` (cleanup must not throw)
- prefer a small stateless functor deleter to avoid extra storage

The deleter type is part of the `unique_ptr` type

`std::unique_ptr<T, D1>` and `std::unique_ptr<T, D2>` are different types. This matters for interfaces and ABI boundaries: choose a stable deleter type (often a named struct).

9.1.7 Arrays: `unique_ptr<T[]>` Has Different Rules

`std::unique_ptr<T[]>` is a separate specialization intended for dynamic arrays. Important differences:

- it uses `delete[]` by default (not `delete`)
- it provides `operator[]` for element access
- it does not provide `operator*` or `operator->`

```cpp
#include <memory>

void fill(std::unique_ptr<int[]>& a, std::size_t n) {
    for (std::size_t i = 0; i < n; ++i) a[i] = static_cast<int>(i);
}
```

Design guidance:

- prefer `std::vector<T>` for resizable arrays and safe size management
- use `unique_ptr<T[]>` for ownership of a raw dynamic array when you truly need it

9.1.8 Incomplete Types: Powerful, but You Must Respect the Rule

The rule

It is valid to declare `std::unique_ptr<T>` where `T` is incomplete (forward-declared), but **`T` must be complete at the point where the `unique_ptr` destructor is instantiated/called** when using the default deleter.

Practical pattern: PImpl

Header:

```cpp
#pragma once
#include <memory>

class Widget {
public:
    Widget();
    ~Widget(); // defined out-of-line where Impl is complete
```

```
    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;

private:
    struct Impl;                // incomplete here
    std::unique_ptr<Impl> p_;   // OK to declare
};
```

Source:

```
#include "Widget.h"

struct Widget::Impl {
    int x = 0;
};

Widget::Widget() : p_(std::make_unique<Impl>()) {}
Widget::~Widget() = default;
Widget::Widget(Widget&&) noexcept = default;
Widget& Widget::operator=(Widget&&) noexcept = default;
```

This works because `Impl` is complete in the translation unit where the destructor is defined.

9.1.9 API Design: Passing and Returning `unique_ptr` Safely

Return by value to transfer ownership

Returning `unique_ptr<T>` is the standard ownership-transfer API.

```
#include <memory>

struct Connection { };
```

```
std::unique_ptr<Connection> connect() {
    return std::make_unique<Connection>();
}
```

Take by value to assume ownership

Taking `unique_ptr<T>` by value expresses that the function assumes ownership.

```
#include <memory>

struct Job { };

void enqueue(std::unique_ptr<Job> j);

void example() {
    auto j = std::make_unique<Job>();
    enqueue(std::move(j)); // explicit transfer
}
```

Use references for non-owning access

If a function does not take ownership, prefer:

- `T*` or `T&` when you only need to observe or mutate
- `const T*` or `const T&` when you only need to read

Do not accept `unique_ptr<T>&` unless you truly need to modify the smart pointer itself (e.g., reset it, swap it).

9.1.10 Common Misuses and How to Avoid Them

Mistake: building a `unique_ptr` from a pointer you do not own

Never do this:

```
void bad(int* p) {
    std::unique_ptr<int> x(p); // WRONG unless p came from new and
    ↪  ownership is transferred
}
```

A `unique_ptr` must be constructed only from a pointer whose ownership is being transferred, typically created by `make_unique` or returned by an API that documents transfer of ownership.

Mistake: letting the raw pointer escape

If you call `get()` and store that pointer somewhere long-lived, you are back to manual lifetime bugs. If the code needs long-lived access, redesign:

- pass references where possible
- transfer ownership if necessary
- use shared ownership only when required by the model

Mistake: expecting `unique_ptr` to solve shared lifetime

`unique_ptr` represents exclusive ownership. If multiple parties need the same lifetime, you need an explicit shared-lifetime design (often `std::shared_ptr`, sometimes a higher-level owner).

9.1.11 Summary

`std::unique_ptr` is the foundational tool for expressing and enforcing **exclusive ownership** in Modern C++:

- create with `std::make_unique`
- transfer ownership with moves
- use custom deleters for non-`delete` resources
- respect the incomplete-type rule (complete type at destruction point with default deleter)
- design APIs so ownership transfer is explicit and obvious

Used consistently, `unique_ptr` eliminates the largest class of memory lifetime bugs caused by raw owning pointers, and it provides the same RAII safety model for many non-memory resources through custom deleters.

9.2 `std::shared_ptr`

9.2.1 What `std::shared_ptr` Is (and What It Is Not)

`std::shared_ptr<T>` is the standard smart pointer for **shared ownership**. Multiple `shared_ptr` objects can own the same object, and the object is destroyed when the *last owning* `shared_ptr` is destroyed or reset.
It is not a general substitute for:

- clear single ownership (`std::unique_ptr` is the default there)
- lifetime-safe borrowing (use references, pointers, or views with clear lifetime rules)

- thread-safety of the pointed-to object (reference counting is not the same as data-race freedom)

Guideline: prefer `unique_ptr` unless you truly need shared ownership.

9.2.2 The Control Block: Where the "Shared" Part Lives

A `shared_ptr` typically owns through a hidden **control block** that stores:

- a strong (owning) reference count
- a weak (non-owning) reference count (for `std::weak_ptr`)
- (sometimes) the deleter and allocator state

The key consequence:

- the object lifetime ends when the strong count reaches zero
- the control block may live longer if weak references remain

This explains why `weak_ptr` can observe expiration without owning the object.

9.2.3 Correct Creation: Prefer `std::make_shared`

Why `make_shared` is usually best

Prefer `std::make_shared<T>(args...)` because it is the idiomatic and efficient way to create a `shared_ptr`. Most implementations allocate the control block and the object together, which typically reduces allocations and improves locality.

```cpp
#include <memory>
#include <string>

struct Session {
    std::string user;
    explicit Session(std::string u) : user(std::move(u)) {}
};

std::shared_ptr<Session> open_session() {
    return std::make_shared<Session>("alice");
}
```

When `make_shared` is not ideal

You may prefer constructing with a custom deleter or allocator when:

- the resource is not released with `delete` (needs a custom close/free)
- the object must be freed in a specific way by an external API
- you need special allocation strategies

In such cases, use the appropriate `shared_ptr` constructor deliberately.

9.2.4 Core Semantics You Must Understand

Copying shares ownership

Copying a `shared_ptr` increases the owning count (logically: “one more owner”).

```cpp
#include <memory>

struct X { int v = 0; };
```

```
void example() {
    auto p = std::make_shared<X>();
    auto q = p;          // q shares ownership with p
    q->v = 7;
} // object destroyed only after both p and q are gone
```

Moving transfers one handle, does not change total ownership

Moving a `shared_ptr` transfers that handle; the moved-from pointer becomes empty.

```
#include <memory>

struct X { };

std::shared_ptr<X> f() {
    auto p = std::make_shared<X>();
    return p; // moved or elided
}
```

`reset()`, `use_count()`, and `unique()`

- `reset()` releases this owner's participation (decrements the strong count).
- `use_count()` reports the number of owning `shared_ptr` instances.
- `unique()` tells whether this is the only owner (historically useful but often a design smell).

Note: `use_count()` is typically for debugging and diagnostics, not for correctness decisions in concurrent code.

9.2.5 Threading: What Is Safe, What Is Not

Reference-count operations are safe across threads

`shared_ptr` is designed so that different `shared_ptr` objects that share ownership can be copied/moved/destroyed across threads without corrupting the control block.

But *the same* `shared_ptr` object is not magically data-race free

A critical rule:

> If multiple threads access the *same* `shared_ptr` object (not just the same pointee), and at least one thread modifies it (assign/reset), you must synchronize.

To share a pointer value between threads without external locking, use atomic operations for the pointer object: `std::atomic<std::shared_ptr<T»` (C++20+) or the corresponding atomic access facilities.

Atomic `shared_ptr` in Modern C++ (C++20 and C++26)

C++ provides a partial specialization `std::atomic<std::shared_ptr<T»` to manipulate `shared_ptr` objects atomically.
Also, the older free-function atomic access APIs for `shared_ptr` were deprecated in C++20 and removed in C++26 in favor of the `std::atomic` specialization.

```
#include <atomic>
#include <memory>

struct Config { int version = 0; };

std::atomic<std::shared_ptr<const Config>> g_cfg;
```

```
void publish_new_config(int v) {
    g_cfg.store(std::make_shared<Config>(Config{v}),
    ↪  std::memory_order_release);
}

int read_config_version() {
    auto local = g_cfg.load(std::memory_order_acquire);
    return local ? local->version : -1;
}
```

Important:

- Atomicity protects the `shared_ptr` value transfer (which object is referenced).
- It does not make `Config` itself thread-safe; you still need immutability or internal synchronization.

9.2.6 The Biggest Pitfall: Cycles and Leaks

Shared ownership can create **reference cycles**: two objects own each other through `shared_ptr`. The strong counts never reach zero, so neither object is destroyed.

Bad: cycle with `shared_ptr`

```
#include <memory>

struct B;

struct A {
    std::shared_ptr<B> b;
};
```

```
struct B {
    std::shared_ptr<A> a;
};

void leak_cycle() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a; // cycle: neither will be destroyed
}
```

Correct: break cycles with `std::weak_ptr`

`std::weak_ptr` is a non-owning observer of a `shared_ptr`'s control block. It can be converted to a `shared_ptr` temporarily by calling `lock()`. Using `weak_ptr` for back-references breaks cycles by removing one edge from strong ownership.

```
#include <memory>

struct B;

struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::weak_ptr<A> a; // non-owning back-reference
};

void no_leak_cycle() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
```

```
    a->b = b;
    b->a = a; // weak: cycle broken
}
```

Guideline: use `weak_ptr` to break cycles of `shared_ptr`.

9.2.7 Owning `this`: `std::enable_shared_from_this`

The problem

If an object needs to create a `shared_ptr` to itself, it must not do so by writing `std::shared_ptr<T>(this)`. That would create a *second, unrelated* control block and usually leads to double deletion.

The correct tool

Inherit from `std::enable_shared_from_this<T>` and call `shared_from_this()` to produce a new `shared_ptr` that shares ownership with the existing one.

```
#include <memory>

struct Node : std::enable_shared_from_this<Node> {
    std::shared_ptr<Node> self() {
        return shared_from_this(); // shares the existing control block
    }
};

void example() {
    auto n = std::make_shared<Node>();
    auto n2 = n->self(); // safe, shared ownership
}
```

Rule: only call `shared_from_this()` when the object is already owned by a `shared_ptr`. Otherwise it throws `std::bad_weak_ptr`.

9.2.8 Advanced Technique: The Aliasing Constructor

Why it exists

`shared_ptr` has an aliasing constructor that lets you share ownership with one object while storing a different pointer value. This is useful to keep a subobject alive by tying its lifetime to the parent object's control block.

Example: keep an owning parent alive while pointing to a member

```
#include <memory>
#include <vector>

struct Buffer {
    std::vector<int> data{1,2,3,4};
};

std::shared_ptr<const int> first_element(std::shared_ptr<Buffer> b) {
    // shares ownership with b, but points to b->data[0]
    return std::shared_ptr<const int>(std::move(b), &b->data[0]);
}
```

Safety note:

- This keeps `Buffer` alive as long as the returned `shared_ptr` lives.
- The returned pointer must remain valid as a subobject for the parent lifetime (no reallocation that invalidates it).

9.2.9 Custom Deleters: Shared Ownership of Non-`delete` Resources

`shared_ptr` can manage resources released by non-`delete` APIs using a custom deleter, the same general idea as `unique_ptr`.

```
#include <cstdio>
#include <memory>
#include <stdexcept>

std::shared_ptr<std::FILE> open_shared_file(const char* path, const char*
↪  mode) {
    std::FILE* f = std::fopen(path, mode);
    if (!f) throw std::runtime_error("fopen failed");

    return std::shared_ptr<std::FILE>(f, [](std::FILE* p) noexcept {
        if (p) std::fclose(p);
    });
}
```

Professional caution:

- Shared ownership of OS handles is often a design smell. Many handles are logically unique owners.
- Prefer `unique_ptr` (or a dedicated unique-handle wrapper) unless shared ownership is required.

9.2.10 API Design Rules for `shared_ptr`

Accept smart pointers only to express lifetime semantics

A strong guideline is to take smart pointers as parameters only when you need to express ownership or shared lifetime. Otherwise, accept references/pointers (`T&`, `T*`, `const T&`,

`const T*`) to avoid unnecessary refcount traffic and unclear ownership.
Common patterns:

- Return `std::shared_ptr<T>` to give shared ownership to the caller.
- Take `std::shared_ptr<T>` *by value* if you will store it (you are becoming a shared owner).
- Take `const std::shared_ptr<T>&` only when you must inspect the pointer object itself (rare).
- Take `T&` or `T*` when you only need to use the object during the call.

9.2.11 Performance and Predictability Considerations

Costs you should assume exist

Shared ownership has inherent runtime costs:

- control block allocation (often avoided or reduced by `make_shared`)
- reference counting on copies/destroys (often implemented with atomic operations)

Engineering guidance:

- Use `shared_ptr` as a design tool, not as a default pointer type.
- Prefer passing references for hot paths.
- Consider object ownership architecture first; performance follows naturally from clarity.

9.2.12 Common Misuses and Their Fixes

Mistake: "shared everywhere" to avoid thinking about ownership

Fix:

- model a single owner with `unique_ptr`
- provide non-owning access via references/views
- introduce `shared_ptr` only when multiple owners are truly required

Mistake: cycles

Fix:

- break cycles using `weak_ptr`
- redesign ownership so there is a clear tree/graph of owners and observers

Mistake: creating a `shared_ptr` from `this`

Fix:

- use `enable_shared_from_this` and `shared_from_this()`
- ensure objects are created under `make_shared` / `shared_ptr` factories

9.2.13 Summary

`std::shared_ptr` is the right tool when **shared ownership is a real requirement**. Used correctly:

- create with `make_shared`

- break cycles with `weak_ptr`
- use `enable_shared_from_this` to safely obtain owning pointers to `this`
- treat thread-safety carefully: the control block is safe, but the pointee and the pointer object still need synchronization
- avoid overuse: prefer `unique_ptr` and borrowing when sharing is not required

9.3 `std::weak_ptr`

9.3.1 What `std::weak_ptr` Is

`std::weak_ptr<T>` is a **non-owning** companion to `std::shared_ptr<T>`. It observes an object that is managed by shared ownership, without extending that object's lifetime.
Conceptually:

- `std::shared_ptr<T>` **owns** (keeps the object alive).
- `std::weak_ptr<T>` **observes** (does not keep it alive).

A `weak_ptr` can be converted to a temporary owning pointer by calling `lock()`. If the object is still alive, `lock()` returns a `shared_ptr` that shares ownership. If the object has already expired, `lock()` returns an empty `shared_ptr`.

9.3.2 The Control Block and What `weak_ptr` Actually Points To

A `weak_ptr` does not point to the object in the ownership sense. Instead, it refers to the same **control block** used by `shared_ptr`. That control block tracks:

- the strong (owning) count: how many `shared_ptr` owners exist
- the weak (observing) count: how many `weak_ptr` observers exist

Lifetime rule:

- when the strong count reaches zero, the managed object is destroyed
- the control block may remain as long as weak references exist

This is why a `weak_ptr` can detect expiration even after the object is gone.

9.3.3 Primary Use Cases

Breaking `shared_ptr` cycles

The most famous use of `weak_ptr` is to break reference cycles. If two objects own each other via `shared_ptr`, neither can reach zero owners, and the objects leak.

Bad: strong cycle leaks

```
#include <memory>

struct B;

struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::shared_ptr<A> a; // strong back-reference creates a cycle
};
```

```
void leak_cycle() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a; // cycle: neither will be destroyed
}
```

Correct: make one edge weak

```
#include <memory>

struct B;

struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::weak_ptr<A> a; // weak back-reference breaks the cycle
};

void no_leak_cycle() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a; // no cycle in strong ownership graph
}
```

Guideline: use `std::weak_ptr` to break cycles of `std::shared_ptr`.

Caches and registries (“keep if alive, otherwise recreate”)

A cache often should not extend lifetime; it should reuse an object *if it still exists*.

```cpp
#include <memory>
#include <string>
#include <unordered_map>

struct Image { /* big resource */ };

class ImageCache {
    std::unordered_map<std::string, std::weak_ptr<Image>> cache_;

public:
    std::shared_ptr<Image> get(const std::string& key) {
        if (auto it = cache_.find(key); it != cache_.end()) {
            if (auto alive = it->second.lock()) { // acquire temporary
            ↪ ownership
                return alive;
            }
        }
        auto fresh = std::make_shared<Image>();
        cache_[key] = fresh; // store weak observation
        return fresh;
    }
};
```

Observer relationships (subscribe without owning)

In event systems, you usually want:

- the subject to keep weak references to observers, so it does not prolong their lifetime
- observers to disappear naturally when their last strong owner goes away

```cpp
#include <memory>
#include <vector>
```

```cpp
#include <algorithm>

struct Listener {
    virtual ~Listener() = default;
    virtual void on_event() = 0;
};

class Subject {
    std::vector<std::weak_ptr<Listener>> listeners_;

public:
    void subscribe(const std::shared_ptr<Listener>& l) {
        listeners_.push_back(l); // weakly
    }

    void publish() {
        listeners_.erase(
            std::remove_if(listeners_.begin(), listeners_.end(),
                [](std::weak_ptr<Listener>& w) {
                    auto s = w.lock();
                    if (!s) return true;   // drop expired
                    s->on_event();
                    return false;
                }),
            listeners_.end());
    }
};
```

9.3.4 Correct Usage: `lock()` First

`lock()` is the safe acquisition primitive

The safe pattern is:

> Call `lock()` and use the resulting `shared_ptr` for the entire access.

This ensures the object stays alive while you use it.

```
#include <memory>

struct X { int v = 0; };

int read_if_alive(const std::weak_ptr<X>& w) {
    if (auto s = w.lock()) {      // either we acquire ownership or we
    ↪  don't
        return s->v;              // safe: object stays alive via s
    }
    return -1;                    // expired
}
```

Avoid `expired()` as a pre-check in multithreaded code

`expired()` is effectively a snapshot of whether the strong count is zero at that moment. Between calling `expired()` and using the object, another thread could release the last owner. Therefore:

- in concurrent code, do not write `if (!w.expired()) use(w);`
- prefer `if (auto s = w.lock()) use(*s);`

The latter is both simpler and race-resistant because it obtains a strong reference for the duration of use.

9.3.5 Key Member Functions and Their Meaning

`expired()` and `use_count()`

- `expired()` returns whether the observed object has no remaining owners.
- `use_count()` reports the number of owning `shared_ptr` instances.

Practical guidance:

- treat `use_count()` as diagnostics, not as a correctness tool
- prefer `lock()` as the only correctness-relevant check

`reset()`

`reset()` makes the `weak_ptr` empty (it stops observing).

`owner_before()`

`owner_before()` provides an **owner-based ordering** for smart pointers. Two pointers compare equivalent in this ordering if they share ownership of the same object (or are both empty), even if their raw pointer values differ (e.g., aliasing `shared_ptr`).
This is useful for ordering in associative containers when you need to key by ownership identity.

9.3.6 Threading and Safety Guarantees

What is safe

- Observing via `weak_ptr` does not extend lifetime.

- Calling `lock()` is the supported way to safely attempt access and, if successful, extend lifetime via a temporary `shared_ptr`.

What is not guaranteed

- `weak_ptr` does not make the pointed-to object thread-safe.
- You still need immutability or synchronization for the object's state.

9.3.7 Relationship to `enable_shared_from_this`

`std::enable_shared_from_this<T>` works by maintaining internal weak ownership information so that `shared_from_this()` can create a `shared_ptr` that shares the same control block. Calling `shared_from_this()` when the object is not already owned by a `shared_ptr` is a contract violation and results in an exception (`std::bad_weak_ptr`).
(Practical rule: only call `shared_from_this()` on objects that were created and are currently managed by `shared_ptr`.)

9.3.8 Design Guidance: Where `weak_ptr` Belongs

Use `weak_ptr` when you want *observation*, not ownership

Typical placements:

- back-references (child → parent) in object graphs
- caches and registries
- observer lists (event subscribers)
- cross-component references where ownership would create cycles or unclear lifetimes

Do not use `weak_ptr` to "patch over" unclear ownership

If the design has no clear owner, `weak_ptr` will not solve the core problem. Start by defining:

- who owns the object (often a single owner with `unique_ptr`)
- who merely borrows/observes (references, pointers, or views)
- where shared ownership is truly required (`shared_ptr`)

9.3.9 Common Mistakes and How to Avoid Them

Mistake: using the raw pointer without a strong reference

Bad pattern:

```
X* p = w.lock().get(); // p becomes dangling if the temporary shared_ptr
↪  is destroyed
```

Correct: keep the `shared_ptr` alive for the whole use.

Mistake: assuming `expired()` makes access safe

In concurrent code, `expired()` can become stale immediately. Correct: use `lock()` and check the result.

Mistake: never cleaning up expired entries in containers

If you store `weak_ptr` in containers, periodically prune expired entries (or prune while iterating, as shown in the observer example).

9.3.10 Summary

`std::weak_ptr` is the standard tool for **safe observation** in shared-ownership systems:

- it does not keep objects alive
- `lock()` safely attempts to obtain temporary ownership
- it breaks `shared_ptr` cycles and supports caches/observers
- it improves design clarity by separating *ownership* from *observation*

The professional rule is simple:

> If you need the object, `lock()` it and use the returned `shared_ptr`.

9.4 Common misuse patterns

9.4.1 Why Smart Pointers Still Get Misused

Smart pointers eliminate many manual `new`/`delete` hazards, but they do *not* eliminate the need to model ownership correctly. The most common failures come from:

- unclear ownership (who owns? who borrows?)
- creating multiple, conflicting owners for the same raw pointer
- letting raw pointers outlive their owners
- using `shared_ptr` as a "default pointer" instead of a deliberate shared-ownership tool

The C++ Core Guidelines emphasize that smart pointers should be used to *represent ownership*, and that `unique_ptr` should be preferred unless shared ownership is truly required.

9.4.2 Misuse 1: Constructing an owning smart pointer from a non-owned raw pointer

The bug

Creating `std::unique_ptr<T>` or `std::shared_ptr<T>` from a raw pointer that you do not own (e.g., stack memory, a pointer owned elsewhere, a pointer managed by another smart pointer) is a direct path to double-delete or invalid free.

Bad

```
#include <memory>

int main() {
    int x = 42;
    std::unique_ptr<int> p(&x); // WRONG: will call delete on stack
    ↪  memory
}
```

Correct

```
#include <memory>

int main() {
    auto p = std::make_unique<int>(42); // correct unique ownership
}
```

Guideline: use `make_unique()` / `make_shared()` to create owners.

9.4.3 Misuse 2: Two `shared_ptr` control blocks for the same object

The bug

This is one of the most dangerous smart-pointer errors: constructing more than one `shared_ptr` independently from the same raw pointer. Each `shared_ptr` then believes it is the sole shared owner (with its own control block), and the object gets deleted twice.

Bad

```
#include <memory>

struct X { };

int main() {
    X* raw = new X;

    std::shared_ptr<X> a(raw);
    std::shared_ptr<X> b(raw); // WRONG: second control block, double
    ↪  delete
}
```

Correct

```
#include <memory>

struct X { };

int main() {
    auto a = std::make_shared<X>();
    auto b = a; // shares the same control block
}
```

Rule: if you need multiple `shared_ptr` to the same object, copy an existing `shared_ptr` (or create via `make_shared`), never "re-wrap" the raw pointer.

9.4.4 Misuse 3: `shared_ptr(this)` instead of `enable_shared_from_this`

The bug

Inside an object method, writing `std::shared_ptr<T>(this)` creates a new control block. If the object is already managed by a `shared_ptr`, this again leads to double deletion.

Bad

```
#include <memory>

struct Node {
    std::shared_ptr<Node> self_bad() {
        return std::shared_ptr<Node>(this); // WRONG: new control block
    }
};
```

Correct

```
#include <memory>

struct Node : std::enable_shared_from_this<Node> {
    std::shared_ptr<Node> self() {
        return shared_from_this(); // shares existing control block
    }
};

int main() {
    auto n = std::make_shared<Node>();
    auto n2 = n->self();
```

```
}
```

Contract: `shared_from_this()` is valid only when the object is already owned by a `shared_ptr`.

9.4.5 Misuse 4: Treating `shared_ptr` as "the default pointer type"

The bug

Using `shared_ptr` everywhere to avoid thinking about ownership often produces:

- hidden global lifetime coupling
- accidental cycles (leaks)
- unnecessary reference-count churn in hot paths

Guideline: prefer `unique_ptr` over `shared_ptr` unless you need to share ownership.

Better architecture

- one clear owner: `unique_ptr<T>` (or a value member)
- many users: references/pointers/views (`T&`, `T*`, `span`, etc.) with documented lifetime
- shared ownership only where the model truly requires it: `shared_ptr<T>`

9.4.6 Misuse 5: Reference cycles with `shared_ptr`

The bug

Two objects owning each other via `shared_ptr` keeps the strong counts non-zero forever: the objects never get destroyed.

Bad

```
#include <memory>

struct B;

struct A { std::shared_ptr<B> b; };
struct B { std::shared_ptr<A> a; };

void leak_cycle() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a; // cycle: leak
}
```

Correct: break cycles with `weak_ptr`

```
#include <memory>

struct B;

struct A { std::shared_ptr<B> b; };
struct B { std::weak_ptr<A> a; }; // weak back-reference

void ok() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a;
}
```

Guideline: use `std::weak_ptr` to break cycles of `shared_ptr`.

9.4.7 Misuse 6: Letting `get()` escape beyond the owner

The bug

`get()` returns a raw pointer *without* extending lifetime. Storing that raw pointer for later use easily creates dangling pointers.

Bad

```
#include <memory>

struct X { int v = 0; };

X* g = nullptr;

void bad() {
    auto p = std::make_unique<X>();
    g = p.get(); // g outlives p -> dangling after return
}
```

Correct patterns

- keep the owner alive for at least as long as any borrowed pointer/reference is used
- if long-lived access is required, redesign ownership (transfer or share intentionally)

For `weak_ptr`, the safe pattern is: **lock, then use the resulting `shared_ptr`.**

9.4.8 Misuse 7: `weak_ptr::expired()` as a "safety check"

The bug

`expired()` is only a snapshot. In concurrent code, the object can expire immediately after the check. Correctness must be based on `lock()`, which creates a strong reference if the object is still alive.

Bad

```
#include <memory>

struct X { void use(); };

void bad(const std::weak_ptr<X>& w) {
    if (!w.expired()) {            // racy check
        w.lock()->use();           // may still be empty
    }
}
```

Correct

```
#include <memory>

struct X { void use(); };

void good(const std::weak_ptr<X>& w) {
    if (auto s = w.lock()) {       // acquire strong reference
        s->use();                  // safe while s is alive
    }
}
```

9.4.9 Misuse 8: Using `use_count()` for correctness

The bug

`use_count()` is not a correctness tool. In concurrent programs it can change immediately after reading, and even in single-threaded code it encourages brittle logic that couples correctness to ownership topology.

Better

- for lifetime safety: keep an owning `shared_ptr` while using the object
- for uniqueness requirements: redesign the ownership model (often `unique_ptr`)
- for debugging/metrics: `use_count()` can be acceptable as diagnostics

9.4.10 Misuse 9: Misunderstanding thread safety

The bug

It is true that `shared_ptr`/`weak_ptr` manage their control block safely across threads, but that does *not* make the pointed-to object thread-safe.
Two common errors:

- assuming `shared_ptr` implies safe concurrent access to the object
- concurrently reading/writing the *same* `shared_ptr` variable without synchronization

Correct approach:

- protect object state with locks, immutability, or a concurrency-safe design
- protect pointer-value updates with synchronization or `std::atomic<std::shared_ptr<T>` (C++20+) when required

9.4.11 Misuse 10: Misusing the aliasing constructor

The bug

The aliasing constructor allows a `shared_ptr` to share ownership with another `shared_ptr` while storing a different pointer value. This is powerful, but it can be misused when the stored pointer does not remain valid for the lifetime of the shared ownership group (e.g., pointing into a vector that later reallocates).

Bad: aliasing a pointer that may be invalidated

```
#include <memory>
#include <vector>

struct Owner {
    std::vector<int> v{1,2,3,4};
};

std::shared_ptr<int> bad(std::shared_ptr<Owner> o) {
    // points to v[0], but later push_back might reallocate and
    ↪  invalidate &v[0]
    return std::shared_ptr<int>(std::move(o), &o->v[0]);
}
```

Safer guidance

- alias only subobjects with stable addresses for the required lifetime
- prefer returning a `shared_ptr<Owner>` plus an index, or returning a view with clearly documented constraints

9.4.12 Misuse 11: Throwing cleanup through a deleter

The bug

Deleters run during destruction and must not throw (just like destructors). If cleanup throws during stack unwinding, the program terminates. Therefore deleters should be `noexcept` by design.

Correct pattern

```
#include <memory>
#include <cstdio>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f); // best-effort, no throw
    }
};
```

If an operation can fail meaningfully (e.g., flushing output), make that failure visible via an explicit function (`flush_or_throw()`, `close_or_error()`, etc.), not via the deleter/destructor.

9.4.13 Misuse 12: Passing smart pointers in APIs without meaning

The bug

Passing `shared_ptr` or `unique_ptr` parameters when the function does not need ownership creates confusion and can introduce unnecessary reference counting or forced heap allocation.
Guideline: take smart pointers as parameters only to explicitly express lifetime semantics.

Better API shapes

- **Borrowing**: `void f(T&)` or `void f(T*)`
- **Read-only borrowing**: `void f(const T&)` or `void f(const T*)`
- **Unique ownership transfer**: `void f(std::unique_ptr<T>)`
- **Shared ownership participation (store it)**: `void f(std::shared_ptr<T>)`

9.4.14 Practical Checklist: Spotting Misuse Quickly

When reviewing code, these are high-signal indicators of smart-pointer misuse:

1. `shared_ptr<T>(raw)` appears more than once for the same raw pointer
2. `shared_ptr(this)` appears anywhere
3. raw pointers from `get()` are stored globally or in long-lived objects
4. `expired()` used as a pre-check in concurrent code (instead of `lock()`)
5. `use_count()` used to decide correctness or ordering of operations
6. pervasive `shared_ptr` without a clear "why shared?" answer
7. `weak_ptr` containers never prune expired entries
8. aliasing constructors that point into containers that might reallocate

9.4.15 Summary

Smart pointers are safe *only when they encode the right ownership model*:

- `unique_ptr` for single ownership and clear transfer by move
- `shared_ptr` only when shared ownership is required
- `weak_ptr` for observation, caches, and breaking cycles

Most real-world bugs come from creating the wrong owners (or too many owners), and from letting raw pointers escape past the lifetime guaranteed by the owning smart pointer.

9.5 Performance considerations

9.5.1 A Practical Cost Model

Smart pointers are primarily *correctness* tools, but they do have predictable runtime and memory costs. The performance-relevant questions are:

- Does this pointer type **own** (and therefore do work on construction/destruction), or is it just a **borrow**?
- Does it perform **reference counting** (often with atomics)?
- Does it allocate a **control block** (extra memory and an extra allocation in some cases)?
- Does it force **indirection / heap allocation** where a value member would have been better?

As a rule of thumb: `unique_ptr` is usually close to the cost of a raw pointer, while `shared_ptr`/`weak_ptr` introduce reference-count and control-block overhead.

9.5.2 `std::unique_ptr`: Usually "Zero-Overhead" in Practice

What it costs

`std::unique_ptr<T, D>` is move-only and typically stores:

- one pointer (`T*`)
- plus a deleter object (`D`) which is often empty (and therefore can be optimized away in size)

Operations are simple:

- move: pointer swap/transfer (constant time)
- destruction: one conditional deleter call

This means `unique_ptr` is generally the right default for owning dynamic objects when you need dynamic lifetime.

Hidden cost to watch for: needless heap allocation

The most common performance mistake is not the pointer itself, but *using heap allocation at all* when value semantics would be faster and simpler:

```
struct Big { /* ... */ };

void better() {
    Big x;             // value, no allocation
    // ...
}

void maybe_slower() {
```

```
    auto p = std::make_unique<Big>(); // allocation + indirection
    // ...
}
```

If the object naturally belongs to one owner and its lifetime is scoped, consider making it a direct member or local value.

9.5.3 `std::shared_ptr`: Reference Counting Is the Cost Center

What it costs

A `shared_ptr` owns through a **control block** and maintains a strong reference count (and usually a weak count).
Costs you should assume exist:

- extra memory for the control block
- increments/decrements of the reference count on copy/move/destroy
- in many implementations, reference-count updates use atomic operations to support cross-thread ownership safely

This is why the C++ Core Guidelines recommend preferring `unique_ptr` unless shared ownership is actually required.

Most expensive pattern: passing `shared_ptr` by value when you do not need ownership

Copying a `shared_ptr` typically modifies the reference count. That is real work in hot paths.
If a function does *not* store the `shared_ptr` and does not participate in ownership, it should normally take a reference/pointer to the object, not a `shared_ptr`.

```cpp
#include <memory>

struct Job { void run() const; };

void run_job_fast(const Job& j) {   // borrowing, no refcount traffic
    j.run();
}

void run_job_costly(std::shared_ptr<Job> j) { // may touch refcount
    j->run();
}
```

Guideline (practical):

- Take `std::shared_ptr<T>` **by value** only if you will store it or otherwise extend shared ownership.
- Otherwise, take `T&`, `T*`, `const T&`, `const T*`.

`use_count()` and `unique()` in hot paths

`use_count()` is typically for diagnostics and debugging, not for performance-critical logic or correctness decisions.
Using it in tight loops often adds extra refcount reads and encourages brittle “count-based” logic.

9.5.4 `make_shared` and `allocate_shared`: Allocation and Locality Tradeoffs

Why `make_shared` is often faster

`std::make_shared<T>` (and `std::allocate_shared`) typically perform a **single allocation** that stores both the control block and the object in one memory block, improving locality and reducing allocation overhead. The standard recommends (but does not require) this; all major implementations do it.

```
#include <memory>

struct X { int a; int b; };

auto p = std::make_shared<X>(); // typically one allocation for control
↪  block + X
```

The non-obvious tradeoff: long-lived `weak_ptr` can retain the whole block

With a single-allocation strategy, if a `weak_ptr` survives long after the last `shared_ptr` is gone, the combined block (object storage + control block) cannot be fully freed until weak references are gone. With separate allocations, only the control block remains. This is a memory-retention tradeoff, not a correctness issue, and it can matter in cache-like designs with long-lived `weak_ptr`.
Practical guidance:

- Prefer `make_shared` by default for speed and locality.
- If you expect many long-lived weak observers and large objects, measure memory retention behavior.

`make_shared_for_overwrite`

C++20 added `make_shared_for_overwrite` for cases where you will overwrite the object storage and want to avoid value-initialization work. This can be a performance win for

large trivially copyable types or buffers, when you will fully initialize them manually.

9.5.5 `std::weak_ptr`: Observation Has a Cost

Cost drivers

A `weak_ptr` typically:

- increments/decrements the weak count in the control block on copy/destroy
- performs a checked operation in `lock()` that attempts to create a new `shared_ptr` only if the object is still alive

Use the canonical pattern:

```
#include <memory>

struct X { void f() const; };

void use_if_alive(const std::weak_ptr<X>& w) {
    if (auto s = w.lock()) { // do one lock per logical use
        s->f();
    }
}
```

Performance guidance:

- Avoid repeated `lock()` calls inside loops when you can lock once and reuse the resulting `shared_ptr`.
- Store `weak_ptr` in caches/registries to avoid extending lifetime, but prune expired entries.

9.5.6 Atomic Shared Ownership: Avoid Accidental Contention

Atomic `shared_ptr` operations are for pointer publication, not for object safety

If you need to publish and update a shared pointer value across threads without external locking, Modern C++ provides `std::atomic<std::shared_ptr<T>>` (C++20+).

```
#include <atomic>
#include <memory>

struct Config { int v; };

std::atomic<std::shared_ptr<const Config>> g;

void publish(int v) {
    g.store(std::make_shared<Config>(Config{v}),
     ↪  std::memory_order_release);
}

int read() {
    auto p = g.load(std::memory_order_acquire);
    return p ? p->v : -1;
}
```

Important performance note:

- Atomic pointer publication is useful, but it is not free.
- If many threads frequently copy/assign the same shared ownership group, reference-count contention can become significant.

C++26 library direction

The older C-style atomic access APIs for `shared_ptr` were deprecated in C++20 and removed in C++26, in favor of the `std::atomic<std::shared_ptr<T>` specialization. This reduces confusion and encourages explicit, type-safe atomic usage.

9.5.7 Deleters and Object Size: Small Details That Matter

Custom deleters can affect the pointer object's size

For `unique_ptr`, the deleter type is part of the pointer type and is stored inside the `unique_ptr`. A stateless deleter is usually cheap; a stateful deleter increases object size and can hurt cache behavior.
For `shared_ptr`, the deleter is typically stored in the control block; copying a `shared_ptr` does not copy the deleter state, but creating the first owner involves setting up that control block.
Practical guidance:

- Prefer small, stateless deleters where possible.
- Avoid capturing large state in deleters unless it is required.

9.5.8 A Decision Guide That Improves Performance and Clarity

Prefer the cheapest correct lifetime model

- **Value member / local value**: fastest and simplest when lifetime is scoped and ownership is clear.
- **Borrowed reference/pointer/view**: no ownership overhead; requires clear lifetime documentation.

- **`unique_ptr`**: exclusive ownership with minimal overhead; default owning pointer.
- **`shared_ptr`**: use only when shared ownership is a real requirement; expect refcount/control-block cost.
- **`weak_ptr`**: observation and cycle-breaking; `lock()` introduces checked acquisition cost.

This ordering matches the Core Guidelines philosophy of using smart pointers to *express ownership* and not as a default substitute for references and values.

9.5.9 Summary

Performance with smart pointers is mostly about choosing the right ownership model:

- `unique_ptr` is typically near raw-pointer cost and is the default owning tool.
- `shared_ptr` introduces control-block and reference-count overhead; avoid copying it in hot paths unless needed.
- `make_shared`/`allocate_shared` often reduce allocations and improve locality, but can increase memory retention when weak observers live long.
- `weak_ptr` is for observation; lock once per use and prune expired entries in containers.
- Atomic shared-pointer publication (`std::atomic<std::shared_ptr<T>>`) is useful but not free; it solves pointer-value races, not object-state races.

Chapter 10

Raw Pointers: When Are They Acceptable?

10.1 Observers vs owners

10.1.1 The Central Distinction

In modern C++ safety, the most important question about any pointer is not "Is it smart or raw?" It is:

> Does this pointer **own** the object (is responsible for destroying it), or does it merely **observe** it (borrow it without controlling lifetime)?

The C++ Core Guidelines summarize this idea with two rules:

- A raw pointer (`T*`) is **non-owning** by default.
- References should **never own**.

This section uses:

- **Owner** = the code that is responsible for eventually destroying/releasing the resource.
- **Observer** = a non-owning view that must never outlive the owned object.

10.1.2 Why the Distinction Matters

If ownership is unclear, you get the classic failures:

- **double delete**: two owners both destroy the same object
- **use-after-free**: an observer outlives the object
- **leaks**: nobody clearly owns, so nobody reliably destroys

The Core Guidelines explicitly aim to make owners identifiable so cleanup is reliable and efficient.

10.1.3 Owners: What Ownership Looks Like in Modern C++

Prefer `std::unique_ptr` for single ownership

Single ownership is the default model in safe designs. Use `std::unique_ptr<T>` (and factories returning it) to express that exactly one place owns the object.

```
#include <memory>

struct Widget { int id; };

std::unique_ptr<Widget> make_widget(int id) {
    return std::make_unique<Widget>(Widget{id});
}
```

Use `std::shared_ptr` only for true shared ownership

Shared ownership is a specific design choice. Passing `shared_ptr` by value means the callee participates in ownership (it keeps the object alive).

```
#include <memory>

struct Service { void use() const; };

void store_for_later(std::shared_ptr<Service> s); // takes shared
↪  ownership

void example() {
    auto s = std::make_shared<Service>();
    store_for_later(s); // caller and callee now share ownership
}
```

Never transfer ownership via a raw pointer

A key interface rule:

> Never transfer ownership by a raw pointer (`T*`).

If an API needs to transfer ownership, use `unique_ptr` or `shared_ptr`.

Annotating legacy ownership: `gsl::owner<T*>`

When you must use raw pointers in legacy or low-level code, the Guidelines Support Library (GSL) provides annotations to make intent explicit:

- `gsl::owner<T*>` for owning raw pointers (legacy interop)
- plain `T*` for non-owning raw pointers (observers)

The Core Guidelines describe this convention directly.

10.1.4 Observers: When Raw Pointers Are Acceptable

Raw pointers are acceptable as non-owning views

Raw pointers are commonly used as observers because:

- they are lightweight (no control block, no refcount)
- they work well with polymorphism and optional "may be null" semantics
- they interoperate with C and OS APIs

But the safety contract is strict:

> A raw pointer observer must never outlive the object it points to.

Borrowing parameters: `T*` and `T&`

A good rule (also emphasized in smart-pointer parameter guidance):

- Take `T&` or `const T&` when the argument must exist (not optional).
- Take `T*` or `const T*` when "no object" is meaningful (nullable).
- Take smart pointers only when you mean ownership transfer or sharing.

```
struct Device { void reset(); };

void reset_device(Device& d) {      // must exist
    d.reset();
}

void maybe_reset(Device* d) {       // optional
    if (d) d->reset();
}
```

Non-null observers: `gsl::not_null<T*>`

If a pointer must never be null, express that contract. The Core Guidelines recommend using `gsl::not_null` for pointers that must not be null.

```
#include <gsl/gsl>

struct Engine { void tick(); };

void tick_engine(gsl::not_null<Engine*> e) {
    e->tick(); // no null checks needed by design
}
```

Observer data members: allowed, but document lifetime

The Guidelines explicitly distinguish between an owning member pointer and a non-owning member pointer.

```
#include <memory>

struct Logger { void info(const char*); };

class Component {
    std::unique_ptr<int> owned_; // owner (example)
    Logger* log_ = nullptr;      // observer: does not own

public:
    Component(std::unique_ptr<int> owned, Logger& log)
        : owned_(std::move(owned)), log_(&log) {}

    void run() {
        log_->info("running");
    }
```

```
};
```

Professional requirement: the code that constructs `Component` must ensure `Logger` outlives `Component`. That is the observer contract.

10.1.5 Recognizing Ownership at the Call Site

Owners are usually created in one of three ways

- **Value members / values**: lifetime is lexical (stack or containing object)
- **`unique_ptr`**: single ownership, moved into the owner
- **`shared_ptr`**: shared ownership, copied to add an owner

The best designs make ownership obvious without reading deep documentation.

Observers are usually passed as references/pointers

If you see:

- `T&` / `const T&`: must exist, borrowed
- `T*` / `const T*`: may be null, borrowed

that is typically observation, not ownership.

10.1.6 The Two Classic Bugs (and Their Fixes)

Bug 1: Owning with a raw pointer by accident

```
struct X { };
```

```
void bad() {
    X* p = new X;  // ownership is unclear
    // ...
    delete p;      // manual cleanup is easy to forget or duplicate
}
```

Fix: express ownership with `unique_ptr` (or a value).

```
#include <memory>

struct X { };

void good() {
    auto p = std::make_unique<X>();
    // ...
} // deterministic cleanup
```

The Guidelines warn against raw owning pointers and aim to make owners explicit.

Bug 2: Observer outlives owner

```
#include <memory>

struct X { int v = 0; };

X* g = nullptr;

void bad() {
    auto p = std::make_unique<X>();
    g = p.get(); // g is an observer, but p will be destroyed at scope
    ↪  exit
} // g becomes dangling here
```

Fix options:

- keep the owner alive as long as any observer is used
- redesign: return the owned object, store it, or share ownership intentionally

10.1.7 A Practical Rule Set for Chapter 10

Raw pointers are acceptable when they are observers

1. A raw `T*` parameter is fine for optional borrowing (nullable) *when the callee does not store it*.
2. A raw `T*` data member is acceptable as an observer if the owning relationship is elsewhere and documented.
3. Use `gsl::not_null<T*>` when null is not a valid state.

Raw pointers are not acceptable for ownership transfer

1. Do not return owning raw pointers from factories.
2. Do not accept owning raw pointers to indicate "caller transfers ownership".
3. Use `unique_ptr` (or `shared_ptr` when sharing is required) to represent that transfer.

10.1.8 Summary

- **Owners** control lifetime: use values, `unique_ptr`, or `shared_ptr`.
- **Observers** do not control lifetime: use `T&`, `T*`, and (when appropriate) `gsl::not_null`.

- Raw pointers are acceptable **as observers**, but unsafe **as owners** unless explicitly annotated for legacy interop.
- Good APIs make ownership obvious: do not transfer ownership by raw pointers.

10.2 Documenting ownership clearly

10.2.1 The Goal: Make Lifetime Obvious Without Reading the Implementation

Ownership documentation is successful when a reader can answer these questions quickly:

- Who **owns** this object (who must eventually destroy it)?
- Who **borrows** it (who may use it temporarily, without extending lifetime)?
- Is the pointer/reference **nullable** or **non-null** by contract?
- Is the lifetime **scoped**, **shared**, **global**, or **external**?

The most reliable way to document ownership is to encode it directly in the *types* and *API shapes*, and then add short, local comments only where lifetime relationships are not obvious.

10.2.2 Document Ownership by Choosing the Right Parameter Types

Borrowing: use references and raw pointers

Use references and raw pointers to express borrowing:

- `T&` / `const T&`: the argument must exist, not optional
- `T*` / `const T*`: the argument is optional (may be null)

```
struct Device { void reset(); };

void reset_device(Device& d) {      // must exist; borrowed
    d.reset();
}

void maybe_reset(Device* d) {       // optional; borrowed
    if (d) d->reset();
}
```

These signatures document the ownership rule implicitly: the callee does not own, and must not store the pointer beyond the call unless the lifetime is explicitly guaranteed.

Non-null borrowing: use `gsl::not_null`

When null is not a valid state, document it in the type. This is especially useful when you need pointer semantics (rebindable, nullable-by-default types, etc.).

```
#include <gsl/gsl>

struct Engine { void tick(); };

void tick_engine(gsl::not_null<Engine*> e) { // borrowed, never null
    e->tick();
}
```

Now the contract is explicit and reviewable: callers cannot pass null accidentally, and callees do not need defensive null checks.

Ownership transfer: use smart pointers by value

Never use raw pointers to transfer ownership. If a function *takes ownership*, accept a smart pointer *by value*:

```cpp
#include <memory>

struct Job { void run(); };

void enqueue(std::unique_ptr<Job> j); // takes ownership (unique)

void example() {
    auto j = std::make_unique<Job>();
    enqueue(std::move(j)); // ownership transfer is explicit at the call
    ↪ site
}
```

If the callee wants to *share* ownership (store a shared reference), take `std::shared_ptr<T>` by value:

```cpp
#include <memory>

struct Service { void use() const; };

class Registry {
    std::shared_ptr<Service> s_;
public:
    void install(std::shared_ptr<Service> s) { // takes shared ownership
        s_ = std::move(s);
    }
};
```

If a function does not need ownership, do not accept a smart pointer parameter just because the caller has one; accept `T&` or `T*` instead.

10.2.3 Document Ownership by Choosing the Right Return Types

Returning a new owner

Factories should return an owning type:

- `std::unique_ptr<T>` for unique ownership
- `std::shared_ptr<T>` for shared ownership

```
#include <memory>

struct Widget { int id; };

std::unique_ptr<Widget> make_widget(int id) {
    return std::make_unique<Widget>(Widget{id});
}
```

Returning an observer

Returning `T*` is acceptable only when it is a non-owning observation of something whose lifetime is managed elsewhere (and that lifetime is documented and enforced).

```
struct Node { Node* next = nullptr; };

Node* find(Node* head, int /*key*/) {
    // returns a borrowed pointer into the list; caller must not outlive
    ↪  the list owner
    return head;
}
```

Prefer returning references when null is not meaningful, and pointers when “not found” is a valid outcome.

10.2.4 Make Ownership Visible in Data Members

Owners as members

A class that owns an object should say so in the member type:

- value member: **direct ownership** (often best)
- `std::unique_ptr<T>`: **unique ownership**
- `std::shared_ptr<T>`: **shared ownership**

```
#include <memory>

struct Impl { int state = 0; };

class Owner {
    std::unique_ptr<Impl> p_; // owns Impl
public:
    Owner() : p_(std::make_unique<Impl>()) {}
};
```

Observers as members (allowed, but must be documented)

Observer members are fine when the owning relationship is external, but they require a clear lifetime statement near the declaration.

```
#include <memory>

struct Logger { void info(const char*); };

class Component {
    Logger* log_ = nullptr; // observer; must outlive *this
```

```
public:
    explicit Component(Logger& log) : log_(&log) {}

    void run() {
        log_->info("running");
    }
};
```

This single comment often prevents months of confusion.

10.2.5 Annotating Legacy Ownership When Raw Owning Pointers Exist

Sometimes legacy APIs or low-level interfaces force raw owning pointers. In such cases, annotate explicitly and isolate the ownership boundary.

Use `gsl::owner<T*>` as an ownership marker

```
#include <gsl/gsl>

struct CHandle;

gsl::owner<CHandle*> acquire();          // returns owning raw pointer
↪  (legacy boundary)
void release(gsl::owner<CHandle*> h);    // consumes ownership
```

Then immediately wrap into RAII on the C++ side:

```
#include <memory>

struct CHandle;
extern CHandle* acquire();
extern void release(CHandle*) noexcept;
```

```
struct HandleDeleter {
    void operator()(CHandle* h) const noexcept { if (h) release(h); }
};

using unique_handle = std::unique_ptr<CHandle, HandleDeleter>;

unique_handle make_handle() {
    return unique_handle{acquire()};
}
```

The documentation is now encoded in types: raw ownership exists only at the boundary, and the internal design is RAII.

10.2.6 Naming Conventions That Communicate Ownership

Names cannot replace types, but they can reinforce intent:

- Suffix observer members with _ and add a short comment: `Logger* log_` *(observer)*
- Use verbs that reveal transfer: `take_ownership`, `adopt`, `release`, `detach`
- Use `create/make` for factories returning owners
- Use `get/find/lookup` for observers (borrowed results)

10.2.7 Documenting Lifetime Assumptions Precisely

When the type alone cannot express the full lifetime relationship, document a short invariant:

```
class Session {
    // Invariant: cfg_ points to a configuration object that outlives
    ↪ *this.
    const void* cfg_ = nullptr; // observer (example placeholder type)
};
```

Good documentation is:

- local (next to the member or parameter)
- testable (an invariant you could assert in debug builds)
- stable (does not depend on hidden call-order assumptions)

10.2.8 Tooling: Enforcing Ownership Documentation

Clear ownership documentation becomes much stronger when tooling can check it. Two practical strategies:

Use guideline-based static analysis

Enable guideline checkers (e.g., Core Guidelines checks) that flag:

- suspicious smart pointer parameters (taking `unique_ptr` by const reference, etc.)
- patterns that violate ownership conventions

Codify project rules

Establish project rules such as:

- "Owning pointers are always `unique_ptr` or `shared_ptr` (or `gsl::owner` at boundaries)."

- “Observer members must carry a lifetime comment.”
- “APIs never transfer ownership by raw pointer.”

This keeps ownership documentation consistent across the codebase.

10.2.9 Summary

Documenting ownership clearly is mostly about **types and API design**:

- Borrow with `T&` / `T*` (and use `gsl::not_null` when null is invalid).
- Transfer ownership with `std::unique_ptr` or `std::shared_ptr` *by value*.
- Mark legacy raw owning pointers with `gsl::owner` and wrap them into RAII immediately.
- Add short lifetime comments only where the type cannot fully express the relationship.

10.3 Modern alternatives

10.3.1 What This Section Means by “Modern Alternatives”

Raw pointers are still useful as *observers*, but modern C++ offers better tools that make intent explicit:

- ownership vs observation
- nullable vs non-null
- single object vs sequence
- “maybe no result” vs “error with details”

The C++ Core Guidelines recommend expressing these semantics directly in interfaces, using types such as `unique_ptr`, `shared_ptr`, `not_null`, and `span`.

10.3.2 Alternative 1: Prefer Values and RAII Objects Over Dynamic Allocation

Before choosing any pointer, ask whether you need a pointer at all.

Value semantics (often the fastest and safest)

Return objects by value and rely on move semantics when needed (and RVO when applicable). The Core Guidelines explicitly list returning by value as the preferred alternative to returning owning raw pointers.

```
#include <vector>

std::vector<int> make_data() {
    std::vector<int> v(1024);
    // fill v...
    return v; // value return
}
```

RAII handles for non-memory resources

Model OS resources (file descriptors, handles, sockets, locks) as small RAII types, not raw pointers. This keeps acquisition and release in constructor/destructor pairs, and makes failure paths safe by default.

10.3.3 Alternative 2: Use Smart Pointers to Represent Ownership (Not "Because Pointers")

`std::unique_ptr` for exclusive ownership

If an object must live beyond a scope and needs dynamic lifetime, prefer `unique_ptr` to represent exactly one owner. The Core Guidelines recommend `unique_ptr` to transfer ownership where a pointer is needed.

```
#include <memory>

struct Task { void run(); };

std::unique_ptr<Task> make_task() {
    return std::make_unique<Task>();
}
```

`std::shared_ptr` for shared ownership

Use `shared_ptr` only when shared ownership is truly required. The Core Guidelines explicitly call out `shared_ptr` as the tool for sharing ownership.

```
#include <memory>

struct Service { void use() const; };

class Registry {
    std::shared_ptr<Service> svc_; // shares ownership
public:
    void install(std::shared_ptr<Service> s) { svc_ = std::move(s); }
};
```

Important: never transfer ownership via `T*`

This is the core rule: do not return or accept owning raw pointers to communicate lifetime transfer. Use value return or smart pointers instead.

10.3.4 Alternative 3: Express Non-Null Contracts with `not_null`

Raw pointers are nullable by default, which often forces defensive checks and makes contracts ambiguous. When null is not valid, say so.
The Core Guidelines recommend `not_null` to clarify intent and even allow optimizations by removing redundant null checks.

```
#include <gsl/gsl>

struct Engine { void tick(); };

void tick_engine(gsl::not_null<Engine*> e) {
    e->tick(); // no null checks needed by contract
}
```

Tooling support: Microsoft's Core Guidelines checks include diagnostics that suggest `gsl::not_null` when a pointer is never null-checked.

10.3.5 Alternative 4: Use Views for Sequences Instead of "Pointer + Size"

`std::span` for contiguous sequences (C++20)

`std::span` is a non-owning view over a contiguous sequence, with explicit size information and optional static extent.
It replaces the fragile "`T*` plus length" pattern.

```
#include <span>
#include <cstddef>

void scale(std::span<int> xs, int factor) {
    for (int& x : xs) x *= factor;
}

void call(int* p, std::size_t n) {
    scale(std::span<int>{p, n}); // explicit view
}
```

The Core Guidelines recommend `span<T>` to designate a half-open sequence.

`std::string_view` for read-only string views (C++17)

`std::basic_string_view` is a non-owning view of a constant contiguous character sequence.

```
#include <string_view>
#include <cstddef>

std::size_t count_colons(std::string_view s) {
    std::size_t n = 0;
    for (char c : s) if (c == ':') ++n;
    return n;
}
```

Safety reminder: `string_view` does not own its data. It is excellent for parameters, but returning it is only safe when the referenced storage outlives the view.

10.3.6 Alternative 5: Use Ranges and Views for General Iteration (C++20+)

For many APIs, the best alternative to raw pointers is not another pointer-like object, but a higher-level range abstraction.
The C++20 ranges library provides lightweight views that compose and avoid error-prone manual iteration and indexing.
When you need contiguous memory specifically, prefer `std::span`; when you need general iteration, consider accepting a range (or iterator pair) and documenting requirements.

10.3.7 Alternative 6: Replace "Nullable Return Pointer" with `std::optional` or `std::expected`

A classic legacy pattern is returning `T*` and using `nullptr` to mean "not found" or "failed". Modern code often separates:

- **no result** (normal outcome) from
- **error** (unexpected failure with an explanation)

Use `std::optional<T>` for "maybe a value"

If "no result" is not an error, return `std::optional<T>`. This avoids raw pointers entirely and makes the absence explicit.

```
#include <optional>

std::optional<int> parse_int(const char* s); // absence = parse failed
↪  (domain choice)
```

Use `std::expected<T,E>` for "value or error" (C++23)

`std::expected` represents either a value of type `T` or an error of type `E`, and it is never valueless.
This is often a better alternative than returning `nullptr` with an out-parameter error code.

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

std::expected<int, std::string> read_port() {
    // ...
    return std::unexpected("permission denied");
}
```

10.3.8 Alternative 7: Store "References" Safely Using `std::reference_wrapper`

C++ references cannot be reseated and cannot be stored directly in standard containers.
When you need "a rebindable reference-like handle" for observation,
`std::reference_wrapper<T>` is the standard tool.

```
#include <functional>
#include <vector>

struct Listener { void on_event(); };

void notify_all(std::vector<std::reference_wrapper<Listener>> listeners)
↪  {
    for (Listener& l : listeners) l.on_event();
}
```

This remains an **observer** mechanism: it does not own the object.

10.3.9 Alternative 8: Prefer Non-Owning Callable Views Over "Function Pointer + void*"

Legacy C APIs often pass callbacks as a function pointer plus a `void*` context pointer. Modern C++ increasingly uses type-safe callable wrappers.
C++26 adds `std::function_ref`: a **non-owning** wrapper of any callable object, explicitly modeling "observer semantics" for callables without forcing heap allocation like `std::function` might.

```
// C++26
#include <functional>

void for_each_item(std::function_ref<void(int)> f);

void example() {
    int sum = 0;
    for_each_item([&](int x) { sum += x; }); // borrowed callable
}
```

10.3.10 When Raw Pointers Still Win (and How to Keep Them Safe)

Even with modern alternatives, raw pointers remain reasonable:

- as observers (`T*`) where optional null is meaningful
- for low-level interop with C/OS APIs
- in hot paths where you deliberately avoid extra abstraction (but still document contracts)

When you do use them:

- never use `T*` to transfer ownership (use value return or smart pointers)
- use `not_null` when null is not valid
- use `span` / `string_view` for sequences instead of pointer arithmetic

10.3.11 Summary

Modern C++ reduces raw-pointer risk by replacing ambiguous pointer idioms with intent-revealing types:

- **Ownership**: values, `unique_ptr`, `shared_ptr`
- **Non-null contract**: `not_null`
- **Sequences**: `std::span`
- **String views**: `std::string_view`
- **Structured error results**: `std::expected`
- **Observer handles**: `reference_wrapper`, and (C++26) `function_ref`

Chapter 11

Resource Management Beyond Memory

11.1 Files

11.1.1 Files Are Resources, Not "Just I/O"

A file is a scarce operating-system resource: opening a file consumes a handle/descriptor, may lock data, and may carry security-sensitive state (permissions, access mode, inheritance across `exec`, etc.). The C++ Core Guidelines treat file handles as first-class resources and recommend RAII to guarantee timely release on all paths (normal return and exceptions).

11.1.2 The RAII Principle Applied to Files

The core rule

Acquire the file resource in a constructor (or factory), and release it in a destructor. This eliminates leaks and makes error paths safe.

Why “close in the destructor” is not enough by itself

Destructors cannot report failure in a reliable way (they must not throw during stack unwinding). For standard iostream file buffers, destruction will close the file if it is open, and any exceptions that occur during destruction (including from `close()`) are caught and not rethrown.
Therefore:

- Destruction is the **last line of defense** for releasing the handle.
- If you must detect close/flush failure, perform an explicit operation that can report errors (for example, check stream state and explicitly `close()` while still in a context where errors can be handled).

(When you call `basic_filebuf::close()` explicitly, exceptions thrown during internal calls are caught and rethrown *after* attempting to close, enabling error handling at the call site.)

11.1.3 Using Standard C++ File Streams Safely

`std::ifstream`, `std::ofstream`, `std::fstream`

These classes provide RAII for a file-backed stream buffer; they open a file and close it when the stream object is destroyed.

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

std::string read_first_line(const std::string& path) {
    std::ifstream in(path);          // acquires file resource
    std::string line;
```

```
    std::getline(in, line);          // may set failbit/eofbit
    return line;                     // file is closed in destructor
}
```

Error reporting model: flags by default, exceptions if you enable them

Iostreams typically record errors by setting state flags (e.g., `failbit`, `badbit`). If you want exceptions, you must opt in by setting the exception mask; failures then throw `std::ios_base::failure`.

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

std::string read_all_or_throw(const std::string& path) {
    std::ifstream in;
    in.exceptions(std::ios::failbit | std::ios::badbit); // opt-in
    ↪ throwing
    in.open(path);

    std::string data((std::istreambuf_iterator<char>(in)),
                     (std::istreambuf_iterator<char>()));
    return data; // destructor closes; close failure cannot be reliably
    ↪ reported here
}
```

Guidance:

- Use flag-checking style when you want explicit local error handling without exceptions.
- Use exception style when you want failures to propagate cleanly and be handled at a higher level.

Either way, RAII ensures the handle is released.

When close/flush correctness matters

If you must ensure data is written and detect problems, do an explicit check before the stream is destroyed:

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

void write_text_strict(const std::string& path, const std::string& text)
↪  {
    std::ofstream out(path, std::ios::binary);
    if (!out) throw std::runtime_error("open failed");

    out.write(text.data(), static_cast<std::streamsize>(text.size()));
    out.flush();
    if (!out) throw std::runtime_error("write/flush failed");

    out.close();  // explicit close while we can still report errors
    if (!out) throw std::runtime_error("close failed");
}
```

Rationale: destruction closes the file, but destruction cannot reliably surface failures.

11.1.4 Bridging to C APIs: `FILE*` with RAII

If you must call C APIs that use `FILE*` (`fopen`/`fclose`), wrap it immediately in a RAII handle, typically `std::unique_ptr` with a custom deleter. The Core Guidelines explicitly call out paired functions like `fopen`/`fclose` as classic RAII targets.

```
#include <cstdio>
#include <memory>
#include <stdexcept>
#include <string>

struct FileCloser {
    void operator()(std::FILE* f) const noexcept {
        if (f) std::fclose(f);
    }
};

using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file_or_throw(const std::string& path, const char* mode)
↪  {
    if (auto f = unique_file(std::fopen(path.c_str(), mode))) {
        return f;
    }
    throw std::runtime_error("fopen failed");
}
```

11.1.5 POSIX File Descriptors: A Minimal `unique_fd`

On POSIX systems, many low-level operations use an `int` file descriptor (`open`/`close`). A descriptor is also a resource and should be wrapped in RAII.

Close-on-exec is a security and correctness default

When spawning processes, descriptors can leak into child processes unless marked close-on-exec. On Linux/POSIX systems, `O_CLOEXEC` requests that behavior at open time.

```
#include <unistd.h>   // close
```

```
#include <fcntl.h>    // open, O_CLOEXEC
#include <stdexcept>

class unique_fd {
    int fd_ = -1;

public:
    unique_fd() noexcept = default;
    explicit unique_fd(int fd) noexcept : fd_(fd) {}

    unique_fd(const unique_fd&) = delete;
    unique_fd& operator=(const unique_fd&) = delete;

    unique_fd(unique_fd&& other) noexcept : fd_(other.fd_) { other.fd_ =
    ↪ -1; }
    unique_fd& operator=(unique_fd&& other) noexcept {
        if (this != &other) {
            reset();
            fd_ = other.fd_;
            other.fd_ = -1;
        }
        return *this;
    }

    ~unique_fd() noexcept { reset(); }

    int get() const noexcept { return fd_; }
    explicit operator bool() const noexcept { return fd_ >= 0; }

    void reset(int new_fd = -1) noexcept {
        if (fd_ >= 0) ::close(fd_);
        fd_ = new_fd;
```

```
    }
};

unique_fd open_readonly_or_throw(const char* path) {
    int fd = ::open(path, O_RDONLY | O_CLOEXEC);
    if (fd < 0) throw std::runtime_error("open failed");
    return unique_fd(fd);
}
```

This is the same RAII idea as `std::ifstream`: one clear owner that releases automatically.

11.1.6 Atomic Update Pattern: Write-Temp, Then Rename

For configuration files, state files, caches, and durable outputs, a robust pattern is:

1. write to a temporary file in the **same directory** (same filesystem)
2. flush and close it
3. rename it over the destination

Why rename is the right primitive (POSIX semantics)

On Linux, `rename()` atomically replaces `newpath` if it already exists, so there is no moment when `newpath` is missing for other processes. Also, rename cannot cross mounted filesystems (e.g., returns `EXDEV`), which is why the temp file must be on the same filesystem.

Standard C++ interface: `std::filesystem::rename`

C++17 provides `std::filesystem::rename`, specified "as if by POSIX rename".

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

namespace fs = std::filesystem;

void atomic_write_text(const fs::path& target, const std::string& text) {
    const fs::path dir = target.parent_path();
    const fs::path tmp = dir / (target.filename().string() + ".tmp");

    {
        std::ofstream out(tmp, std::ios::binary | std::ios::trunc);
        if (!out) throw std::runtime_error("open tmp failed");
        out.write(text.data(),
        ↪  static_cast<std::streamsize>(text.size()));
        out.flush();
        if (!out) throw std::runtime_error("write/flush tmp failed");
        out.close();
        if (!out) throw std::runtime_error("close tmp failed");
    }

    fs::rename(tmp, target); // atomic replacement on POSIX-like
    ↪  semantics
}
```

Practical notes:

- The rename step is atomic with respect to path replacement, but it does not automatically make data durable against power loss; durability requirements depend on the OS/filesystem and may require explicit syncing at the OS layer.
- Always handle cleanup for the temporary file on failure paths (e.g., best-effort remove).

11.1.7 Filesystem Facilities for File Management

C++17 `std::filesystem` provides standardized operations for paths and files.

Paths are syntactic objects

`std::filesystem::path` represents paths and manipulates them syntactically; a `path` may point to a non-existing object.

Copying files and controlling behavior

Use `copy_file`, `copy`, and `copy_options` instead of ad-hoc loops when appropriate.

Permissions

Use `std::filesystem::permissions` when you need to manipulate file permissions portably at the C++ layer.

```
#include <filesystem>

namespace fs = std::filesystem;

void make_readonly(const fs::path& p) {
    fs::permissions(p,
        fs::perms::owner_write,
        fs::perm_options::remove);
}
```

11.1.8 Checklist: File Safety Rules That Prevent Common Bugs

RAII and ownership

- Prefer `std::ifstream/ofstream/fstream` for standard file I/O; they close automatically.
- For C/POSIX handles (`FILE*`, `int fd`), wrap immediately in an owning RAII type.

Failure handling

- If you need exceptions, enable them on streams and handle `std::ios_base::failure`.
- If you must detect write/flush/close failure, do explicit checks before destruction; destruction cannot reliably report errors.

Atomic replace

- For "replace file content safely", write temp + rename within the same filesystem.

Security hygiene

- Prefer close-on-exec when opening descriptors on POSIX (`O_CLOEXEC`) to avoid leaking handles to child processes.

11.1.9 Summary

Files are resources and should be managed like any other scarce resource:

- Use RAII so file handles close on all paths.
- For standard streams, understand the error model: flags by default, exceptions when enabled.

- When close/flush errors matter, check explicitly before destruction because destructors cannot reliably report failure.
- For safe updates, use write-temp + rename; rename provides atomic replacement and requires same filesystem.

11.2 Locks

11.2.1 Locks Are Resources (Just Like Files and Memory)

A mutex (or shared mutex) is an operating-system or runtime synchronization resource. Lock ownership must be acquired and released reliably, on every control path. This is exactly the kind of problem RAII solves: acquire in a constructor, release in a destructor.
In safe C++ concurrency, the fundamental rule is:

> Never manually "pair" `lock()` and `unlock()` in multiple branches; instead, bind the lock lifetime to a scope.

11.2.2 The Core RAII Lock Wrappers

`std::lock_guard`: the simple, fastest default for one mutex

`std::lock_guard<Mutex>` provides a minimal RAII mechanism: it locks in the constructor and unlocks in the destructor, and it cannot be copied.
Use it when:

- you lock exactly one mutex
- you hold the lock for the entire lexical scope
- you do not need to unlock early

```cpp
#include <mutex>
#include <vector>

class SafeQueue {
    std::mutex m_;
    std::vector<int> v_;

public:
    void push(int x) {
        std::lock_guard<std::mutex> lk(m_); // lock for this scope
        v_.push_back(x);
    }
};
```

`std::unique_lock`: flexible ownership (needed for condition variables)

`std::unique_lock<Mutex>` is the general-purpose wrapper: it supports deferred locking, timed locking, try-locking, manual unlock/relock, ownership transfer (move), and is designed to work with condition variables.
Use it when:

- you must unlock before leaving the scope (lock splitting)
- you need `try_lock`/timed locking
- you use `std::condition_variable` (`wait` requires a `unique_lock`)

```cpp
#include <condition_variable>
#include <mutex>
#include <queue>

class BlockingQueue {
```

```
    std::mutex m_;
    std::condition_variable cv_;
    std::queue<int> q_;

public:
    void push(int x) {
        {
            std::lock_guard<std::mutex> lk(m_);
            q_.push(x);
        }
        cv_.notify_one();
    }

    int pop() {
        std::unique_lock<std::mutex> lk(m_);
        cv_.wait(lk, [&] { return !q_.empty(); }); // handles spurious
        ↪ wakeups
        int x = q_.front();
        q_.pop();
        return x;
    }
};
```

The predicate form of `wait` is specified as a loop that re-checks the predicate to handle spurious wakeups.

`std::scoped_lock`: RAII for one or more mutexes (deadlock-avoidance helper)

`std::scoped_lock` (C++17) is a RAII wrapper that can own zero or more mutexes for the duration of a scope.
Its key advantage is multi-locking in one expression, typically using a deadlock-avoidance strategy internally.

```
#include <mutex>

struct Pair {
    std::mutex a;
    std::mutex b;
    int value = 0;
};

void update_both(Pair& p) {
    std::scoped_lock lk(p.a, p.b); // lock both without manual ordering
    ++p.value;
}
```

11.2.3 Avoiding Deadlocks When Multiple Locks Are Involved

Use `std::lock` or `std::scoped_lock`, not ad-hoc ordering

When you need to lock multiple mutexes, doing it manually is error-prone (inconsistent lock order across code paths leads to deadlock). `std::lock` locks multiple lockables using a deadlock-avoidance algorithm.
A standard pattern is:

1. lock multiple mutexes using `std::lock`
2. adopt them into RAII guards using `std::adopt_lock`

```
#include <mutex>

void safe_transfer(std::mutex& m1, std::mutex& m2) {
    std::lock(m1, m2); // deadlock avoidance algorithm
    std::lock_guard<std::mutex> lk1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> lk2(m2, std::adopt_lock);
```

```
    // critical section holding both locks
}
```

The adoption mechanism is standardized via lock tags: `std::adopt_lock` assumes the calling thread already owns the mutex; `std::defer_lock` means “do not lock yet”; `std::try_to_lock` means “try without blocking”.
With C++17, `std::scoped_lock` is often the simplest way to express multi-locking safely.

Design rule: reduce lock coupling

Even with safe multi-locking, acquiring multiple locks increases contention risk and can harm scalability. Prefer designs where:

- each invariant is protected by exactly one mutex
- lock ordering is rare and well-documented when unavoidable

11.2.4 Shared Locks for Read-Mostly Data

`std::shared_mutex` / `std::shared_timed_mutex`

For read-mostly workloads, a shared mutex allows:

- **shared (reader)** ownership by multiple threads concurrently
- **exclusive (writer)** ownership by one thread

`std::shared_lock`: RAII for shared ownership

`std::shared_lock` (C++14) is the shared-mode equivalent of `unique_lock`: it supports deferred locking, timed locking, and ownership transfer, but locks the associated shared mutex in shared mode.

```
#include <shared_mutex>
#include <string>

class Config {
    mutable std::shared_mutex m_;
    std::string value_;

public:
    std::string read() const {
        std::shared_lock lk(m_);    // shared (reader) lock
        return value_;
    }

    void write(std::string v) {
        std::unique_lock lk(m_);    // exclusive (writer) lock
        value_ = std::move(v);
    }
};
```

11.2.5 Timed Locking and Try-Locking

Timed mutexes (e.g., `std::timed_mutex`) support `try_lock_for` and `try_lock_until`.

Important correctness notes:

- `try_lock` may fail spuriously (return false even if the mutex is not locked).
- Calling `try_lock` on `std::mutex` from a thread that already owns it is undefined behavior.

A safe pattern is to treat try-lock as “best effort” and always have a correct fallback:

```
#include <mutex>

bool try_update(std::mutex& m, int& x) {
    if (m.try_lock()) {                  // may fail spuriously
        std::lock_guard<std::mutex> lk(m, std::adopt_lock);
        ++x;
        return true;
    }
    return false;
}
```

11.2.6 Common Lock Misuse Patterns (and How RAII Prevents Them)

Misuse 1: Manual unlock on multiple paths

```
#include <mutex>

void bad(std::mutex& m, bool ok) {
    m.lock();
    if (!ok) { m.unlock(); return; }
    // ...
    m.unlock();
}
```

Fix: RAII makes all exits safe:

```
#include <mutex>

void good(std::mutex& m, bool ok) {
    std::lock_guard<std::mutex> lk(m);
    if (!ok) return;
    // ...
}
```

Misuse 2: Holding locks across blocking I/O or callbacks

This is a design hazard, not just a style issue:

- I/O can block unpredictably and increases contention
- callbacks can re-enter code that tries to lock the same mutex (risking deadlock)

Guidance: keep critical sections small; copy the needed state out under the lock, then perform I/O/callbacks after unlocking.

Misuse 3: Using `recursive_mutex` as a "quick fix"

`std::recursive_mutex` and `std::recursive_timed_mutex` allow the same thread to lock the same mutex multiple times.
They can be useful for specific re-entrant designs, but they frequently hide design problems: uncontrolled re-entrancy, unclear invariants, and accidental deep lock nesting. Prefer redesign (split responsibilities, avoid re-entrant locking) when possible.

11.2.7 Lock Ownership as an Explicit State (Lock Tags)

Lock tags are standardized to express ownership strategy precisely:

- `std::defer_lock`: construct the lock wrapper without locking yet
- `std::try_to_lock`: try locking without blocking
- `std::adopt_lock`: adopt an already-locked mutex

Example: defer lock, then coordinate multi-locking with `std::lock`:

```cpp
#include <mutex>

void example(std::mutex& a, std::mutex& b) {
    std::unique_lock<std::mutex> lk1(a, std::defer_lock);
    std::unique_lock<std::mutex> lk2(b, std::defer_lock);

    std::lock(lk1, lk2); // locks both without deadlock
    // critical section
}
```

11.2.8 Forward-Looking Note for C++26 and Beyond

The concurrency library continues to evolve. For example, a 2025 WG21 proposal introduces `std::multi_lock` as a RAII type combining multi-mutex locking with the flexibility of `unique_lock` (deferred locking, try/timed locking, ownership transfer), addressing limitations of `scoped_lock`.
This is not standardized as of this writing; treat it as a direction of travel, not a guaranteed facility.

11.2.9 Summary

- Locks are resources: manage them with RAII to guarantee release on all paths.
- Use `lock_guard` for simple scoped locking of one mutex.
- Use `unique_lock` when you need flexibility or condition variables.
- Use `scoped_lock` (or `std::lock` + adopt) to acquire multiple locks without deadlock.

- For read-mostly data, use `shared_mutex` + `shared_lock` for shared reads and exclusive writes.

- Treat try-lock and timed locking as best-effort primitives and respect their specified edge cases (spurious failure and UB for re-locking `std::mutex` by the owning thread).

11.3 Sockets

11.3.1 Sockets Are Resources (Descriptors and Kernel State)

A socket is not "just an API call"; it is an operating-system resource that consumes:

- a handle/descriptor that must be released

- kernel memory and protocol state (TCP state machine, buffers, queues)

- optional OS-level configuration (non-blocking mode, timeouts, keepalive, linger)

On POSIX systems, sockets are file descriptors created by `socket()` and configured/used via `bind`, `connect`, `listen`, `accept`, `send`/`recv`, etc. On Windows, sockets are Winsock handles and must be closed with `closesocket()` (not `CloseHandle`).
The RAII goal is identical to files:

> Once a socket is successfully created, it must be released exactly once, on every path (success, error, early return, exception).

This is a textbook RAII use case: acquire in a constructor/factory, release in a destructor.

11.3.2 Standard C++ Reality: No Portable Sockets in `std::` Yet

As of the C++11 → C++26 timeframe, the ISO C++ standard library does not provide a fully standardized, portable socket API in the `std::` namespace. There is a published *Networking TS* (Extensions for Networking) based on Asio, and implementations may expose it as an experimental library.
Practical implication for professional code:

- Use a proven networking library (commonly Asio/Boost.Asio) *or*
- Use platform APIs (POSIX / Winsock) behind a small RAII wrapper and a narrow interface.

This section focuses on the second approach: RAII wrappers that make platform APIs safe to use.

11.3.3 Core RAII Pattern: A Move-Only Socket Owner

Design requirements for a safe socket handle type

A correct socket-owning type should:

- be **move-only** (unique ownership of the underlying handle)
- close the handle in a **noexcept** destructor
- expose `get()` for interop and `release()`/`reset()` for ownership transfer
- use **invalid sentinel** values consistently (`-1` on POSIX; `INVALID_SOCKET` on Windows)

POSIX: `unique_socket` as `unique_fd` specialized for sockets

On POSIX, a socket is a file descriptor created by `socket(2)` and closed with `close(2)`. A minimal RAII owner can use `close` the same way as a file descriptor.

```
#include <unistd.h>   // close
#include <sys/socket.h>

class unique_socket {
    int fd_ = -1;

public:
    unique_socket() noexcept = default;
    explicit unique_socket(int fd) noexcept : fd_(fd) {}

    unique_socket(const unique_socket&) = delete;
    unique_socket& operator=(const unique_socket&) = delete;

    unique_socket(unique_socket&& other) noexcept : fd_(other.fd_) {
    ↪  other.fd_ = -1; }
    unique_socket& operator=(unique_socket&& other) noexcept {
        if (this != &other) {
            reset();
            fd_ = other.fd_;
            other.fd_ = -1;
        }
        return *this;
    }

    ~unique_socket() noexcept { reset(); }

    int get() const noexcept { return fd_; }
    explicit operator bool() const noexcept { return fd_ >= 0; }
```

```
    int release() noexcept {
        int tmp = fd_;
        fd_ = -1;
        return tmp;
    }

    void reset(int new_fd = -1) noexcept {
        if (fd_ >= 0) ::close(fd_);
        fd_ = new_fd;
    }
};
```

Windows: `unique_socket` using `closesocket`

On Windows, a socket must be released with `closesocket`, and the handle can be reused immediately after close, so using a stale socket value is unsafe.

```
#ifdef _WIN32
#include <winsock2.h>

class unique_socket {
    SOCKET s_ = INVALID_SOCKET;

public:
    unique_socket() noexcept = default;
    explicit unique_socket(SOCKET s) noexcept : s_(s) {}

    unique_socket(const unique_socket&) = delete;
    unique_socket& operator=(const unique_socket&) = delete;

    unique_socket(unique_socket&& other) noexcept : s_(other.s_) {
    ↪  other.s_ = INVALID_SOCKET; }
```

```
    unique_socket& operator=(unique_socket&& other) noexcept {
        if (this != &other) {
            reset();
            s_ = other.s_;
            other.s_ = INVALID_SOCKET;
        }
        return *this;
    }

    ~unique_socket() noexcept { reset(); }

    SOCKET get() const noexcept { return s_; }
    explicit operator bool() const noexcept { return s_ !=
    ↪  INVALID_SOCKET; }

    SOCKET release() noexcept {
        SOCKET tmp = s_;
        s_ = INVALID_SOCKET;
        return tmp;
    }

    void reset(SOCKET news = INVALID_SOCKET) noexcept {
        if (s_ != INVALID_SOCKET) ::closesocket(s_);
        s_ = news;
    }
};
#endif
```

11.3.4 Shutdown vs Close: Correct Semantics and Correct Intent

`shutdown()` is not `close()`

Both POSIX and Winsock define `shutdown` as a way to disable sending and/or receiving on a connected socket:

- POSIX: `shutdown()` "causes all or part of a full-duplex connection" to be shut down.
- Winsock: `shutdown` does not free resources; the socket is not released until `closesocket` is called.

Professional rule:

> Use `shutdown` to signal end-of-stream (half-close), but always call close/closesocket to release the handle.

Half-close: "I will not send more"

On TCP:

- POSIX: `shutdown(fd, SHUT_WR)` disables further writes (peer will see EOF when it reads after data is drained).
- Winsock: `shutdown(s, SD_SEND)` disallows further sends; for TCP a FIN is sent after pending data is transmitted and acknowledged.

This is useful in request/response protocols where the peer expects an explicit end-of-request.

11.3.5 Avoiding Common Socket Resource Bugs

Bug 1: Leaking sockets on early return or exception

Without RAII, every failure path must remember to close the socket. With RAII, cleanup is automatic:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdexcept>

unique_socket connect_ipv4_or_throw(const char* ip, unsigned short port)
↪  {
    int fd = ::socket(AF_INET, SOCK_STREAM, 0);    // may fail
    if (fd < 0) throw std::runtime_error("socket failed");
    unique_socket s(fd);                            // now owned (will
    ↪  close on exceptions)

    sockaddr_in addr{};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    if (::inet_pton(AF_INET, ip, &addr.sin_addr) != 1)
        throw std::runtime_error("inet_pton failed");

    if (::connect(s.get(), reinterpret_cast<sockaddr*>(&addr),
    ↪  sizeof(addr)) != 0)
        throw std::runtime_error("connect failed");

    return s; // move out
}
```

The ownership transfer at return is explicit and safe because the wrapper is move-only.

Bug 2: Closing a socket while another thread is using it

On Windows, the documentation explicitly warns: a client must never call `closesocket` concurrently with another Winsock function call on the same socket.
Professional rule:

Socket lifetime must be coordinated. A handle must not be closed while another thread may still call `send`/`recv` on it.

Typical solution patterns:

- confine the socket to one I/O thread (message passing)
- reference-count a higher-level "connection" object and ensure shutdown sequencing
- use cancellation mechanisms provided by your event loop/library (e.g., asynchronous I/O framework)

Bug 3: Ignoring SIGPIPE / EPIPE on POSIX

On many POSIX systems, writing to a closed peer on a stream socket can raise `SIGPIPE` (which by default terminates the process). POSIX defines `MSG_NOSIGNAL` for `sendmsg`/`send` to request that `SIGPIPE` not be generated, while still returning `EPIPE`. A robust send wrapper often includes an option to suppress signal-based termination:

```
#include <sys/socket.h>
#include <cerrno>

ssize_t safe_send_nosignal(int fd, const void* buf, size_t len) {
    // MSG_NOSIGNAL is a POSIX extension (present on Linux); portability
    ↪  requires feature checks.
    return ::send(fd, buf, len, MSG_NOSIGNAL);
}
```

Portability note: behavior varies across Unix-like systems; a cross-platform networking library typically abstracts these differences.

11.3.6 Close-on-exec Hygiene (POSIX)

If your process spawns other processes, file descriptors (including sockets) may be inherited across `exec` unless marked close-on-exec. For sockets created on Linux/POSIX, setting `CLOEXEC` at creation time is a best practice. `socket` supports `SOCK_CLOEXEC` as a type flag on Linux.

```
int fd = ::socket(AF_INET, SOCK_STREAM | SOCK_CLOEXEC, 0); // Linux:
↪  close-on-exec
```

If `SOCK_CLOEXEC` is unavailable, set close-on-exec via `fcntl` immediately after creation (still has a small race window).

11.3.7 Graceful Shutdown Strategy

A professional shutdown sequence depends on your protocol, but a common safe pattern for TCP is:

1. Stop initiating new writes.
2. `shutdown(SHUT_WR)` (or `shutdown(SD_SEND)`) to signal end-of-stream.
3. Continue reading until EOF (peer closes) or until protocol completion.
4. Close the socket handle (RAII destructor or explicit reset).

The key correctness point remains: `shutdown` changes stream directionality, but does not release resources; release still requires `close/closesocket`.

11.3.8 Error Handling: Platform Error Codes vs C++ Error Types

Socket APIs commonly report errors via:

- POSIX: return value + `errno`
- Winsock: return value + `WSAGetLastError()` (not shown here)

For modern C++ interfaces, prefer returning:

- `std::error_code` alongside a value, or
- `std::expected<T, std::error_code>` (C++23) to cleanly model "value or error"

This keeps RAII (resource safety) separate from error reporting (operational correctness).

11.3.9 Recommended Practice: Library-Based Networking When Portability Matters

Because portable socket behavior includes subtle OS differences (signals, close semantics, non-blocking modes, event models), many teams adopt a mature library. The Networking TS itself is based on Asio, reflecting the committee's recognition of the complexity of standardizing networking.
Even when you use a library, the guiding rule remains:

> Sockets are resources; ownership must be explicit and lifetime must be deterministic.

11.3.10 Summary

- A socket is an OS resource and must be released exactly once; RAII is the correct C++ model.
- Standard C++ still lacks a fully standardized socket API; the Networking TS exists as experimental extensions based on Asio.
- Use move-only RAII wrappers (`unique_socket`) to prevent leaks and double-close.
- `shutdown` changes stream directionality but does not release the handle; you must still close/closesocket.
- Coordinate lifetime across threads; do not close a socket concurrently with other calls on it (explicitly warned by Winsock).
- On POSIX, avoid process termination on broken pipes by using facilities such as `MSG_NOSIGNAL` where available.

11.4 OS handles

11.4.1 What an "OS handle" really is

An OS handle is a capability-like token issued by the operating system that refers to a kernel object or kernel-managed state. Examples include:

- POSIX file descriptors (`int`): files, sockets, pipes, epoll, eventfd, etc.
- Windows `HANDLE`: files, events, mutexes, processes, threads, registry keys (via specific handle types), and more

Handles are **resources**. They are scarce, they have lifetime rules, and leaking them can break systems just as surely as leaking memory. The C++ Core Guidelines explicitly recommend managing resources automatically using RAII and resource handles.

11.4.2 The RAII contract for handles

Core invariant

A correct owner of an OS handle must satisfy:

- The handle is released **exactly once**.
- Release happens on **every** control path (normal return, early return, exception).
- After release, the handle value must be treated as **invalid** (do not reuse it).

This is exactly the kind of invariant RAII is designed to enforce.

Why raw handle values are dangerous

Raw handle values (`int fd` or `HANDLE h`) are just integers/pointers. Nothing in the type system tells you:

- who owns it
- whether it must be closed
- whether it is shared or unique
- whether it is valid

Therefore: if a handle has **ownership**, put it inside a **move-only RAII wrapper**.

11.4.3 POSIX: file descriptors (`int`)

Semantics you must remember

On POSIX systems, `close()` releases a file descriptor so it no longer refers to an open file description, and the integer value may be reused by the OS for future opens. This reuse is the reason dangling descriptors are so dangerous: a stale `fd` might suddenly refer to a different resource.

Close-on-exec hygiene

By default, file descriptors may remain open across `execve` unless marked close-on-exec. Linux provides `O_CLOEXEC` at open time to avoid the classic race of setting the flag later via `fcntl`.

A minimal move-only `unique_fd`

```
#include <unistd.h>   // close
#include <fcntl.h>    // open, O_CLOEXEC
#include <stdexcept>

class unique_fd {
    int fd_ = -1;

public:
    unique_fd() noexcept = default;
    explicit unique_fd(int fd) noexcept : fd_(fd) {}

    unique_fd(const unique_fd&) = delete;
    unique_fd& operator=(const unique_fd&) = delete;
```

```
    unique_fd(unique_fd&& other) noexcept : fd_(other.fd_) { other.fd_ =
    ↪ -1; }
    unique_fd& operator=(unique_fd&& other) noexcept {
        if (this != &other) {
            reset();
            fd_ = other.fd_;
            other.fd_ = -1;
        }
        return *this;
    }

    ~unique_fd() noexcept { reset(); }

    int get() const noexcept { return fd_; }
    explicit operator bool() const noexcept { return fd_ >= 0; }

    int release() noexcept {
        int tmp = fd_;
        fd_ = -1;
        return tmp;
    }

    void reset(int new_fd = -1) noexcept {
        if (fd_ >= 0) ::close(fd_);
        fd_ = new_fd;
    }
};

unique_fd open_readonly_or_throw(const char* path) {
    int fd = ::open(path, O_RDONLY | O_CLOEXEC);
    if (fd < 0) throw std::runtime_error("open failed");
    return unique_fd(fd);
```

```
}
```

Key properties:

- Move-only enforces single ownership.
- Destructor is `noexcept` and always releases the descriptor if valid.
- `release()` transfers ownership intentionally.

11.4.4 Windows: `HANDLE` and `CloseHandle`

What `CloseHandle` does

Windows kernel objects are reference-counted by handles. `CloseHandle` invalidates the handle and decrements the object's handle count. When the last handle is closed, the object is removed from the system (subject to the object's own retention rules).
Professional consequence:

> If you own a `HANDLE`, you must call `CloseHandle` exactly once.

Do not guess the correct "close" function

Windows uses different close functions for different handle families (for example, Winsock sockets use `closesocket`). Only use `CloseHandle` for the handle types that specify it.

A minimal move-only `unique_handle`

```
#ifdef _WIN32
#define NOMINMAX
#include <windows.h>
```

```
class unique_handle {
    HANDLE h_ = nullptr;

public:
    unique_handle() noexcept = default;
    explicit unique_handle(HANDLE h) noexcept : h_(h) {}

    unique_handle(const unique_handle&) = delete;
    unique_handle& operator=(const unique_handle&) = delete;

    unique_handle(unique_handle&& other) noexcept : h_(other.h_) {
    ↪  other.h_ = nullptr; }
    unique_handle& operator=(unique_handle&& other) noexcept {
        if (this != &other) {
            reset();
            h_ = other.h_;
            other.h_ = nullptr;
        }
        return *this;
    }

    ~unique_handle() noexcept { reset(); }

    HANDLE get() const noexcept { return h_; }
    explicit operator bool() const noexcept { return h_ != nullptr && h_
    ↪  != INVALID_HANDLE_VALUE; }

    HANDLE release() noexcept {
        HANDLE tmp = h_;
        h_ = nullptr;
        return tmp;
    }
```

```
    void reset(HANDLE nh = nullptr) noexcept {
        if (h_ != nullptr && h_ != INVALID_HANDLE_VALUE) {
            ::CloseHandle(h_);
        }
        h_ = nh;
    }
};
#endif
```

Notes:

- Some Win32 APIs use `INVALID_HANDLE_VALUE` rather than `nullptr` to report failure. The wrapper treats both as invalid by policy.
- Destructors must not throw; `CloseHandle` reports failure via return value, so this wrapper ignores it. If close failure matters for your resource type, provide an explicit `close()` member that can report errors.

11.4.5 Encoding ownership with `std::unique_ptr` (when appropriate)

Custom deleters

`std::unique_ptr<T, D>` is a general-purpose RAII building block: it calls the stored deleter when destroyed. It is a good fit for C APIs that return pointers (e.g., `FILE*`) or opaque pointer-like handles.
For Windows `HANDLE`, a dedicated wrapper type is often clearer than forcing the handle into a pointer-shaped type. However, the underlying RAII principle is the same.

11.4.6 Ownership transfer and sharing rules

Unique ownership by default

Design APIs so there is one clear owner of a handle:

- Returning a `unique_fd` / `unique_handle` transfers ownership explicitly (move).
- Passing a handle wrapper by value transfers ownership into the callee.

Borrowing a handle vs owning a handle

A borrowed handle is a non-owning view:

- POSIX: `int fd` parameter often means "borrowed" (callee must not close).
- Windows: `HANDLE` parameter often means "borrowed" (callee must not close), unless the API explicitly states it takes ownership.

To avoid ambiguity, adopt a project rule:

> Functions that **close** must accept an owning wrapper type; functions that **use** accept a borrowed raw handle.

This directly follows the Core Guidelines goal of making ownership explicit via types and RAII.

11.4.7 Common OS-handle failure modes (and how RAII prevents them)

Leak on early return or exception

```
// bad: manual close is easy to miss
int fd = ::open(path, O_RDONLY);
```

```
if (fd < 0) return -1;
if (!precondition()) return -2;  // leak

// good: RAII closes on all paths
unique_fd fd2 = open_readonly_or_throw(path);
if (!precondition()) return -2;  // safe: fd2 destructor closes
```

Double-close and use-after-close

Because OS may reuse handle values, a stale handle can appear valid but refer to a different object. Move-only wrappers and `reset()`/`release()` patterns reduce the chance of accidental reuse by clearing the stored value.

Inheritance across exec (POSIX)

Using `O_CLOEXEC` at open time avoids unintentionally leaking descriptors to child processes.

11.4.8 Checklist: Handle safety rules

- Treat every handle as a resource: it must have a clearly documented owner.
- Represent ownership with a move-only RAII type (`unique_fd`, `unique_handle`).
- Ensure the destructor is `noexcept` and always releases when valid.
- Provide `get()` for borrowing, and `release()`/`reset()` for explicit ownership transfer.
- On POSIX, prefer `O_CLOEXEC` for descriptors that must not cross `exec`.

- On Windows, call the correct close function for the handle family; for many kernel handles this is `CloseHandle`.

11.5 Transaction-style RAII

11.5.1 What “transaction-style RAII” means

Transaction-style RAII is the pattern of structuring a multi-step operation so that it has **commit-or-rollback semantics**:

- if the operation completes successfully, changes are *committed*
- if the operation exits early (error, exception, return), partial effects are *rolled back*

This idea is closely related to the **strong exception-safety guarantee**: either the operation succeeds completely, or it has no observable side effects.
RAII is the mechanism that makes this reliable: rollback actions are tied to scope exit, so they run automatically even when control flow is non-linear (exceptions, early returns).

11.5.2 Why you need it beyond “release on scope exit”

Simple RAII answers: *“Will the resource be released?”* Transaction-style RAII answers the harder question: *“Will the system be left in a correct state if we fail halfway?”*
Typical failure-sensitive operations:

- updating multiple related data structures (must maintain invariants)
- changing external state (filesystem, sockets, configuration, database-like files)
- performing sequences of OS calls that must be undone on failure

11.5.3 The three scope-guard forms: always, on-fail, on-success

A standardized and widely implemented scope-guard design is described in WG21 paper P0052R10, which specifies three scope guard class templates: `scope_exit`, `scope_fail`, and `scope_success`.

- **`scope_exit`**: run an action whenever the scope is exited
- **`scope_fail`**: run an action only if the scope exits because of an exception
- **`scope_success`**: run an action only if the scope exits normally

Many ecosystems provide these facilities either via the Library Fundamentals TS (as `std::experimental::scope_*`) or via Boost.Scope.

Why `scope_fail`/`scope_success` require `std::uncaught_exceptions`

The on-fail/on-success distinction is typically implemented by sampling the number of uncaught exceptions on entry, then checking it again on exit. The P0052R10 design explicitly uses this mechanism, so these forms depend on `std::uncaught_exceptions()` (available since C++17).

11.5.4 A practical approach across C++11 → C++26

Because the exact availability may vary by standard library and compiler, the safest guidance is:

- If your environment provides `std::experimental::scope_*` or Boost.Scope, use it.
- Otherwise, implement a minimal `scope_exit` locally (easy, safe, and works in C++11).

A minimal C++11 `scope_exit`-style guard (always-on-exit):

```
#include <utility>

template <class F>
class scope_exit {
    F f_;
    bool active_ = true;

public:
    explicit scope_exit(F f)
    ↪  noexcept(std::is_nothrow_move_constructible<F>::value)
        : f_(std::move(f)) {}

    scope_exit(const scope_exit&) = delete;
    scope_exit& operator=(const scope_exit&) = delete;

    scope_exit(scope_exit&& other)
    ↪  noexcept(std::is_nothrow_move_constructible<F>::value)
        : f_(std::move(other.f_)), active_(other.active_) {
        other.active_ = false;
    }

    ~scope_exit() noexcept {
        if (active_) f_();
    }

    void release() noexcept { active_ = false; }
};

template <class F>
scope_exit<F> make_scope_exit(F f) {
    return scope_exit<F>(std::move(f));
```

```
}
```

This matches the fundamental scope-guard idea described in the P0052 design (a callable stored and invoked on scope exit).

11.5.5 Pattern 1: “rollback unless committed”

This is the most common transaction-style RAII form:

- install a rollback action immediately
- do the work
- mark the guard as committed (disable rollback)

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

void push_two_or_nothing(std::vector<int>& v, int a, int b) {
    const std::size_t old_size = v.size();

    auto rollback = make_scope_exit([&] {
        v.resize(old_size);          // rollback to original state
    });

    v.push_back(a);                 // may throw (allocation)
    v.push_back(b);                 // may throw (allocation)

    rollback.release();             // commit
}
```

This provides *commit-or-rollback* behavior for the container’s size. The same idea scales to multiple invariants: capture the minimal state needed to restore correctness.

11.5.6 Pattern 2: Strong guarantee via “stage, then commit” (copy-and-swap family)

When an operation modifies an object’s internal state in many steps, a robust way to get strong guarantees is: build the new state in a temporary, then commit in one non-throwing swap (or equivalent atomic step).
A classic example is copy assignment using copy-and-swap, which (in the canonical form) provides a strong guarantee: either assignment succeeds, or the target object remains unchanged.

```
#include <utility>

struct Buffer {
    int* p = nullptr;
    std::size_t n = 0;

    // assume proper ctor/dtor omitted for brevity

    friend void swap(Buffer& a, Buffer& b) noexcept {
        using std::swap;
        swap(a.p, b.p);
        swap(a.n, b.n);
    }

    Buffer& operator=(Buffer other) noexcept { // copy or move constructs
     ↪  'other'
        swap(*this, other);                    // commit step
        return *this;                          // old state released with
         ↪  'other'
    }
};
```

The key concept is transactional:

- construct a new candidate state first (may throw)
- commit by swapping states (ideally `noexcept`)

This style is consistent with strong exception safety as “commit or rollback”.

11.5.7 Pattern 3: External transactions (filesystem) with compensating actions

For external state, you often cannot undo everything perfectly, but you can still design **compensating actions** that restore a safe and expected state.
A common file-update transaction:

- write to a temporary file
- on success: rename into place (commit)
- on failure: delete the temporary file (rollback)

```
#include <filesystem>
#include <fstream>
#include <stdexcept>
#include <string>

namespace fs = std::filesystem;

void write_config_transactionally(const fs::path& target, const
↪  std::string& text) {
    const fs::path dir = target.parent_path();
    const fs::path tmp = dir / (target.filename().string() + ".tmp");
```

```
    auto rollback = make_scope_exit([&] {
        std::error_code ec;
        fs::remove(tmp, ec); // best-effort rollback (do not throw from
        ↪  rollback)
    });

    {
        std::ofstream out(tmp, std::ios::binary | std::ios::trunc);
        if (!out) throw std::runtime_error("open tmp failed");

        out.write(text.data(),
        ↪  static_cast<std::streamsize>(text.size()));
        out.flush();
        if (!out) throw std::runtime_error("write/flush tmp failed");
    }

    fs::rename(tmp, target);  // commit step (may throw)
    rollback.release();       // commit: keep file, no cleanup
}
```

This is transaction-style RAII even though the "transaction" is not a database transaction: you still have a clear rollback action and a clear commit point.

11.5.8 Design rules for correct transaction-style RAII

Rule 1: Install rollback first

Create the rollback guard immediately after acquiring or beginning a multi-step change. If you wait, an exception can occur before rollback is installed.

Rule 2: Rollback actions must not throw

Rollback typically runs during exception unwinding, and throwing from a destructor during unwinding leads to termination. Therefore, rollback actions should be `noexcept` by design (best-effort cleanup, error logging, or error-code APIs).
This is the same reasoning behind general guidance about not letting cleanup throw.

Rule 3: Make commit explicit and single

The commit point should be obvious:

- `rollback.release()` or `commit()` call
- a single swap/rename step

Avoid "half commits" spread across many lines; they are hard to reason about.

Rule 4: Keep rollback minimal

Capture only what you need to restore invariants:

- old size, old pointer, old handle
- temp file path to delete
- previous mode/flags to restore

Large rollbacks are themselves failure-prone.

Rule 5: Prefer strong guarantee when it is cheap, basic guarantee when it is not

The strong guarantee is desirable but may require extra buffering, copying, or staging. The Boost exception-safety documentation describes the strong guarantee as full commit-or-rollback semantics and notes that it may require additional work.

Professional practice:

- if you can stage cheaply (small objects, small buffers), do it
- if staging is expensive, at least preserve invariants and avoid leaks (basic guarantee), and document the behavior

11.5.9 Summary

Transaction-style RAII combines two ideas:

- RAII ensures cleanup actions occur reliably on scope exit.
- Transaction design (strong exception safety) structures operations as “commit or rollback”.

In practice, you implement it using scope guards (P0052-style `scope_exit`/`scope_fail`/`scope_success`) or a minimal local `scope_exit`, and you enforce a single explicit commit point.

Part IV

Concurrency, Threads, and Memory Safety

Chapter 12

The C++ Memory Model (Simplified)

12.1 Data Races

12.1.1 What a Data Race Is (and Why It Matters)

In the C++ memory model, a **data race** is not merely a "bug" or a "timing issue". It is a **language-level violation** that makes the program's behavior **undefined**. In practice, this means that once a data race exists, the compiler is allowed to assume it *never happens*, and it may apply optimizations that produce outcomes that look impossible to humans. This is why data races are among the most dangerous defects in modern C++. **A program with a data race is not just "sometimes wrong"; it is not a well-defined C++ program.**

12.1.2 The Standard Definition (Simplified but Accurate)

C++ defines a data race using these core ideas:

- Two operations are **potentially concurrent** if they occur in different threads (or

involve certain signal-handler cases).

- Two operations are **conflicting** if they access the same memory location (or overlapping storage), and at least one of them is a write (or an operation that starts/ends an object's lifetime in that storage).
- The execution contains a **data race** if it has two potentially concurrent **conflicting** operations, **at least one is not atomic**, and **neither happens-before the other**.
- **Any data race results in undefined behavior.**

This is the key sentence to remember:

> **Non-atomic conflicting access + no happens-before ordering = data race (UB).**

12.1.3 Conflicting Access: "Same Memory Location" in Practice

Two operations conflict when they refer to the **same scalar object** or **overlapping bytes of storage**, and at least one is a write. A subtle but important part of the C++ definition is that conflict is not limited to "read vs write" and "write vs write"; it also includes some lifetime-related events (constructing/destructing objects in overlapping storage).

12.1.4 Happens-Before: The Only Valid Way to "Make It Safe"

A common misconception is: "The threads do not run at exactly the same time, so it is safe." That is incorrect. The relevant concept is not wall-clock time; it is **ordering in the abstract machine**.

C++ uses **happens-before** to express a guaranteed order between actions in different threads. If one action *happens-before* another, then the second action is required to observe the effects

of the first in a well-defined way (subject to the rules of atomics, fences, and synchronization primitives). If there is no happens-before relation between conflicting non-atomic actions, the program has a data race.

12.1.5 Minimal Example: A Real Data Race (Undefined Behavior)

```
#include <thread>
#include <iostream>

int x = 0;

void writer() {
    x = 42;               // non-atomic write
}

void reader() {
    if (x == 42) {        // non-atomic read
        // ...
    }
}

int main() {
    std::thread t1(writer);
    std::thread t2(reader);
    t1.join();
    t2.join();
}
```

This program has a data race: one thread writes `x` while another reads `x`, `x` is non-atomic, and there is no happens-before ordering between the operations. Therefore the behavior is undefined.

12.1.6 Fix Strategy 1: Use a Mutex (Establish Happens-Before)

Mutex lock/unlock operations provide synchronization that establishes the needed ordering.

```
#include <thread>
#include <mutex>

int x = 0;
std::mutex m;

void writer() {
    std::lock_guard<std::mutex> lock(m);
    x = 42;
}

void reader() {
    std::lock_guard<std::mutex> lock(m);
    if (x == 42) {
        // safe
    }
}
```

Because both accesses are protected by the same mutex, there is no data race. Correctly used locks create the required ordering between critical sections.

12.1.7 Fix Strategy 2: Use Atomics (Avoid Non-Atomic Conflicts)

If the data is shared and frequently accessed, a mutex may be too heavy. In that case, use `std::atomic` and choose a memory order appropriate to the communication pattern.

```
#include <atomic>
#include <thread>
```

```
std::atomic<int> x{0};

void writer() {
    x.store(42, std::memory_order_release);
}

void reader() {
    if (x.load(std::memory_order_acquire) == 42) {
        // safe: acquire observes release in the intended synchronization
        ↪ pattern
    }
}
```

Atomic operations on the same atomic object do not create a data race. The release/acquire pair is a common way to establish inter-thread ordering and visibility.

12.1.8 A Professional Pattern: Publish Data Safely

A classic scenario: one thread builds an object, then publishes it to other threads. Do not publish a raw pointer with unsynchronized non-atomic access.

Incorrect Publishing (Data Race on the Pointer and/or the Data)

```
#include <thread>

struct Payload { int a; int b; };

Payload* p = nullptr; // non-atomic shared pointer

void producer() {
    auto* tmp = new Payload{1, 2};
```

```
    p = tmp;            // non-atomic write to shared variable
}

void consumer() {
    if (p) {            // non-atomic read
        // reading p->a / p->b is also not synchronized
        int x = p->a;
        (void)x;
    }
}
```

Correct Publishing with Atomics

```
#include <atomic>
#include <thread>

struct Payload { int a; int b; };

std::atomic<Payload*> p{nullptr};

void producer() {
    auto* tmp = new Payload{1, 2};
    p.store(tmp, std::memory_order_release);
}

void consumer() {
    Payload* ptr = p.load(std::memory_order_acquire);
    if (ptr) {
        int x = ptr->a; // safe with the publish pattern
        (void)x;
    }
}
```

The `release` store publishes the initialized object, and the matching `acquire` load ensures the consumer observes that initialization as intended by the model.

12.1.9 What `volatile` Does *Not* Do

In C++, `volatile` is **not** a thread-synchronization mechanism. It does not establish happens-before ordering, it does not prevent data races, and it does not provide atomicity. It exists primarily for special cases such as memory-mapped I/O and interactions with hardware where the compiler must not elide reads/writes. Therefore, replacing shared variables with `volatile int` does *not* fix a data race.

12.1.10 Data Race vs Race Condition

A **race condition** (in general engineering usage) means the program's result depends on timing or interleaving. A program may have a race condition even when it is data-race-free (for example, logically incorrect ordering of operations despite correct synchronization). A **data race** has a precise meaning in the C++ memory model and results in undefined behavior. The important professional takeaway is:

> **All data races are serious; removing a data race is a correctness prerequisite, not an optimization.**

12.1.11 Detection and Verification in Real Projects

Even experienced engineers miss data races during code review because they depend on interleavings that rarely appear in tests. Use specialized tools:

- **ThreadSanitizer (TSan)** to detect data races dynamically during testing (CI, stress tests, fuzzing).

- Prefer designs that minimize shared mutable state: message passing, immutable snapshots, thread confinement, and higher-level concurrency abstractions.

These practices do not replace correct use of mutexes/atomics, but they dramatically reduce risk and improve maintainability.

12.1.12 Professional Checklist

- Any shared mutable variable: either protect it with a mutex *or* make it atomic (or redesign to avoid sharing).
- Never mix atomic and non-atomic access to the same object.
- Do not rely on timing, sleeps, or "it usually works" reasoning.
- Do not use `volatile` for thread safety.
- For publish/subscribe patterns: use release/acquire or a mutex.

12.2 Happens-before

12.2.1 Why "happens-before" exists

In concurrent programs, the CPU and compiler are allowed to reorder operations in ways that are invisible in single-threaded code. The C++ memory model provides a precise way to say:

> Which operations must be observed in which order by other threads?

The central tool is the **happens-before** relation. Informally:

> If `A happens-before B`, then the effects of `A` are visible to `B` (and everything after `B` in that thread), and `B` cannot observe a state where `A` "didn't happen yet".

C++ uses happens-before primarily to define when a program is **data-race-free**, because any data race results in undefined behavior.

12.2.2 The building blocks

Sequenced-before (single-thread ordering)

Sequenced-before is an ordering relation between evaluations *within a single thread*. If evaluation `A` is sequenced-before `B`, then `A` completes before `B` begins.
This is the modern replacement for the old "sequence point" rules.

Synchronizes-with (cross-thread ordering)

Synchronizes-with is the key cross-thread relation that creates a visibility edge. It is established by well-defined synchronization operations, such as:

- releasing a mutex and later acquiring the same mutex
- a release atomic store whose value is read by an acquire atomic load
- thread completion and `join()`

These are specified in the standard library contracts and referenced through the memory model definitions.

Inter-thread happens-before and happens-before

Cppreference presents the standard definitions in terms of:

- **inter-thread happens-before** (built from synchronizes-with, dependency ordering, and transitivity)
- **happens-before** (the union of sequenced-before and inter-thread happens-before)

For the purpose of safe C++ concurrency, the most important implications are:

1. If `A is sequenced-before B`, then `A happens-before B`.
2. If `A synchronizes-with B`, then `A happens-before B`.
3. Happens-before is **transitive**: if `A hb B` and `B hb C`, then `A hb C`.

12.2.3 Why happens-before matters: data races and undefined behavior

The execution of a program contains a **data race** if two conflicting memory accesses are potentially concurrent, at least one is non-atomic, and neither happens-before the other. Any data race is undefined behavior.

Example: a data race (undefined behavior)

```
#include <thread>

int x = 0;

int main() {
    std::thread t([&] { x = 1; });  // write
    x = 2;                          // write in main
    t.join();
}
```

Two writes to the same non-atomic variable are potentially concurrent and not ordered by happens-before, so this is a data race and therefore undefined behavior.

12.2.4 How to create happens-before edges in real code

1) Mutex lock/unlock

Unlocking a `std::mutex` **synchronizes-with** a subsequent successful lock on the same mutex that obtains ownership.
This is why a mutex protects both:

- mutual exclusion (only one thread in the critical section)
- visibility (writes before unlock become visible after lock)

```
#include <mutex>
#include <thread>

std::mutex m;
int data = 0;

void producer() {
    std::lock_guard<std::mutex> lk(m);
    data = 42; // write protected by m
} // unlock happens here

void consumer() {
    std::lock_guard<std::mutex> lk(m); // lock synchronizes-with
    ↪ producer's unlock
    int v = data;                      // guaranteed to see 42 after
    ↪ producer finishes
    (void)v;
```

```
}

int main() {
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join();
    t2.join();
}
```

2) Thread completion and `join()`

The completion of the thread function **synchronizes-with** the successful return from `join()` on that thread.
So, any writes performed by a thread are visible to the joining thread after `join()` returns.

```
#include <thread>

int result = 0;

int main() {
    std::thread t([&] { result = 7; }); // write in worker thread
    t.join();                           // synchronizes-with thread
     ↪  completion
    int v = result;                     // guaranteed to see 7
    (void)v;
}
```

Also, the completion of the `std::thread` constructor invocation **synchronizes-with** the beginning of the invocation of the function object on the new thread (notably specified since C++20).

3) Condition variables (mutex + notify/wait)

A `std::condition_variable` is used together with a mutex to coordinate waiting and notification. The correct pattern always uses a predicate, because wakeups can be spurious and notifications can be missed if you wait incorrectly.

```
#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex m;
std::condition_variable cv;
bool ready = false;
int payload = 0;

void producer() {
    {
        std::lock_guard<std::mutex> lk(m);
        payload = 123;
        ready = true;
    }                   // unlock
    cv.notify_one();
}

void consumer() {
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [&]{ return ready; }); // specified as a loop on the
    ↪ predicate
    int v = payload;                   // safe: protected by the same
    ↪ mutex
    (void)v;
}
```

The key safety property comes from the mutex:

- producer writes under the lock
- producer unlocks
- consumer later locks and reads under the lock

and that unlock/lock pair creates the happens-before edge.

4) Atomics: release/acquire synchronization

For atomic operations, the strongest practical rule is:

> A `store(memory_order_release)` that is observed by a `load(memory_order_acquire)` creates a synchronizes-with edge, therefore a happens-before edge.

Cppreference defines happens-before and inter-thread happens-before in terms of synchronization relations, and documents release/acquire behavior and its transitive visibility consequences.

```
#include <atomic>
#include <thread>

std::atomic<bool> flag{false};
int data = 0;

void producer() {
    data = 99;                                  // non-atomic write
    flag.store(true, std::memory_order_release);
}
```

```
void consumer() {
    while (!flag.load(std::memory_order_acquire)) { }
    int v = data;                                  // guaranteed to see 99
    (void)v;
}

int main() {
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join();
    t2.join();
}
```

Why this is safe:

- the non-atomic write to `data` is sequenced-before the release store in `producer`
- the acquire load that reads-from that store synchronizes-with it
- by transitivity, the write to `data` happens-before the read in `consumer`

This is exactly the intended use of happens-before to make non-atomic data safely visible across threads.

5) Fences: ordering without tying to one specific atomic operation

`std::atomic_thread_fence` can establish ordering for non-atomic and relaxed atomic operations, but it still requires at least one atomic operation to complete a synchronization relationship.
Use fences only when you truly need them; release/acquire on an atomic variable is usually clearer and less error-prone.

12.2.5 A simplified mental model (that stays correct)

When reading or writing shared state, ask two questions:

Question 1: Is there a data race?

If two threads access the same location and at least one access is a write, then either:

- both accesses are atomic, or
- you must establish happens-before between them

Otherwise the behavior is undefined.

Question 2: What creates the happens-before edge?

The usual answers are:

- mutex unlock $\rightarrow$ later lock (same mutex)
- thread completion $\rightarrow$ `join()` return
- release store $\rightarrow$ acquire load that reads its value

If you cannot point to a specific edge, you should assume there is no ordering and therefore no safety.

12.2.6 Summary

- **Sequenced-before** orders evaluations within a thread.
- **Synchronizes-with** is the cross-thread relation created by synchronization primitives (mutexes, atomics, join).

- **Happens-before** combines these relations and is transitive; it defines visibility and is used to define data races.
- Any data race (conflicting non-atomic accesses without a happens-before order) is undefined behavior.

12.3 Visibility vs ordering

12.3.1 Two different questions (that are easy to mix up)

In the C++ memory model, **ordering** and **visibility** are related but not identical.

Ordering

Ordering answers:

> Which operations are constrained from being reordered (by the compiler and/or CPU) relative to other operations?

C++ expresses ordering constraints primarily through:

- mutex lock/unlock and other synchronization primitives
- atomic operations with specific `std::memory_order` values
- fences (thread fences)

Visibility

Visibility answers:

> When a thread reads a value, which write is it allowed (or required) to observe?

Visibility is governed by the **happens-before** relation and the atomic rules (e.g., modification order and synchronization). In particular, the standard's notion of a write becoming *visible* to another thread is not "physical time". It is a semantic guarantee: a read is constrained to see at least certain writes if the required ordering exists.

12.3.2 Why ordering and visibility can diverge

Key fact: different threads may observe different orders

Without constraints, one thread can observe changes to multiple variables in an order different from the order another thread wrote them. Different reader threads can even disagree about the apparent order of changes.
This is the simplest way to remember the difference:

> "Ordering" is about preventing reordering; "visibility" is about what values become observable when.

Key fact: atomicity does not imply cross-variable ordering

An atomic operation guarantees atomicity for that one atomic object. But if you use `memory_order_relaxed`, you get *no ordering* between that atomic and other memory operations.
That means a relaxed atomic can be used as a *counter* safely, but it is not (by itself) a safe way to publish data.

12.3.3 A concrete mental model that stays correct

Ordering: “reordering barriers” around an operation

Cppreference describes `std::memory_order` as specifying how memory accesses (including regular non-atomic accesses) are ordered around an atomic operation.
A simplified view:

- `release` prevents earlier reads/writes in the same thread from moving after the release store
- `acquire` prevents later reads/writes in the same thread from moving before the acquire load

Visibility: “publication” through a synchronization edge

Visibility of ordinary (non-atomic) writes across threads is not guaranteed unless there is a synchronization relationship that creates a happens-before edge (for example, release/acquire or mutex unlock/lock).

12.3.4 Example 1: Relaxed atomic gives atomicity, not safe publication

Incorrect pattern: using relaxed as a publish flag

```
#include <atomic>
#include <thread>

std::atomic<bool> ready{false};
int data = 0;

void producer() {
    data = 123; // non-atomic write
    ready.store(true, std::memory_order_relaxed);
}
```

```
void consumer() {
    while (!ready.load(std::memory_order_relaxed)) { }
    int v = data;  // not guaranteed to see 123
    (void)v;
}
```

Why this is wrong:

- the atomic flag is updated atomically
- but `memory_order_relaxed` does not establish the ordering/visibility needed to publish `data`

The memory model explicitly allows different observed orders without constraints.

Correct pattern: release/acquire publish

```
#include <atomic>
#include <thread>

std::atomic<bool> ready{false};
int data = 0;

void producer() {
    data = 123;
    ready.store(true, std::memory_order_release);
}

void consumer() {
    while (!ready.load(std::memory_order_acquire)) { }
    int v = data;  // guaranteed to see 123
    (void)v;
}
```

Here, release/acquire creates the intended cross-thread visibility for the preceding non-atomic write (`data = 123`) through the atomic variable.

12.3.5 Example 2: Ordering without ownership of a particular atomic (fences)

`std::atomic_thread_fence` provides ordering constraints not tied to a specific atomic object. Cppreference explains that a release fence prevents prior operations from moving past relevant subsequent stores, and shows how fences can synchronize non-atomic and relaxed atomic accesses when paired appropriately.
A key practical point:

> Fences are powerful but easier to misuse; prefer direct release/acquire on an atomic object when possible.

(Release/acquire makes the synchronization edge explicit at the read/write site.)

12.3.6 Example 3: Mutex ordering implies visibility for protected data

Mutexes provide both:

- mutual exclusion (only one thread in the critical section)
- a visibility guarantee (writes before unlock become visible after a later successful lock)

This is the common "ordering ⇒ visibility" story, but it is *specifically because* the mutex operations synchronize.

```
#include <mutex>
#include <thread>
```

```
std::mutex m;
int data = 0;

void producer() {
    std::lock_guard<std::mutex> lk(m);
    data = 42;
}

void consumer() {
    std::lock_guard<std::mutex> lk(m);
    int v = data; // guaranteed to see 42 if producer executed first
    (void)v;
}
```

12.3.7 What to remember about `memory_order_seq_cst`

`memory_order_seq_cst` is the strongest and simplest atomic ordering: it provides a single total order of all sequentially consistent operations, in addition to the acquire/release-style constraints.
Practical guidance:

- Use `seq_cst` when you want the easiest-to-reason-about global ordering across threads.
- Use release/acquire when you want clear publication without global total ordering.
- Use relaxed when you only need atomicity (e.g., statistics counters) and not publication.

These are directly aligned with how `std::memory_order` is described (ordering around atomic operations and the fact that, without constraints, observation order can differ).

12.3.8 A safety checklist for visibility vs ordering

Checklist

- If you share **non-atomic** data across threads, you must protect it with synchronization (mutex or release/acquire publication). Otherwise you risk a data race and undefined behavior.
- `memory_order_relaxed` gives atomicity for that atomic object, but does not publish other data.
- Release/acquire is the standard publication pattern: it constrains ordering and provides visibility via the atomic variable.
- Prefer direct release/acquire edges over fences unless you have a specific reason.

12.3.9 Summary

- **Ordering** constrains reordering around operations (what can move before/after what).
- **Visibility** constrains what values a read may observe, typically through happens-before created by synchronization.
- Atomic operations can be ordered in different strengths; choosing the right `memory_order` is choosing the right ordering/visibility contract.

Chapter 13

Race Conditions and Data Corruption

13.1 Why “it works on my machine” fails

13.1.1 The uncomfortable truth: concurrency bugs are often *allowed* to look correct

Concurrency failures often do not appear as clean crashes. They appear as:

- a test that passes 10,000 times and fails once
- a bug that disappears when you add logging or run under a debugger
- a production-only corruption that never reproduces on a developer laptop

This is not mysterious once you accept two standard facts:

1. If a **data race** occurs, the program has **undefined behavior**.
2. Without explicit synchronization, different threads are allowed to observe changes in different orders.

Undefined behavior means the standard imposes no requirements on what happens, so “it seemed fine” is not evidence of correctness.

13.1.2 Four reasons your machine can hide a race that production reveals

1) Scheduling is not deterministic

Thread execution order depends on:

- OS scheduler decisions
- CPU load, core counts, NUMA placement
- interrupts, I/O timing, power management

A race condition may require a specific interleaving (timing window). On your machine, the window might be rare; under production load, it might be frequent.

2) Optimization changes the program you think you wrote

Compilers transform code aggressively, especially under optimization. If your program has a data race, the compiler is allowed to assume the race does not happen and optimize accordingly, because the language definition makes data-racy executions undefined.
That is why:

- Debug builds can “work”
- Release builds fail
- Adding `printf` “fixes” the bug (it changes timing and optimization)

3) Hardware memory ordering differs across CPUs

Even if your code seems to work on one CPU, other CPUs can legitimately expose different behaviors because the hardware may reorder memory operations differently. C++ formalizes this by requiring you to use synchronization (mutexes or atomics with proper ordering) rather than relying on a particular machine's behavior.

4) "Atomic" does not automatically mean "properly synchronized"

Atomics have multiple ordering modes. If you use `memory_order_relaxed`, you get atomicity for that atomic object but no synchronization for other data. Cppreference explicitly notes that, absent constraints, different threads may observe writes in different orders.
So an atomic flag can still fail as a publication mechanism if it is used with the wrong ordering.

13.1.3 Canonical example: the increment that "usually works"

The bug: data race on a plain integer

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

int counter = 0;

int main() {
    std::vector<std::thread> ts;
    for (int i = 0; i < 8; ++i) {
        ts.emplace_back([] {
            for (int j = 0; j < 1'000'000; ++j)
                ++counter; // data race: read-modify-write on non-atomic
```

```
        });
    }
    for (auto& t : ts) t.join();
    std::cout << counter << "\n"; // often "looks close", sometimes
    ↪ wildly wrong
}
```

This is a data race, so the behavior is undefined. On some machines/builds it may print the expected value occasionally. That does not make it correct.

The fix: atomic or mutex

```
#include <atomic>
#include <thread>
#include <vector>
#include <iostream>

std::atomic<int> counter{0};

int main() {
    std::vector<std::thread> ts;
    for (int i = 0; i < 8; ++i) {
        ts.emplace_back([] {
            for (int j = 0; j < 1'000'000; ++j)
                counter.fetch_add(1, std::memory_order_relaxed);
        });
    }
    for (auto& t : ts) t.join();
    std::cout << counter.load() << "\n";
}
```

This is correct because the counter itself is atomic. Relaxed ordering is sufficient here because we only need atomicity for the counter, not publication of other data.

13.1.4 Canonical example: the “publish a pointer” bug

The bug: using a flag without establishing visibility

```
#include <atomic>
#include <thread>

std::atomic<bool> ready{false};
int payload = 0;

void producer() {
    payload = 123; // non-atomic write
    ready.store(true, std::memory_order_relaxed); // not a publication
    ↪ barrier
}

void consumer() {
    while (!ready.load(std::memory_order_relaxed)) { }
    int v = payload; // not guaranteed to see 123
    (void)v;
}
```

Why it fails:

- the flag update is atomic
- but relaxed ordering does not create the synchronization needed to make `payload` visible

Without constraints, readers may observe writes in different orders.

The fix: release/acquire publication

```
#include <atomic>
```

```cpp
#include <thread>

std::atomic<bool> ready{false};
int payload = 0;

void producer() {
    payload = 123;
    ready.store(true, std::memory_order_release);
}

void consumer() {
    while (!ready.load(std::memory_order_acquire)) { }
    int v = payload; // guaranteed to see 123
    (void)v;
}
```

Release/acquire is the standard publication pattern: it creates the required cross-thread ordering and visibility.

13.1.5 Data-race-free programs behave *much* better

The C++ concurrency model is designed around a practical principle: if a program is **data-race-free**, it gets strong semantic guarantees (commonly described as a form of sequential consistency for race-free programs), while executions with data races are undefined.

This is exactly why “works on my machine” is not acceptable as a correctness argument: the same buggy program can appear stable under one set of timings and fall apart under another.

13.1.6 How to stop trusting luck

1) Follow the one rule that matters most

The C++ Core Guidelines are blunt about this: avoid data races.
Practical translation:

> Every shared mutable object must be protected by *one* synchronization strategy: a mutex, or atomics with a clear happens-before story.

2) Use tools that expose the bug instead of hiding it

Dynamic race detectors can reveal data races that tests miss. ThreadSanitizer (TSan) is specifically designed to detect data races in C/C++ via compiler instrumentation and a runtime library.
TSan is especially valuable because it turns "rare timing windows" into deterministic reports.

3) Test under conditions that destroy timing assumptions

To make races visible:

- run on different CPU architectures and core counts when possible
- vary optimization levels (Debug vs Release)
- add stress: higher thread counts, randomized sleeps, CPU pinning, heavy I/O

These steps do not *prove* correctness, but they reduce the chance that you are only seeing the lucky schedule.

13.1.7 Summary

- Concurrency bugs often "work" until scheduling, optimization, or hardware changes.

- If a data race occurs, behavior is undefined, so passing tests is not evidence.
- Without synchronization, different threads may legitimately observe writes in different orders.
- Data-race-free design gives you strong guarantees; data races destroy reasoning.
- Use the standard synchronization tools (mutexes, release/acquire) and validate with race detectors such as TSan.

13.2 Common concurrency bugs

13.2.1 Why these bugs are so common

Concurrency bugs are common because correctness depends on *relationships* between operations:

- Is there a happens-before edge?
- Are two operations conflicting and potentially concurrent?
- Does a waiting thread re-check the condition after waking?

If these relationships are wrong, the C++ standard often gives you no safety net. In particular, **any data race is undefined behavior**.

13.2.2 Bug 1: Data races on non-atomic shared state

What it is

Two threads access the same memory location concurrently, at least one access is a write, and the accesses are not ordered by happens-before. If it happens, the behavior is undefined.

Typical symptom

- "It works on my machine" until optimization, CPU, or load changes
- rare corruptions, impossible invariants, or crashes far from the real bug

Example (undefined behavior)

```
#include <thread>

int x = 0;

int main() {
    std::thread t([] { x = 1; });
    x = 2;          // concurrent write to the same non-atomic object
    t.join();
}
```

Fix

Choose exactly one strategy per shared object:

- protect it with a mutex, or
- make it atomic (and use correct ordering when publishing other data)

The Core Guidelines explicitly emphasize avoiding data races.

13.2.3 Bug 2: "Atomic flag" used with `memory_order_relaxed` to publish data

What it is

A relaxed atomic operation is atomic for that one object, but it provides no ordering that would publish non-atomic data. So a reader can see the flag change but still read stale payload values.

Bad example (not a safe publication pattern)

```
#include <atomic>

std::atomic<bool> ready{false};
int payload = 0;

void producer() {
    payload = 123;
    ready.store(true, std::memory_order_relaxed);
}

int consumer() {
    while (!ready.load(std::memory_order_relaxed)) { }
    return payload; // not guaranteed to observe 123
}
```

Cppreference warns that, absent constraints, threads may observe writes in different orders.

Fix (release/acquire publication)

```
#include <atomic>

std::atomic<bool> ready{false};
int payload = 0;
```

```
void producer() {
    payload = 123;
    ready.store(true, std::memory_order_release);
}

int consumer() {
    while (!ready.load(std::memory_order_acquire)) { }
    return payload; // guaranteed to observe 123
}
```

13.2.4 Bug 3: Check-then-act races (TOCTOU) under concurrency

What it is

Thread A checks a condition, then acts assuming the condition is still true. Thread B changes the condition in between.

Classic example

```
#include <vector>

std::vector<int> v; // shared

// Thread A:
if (!v.empty()) {
    int x = v.back(); // unsafe: another thread could pop_back()
    (void)x;
}
```

Fix

Make the check and the act part of the same critical section (mutex), or use a lock-free algorithm designed for the structure.

13.2.5 Bug 4: Lost wakeups and spurious wakeups with condition variables

What it is

A condition variable can wake spuriously; notifications can also be missed if you wait incorrectly. Therefore, waiting must always be done in a loop that re-checks the predicate.

Bad example (incorrect waiting)

```
#include <condition_variable>
#include <mutex>

std::mutex m;
std::condition_variable cv;
bool ready = false;

void consumer_bad() {
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk);            // may return spuriously
    if (ready) { /* ... */ }  // not reliable
}
```

Correct pattern (predicate wait)

Cppreference specifies that `wait(lock, pred)` is equivalent to a loop that re-checks the predicate.

```
#include <condition_variable>
#include <mutex>

std::mutex m;
std::condition_variable cv;
bool ready = false;

void consumer_good() {
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [&]{ return ready; }); // re-checks predicate
    // safe to proceed
}
```

13.2.6 Bug 5: Deadlocks from inconsistent lock ordering

What it is

Two or more locks are acquired in different orders on different code paths, creating a cycle.

Bad example

```
#include <mutex>

std::mutex a, b;

void f() {
    std::lock_guard<std::mutex> l1(a);
    std::lock_guard<std::mutex> l2(b);
```

```
}

void g() {
    std::lock_guard<std::mutex> l2(b);
    std::lock_guard<std::mutex> l1(a); // opposite order: deadlock risk
}
```

Fix

Use deadlock-avoiding multi-lock facilities (or enforce a global lock order). C++ provides `std::lock` to lock multiple mutexes without deadlock (as a library contract).

```
#include <mutex>

std::mutex a, b;

void safe() {
    std::lock(a, b);
    std::lock_guard<std::mutex> l1(a, std::adopt_lock);
    std::lock_guard<std::mutex> l2(b, std::adopt_lock);
    // critical section holding both locks
}
```

13.2.7 Bug 6: Livelock and starvation

Livelock

Threads keep reacting to each other (retrying, backing off) and make no progress.

Starvation

One thread is continually denied access to a needed resource because other threads keep winning the race.

Common cause

Overuse of `try_lock` loops with poorly designed backoff. If you rely on luck for fairness, you will eventually see pathological schedules.

13.2.8 Bug 7: Holding locks while calling unknown code

What it is

You hold a mutex, then call:

- a callback
- a virtual function
- a function that may block on I/O

This can cause:

- re-entrancy deadlocks (the callback tries to acquire the same mutex or another mutex)
- unbounded contention (I/O under a lock)

Fix

Keep critical sections small and well-defined: copy the state you need under the lock, then release the lock before calling out.

13.2.9 Bug 8: Misusing `volatile` for synchronization

What it is

Using `volatile` in place of atomics or mutexes. `volatile` is not a C++ synchronization primitive and does not create happens-before relationships.

The Core Guidelines explicitly warn: do not try to use `volatile` for synchronization.

Fix

Use:

- `std::atomic` for atomic state, and
- mutexes/condition variables for blocking coordination

13.2.10 Bug 9: Thread lifetime bugs (dangling references, detached threads)

What it is

A thread captures references or pointers to objects that are destroyed before the thread finishes, or a detached thread outlives the resources it touches.

Bad example

```
#include <thread>
#include <string>

void start() {
    std::string s = "hello";
    std::thread t([&] { /* uses s */ });
    t.detach(); // s will be destroyed when start() returns
}
```

Fix

Make lifetime explicit:

- join threads, or use a scoped thread abstraction
- pass ownership (move) into the thread (e.g., `std::shared_ptr` or moved value)

13.2.11 Bug 10: Non-atomic object teardown while another thread uses it

What it is

One thread destroys or closes a resource while another thread is still operating on it (e.g., closing a socket while another thread is in `send`/`recv`).

Fix

Coordinate shutdown:

- use a clear ownership model for the resource
- signal stop, then join (or otherwise prove no concurrent use remains), then destroy/close

13.2.12 Bug 11: False sharing (performance bug that looks like a concurrency bug)

What it is

Two threads update different variables that happen to reside on the same cache line. This is not incorrect, but it can create severe contention and timing changes that make races easier to reproduce or hide.

Fix

Separate frequently-written independent variables (padding, alignment, structure redesign) and reduce shared writes.

13.2.13 Bug 12: Misinterpreting tool results (or not using tools at all)

Concurrency bugs are often non-reproducible by testing alone. ThreadSanitizer is designed specifically to detect data races using compiler instrumentation and a runtime library. The Core Guidelines also recommend using tools to validate concurrent code.

13.2.14 A practical checklist

- Every shared mutable object has exactly one synchronization strategy (mutex or atomics with a clear publication rule).
- Condition variable waits are always predicate-based (loop).
- Multi-locking uses `std::lock` (or a consistent global lock order).
- Never use `volatile` for synchronization.
- Run ThreadSanitizer for race detection in testing pipelines.

13.2.15 Summary

Common concurrency bugs cluster into a few families:

- **Correctness failures**: data races (UB), broken publication, lost/spurious wakeups, deadlocks, lifetime races.
- **Progress failures**: livelock, starvation.

- **Performance traps**: false sharing, over-locking, lock contention.

The fastest path to robust code is to make synchronization relationships explicit (happens-before edges), follow the standard patterns for mutexes/atomics/condition variables, and validate with race-detection tools.

13.3 Diagnosing race conditions

13.3.1 What makes race conditions hard to diagnose

Race conditions are difficult because they are:

- **nondeterministic**: the failing interleaving may be rare
- **sensitive to observation**: logging, debugging, and timing changes can hide the bug
- **far from the symptom**: corruption can appear long after the real violation

In C++, the most important reason is semantic: if a **data race** occurs, the program has **undefined behavior**. Once behavior is undefined, "it sometimes works" provides no safety evidence.

13.3.2 Start with the one question that matters

Is it a data race or a higher-level race?

- **Data race**: conflicting accesses (same location, at least one write) that are not ordered by happens-before. This is undefined behavior.
- **Higher-level race**: all accesses are synchronized, but the program logic is wrong (e.g., wrong lock granularity, wrong condition predicate, wrong protocol state machine).

Diagnosis is faster if you separate these early:

> First eliminate data races (UB). Then debug remaining logical races.

13.3.3 A professional workflow: from symptom to root cause

Step 1: Make failure reproducible (or recordable)

Use these levers to increase failure probability:

- raise thread counts and iterations
- add contention (shared hot paths, CPU load)
- run optimized builds as well as debug builds (optimization changes behavior)

When reproducibility is difficult, record/replay tools can help. `rr` is designed for recording and deterministically replaying executions to debug low-frequency nondeterministic failures.

Step 2: Turn on race detection (dynamic)

ThreadSanitizer (TSan). ThreadSanitizer is a dynamic data-race detector based on compile-time instrumentation plus a runtime library. It is one of the most effective tools for diagnosing real data races.
Typical Clang/GCC build flags:

```
# Build with ThreadSanitizer (Clang or GCC)
CXX=clang++  # or g++
CXXFLAGS="-O1 -g -fno-omit-frame-pointer -fsanitize=thread"
LDFLAGS="-fsanitize=thread"

$CXX $CXXFLAGS main.cpp -o app $LDFLAGS
./app
```

Notes:

- `-O1` is commonly recommended with sanitizers to improve analysis quality while keeping overhead manageable.
- `-fno-omit-frame-pointer` improves stack traces in sanitizer reports.

Interpreting TSan output. A useful report typically provides:

- the two (or more) conflicting accesses
- stack traces for each access
- the synchronization history (locks, happens-before arcs) leading to the conclusion

Treat the first reported race as high priority: a single data race can invalidate later observations.

When TSan is not available or not suitable. Helgrind is a Valgrind tool for detecting synchronization errors and data races for programs using POSIX pthread primitives. It is often slower than TSan, but can be useful in specific environments or debugging setups.

Step 3: Add static diagnostics (compile-time)

Static analysis cannot prove thread safety in general, but it can detect many common mistakes with low cost.
Clang Thread Safety Analysis is a compile-time analysis that warns about potential race conditions when you annotate locking policies.
The practical value:

- it prevents regressions (a missing lock becomes a compile warning)
- it documents the intended locking discipline in code

Step 4: Reduce the bug to a minimal reproducer

Once a tool report identifies a race, reduce the code until:

- the race is still reported
- the reproducer is small enough to reason about

This step is not optional in professional work. Races are relationship bugs; reduction clarifies the relationships.

13.3.4 Recognizing the most common signatures in reports

Signature A: “non-atomic shared integer”

Typical report pattern:

- write vs write, or read vs write on a plain variable
- no lock held, no atomic ordering present

Minimal example (data race, UB):

```
#include <thread>

int counter = 0;

int main() {
    std::thread t1([] { ++counter; });
    std::thread t2([] { ++counter; });
    t1.join();
    t2.join();
}
```

Fix options:

- protect `counter` with a mutex
- make it `std::atomic<int>` (often with `memory_order_relaxed` if it is only a counter)

Signature B: "published pointer/data without a happens-before edge"

The pattern:

- Thread A writes payload, then sets a flag
- Thread B sees the flag but reads stale payload

Without proper synchronization, different threads may observe writes in different orders. Correct publication is typically done with release/acquire semantics on the atomic that carries the synchronization edge.

Signature C: "condition variable wait used without a predicate"

Spurious wakeups are permitted, so a condition variable wait must re-check a predicate. The predicate overload of `wait` is specified as a loop that checks the predicate.
Correct form:

```
cv.wait(lock, [&]{ return ready; }); // re-checks in a loop
```

If a bug disappears when you add logging, incorrect waiting logic is a frequent cause.

Signature D: "deadlock or lock-order cycle"

Symptoms:

- threads stuck waiting on locks
- long stalls that vanish under debugging

Diagnosis approach:

- capture stacks of blocked threads
- check lock acquisition order across code paths
- use consistent lock ordering or deadlock-avoiding multi-lock primitives

13.3.5 Do not sabotage your own diagnosis

Avoid "fixing" races by adding logging

Logging changes timing and scheduling, and can mask races. Use it to add *invariants* and *state tracing*, but never accept "the bug went away" as evidence.

Do not use `volatile` as a debugging or synchronization tool

`volatile` is not a C++ synchronization mechanism and does not create happens-before relationships. Rely on atomics/mutexes and proper tooling instead. (This rule is emphasized in modern C++ guidance, including the Core Guidelines tool ecosystem discussion around concurrency.)

13.3.6 A practical checklist for diagnosing races

1. Confirm whether the failure could be a data race (shared non-atomic mutation).
2. Build and run with ThreadSanitizer; treat the first race report as urgent.
3. If needed, use Helgrind on POSIX/pthreads workloads.

4. Add static checks (e.g., Clang Thread Safety Analysis) for long-term prevention.
5. Reduce to a minimal reproducer and fix by creating a clear happens-before edge (mutex or atomics with correct ordering).
6. Re-run sanitizers and add stress tests so the fix is validated under load.

13.3.7 Summary

Diagnosing race conditions is largely about replacing guesswork with evidence:

- A data race is undefined behavior, so testing alone cannot establish correctness.
- Use dynamic detectors (especially ThreadSanitizer) to find conflicting accesses and their stacks.
- Use static analysis (e.g., Clang Thread Safety Analysis) to prevent common locking mistakes.
- When nondeterminism blocks debugging, use record/replay (rr) to make failures repeatable.

Chapter 14

Mutexes and Locks the Right Way

14.1 `std::mutex`

14.1.1 What `std::mutex` is for

`std::mutex` is the standard C++ primitive for **mutual exclusion**: it ensures that at most one thread at a time enters a critical section that manipulates shared state.

Just as important as exclusion is **memory synchronization**: locking and unlocking a mutex creates a well-defined *happens-before* relationship between threads, so writes performed by one thread inside the critical section become visible to another thread that later acquires the same mutex.

14.1.2 Key properties and lifetime rules

Non-copyable and non-movable

A `std::mutex` cannot be copied or moved. This is intentional: a mutex represents a unique synchronization object.

Destruction while locked is undefined behavior

Destroying a mutex while it is owned by any thread is undefined behavior. Also, if a thread terminates while owning a mutex, the behavior is undefined.
Practical meaning:

- a mutex must outlive all code that might lock it
- never let a thread exit while holding a lock (use RAII locking)

14.1.3 The synchronization guarantee (why mutexes also solve visibility)

The rule you must know

Unlocking a mutex **synchronizes-with** any subsequent successful lock operation on the same mutex that obtains ownership.
This creates a happens-before edge:

> All writes in Thread A that are sequenced-before `unlock()` become visible to Thread B after it successfully `lock()`s the same mutex.

That is exactly why a mutex is a correct tool for protecting non-atomic shared data.

14.1.4 Correct usage: RAII locking is the default

`std::lock_guard`: simplest, safest

`std::lock_guard<std::mutex>` locks in its constructor and unlocks in its destructor. This prevents leaks of the lock on early return or exceptions.

```
#include <mutex>
#include <vector>

class SafeQueue {
    std::mutex m_;
    std::vector<int> q_;

public:
    void push(int x) {
        std::lock_guard<std::mutex> lk(m_);
        q_.push_back(x);
    }

    bool try_pop(int& out) {
        std::lock_guard<std::mutex> lk(m_);
        if (q_.empty()) return false;
        out = q_.back();
        q_.pop_back();
        return true;
    }
};
```

`std::unique_lock`: when you need more control

Use `std::unique_lock` when you need one of these:

- deferred locking (`std::defer_lock`)

- manual unlock/relock (split critical sections intentionally)
- interaction with `std::condition_variable` (which requires a `std::unique_lock`)

Keep the principle: lock ownership should still be scoped, explicit, and exception-safe.

14.1.5 Preconditions and undefined behavior traps

`unlock()` requires ownership

Calling `unlock()` when the current thread does not own the mutex is undefined behavior.

`lock()` and `try_lock()` on a mutex you already own

If `lock()` is called by a thread that already owns the mutex, the behavior is undefined (in practice, it may deadlock).
Similarly, calling `try_lock()` from a thread that already owns the same `std::mutex` is undefined behavior.
Practical rule:

> If you need re-entrant locking, redesign the code; do not "accidentally" rely on recursive locking.

`try_lock()` may fail spuriously

`try_lock()` is allowed to return `false` even when no other thread currently holds the mutex.
So you must not write correctness logic that assumes "false means someone else owns it right now". Treat `try_lock()` as a best-effort optimization.

14.1.6 Avoiding deadlock with multiple mutexes

The classic failure: inconsistent lock order

Deadlocks commonly happen when different code paths lock multiple mutexes in different orders.

The standard solution: `std::lock` + `adopt_lock`

When you must lock two (or more) mutexes together, use `std::lock` to acquire them without deadlock, then adopt those locks into RAII wrappers.

```
#include <mutex>

std::mutex a, b;

void safe_lock_both() {
    std::lock(a, b); // lock both without deadlock
    std::lock_guard<std::mutex> l1(a, std::adopt_lock);
    std::lock_guard<std::mutex> l2(b, std::adopt_lock);
    // critical section holding both a and b
}
```

This preserves RAII while also using a deadlock-avoiding multi-lock algorithm.

14.1.7 When `std::mutex` is not the right mutex

If you need read-mostly concurrency

If many threads only read shared state, a reader-writer lock can improve concurrency (e.g., `std::shared_mutex`). But the safety story remains identical: you need a clear locking discipline and a clear happens-before edge.

If you think you need recursion

If you believe you need `std::recursive_mutex`, treat it as a design smell first. Recursive mutexes exist, but they can mask poor structure and make correctness reasoning harder. If you still need it, make that decision explicit and document why.

14.1.8 Safety rules to adopt as project policy

These rules are consistent with standard contracts and modern best practice:

- Use RAII locking (`lock_guard` / `unique_lock`); avoid manual `lock()/unlock()` in application code.
- Keep critical sections small; do not perform blocking I/O or call unknown callbacks while holding a lock.
- Do not destroy a mutex while any thread might still lock it.
- Never call `unlock()` without ownership.
- For multiple mutexes, use `std::lock` (or an enforced global lock order).
- Prefer one mutex per invariant: decide which data is protected by which mutex, and document it (in code comments or naming).

The C++ Core Guidelines maintain a dedicated concurrency section emphasizing avoiding data races and using the standard concurrency tools correctly.

14.1.9 Summary

- `std::mutex` provides mutual exclusion and a cross-thread synchronization guarantee: `unlock()` synchronizes-with a later successful `lock()` on the same mutex.

- Misuse is often undefined behavior: unlocking without ownership, re-locking a non-recursive mutex you already own, or destroying a locked mutex.

- Default to RAII locking and use `std::lock` for multi-mutex deadlock avoidance.

14.2 `std::lock_guard`

14.2.1 What `std::lock_guard` is

`std::lock_guard<Mutex>` is a small RAII wrapper for a mutex-like object:

- it **locks** the mutex when the guard is constructed

- it **unlocks** the mutex when the guard is destroyed (scope exit)

It is non-copyable by design, so lock ownership cannot be duplicated accidentally.
This is the default tool for protecting a short critical section because it makes the lock-release unconditional (every return path, including exceptions, releases the lock).

14.2.2 Why `std::lock_guard` is the "right default"

It prevents the most common locking bug: missing `unlock()`

Manual `lock()/unlock()` is fragile: an early return or thrown exception can easily skip the `unlock()`.
`std::lock_guard` ties the lock lifetime to a scope, so the destructor reliably releases the lock.

Mutex locking also provides visibility guarantees

A mutex is not only mutual exclusion; it also provides synchronization: unlocking a mutex synchronizes-with a later successful lock on the same mutex. That is what makes writes in one critical section visible to a later critical section in another thread.
Using `std::lock_guard` is the idiomatic way to acquire that guarantee safely.

14.2.3 Basic usage pattern

```
#include <mutex>
#include <vector>

class Counter {
    std::mutex m_;
    int value_ = 0;

public:
    void inc() {
        std::lock_guard<std::mutex> lk(m_);
        ++value_;
    }

    int get() const {
        // Even reads must be synchronized if other threads can write.
        // (You may prefer std::shared_mutex for read-mostly designs.)
        std::lock_guard<std::mutex> lk(m_);
        return value_;
    }
};
```

Rule of thumb:

If a piece of shared state is protected by a mutex, then *every* access to that state follows the same locking discipline.

14.2.4 Construction modes and their meaning

Default constructor form: locks immediately

When a `lock_guard` is constructed with a mutex reference, it attempts to take ownership (by locking).
This is what you want for the majority of critical sections.

`std::adopt_lock`: "I already locked it"

`std::adopt_lock` constructs a guard that *assumes* the current thread already owns the mutex. If that is not true, the program has undefined behavior (because the guard will attempt to unlock a mutex it does not own).
Correct usage: lock first, then adopt.

```
#include <mutex>

std::mutex m;

void f() {
    m.lock(); // acquire ownership first (may throw only in extremely
    ↪ unusual implementations)
    std::lock_guard<std::mutex> lk(m, std::adopt_lock); // now lk will
    ↪ unlock on scope exit
    // critical section
}
```

In modern code, `adopt_lock` is most useful when combined with algorithms that lock multiple mutexes safely, such as `std::lock`.

```cpp
#include <mutex>

std::mutex a, b;

void safe_two_mutex_section() {
    std::lock(a, b); // deadlock-avoidance algorithm
    std::lock_guard<std::mutex> lk1(a, std::adopt_lock);
    std::lock_guard<std::mutex> lk2(b, std::adopt_lock);
    // critical section holding both locks
}
```

14.2.5 Common misuse patterns (and why they are dangerous)

Misuse 1: guarding too much code

A common "works at first" bug is holding the lock while doing expensive work or calling unknown code (callbacks, virtuals, I/O). This increases contention and can create deadlocks. Fix: keep critical sections small. Copy or move data out under the lock, then release the lock before heavy work.

Misuse 2: assuming `lock_guard` can be unlocked early

`std::lock_guard` has a deliberately small interface: it does not support manual `unlock()` or `lock()`.
If you need to unlock early (split the critical section), use `std::unique_lock` instead. This separation is intentional: `lock_guard` expresses "pure RAII locking."

Misuse 3: using `adopt_lock` without owning the mutex

If `adopt_lock` is used when the mutex is not already locked by the current thread, the destructor will attempt to unlock a mutex the thread does not own, which is undefined

behavior.

14.2.6 Choosing between `lock_guard`, `unique_lock`, and `scoped_lock`

`std::lock_guard`: simplest single-lock ownership

Use `lock_guard` when:

- you lock one mutex
- you hold it for the whole scope
- you do not need condition variables or deferred locking

This is the "default" locking form in C++11 and later.

`std::unique_lock`: more control

Use `unique_lock` when you need:

- deferred locking (`defer_lock`)
- try-lock patterns with stateful ownership
- manual unlock/relock
- `std::condition_variable` waits (which require `unique_lock`)

`std::scoped_lock`: multiple mutexes, deadlock-avoidance (C++17+)

`std::scoped_lock` can lock multiple mutexes at once using a deadlock-avoidance strategy, conceptually like `std::lock`.
For single-mutex cases, `lock_guard` remains perfectly correct and often the clearest expression of intent.

14.2.7 Practical rules for professional codebases

- Prefer RAII locks over manual `lock()/unlock()` to guarantee release on all paths.
- Use `std::lock_guard` as the default for simple critical sections.
- Use `std::lock` + `adopt_lock` (or `std::scoped_lock` in C++17+) for multi-mutex locking.
- Keep critical sections minimal and avoid calling unknown code while holding the lock.
- Document which mutex protects which data (even a one-line comment can prevent years of bugs).

14.2.8 Summary

`std::lock_guard` is the simplest correct way to acquire a mutex in C++:

- lock at construction, unlock at destruction (scope exit)
- non-copyable to prevent accidental sharing of ownership
- pairs naturally with the mutex synchronization guarantee that makes shared state both exclusive and visible across threads

Use it by default, and move to `unique_lock` or `scoped_lock` only when you need their extra capabilities.

14.3 `std::unique_lock`

14.3.1 What `std::unique_lock` is (and why it exists)

`std::unique_lock<Mutex>` is a general-purpose RAII wrapper that owns (or does not own) a mutex. Like `std::lock_guard`, it can lock a mutex on construction and unlock it on destruction, but it is designed to support **more workflows**:

- deferred locking (construct now, lock later)
- try-lock and timed lock attempts (when the underlying mutex supports them)
- manual unlock/relock (split a critical section intentionally)
- transfer of ownership (movable lock objects)
- integration with `std::condition_variable` (which works with `std::unique_lock<std::mutex>`)

14.3.2 Core ownership model

Movable, not copyable

`std::unique_lock` is movable but not copyable. This expresses the key rule: *lock ownership is unique*.
Practical consequences:

- you can return a lock from a helper (move it)

- you can store a lock in an object and move it between states
- you cannot accidentally duplicate lock ownership (copy)

The lock can be “empty”

A `unique_lock` can exist with:

- no associated mutex, or
- an associated mutex but not currently owning it (unlocked state)

This is explicitly supported by its constructors.

14.3.3 Construction strategies (how you choose the initial state)

Immediate locking (the default)

Construct with a mutex reference to lock immediately:

```
#include <mutex>

std::mutex m;

void f() {
    std::unique_lock<std::mutex> lk(m); // locks now
    // critical section
} // unlocks automatically
```

Deferred locking: `std::defer_lock`

`std::defer_lock` constructs the lock object but does not lock the mutex yet.
Use this when you want to decide later whether/when to acquire the mutex, while still keeping RAII semantics.

```cpp
#include <mutex>

std::mutex m;

void f() {
    std::unique_lock<std::mutex> lk(m, std::defer_lock);
    // do work that does not require the lock yet
    lk.lock(); // acquire when needed
    // critical section
}
```

Try locking: `std::try_to_lock`

`std::try_to_lock` attempts to lock without blocking. Whether it succeeds is observable via `owns_lock()`.

```cpp
#include <mutex>

std::mutex m;

bool try_update() {
    std::unique_lock<std::mutex> lk(m, std::try_to_lock);
    if (!lk.owns_lock()) return false; // did not acquire
    // critical section
    return true;
}
```

Adopting an existing lock: `std::adopt_lock`

`std::adopt_lock` means: *"the current thread already owns this mutex; just manage its release with RAII."* The tag types are specifically defined for `std::unique_lock` (and `std::shared_lock`).

Correct pattern: lock first, then adopt:

```
#include <mutex>

std::mutex m;

void f() {
    m.lock();
    std::unique_lock<std::mutex> lk(m, std::adopt_lock);
    // critical section
} // lk unlocks m
```

Project rule:

> Only use `adopt_lock` immediately after the lock is acquired, in the same scope.

14.3.4 The main feature of `unique_lock`: controlled unlock/relock

Why manual unlock exists

Sometimes you need a lock to protect a small shared state update, but you must not hold the lock while doing expensive work or calling code you do not control.

```
#include <mutex>
#include <string>

std::mutex m;
std::string shared;

void append_and_compute(const std::string& s) {
    std::unique_lock<std::mutex> lk(m);
```

```
    shared += s;        // protected update

    lk.unlock();        // release early: reduce contention
    // expensive work not touching shared
    // ...
    lk.lock();          // optionally re-acquire if needed later
}
```

Safety preconditions of `unlock()`

`unique_lock::unlock()` releases ownership of the associated mutex, and throws `std::system_error` if there is no associated mutex or if the mutex is not locked. This is a major behavioral difference from `std::mutex::unlock()`, where unlocking without ownership is undefined behavior. In other words:

> `unique_lock` can report misuse via exceptions; `mutex` misuse is often undefined behavior.

(Do not rely on exceptions as a correctness strategy; rely on a disciplined ownership model.)

14.3.5 The reason `condition_variable` uses `unique_lock`

The required behavior: atomically unlock and wait, then re-lock

`std::condition_variable::wait` works by:

- atomically unlocking the lock
- blocking until notification or spurious wakeup
- re-locking the lock before returning

This behavior is specified directly.
Because `wait` must call `lock.unlock()` and `lock.lock()` as part of its contract, it requires a lock type that supports those operations, which `std::unique_lock` does.

Correct waiting pattern (predicate form)

`wait(lock, pred)` is defined as a loop that repeatedly waits until the predicate is true, which is essential because spurious wakeups are permitted.

```
#include <condition_variable>
#include <mutex>

std::mutex m;
std::condition_variable cv;
bool ready = false;

void consumer() {
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [&]{ return ready; }); // re-checks predicate in a loop
    // proceed
}
```

14.3.6 Timed locking (when supported)

`std::unique_lock` supports time-constrained locking when used with mutex types that provide timed lock operations (e.g., `std::timed_mutex`, `std::recursive_timed_mutex`). The constructors include timed overloads that accept `std::chrono` durations or time points.
Guideline:

> Timed locks are a tool for bounded waiting and responsiveness, not a substitute for correct design.

14.3.7 Choosing between `lock_guard`, `unique_lock`, and `scoped_lock`

`std::lock_guard`

Use when you:

- lock one mutex
- hold it for the entire scope
- do not need manual unlock or condition variables

It is minimal and expresses intent clearly.

`std::unique_lock`

Use when you need:

- **condition variables** (`wait` requires `std::unique_lock<std::mutex>`)
- deferred lock / try lock / timed lock strategies
- unlock/relock within a scope
- move ownership to another scope/function

`std::scoped_lock` (C++17+)

Use when you need to lock *multiple* mutexes together with RAII. It can own zero or more mutexes and is intended to support multi-locking patterns.

14.3.8 Common misuse patterns (and how to avoid them)

Misuse 1: forgetting to lock after `defer_lock`

If you construct with `defer_lock`, you must explicitly `lock()` before entering the critical section. Use `owns_lock()` in debug assertions if helpful.

Misuse 2: holding a lock across blocking operations

If you hold a lock while performing I/O, waiting on other locks, or calling unknown code, you increase contention and deadlock risk. Prefer: lock, copy necessary state, unlock, then do the expensive work.

Misuse 3: waiting without a predicate

Always use the predicate overload (or an explicit `while` loop), because spurious wakeups are permitted and the predicate overload is specified as a loop.

14.3.9 Project policy: safe default rules

- Use `std::lock_guard` by default; upgrade to `std::unique_lock` only when you need its capabilities.
- For condition variables, always use `std::unique_lock<std::mutex>` and the predicate overload of `wait`.
- Keep critical sections small; avoid blocking and callbacks while holding locks.
- Prefer `std::scoped_lock` for multi-mutex RAII in C++17+ codebases.

14.3.10 Summary

`std::unique_lock` is the flexible RAII lock:

- movable, not copyable (unique ownership)
- supports deferred/try/timed strategies via tag types
- supports explicit `unlock()` and is required for `std::condition_variable` waiting semantics

Use it when you need control; otherwise prefer simpler RAII locks.

14.4 Avoiding deadlocks with RAII

14.4.1 Deadlock in one sentence

A deadlock occurs when two or more threads wait forever for each other's locks. The most common cause in C++ is **inconsistent lock acquisition order** across code paths.
The best prevention strategy is simple:

> Make lock ownership explicit, scope-bound, and exception-safe, and use the standard deadlock-avoiding primitives when locking multiple mutexes.

The C++ Core Guidelines explicitly recommend RAII for locks and recommend `std::lock` / `std::scoped_lock` for multiple mutexes.

14.4.2 Why RAII helps: it prevents accidental lock leaks

Deadlocks are often triggered or amplified when a lock is held longer than intended, especially on early-return paths or exceptions.

RAII lock wrappers (such as `std::lock_guard`, `std::unique_lock`, `std::scoped_lock`) ensure:

- the mutex is released on *every* exit path
- the lock lifetime is visible in the code structure (scope)

The Core Guidelines summarize this as: use RAII, never plain `lock() / unlock()`.

14.4.3 Rule 1: Never manually pair `lock()` and `unlock()` in application code

Bad pattern: easy to leak locks

```
#include <mutex>

std::mutex m;

void f(bool fail) {
    m.lock();
    if (fail) return; // BUG: m never unlocked
    // ...
    m.unlock();
}
```

This bug can turn into a deadlock immediately once any other thread tries to lock `m`.

Good pattern: scope-based locking

```
#include <mutex>

std::mutex m;
```

```
void f(bool fail) {
    std::lock_guard<std::mutex> lk(m); // unlocks on every path
    if (fail) return;
    // ...
}
```

14.4.4 Rule 2: If you lock more than one mutex, do not "hand-roll" ordering

Even experienced teams get lock ordering wrong over time as code evolves. Instead, use the standard library tools designed to avoid deadlocks.

14.4.5 Technique A (C++11+): `std::lock` + `adopt_lock`

What `std::lock` guarantees

`std::lock(lock1, lock2, ...)` locks multiple lockable objects using a **deadlock avoidance algorithm**. Cppreference also notes that it may use an unspecified series of `lock`, `try_lock`, and `unlock` calls, and that if an exception occurs during locking/unlocking, any already-locked objects are unlocked before rethrowing.

RAII adoption: make the locks exception-safe

After `std::lock`, immediately adopt the already-acquired locks into RAII guards:

```
#include <mutex>

std::mutex a, b;

void safe_transfer() {
```

```
    std::lock(a, b); // deadlock avoidance algorithm
    std::lock_guard<std::mutex> lk1(a, std::adopt_lock);
    std::lock_guard<std::mutex> lk2(b, std::adopt_lock);

    // critical section holding both a and b
}
```

The `adopt_lock` tag is a standard locking-strategy tag used to indicate that the calling thread already owns the mutex.

Project rule. If you use `adopt_lock`, it should appear immediately after the call that actually acquired the lock(s) (`std::lock`, or a direct `lock()`), and in the same scope.

14.4.6 Technique B (C++17+): `std::scoped_lock` for multiple mutexes

`std::scoped_lock` is an RAII lock that can manage one or more mutexes:

- it attempts to take ownership of the given mutexes on construction
- it releases them on destruction (scope exit)

If multiple mutexes are given, it uses a deadlock-avoidance algorithm *as if by* `std::lock`.

```
#include <mutex>

std::mutex a, b;

void safe_section() {
    std::scoped_lock lk(a, b); // RAII + deadlock avoidance
    // critical section holding both a and b
}
```

Core Guidelines explicitly recommend `std::lock()` or `std::scoped_lock` when acquiring multiple mutexes.

14.4.7 Technique C: `std::unique_lock` + `std::lock` (for complex flows)

Sometimes you need multi-locking plus the ability to unlock/relock or pass lock ownership. `std::unique_lock` supports deferred locking and can be used with `std::lock`.

```
#include <mutex>

struct Pair {
    std::mutex m;
    int value = 0;
};

void swap_values(Pair& x, Pair& y) {
    std::unique_lock<std::mutex> lx(x.m, std::defer_lock);
    std::unique_lock<std::mutex> ly(y.m, std::defer_lock);

    std::lock(lx, ly); // locks both using a deadlock-avoidance algorithm

    int tmp = x.value;
    x.value = y.value;
    y.value = tmp;
}
```

This pattern is also shown in cppreference's documentation around lock strategy tags and multi-locking.

14.4.8 Rule 3: Don’t call unknown code while holding a lock

Even if you avoid classic “two-mutex” deadlocks, you can create *re-entrancy* and *indirect* deadlocks by holding a lock while calling code you do not control:

- callbacks
- virtual functions
- user-provided functors

The Core Guidelines state this rule explicitly: never call unknown code while holding a lock.

Bad pattern

```
#include <mutex>
#include <functional>

std::mutex m;
std::function<void()> cb;

void update_and_notify() {
    std::lock_guard<std::mutex> lk(m);
    // update shared state...
    cb(); // unknown code while holding m: can deadlock or recurse
}
```

Safer pattern: copy state, unlock, then call out

```
#include <mutex>
#include <functional>

std::mutex m;
```

```
std::function<void()> cb;

void update_and_notify() {
    std::function<void()> local;
    {
        std::lock_guard<std::mutex> lk(m);
        // update shared state...
        local = cb; // copy callback under lock
    } // unlock before calling unknown code
    if (local) local();
}
```

14.4.9 Rule 4: Keep lock scope small and intentional

RAII makes it easy to see the lifetime of a lock. Use that visibility:

- lock only around the data that needs protection
- avoid holding locks across I/O, long computations, sleeps, or waits on other locks

Smaller critical sections reduce contention and reduce the surface area for deadlocks.

14.4.10 A compact deadlock-prevention checklist

- Use RAII locks; do not write naked `lock()/unlock()` pairs.
- When acquiring multiple mutexes, use `std::scoped_lock` (C++17+) or `std::lock` + `adopt_lock`.
- Never call unknown code while holding a lock.
- Keep critical sections small and document the locking discipline (which mutex protects which data).

14.4.11 Summary

RAII prevents deadlocks primarily by preventing *lock leaks* and making lock lifetimes explicit. For multi-mutex scenarios, the standard library provides deadlock-avoidance mechanisms:

- `std::lock`: locks multiple mutexes using a deadlock-avoidance algorithm
- `std::scoped_lock`: RAII wrapper that locks multiple mutexes using `std::lock`-style deadlock avoidance

Combined with the rule "never call unknown code while holding a lock," these practices eliminate the majority of practical deadlocks in C++ systems code.

Chapter 15

Atomics: When Mutexes Are Too Much

15.1 When atomics are appropriate

15.1.1 The decision you are really making

Choosing between atomics and mutexes is not mainly about speed. It is about **what kind of correctness guarantee you need**:

- A **mutex** protects a *compound invariant* (a transaction over multiple steps/variables) and provides blocking coordination.
- An **atomic** protects *one atomic object* (a single variable or a single machine-word-sized state) and provides non-blocking coordination.

C++ concurrency rules treat **data races as undefined behavior**, so your first goal is always to establish a correct synchronization story.

15.1.2 Appropriate uses of atomics (high-confidence patterns)

1) Simple counters and statistics

If you only need atomicity (no ordering with other memory), atomics are ideal. Cppreference explicitly points to relaxed ordering as a typical choice for counters, including reference counters.

```
#include <atomic>

std::atomic<unsigned long long> requests{0};

void on_request() {
    requests.fetch_add(1, std::memory_order_relaxed);
}
```

Use this pattern when:

- the counter value does not publish or protect other state
- occasional reordering relative to other operations is acceptable

2) One-bit or small state flags (with correct publication)

Flags are appropriate when they represent *a single state variable*, but you must choose the correct ordering.

Publish/consume (the classic case). When a flag is used to publish other (non-atomic) data, use release/acquire:

```
#include <atomic>
```

```
std::atomic<bool> ready{false};
int payload = 0; // ordinary data

void producer() {
    payload = 42;
    ready.store(true, std::memory_order_release);
}

int consumer() {
    while (!ready.load(std::memory_order_acquire)) { }
    return payload; // guaranteed to observe 42
}
```

Cppreference explains that ordering/synchronization semantics are what allow one thread's writes to become visible to another in a defined way.

Relaxed flags are *not* publication. If the flag is only a flag (no other data to publish), relaxed may be fine. But if you use relaxed as a publish mechanism, different threads may observe writes in different orders, which is exactly the failure mode you must avoid.

3) Single-variable state machines

Atomics work well when the shared state is representable as one atomic value:

- an enum state
- a pointer
- an index in a ring buffer (with the rest of the protocol designed around it)

Example: a simple one-time initialization state (conceptual pattern):

```
#include <atomic>

enum class State : unsigned { idle, initializing, ready };

std::atomic<State> state{State::idle};

bool try_become_initializer() {
    State expected = State::idle;
    return state.compare_exchange_strong(expected, State::initializing,
                                         std::memory_order_acq_rel);
}
```

This is appropriate when the correctness condition is “the state changes atomically” (not “several variables change together”).

4) Low-level building blocks inside concurrent libraries

Atomics are appropriate when you are implementing:

- lock-free data structures
- fast-path synchronization (e.g., adaptive locks, specialized queues)
- reference counting and lifetime coordination primitives

This is expert territory: you must reason about memory ordering, progress guarantees, and ABA-like hazards. The standard also makes it clear that lock-freedom is implementation-dependent (via `is_lock_free` and related wording).

15.1.3 When atomics are *not* the right tool

1) When you must preserve a multi-variable invariant

If correctness depends on multiple variables changing together (a transaction), prefer a mutex. Atomics do not automatically make a multi-step protocol atomic.
Examples that are usually mutex problems:

- "remove an item from a container, update size, update statistics"
- "transfer money between two accounts"
- "check then act" logic over multiple fields

2) When you need blocking coordination

Atomics do not put threads to sleep by themselves. If threads need to wait efficiently for a condition, you typically want:

- a mutex + `std::condition_variable`, or
- higher-level concurrency primitives built on those

3) When fairness and starvation avoidance matter

Atomics often produce busy-wait loops (spinning). Spinning can be appropriate in very short critical paths, but it is dangerous for fairness and can waste CPU under contention.
If your system must remain responsive under load, mutexes (and condition variables) frequently produce more predictable behavior.

4) When you cannot clearly explain the ordering story

If you cannot answer:

> What establishes the happens-before edge that makes writes visible to readers?

then you should not be using atomics for that purpose.

15.1.4 A practical selection guide (safe defaults)

Use a mutex when

- you protect a compound invariant (multiple related variables)
- you need to block/wait
- the code must stay easy to review and hard to misuse

Use atomics when

- the shared state is naturally one variable
- you only need atomicity (counters) *or* you can state a clear publication rule (release/acquire)
- you are implementing a well-specified concurrent algorithm

The Core Guidelines summarize the same core idea: use atomics where you might otherwise misuse `volatile`, and use a mutex for more complicated examples.

15.1.5 Memory-order choices: what is appropriate in practice

Cppreference's `std::memory_order` page is a good canonical reference for practical intent.

`memory_order_relaxed`

Appropriate for:

- counters, statistics, reference counts (when not publishing data)

because it provides atomicity without synchronization.

`memory_order_release` / `memory_order_acquire`

Appropriate for:

- publish/subscribe patterns (a flag or pointer that publishes data)
- hand-off protocols (producer/consumer state transitions)

because it establishes a clear synchronization edge for visibility.

`memory_order_seq_cst`

Appropriate when:

- you want the simplest global reasoning model (strongest ordering)
- performance is not yet a bottleneck, and clarity is the priority

It is often a safe starting point, then relax only with measured justification.

Fences

Fences are appropriate mainly for advanced patterns where you deliberately separate ordering from a specific atomic access. Cppreference emphasizes that fences establish ordering but still require at least one atomic operation to set up synchronization. In most application code, direct release/acquire operations on an atomic variable are clearer.

15.1.6 How to validate that your atomic design is correct

Because races can be subtle, professional practice is:

- use dynamic race detection (e.g., ThreadSanitizer) to catch data races early
- treat any reported data race as urgent, because data races are undefined behavior in C++11 and later

Large real-world systems have demonstrated that systematically deploying TSan finds impactful races that tests miss.

15.1.7 Summary

Atomics are appropriate when the shared state is fundamentally **one variable** and you can clearly state the ordering/visibility contract:

- **Counters**: relaxed atomic operations
- **Publication flags**: release/acquire synchronization
- **Small state machines and lock-free internals**: only with rigorous memory-order reasoning and validation

Use a mutex when the operation is a multi-step transaction, when you need blocking coordination, or when you cannot clearly describe the happens-before story.

15.2 Common misuse of atomics

15.2.1 Why atomic misuse is so dangerous

Atomics can make a program *well-defined* in the presence of concurrency, but only if you use them with a correct memory-ordering and synchronization story. If you do not, you typically get one of two outcomes:

- the program still has a **data race** (undefined behavior), or
- the program becomes **race-free but still wrong** due to missing ordering/visibility guarantees.

C++ is explicit: if a data race occurs, the program's behavior is undefined.

15.2.2 Misuse #1: Using `volatile` as if it were atomic or synchronization

In C++, `volatile` is not a concurrency primitive. It does not provide atomicity, does not synchronize between threads, and does not prevent reordering. The C++ Core Guidelines state this directly (CP.8).

Bad (still a data race)

```
volatile int free_slots = 100;

int use_slot() {
    return free_slots--; // data race: read-modify-write is not atomic
}
```

Good (make the shared variable atomic)

```
#include <atomic>
```

```
std::atomic<int> free_slots{100};

int use_slot() {
    return free_slots.fetch_sub(1, std::memory_order_relaxed);
}
```

Note: `memory_order_relaxed` is fine here only if the counter does not publish or protect other data. Relaxed operations are not synchronization operations.

15.2.3 Misuse #2: Treating `memory_order_relaxed` as if it "makes other writes visible"

`memory_order_relaxed` guarantees atomicity and participates in a single atomic's modification order, but it does not create inter-thread synchronization and does not impose ordering among other memory accesses.

Bad: relaxed flag used as publication

```
#include <atomic>

std::atomic<bool> ready{false};
int payload = 0;

void producer() {
    payload = 42;
    ready.store(true, std::memory_order_relaxed); // WRONG for
    ↪ publication
}

int consumer() {
```

```
    while (!ready.load(std::memory_order_relaxed)) { }
    return payload; // may observe stale payload
}
```

Good: release/acquire publication

```
#include <atomic>

std::atomic<bool> ready{false};
int payload = 0;

void producer() {
    payload = 42;
    ready.store(true, std::memory_order_release);
}

int consumer() {
    while (!ready.load(std::memory_order_acquire)) { }
    return payload; // publication is defined
}
```

Cppreference explains that `std::memory_order` controls how other memory accesses are ordered around an atomic operation.

15.2.4 Misuse #3: Mixing atomic and non-atomic access to the same object

A very common mistake is to use an atomic sometimes, but still read/write the same object non-atomically elsewhere.

Bad: atomic write + non-atomic read (data race)

```
#include <atomic>
```

```
std::atomic<int> x{0};

int racy_read() {
    return *reinterpret_cast<int*>(&x); // WRONG: non-atomic access to
    ↪ the same object
}
```

The rule is simple: if an object is shared and written by one thread while another reads or writes it, then the accesses must be synchronized (often by making them atomic, or by locking). Otherwise, a data race occurs and the behavior is undefined.

Special case: `std::atomic_ref` is not a "partial fix"

`std::atomic_ref` can apply atomic operations to an existing object, but while any `atomic_ref` exists, the referenced object must be accessed *exclusively* through those `atomic_ref` instances (no concurrent non-atomic access).

15.2.5 Misuse #4: Using atomics to protect a multi-variable invariant

Atomics make operations on *one variable* atomic. They do not automatically make a multi-step protocol "transactional."

Bad: check-then-act over multiple variables

```
#include <atomic>
#include <vector>

std::atomic<bool> has_item{false};
std::vector<int> items; // not thread-safe
```

```
int pop_one() {
    if (has_item.load(std::memory_order_acquire)) {
        int v = items.back();   // data race / UB (items is unprotected)
        items.pop_back();
        // ...
        return v;
    }
    return -1;
}
```

This mixes an atomic flag with unsynchronized accesses to non-atomic shared state, which violates the data-race rules.

Good default: protect the invariant with a mutex

When you need to mutate a container, maintain consistent size/count, and coordinate multiple steps, a mutex is the normal tool. (Atomics are best when the shared state *is* one variable, not when it merely *points to* many variables.)

15.2.6 Misuse #5: Incorrect `compare_exchange` usage

Mistake A: Not looping on `compare_exchange_weak`

`compare_exchange_weak` is allowed to fail spuriously, so it is normally used in a loop.

```
#include <atomic>

std::atomic<int> x{0};

void inc_bad() {
    int expected = x.load(std::memory_order_relaxed);
    x.compare_exchange_weak(expected, expected + 1,
```

```
                                std::memory_order_relaxed); // WRONG: single
                                ↪  attempt
}
```

Correct: CAS loop, and remember `expected` is updated on failure

```
#include <atomic>

std::atomic<int> x{0};

void inc_ok() {
    int expected = x.load(std::memory_order_relaxed);
    while (!x.compare_exchange_weak(expected, expected + 1,
                                    std::memory_order_relaxed,
                                    std::memory_order_relaxed)) {
        // on failure, expected has been overwritten with the current
        ↪  value
    }
}
```

Mistake B: Invalid/unsafe failure memory order

For compare-exchange, the failure order is used for the load that happens on failure. Cppreference documents that using a failure order stronger than the success order is undefined behavior, and (until C++17) certain failure orders are forbidden.

Mistake C: Assuming CAS on user-defined structs is “just fine”

Compare-exchange comparisons are bitwise (like `memcmp`), and padding bytes can matter. This is one reason many lock-free designs avoid CAS directly on structs unless the type is carefully designed for it.

15.2.7 Misuse #6: Assuming “lock-free” is guaranteed (or always faster)

Whether an atomic type is lock-free is implementation-dependent. C++ exposes this via `is_lock_free` and the `ATOMIC_*_LOCK_FREE` macros, and only guarantees that the result is consistent within a program for a given type.
Two common errors:

- **Correctness error**: writing an algorithm that relies on lock-freedom for progress guarantees without checking and without a fallback design.
- **Performance error**: assuming atomics beat locks under contention. High contention can make atomics slower due to cache-line ping-pong and spinning.

15.2.8 Misuse #7: Using condition variables with an atomic “condition” incorrectly

A condition variable is designed to be used with a mutex and a shared condition protected by that mutex.
A common misuse is to set an `atomic<bool>` outside the mutex and think this replaces the mutex discipline. The Core Guidelines explicitly warn that “atomic bool whose value is set outside of the mutex is wrong” for `condition_variable` usage.

Good pattern: condition protected by the same mutex

```
#include <condition_variable>
#include <mutex>

std::mutex m;
std::condition_variable cv;
bool ready = false; // protected by m
```

```
void producer() {
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
    }
    cv.notify_one();
}

void consumer() {
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{ return ready; });
}
```

Cppreference documents that `wait` atomically unlocks the mutex and blocks, then re-locks before returning, and that the predicate form is the standard way to handle spurious wakeups.

15.2.9 Misuse #8: Treating fences as "magic synchronization"

`std::atomic_thread_fence` imposes ordering constraints, but fences are advanced tools and are often misapplied. Cppreference emphasizes that fences establish ordering constraints as instructed by the order, without being an atomic read-modify-write by themselves.
In most application code, prefer explicit release/acquire operations on a specific atomic variable instead of fences, because the synchronization intent is clearer and easier to review.

15.2.10 Misuse #9: Forgetting what atomics do *not* guarantee

Atomics do not automatically provide:

- **mutual exclusion** for compound operations on non-atomic data

- **fairness** (spinning can starve other threads)
- **a correct design** if the algorithm itself is flawed

If you cannot explain, in one paragraph, the happens-before relationship that makes your design correct, assume the atomic usage is wrong and use a mutex until proven otherwise. The Core Guidelines start with “avoid data races” as a foundational rule.

15.2.11 A professional “atomic misuse” checklist

- Are all accesses to the shared object atomic (or all protected by the same mutex), with no mixing?
- Did you avoid using `volatile` for synchronization?
- Are you using relaxed operations only when you truly need atomicity but not ordering?
- If publishing data, did you use release/acquire (or stronger) in the correct places?
- If using CAS, is it a loop for `compare_exchange_weak`, and are the memory orders valid?
- If using `atomic_ref`, is the referenced object accessed exclusively through `atomic_ref` while it exists?
- If using `condition_variable`, is the condition protected by the same mutex (not “an atomic flag on the side”)?
- Are you relying on lock-freedom without checking `is_lock_free` and without a fallback plan?

15.3 Performance vs correctness

15.3.1 The rule that prevents most mistakes

In concurrent C++, **correctness comes first**. If your program has a **data race**, its behavior is **undefined**. No performance win matters if the result is not well-defined.
The practical ordering is:

1. Make the program **race-free** (well-defined).
2. Make the synchronization strategy **reviewable** (easy to reason about).
3. Only then, consider performance improvements and measure them.

15.3.2 Why "atomics are faster" is not a reliable rule

Atomics are sometimes faster than mutexes, sometimes slower, and sometimes catastrophically slower under contention. The performance depends primarily on:

- contention level (how many threads touch the same cache line)
- whether waiting is mostly spinning (atomics) or mostly blocking (mutex)
- memory ordering strength (relaxed vs acquire/release vs seq_cst)
- CPU and OS behavior (scheduling, preemption, topology)

Mutexes also provide a clear synchronization guarantee: `unlock()` **synchronizes-with** a later successful `lock()` on the same mutex. Atomics provide many possible ordering models, and the chosen model determines both correctness and cost.

15.3.3 The performance model: what you are paying for

Atomics

Atomic operations can be cheap when uncontended, but they still have costs:

- cache coherence traffic (ownership of the cache line must move between cores)
- stronger memory order may add barriers or constrain reordering
- busy-wait loops can burn CPU and starve other work

Relaxed atomics are **not synchronization operations** (no ordering for other memory), so they can be cheaper, but they cannot safely publish other data.

Mutexes

Mutexes typically have:

- fast paths when uncontended (implementation-dependent)
- blocking behavior under contention (threads may sleep and later wake)

Blocking can be slower for very short critical sections, but it can also be *more scalable* than spinning when contention is high, because blocked threads do not continuously consume CPU.

15.3.4 Correctness constraints that shape performance choices

Constraint 1: visibility and ordering are part of correctness

Absent constraints, different threads can observe memory updates in different orders. This is why “it works in tests” is not evidence.

If you need a thread to see the results of another thread's work, you must create a proper synchronization edge:

- mutex unlock → mutex lock (synchronizes-with)
- atomic release store → atomic acquire load (when the acquire reads from that release)

Constraint 2: relaxed atomics are not a "cheap mutex"

Relaxed operations guarantee atomicity of the atomic object, but provide no ordering among other memory accesses. So if correctness requires publication, relaxed is not an option.

15.3.5 A safe way to think about memory orders (performance through minimal correctness)

`memory_order_relaxed`: fastest when you only need atomicity

Use when:

- the atomic is a counter/statistic
- you do not rely on it to publish or protect other data

```
#include <atomic>

std::atomic<unsigned long long> hits{0};

void record_hit() {
    hits.fetch_add(1, std::memory_order_relaxed);
}
```

This is correct because the counter value does not define ordering for other memory.

`memory_order_release` / `memory_order_acquire`: correct publication at moderate cost

Use when:

- a flag or pointer publishes initialized data
- you need a clear "handoff" between threads

```
#include <atomic>

std::atomic<bool> ready{false};
int payload = 0;

void producer() {
    payload = 123;
    ready.store(true, std::memory_order_release);
}

int consumer() {
    while (!ready.load(std::memory_order_acquire)) { }
    return payload; // guaranteed to observe 123
}
```

Cppreference defines memory orders as controlling how regular (non-atomic) accesses are ordered around atomic operations.

`memory_order_seq_cst`: simplest reasoning, potentially higher cost

Sequentially consistent operations impose the strongest ordering constraints. They can be a reasonable default when:

- correctness is still being established

- performance is not yet measured
- you want the most conservative ordering model

Then, relax only with evidence and clear reasoning.

15.3.6 When a mutex beats atomics for performance

This can happen in practice when:

- contention is high and the atomic approach spins
- the atomic variable sits on a hot cache line shared across cores (cache-line ping-pong)
- the critical section is not actually tiny (busy-waiting is wasteful)
- you need to protect a compound invariant (many atomics or retries are more expensive than one lock)

A mutex blocks rather than spinning, and also provides a clear synchronization guarantee (unlock synchronizes-with later lock).

15.3.7 When atomics beat a mutex for performance

This can happen when:

- the shared state is truly one variable (counter, flag, pointer, small state machine)
- contention is low or moderate
- you can use the weakest correct ordering (often relaxed for counters, acquire/release for publication)

The key phrase is *weakest correct ordering*. **Correctness determines the minimum ordering**. Performance only determines whether that design meets goals.

15.3.8 Correctness pitfalls that look like “performance optimizations”

Pitfall 1: replacing a correct mutex with a relaxed atomic flag

This often yields a program that is faster in tests and wrong in production. Relaxed atomics are not synchronization operations.

Pitfall 2: using atomics where you need a transaction

If multiple variables must change together, using separate atomics usually creates intermediate states that other threads can observe. That can be logically incorrect even if there is no data race.

Pitfall 3: assuming “lock-free” means “fast”

Whether an atomic is lock-free is implementation-dependent, and lock-free code can still be slower due to contention and retries. Treat lock-free as a progress property, not a speed guarantee.

15.3.9 How to measure safely without fooling yourself

Always validate correctness with race detection

Before tuning performance, validate that there are no data races: ThreadSanitizer is a dedicated data race detector for C/C++, and explicitly notes that C++11 bans data races as undefined behavior.

Benchmark under realistic contention

Microbenchmarks can be misleading because:

- the OS scheduler changes timing and contention
- cache effects dominate in ways that tests do not reproduce
- different CPUs behave differently

Measure with representative thread counts, real workloads, and real deployment settings.

15.3.10 Practical guidance (what to put in a team coding standard)

- Prefer a mutex for compound invariants and any design that requires blocking coordination.
- Prefer atomics for truly single-variable shared state (counters, flags, pointers) with a clearly stated ordering rule.
- Use `memory_order_relaxed` only when atomicity is enough; otherwise use acquire/release (or stronger).
- Use tools (especially data-race detection) as part of the workflow.

15.3.11 Summary

Performance and correctness are not competing goals; correctness defines the legal designs, and performance chooses among them.

- Data races are undefined behavior, so a “fast” racy program is not a valid baseline.
- Atomics can be excellent for single-variable coordination, but memory order determines both correctness and cost.

- Mutexes remain a strong default for compound invariants and high-contention coordination, with a clear synchronization guarantee (unlock synchronizes-with later lock).

Part V

Polymorphism, Templates, and Type Safety

Chapter 16

Virtual Functions and Object Destruction

16.1 Vtable Basics

16.1.1 What a “vtable” is (and what it is not)

In everyday C++ discussions, *vtable* (virtual function table) and *vptr* (virtual table pointer) are names for the most common **implementation technique** used by compilers to support **dynamic dispatch** of virtual functions.

- The **C++ standard does not require** any particular memory layout for objects, and it does not mandate that a “vtable” exists as a concrete structure. What the standard **does** specify is the *observable behavior*: which overrider is selected, what happens during construction/destruction, and the rules for object lifetime and polymorphic deletion.
- In practice, mainstream compilers implement virtual dispatch using a per-class table of function pointers (the *vtable*) and a per-object (or per-subobject) hidden pointer (the *vptr*) that selects the appropriate table.

A class is commonly called **polymorphic** if it declares or inherits at least one virtual function. Polymorphic objects typically carry at least one hidden pointer used for dispatch.

16.1.2 The key mental model

For most ABIs used in real systems, you can use this simple model:

1. Every polymorphic *subobject* has (at least) one hidden pointer (**vptr**).
2. The vptr points to a read-only table for the dynamic type (**vtable**).
3. A virtual call uses the vptr to fetch the function address from a fixed slot in the table.

Conceptually, a call like `p->f()` (where `f` is virtual) behaves like:

```
// Pseudocode mental model (NOT real C++).
auto table = p->vptr;            // hidden
auto fn    = table[slot_of_f];   // slot is determined by the class
↪  layout rules of the ABI
fn(p);                           // 'this' is passed
```

This model explains two practical truths:

- Virtual calls are **indirect calls** (they typically inhibit inlining and certain optimizations).
- The exact table structure is **ABI- and compiler-dependent**, not something portable C++ code may assume.

16.1.3 What the standard *does* guarantee about virtual dispatch

Even though the standard does not define vtables, it precisely defines **dispatch rules**. Two rules are essential for safety:

Rule 1: Dispatch is based on the *dynamic type* (in normal execution)

If you call a virtual function through a base pointer or reference, the most-derived override (*final overrider*) is selected at runtime.

Rule 2: During construction and destruction, dispatch is restricted

If a virtual function is called (directly or indirectly) from a **constructor** or **destructor** for an object that is currently under construction or destruction, the call does **not** dispatch to a more-derived override. Instead, it resolves to the final overrider in the constructor's or destructor's class.
This rule is crucial because, during base construction, the derived part does not exist yet; during base destruction, the derived part has already been destroyed.

```
#include <iostream>

struct Base {
  Base() { f(); }                // calls Base::f(), not Derived::f()
  virtual ~Base() { f(); }       // calls Base::f(), not Derived::f()
  virtual void f() { std::cout << "Base::f\n"; }
};

struct Derived : Base {
  void f() override { std::cout << "Derived::f\n"; }
};

int main() {
  Derived d;
}
```

The output is typically `Base::f` twice, even though the dynamic type is `Derived` in `main`. This is **by language rule**, not an "ABI quirk".

16.1.4 Object layout in practice: common ABI patterns

The details below describe *common implementations*, not portable language rules.

Single inheritance (the simplest common case)

With single inheritance and no virtual inheritance, many ABIs place one vptr in the object, often at or near the beginning of the most-derived object (this is common but not guaranteed).

```
struct A {
  virtual void f();
  int x;
};

struct B : A {
  void f() override;
  int y;
};
```

Typical outcome (conceptual):

- Objects of `A` and `B` carry a vptr for the `A` subobject.
- `B`'s vptr points to `B`'s vtable, so `A::f` dispatches to `B::f`.

Multiple inheritance

With multiple inheritance, a derived object contains multiple base subobjects, and **each polymorphic base subobject may have its own vptr**. As a result:

- There can be **multiple vptrs** inside one most-derived object.
- A conversion from `Derived*` to `Base2*` may adjust the pointer value.

- Vtables often contain **thunks** (small adapter functions) to adjust `this` before calling the final overrider.

This explains why the following may require pointer adjustment:

```
struct B1 { virtual void f(); };
struct B2 { virtual void g(); };

struct D : B1, B2 {
  void f() override;
  void g() override;
};

void use(B2* p) { p->g(); }   // 'p' points to the B2 subobject inside D
```

Virtual inheritance

Virtual inheritance adds another layer: the location of a virtual base subobject is not a simple fixed offset known at compile time for all uses. ABIs typically store extra data (e.g., virtual base offsets) so the runtime can find the virtual base correctly. This is one reason virtual inheritance is often larger and slower than non-virtual inheritance in low-level code.

16.1.5 Virtual destructors and safe polymorphic deletion

The safety problem

Deleting an object through a base pointer requires that the base destructor is **virtual**. Otherwise, deleting through the base pointer has undefined behavior (the derived destructor may not run, and the deallocation may be wrong).

```
#include <cstdlib>
```

```
struct Base {
  // BUG: non-virtual destructor in a base meant for polymorphic use
  ~Base() = default;
  virtual void f() {}
};

struct Derived : Base {
  ~Derived() { /* release Derived resources */ }
};

int main() {
  Base* p = new Derived{};
  delete p; // Undefined behavior: Base::~Base is not virtual
}
```

The professional rule

If a class is intended to be used polymorphically (i.e., deleted through a base pointer), give it a **virtual destructor**. A common safe pattern is:

```
struct Base {
  virtual ~Base() = default;
  virtual void f() = 0;  // often abstract
};
```

With a virtual destructor, the delete-expression performs dynamic dispatch to the correct most-derived destructor. In typical vtable-based implementations, destructors occupy vtable entries (often more than one entry due to ABI needs such as deleting destructors), but you must treat that as ABI detail.

16.1.6 “Pure virtual” and vtable-related undefined behavior

Calling a pure virtual function

Calling a pure virtual function (directly or indirectly) during construction or destruction is a common source of undefined behavior in real code bases, especially when base constructors call virtual hooks.

```
struct Base {
  Base() { f(); }              // dangerous design
  virtual void f() = 0;        // pure virtual
  virtual ~Base() = default;
};

struct Derived : Base {
  void f() override {}         // never reached from Base::Base()
};

int main() {
  Derived d;                   // UB may occur during Base() when it calls
   ↪  f()
}
```

Even when the virtual is not pure, calling virtuals from constructors/destructors is usually a design smell because the dispatch rules intentionally do not behave like “normal runtime polymorphism” at those times.

16.1.7 Performance reality: cost is often indirect

A vtable-based virtual call usually costs more than a direct call, but the **biggest** cost is often **optimization loss**:

- Inlining is harder (sometimes impossible) across a virtual call.

- The compiler may not propagate constants or remove branches that would be removable with a direct call.
- Indirection can hurt branch prediction and instruction-cache locality in hot paths.

Modern compilers can often remove the cost by **devirtualization** when they can prove the dynamic type (or when the class/function is marked `final` and the optimizer can exploit that). This is optimization behavior, not a language guarantee.

16.1.8 Security and robustness implications (why vtable basics matter)

Understanding vtable-based dispatch helps explain several high-impact bug classes:

Use-after-free on polymorphic objects

If a polymorphic object is used after being destroyed, the vptr may point to memory that has been reused. A virtual call may jump through a corrupted vptr/vtable, turning a memory-safety bug into control-flow hijacking in unsafe code bases.

Type confusion

Mis-casting pointers in complex inheritance hierarchies (especially with multiple/virtual inheritance) can produce an invalid subobject pointer. Virtual calls then use the wrong vptr, leading to undefined behavior.

Slicing

Object slicing removes the derived part by copying into a base object by value, which also discards the derived dynamic type. Virtual dispatch then follows the base object's dynamic type, not the original.

```cpp
#include <iostream>

struct Base {
  virtual ~Base() = default;
  virtual void f() const { std::cout << "Base\n"; }
};

struct Derived : Base {
  void f() const override { std::cout << "Derived\n"; }
};

void takes_by_value(Base b) { b.f(); } // slices

int main() {
  Derived d;
  takes_by_value(d); // prints "Base"
}
```

16.1.9 Practical guidance: what to rely on, what to avoid

Safe assumptions

- Rely on the **language rules** of virtual dispatch and destruction.
- Use **virtual destructors** for polymorphic bases that may be deleted through a base pointer.
- Prefer designs that avoid virtual calls during construction/destruction.

Unsafe assumptions

- Do not assume where the vptr is stored, how many exist, or what a vtable looks like.

- Do not inspect or patch vtables in portable C++.
- Do not assume a specific ABI unless you explicitly target one platform/toolchain and treat it as non-portable.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard), clauses covering virtual functions, object lifetime, and construction/destruction dispatch rules.
- *Itanium C++ ABI*, sections on virtual tables and object layout (widely followed by GCC and Clang on many non-Windows targets).
- cppreference.com, “virtual function specifier” (rules for virtual dispatch, including behavior during construction and destruction).
- ISO C++ (isocpp.org) FAQ, “virtual functions” (high-level explanation of dispatch via vptr/vtable).
- LLVM Project documentation and sources related to devirtualization and vtable handling (for optimization reality in modern compilers).

16.2 Virtual Destructors

16.2.1 Why virtual destructors exist

A destructor is the mechanism that ends an object’s lifetime and releases its resources. In C++, **polymorphism** allows you to manipulate objects through pointers/references to a base class, even when the dynamic type is a derived class.
That flexibility becomes dangerous at destruction time.

- If you **delete through a base pointer**, C++ must destroy the *most-derived* object (derived destructor first, then base destructors), and it must deallocate the correct amount of memory using the correct deallocation function.
- This is only guaranteed when the base class destructor is **virtual**.

The rule is simple and strict:

> **If a class is used polymorphically and is deletable through a base pointer, the base destructor must be virtual.**

This rule is stated in the C++ standard's requirements for the delete-expression and is explained in authoritative references such as cppreference and vendor documentation.

16.2.2 The core safety rule: deleting through a base without a virtual destructor

Deleting a derived object through a base pointer when the base destructor is **not virtual** results in **undefined behavior**.

What goes wrong

When the base destructor is non-virtual, the delete-expression is not required to call the derived destructor. That can cause:

- **Resource leaks** (the derived destructor never runs).
- **Incomplete cleanup** (file handles, locks, sockets, memory owned by derived members).

- **Wrong deallocation** (especially if classes define custom `operator delete`, use aligned allocation, or rely on sized deallocation).
- **Hard-to-debug corruption** because the program has entered UB; anything can happen.

Incorrect example (undefined behavior)

```
#include <iostream>

struct Base {
  ~Base() { std::cout << "~Base\n"; }   // NOT virtual
  virtual void f() {}                  // polymorphic type
};

struct Derived : Base {
  ~Derived() { std::cout << "~Derived\n"; }
};

int main() {
  Base* p = new Derived{};
  delete p; // Undefined behavior
}
```

Even if this *seems* to print something reasonable in one build, it is still undefined behavior and may break under optimization, LTO, a different compiler, or different allocator settings.

16.2.3 Correct design: virtual destructor in a polymorphic base

The standard, safe pattern

```
struct Base {
```

```
  virtual ~Base() = default;  // enables safe polymorphic deletion
  virtual void f() = 0;       // often an interface
};

struct Derived : Base {
  ~Derived() override = default;
  void f() override {}
};

int main() {
  Base* p = new Derived{};
  delete p; // well-defined: calls ~Derived then ~Base
}
```

Important details

- **Destructors are special**, but they still participate in virtual dispatch. If the base destructor is virtual, the derived destructor *overrides* it automatically, even though destructors are not "inherited" like normal member functions.
- Marking the derived destructor `override` is optional but recommended: it makes the intent explicit and lets the compiler diagnose mistakes (for example, if the base destructor was accidentally not virtual).

16.2.4 When you should *not* make the destructor public and virtual

Sometimes you intentionally design a base class to be polymorphic, but you do **not** want users to delete through a base pointer. Common cases:

- The object has **non-heap lifetime management** (owned by a factory, an arena, reference counting, intrusive lists, or a framework).

- The base class is intended only as a **view** or **non-owning interface**.

In such cases, a safe pattern is a **protected non-virtual destructor** (or sometimes protected virtual, depending on the framework) to prevent `delete base_ptr;` from compiling:

```
struct NonDeletableBase {
  virtual void f() = 0;

protected:
  ~NonDeletableBase() = default; // not public: prevents deletion through
  ↪  base
};
```

This design makes the ownership rule enforceable at compile time.

16.2.5 Pure virtual destructors (abstract bases) and the "definition required" rule

A destructor may be declared **pure virtual** to make a class abstract. However, the destructor still runs during destruction of derived objects, so it still needs a definition.

```
struct AbstractBase {
  virtual ~AbstractBase() = 0; // pure virtual
};

AbstractBase::~AbstractBase() = default; // REQUIRED definition
```

If you omit the definition, you will typically get a linker error, because the destructor is odr-used when destroying derived objects.

16.2.6 Implementation reality: what "virtual destructor" usually means in the ABI

This section explains common practice, not portable layout rules.
When a base destructor is virtual, the delete-expression needs two actions:

1. Destroy the object (run the complete chain of destructors for the most-derived object).
2. Deallocate the storage using the correct deallocation function.

Many ABIs therefore model virtual destruction using more than one internal entry point (often described as *complete object destructor* and *deleting destructor*). This supports the fact that `delete p;` is not merely "call destructor"; it is "destroy and then deallocate".
You should **never** write code that depends on the number of vtable slots or the names of compiler-generated destructors. Treat these as ABI details.

16.2.7 Interaction with `operator delete` (including sized and aligned deallocation)

Modern C++ allows customization of deallocation:

- class-specific `operator delete`
- (since C++14) **sized deallocation** overloads
- (since C++17) **over-aligned** new/delete overloads

A virtual destructor helps ensure that:

- The **correct deallocation function** is selected for the dynamic type when deleting through a base pointer (as required by the standard's rules for `delete`).

- Any dynamic-type-specific destruction logic runs before the memory is returned to the allocator.

In safety-critical code, this matters because mismatching allocation and deallocation strategies is a common source of corruption.

16.2.8 Cost model and when it matters

A virtual destructor does not make every destruction expensive; it only matters when destruction is performed through a base pointer/reference (or when RTTI and other polymorphic features are used). The practical costs are:

- A polymorphic object typically stores at least one hidden pointer (commonly a vptr).
- A virtual call to the destructor through a base pointer involves an indirection.

For most systems, this overhead is negligible compared to the safety benefit. If you need maximum performance and you know the static type at the call site, prefer automatic storage (stack) and RAII, or avoid polymorphic deletion entirely by using value semantics.

16.2.9 Professional guidelines (safe defaults)

Do this

- If a base class has any virtual function and you expect deletion through base pointers, declare: `virtual ~Base() = default;`
- Mark derived destructors `override` (recommended).
- Prefer owning smart pointers (`std::unique_ptr<Base>`) when owning polymorphic objects.

- Consider `final` on derived classes when appropriate to prevent further inheritance and enable better optimization and reasoning.

Avoid this

- A polymorphic base class with a **public non-virtual destructor**.
- Calling virtual functions from constructors/destructors expecting derived behavior (dispatch is restricted during construction/destruction).
- "Manual" ownership conventions that allow raw `new`/`delete` across module boundaries without a clear rule for who deletes.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard), delete-expression rules and requirements for polymorphic deletion (`[expr.delete]`).
- cppreference.com, pages: *virtual* (virtual destructor rules) and *destructor* (deleting through base and UB).
- ISO C++ (isocpp.org) FAQ, destructors and polymorphic use guidance.
- Microsoft Learn (MSVC documentation), destructors and virtual destructors in C++.
- Itanium C++ ABI specification, definitions of complete object destructor and deleting destructor (ABI-level explanation).

16.3 Object Slicing

16.3.1 Definition: what “object slicing” means

Object slicing happens when an object of a derived class is **copied (or assigned) by value** into an object of a base class. Only the *base subobject* is copied; the derived part is discarded. After slicing:

- The resulting object is a **pure base object**, not a derived object.
- Any derived data members are lost.
- Virtual dispatch now behaves as the base object’s dynamic type (the base), because the object *is* a base object.

Slicing is not undefined behavior by itself. It is usually a **logic error** that silently breaks polymorphism and can cause resource and correctness issues.

16.3.2 The root cause: value semantics with a polymorphic base

Slicing is fundamentally caused by **treating a polymorphic hierarchy as a value type**. This typically occurs in these patterns:

- Passing a derived object to a function that takes **Base by value**.
- Returning **Base by value** from a function that constructs a derived.
- Storing derived objects inside containers of **Base values** (e.g., `std::vector<Base>`).
- Assigning `Base b = derived;` or `b = derived;`

16.3.3 Canonical example: passing by value slices

```
#include <iostream>

struct Base {
  virtual ~Base() = default;
  virtual void print() const { std::cout << "Base\n"; }
};

struct Derived : Base {
  int secret = 42;
  void print() const override { std::cout << "Derived secret=" << secret
  ↪  << "\n"; }
};

void log_by_value(Base b) {   // SLICES
  b.print();                  // prints "Base"
}

void log_by_ref(const Base& b) { // preserves polymorphism
  b.print();                     // prints dynamic type
}

int main() {
  Derived d;
  log_by_value(d);   // slicing: Derived -> Base
  log_by_ref(d);     // no slicing
}
```

Key observation:

- `log_by_value` receives a **Base object**, not a reference.
- The derived part is discarded during copy construction.

16.3.4 Slicing in containers: `std::vector<Base>` is a red flag

This is one of the most common real-world slicing bugs.

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

struct Base {
  virtual ~Base() = default;
  virtual void print() const { std::cout << "Base\n"; }
};

struct Derived : Base {
  void print() const override { std::cout << "Derived\n"; }
};

int main() {
  std::vector<Base> v;
  v.push_back(Derived{});   // slicing occurs here (Base is stored)

  for (const Base& b : v) {
    b.print();              // prints "Base"
  }
}
```

The container stores **Base values**. Even though a virtual function exists, the dynamic type of each element is **Base**, because slicing has already happened.

16.3.5 Why slicing is a safety problem (not just “wrong output”)

Slicing commonly escalates from a logic bug into a safety bug in these ways:

Lost invariants and partial state

The derived part may enforce invariants (bounds, permissions, reference counting, validation, synchronization). After slicing, those invariants no longer exist.

Resource handling and ownership confusion

Even with a virtual destructor, slicing changes what is owned by the object. If the design assumes derived members manage critical resources, slicing can silently drop that management layer and cause leaks or missing cleanup actions at higher layers.

Security assumptions

Type-based decisions (authorization, policy objects, audit hooks) may be implemented via overrides. Slicing removes the override behavior and can bypass intended checks.

16.3.6 How to prevent slicing by design

Rule 1: Pass polymorphic types by reference or pointer

- Use `const Base&` for read-only polymorphic use.
- Use `Base&` when modification is needed.
- Use `Base*` / `const Base*` when nullability is part of the API.

Rule 2: Store polymorphic objects via owning pointers

Prefer `std::unique_ptr<Base>` (single ownership) or `std::shared_ptr<Base>` (shared ownership).

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

struct Base {
  virtual ~Base() = default;
  virtual void print() const = 0;
};

struct Derived : Base {
  void print() const override { std::cout << "Derived\n"; }
};

int main() {
  std::vector<std::unique_ptr<Base>> v;
  v.push_back(std::make_unique<Derived>());

  for (const auto& p : v) {
    p->print(); // prints "Derived"
  }
}
```

Rule 3: Make slicing a compile-time error when appropriate

If a base class is meant to be used polymorphically, it often should not be copyable as a *value*. A robust technique is to **delete** copy operations in the base:

```
struct Base {
  Base() = default;
  Base(const Base&) = delete;
  Base& operator=(const Base&) = delete;
```

```
  virtual ~Base() = default;
  virtual void f() = 0;
};
```

This prevents accidental pass-by-value, container-by-value, and assignment that would slice.

16.3.7 When value semantics are desirable: alternatives to inheritance

Sometimes you want heterogeneous behavior *and* value semantics. Prefer one of these:

`std::variant` (closed set of types)

Use when the set of alternatives is known and fixed.

```
#include <variant>
#include <iostream>

struct Circle { int r; };
struct Square { int a; };

using Shape = std::variant<Circle, Square>;

int main() {
  Shape s = Circle{3};
  std::visit([](const auto& x) {
    if constexpr (std::is_same_v<std::decay_t<decltype(x)>, Circle>)
      std::cout << "Circle r=" << x.r << "\n";
    else
      std::cout << "Square a=" << x.a << "\n";
  }, s);
}
```

Type erasure (open set of types)

Use when you want runtime polymorphism but also want value-like behavior. Examples include standard facilities such as `std::function` and many library "any-shape" wrappers. A custom type-erased wrapper can provide deep-copy, move, and destruction without slicing.

16.3.8 Advanced pattern: polymorphic copy via `clone()`

If you truly need "copying" of polymorphic objects, define a virtual `clone` that returns an owning pointer.

```
#include <memory>

struct Base {
  virtual ~Base() = default;
  virtual std::unique_ptr<Base> clone() const = 0;
};

struct Derived : Base {
  int x = 7;
  std::unique_ptr<Base> clone() const override {
    return std::make_unique<Derived>(*this);
  }
};
```

This preserves the dynamic type and avoids slicing. It also makes copying explicit (and usually more expensive), which is exactly what you want for safety.

16.3.9 Common misconceptions and clarifications

“But I have virtual functions; why did it print Base?”

Because slicing changes the dynamic type. After slicing, the object really is a `Base` object, so virtual dispatch correctly selects `Base::print()`.

Slicing is not the same as deleting through a base pointer

- Slicing is about **copying by value** and losing the derived part.
- Polymorphic deletion is about **destroying via base pointer**, which requires a virtual destructor.

Slicing can happen even with a virtual destructor

A virtual destructor protects **deletion**. It does not prevent **value copies** from slicing.

16.3.10 Professional checklist

- Do not pass polymorphic bases by value.
- Do not store polymorphic bases by value.
- If a base is polymorphic, consider deleting copy/move operations to prevent slicing.
- Use `std::unique_ptr<Base>` or `std::shared_ptr<Base>` for ownership.
- If you need value semantics for heterogeneous types, consider `std::variant` or type erasure.
- If you need polymorphic copying, use an explicit `clone()` pattern.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard), object model rules for base-class subobjects, copy initialization, and class object semantics.
- cppreference.com, topics: inheritance, object slicing (explained via copy initialization), virtual functions, and polymorphic types.
- ISO C++ Core Guidelines, rules discouraging slicing and recommending pointer/reference semantics for polymorphic hierarchies.
- *Effective C++* / *Effective Modern C++* (Scott Meyers), items discussing polymorphic base classes, passing by reference, and preventing slicing (conceptual guidance aligned with standard rules).

16.4 Polymorphic Ownership

16.4.1 What "polymorphic ownership" means

Polymorphic ownership is the problem of *who owns and destroys* a dynamically allocated object whose **dynamic type** is derived, but which is accessed through a **base interface**:

- `Base* p = new Derived;`
- `std::unique_ptr<Base> p = std::make_unique<Derived>();`

Ownership is not about virtual dispatch itself. It is about **lifetime**: who guarantees the object stays alive, and who performs destruction and deallocation exactly once.
A correct design must answer, in a way the compiler can help enforce:

1. **Who owns the object?** (single owner, shared owners, or external owner)

2. **How is deletion performed?** (virtual destructor, custom deleter, factory destroy function)
3. **How is borrowing expressed?** (references, raw pointers, observers, views)
4. **Where do boundaries exist?** (shared library boundary, plugin ABI, allocator boundary)

16.4.2 The hard rule: polymorphic deletion requires a virtual destructor

If an object may be deleted through a base pointer, the base must have a **public virtual destructor**. Otherwise `delete basePtr;` is undefined behavior.

```
struct Base {
  virtual ~Base() = default;  // required for safe deletion via Base*
  virtual void run() = 0;
};
```

This is the foundation of safe polymorphic ownership.

16.4.3 Preferred model: single ownership with `std::unique_ptr<Base>`

For most systems code, **single ownership** is the safest default. It is explicit, efficient, and easy to reason about.

Factory returns owning pointer

```
#include <memory>
#include <string>
```

```cpp
struct Base {
  virtual ~Base() = default;
  virtual std::string name() const = 0;
};

struct Derived final : Base {
  std::string name() const override { return "Derived"; }
};

std::unique_ptr<Base> make_object() {
  return std::make_unique<Derived>();
}

int main() {
  auto p = make_object();  // ownership is clear
}
```

Why this is the default

- **No leaks by construction**: RAII ties destruction to scope exit.
- **No double delete**: move-only semantics prevent accidental copying.
- **Clear contracts**: if you have a `unique_ptr`, you own.
- **Works with polymorphism**: virtual destructor ensures correct destruction.

16.4.4 Borrowing vs owning: express intent in the type

A major safety failure in polymorphic designs is **mixing ownership and borrowing** using raw pointers without a clear rule.

Recommended conventions

- **Owning**: `std::unique_ptr<Base>` or `std::shared_ptr<Base>`
- **Borrowing (non-owning)**: `Base&`, `const Base&`, `Base*`, `const Base*`
- **Optional borrow**: pointer (nullable) expresses optionality; reference expresses non-null.

```
void execute(Base& b);           // does not own, cannot be null
void execute_optional(Base* b); // does not own, may be null
```

Do not return raw owning pointers

Returning `Base*` from a factory invites leaks and confusion:

```
// Avoid
Base* make_bad(); // who deletes? when? with which allocator?
```

Prefer returning an owning smart pointer.

16.4.5 Shared ownership: `std::shared_ptr<Base>` when it is truly needed

Shared ownership is appropriate only when the object must remain alive as long as *any* user still needs it, and there is no clear single owner.

Correct use pattern

```
#include <memory>

struct Base {
```

```
  virtual ~Base() = default;
  virtual void tick() = 0;
};

struct Derived : Base {
  void tick() override {}
};

int main() {
  std::shared_ptr<Base> p = std::make_shared<Derived>();

  auto p2 = p; // shared ownership
  p->tick();
}
```

Risks and costs

- **Cost**: reference counting implies atomic increments/decrements in many implementations.
- **Cycles leak**: `shared_ptr` cycles do not reclaim automatically.
- **Ownership becomes fuzzy**: “everyone owns” can become “no one is responsible”.

Break cycles with `std::weak_ptr`:

```
#include <memory>

struct Node {
  std::shared_ptr<Node> next;
  std::weak_ptr<Node>   prev; // weak breaks cycle
};
```

16.4.6 Custom destruction: polymorphic ownership across ABI or allocator boundaries

Polymorphic ownership gets more dangerous when creation and destruction happen in different modules (plugins, shared libraries) or when allocators differ.

Why this matters

- A `delete` expression must match the allocation mechanism (`new` vs custom allocator, alignment, sized deallocation).
- Across module boundaries, you can encounter different CRTs or allocators. The safe rule is: **destroy in the same module that created**.

Pattern 1: factory returns `unique_ptr` with a custom deleter

```
#include <memory>

struct Base {
  virtual void run() = 0;
protected:
  // Not necessarily public if deletion must be controlled.
  virtual ~Base() = default;
};

struct Deleter {
  void operator()(Base* p) const noexcept {
    // Calls into the module that created the object.
    destroy_in_module(p);
  }
```

```
  static void destroy_in_module(Base* p) noexcept; // provided by the
  ↪ module
};

using BaseHandle = std::unique_ptr<Base, Deleter>;
```

This expresses ownership while ensuring destruction uses the correct mechanism.

Pattern 2: explicit `destroy()` virtual function

```
struct Base {
  virtual void destroy() noexcept = 0; // calls correct deallocation
  ↪ inside module
  virtual void run() = 0;

protected:
  ~Base() = default; // prevent delete via Base*
};

struct Derived : Base {
  void destroy() noexcept override { delete this; } // module-controlled
  ↪ rule
  void run() override {}
};
```

This can be useful in plugin systems, but it is easier to misuse than a `unique_ptr` with a custom deleter because callers must remember to call `destroy()`.

16.4.7 Preventing accidental slicing and unsafe copying

Polymorphic bases often should not be copyable by value.

Disable copying in polymorphic bases

```
struct Base {
  Base() = default;
  Base(const Base&) = delete;
  Base& operator=(const Base&) = delete;

  virtual ~Base() = default;
  virtual void run() = 0;
};
```

If polymorphic copying is required, use an explicit `clone()` protocol:

```
#include <memory>

struct Base {
  virtual ~Base() = default;
  virtual std::unique_ptr<Base> clone() const = 0;
};

struct Derived : Base {
  int x = 0;
  std::unique_ptr<Base> clone() const override {
    return std::make_unique<Derived>(*this);
  }
};
```

16.4.8 Non-owning access patterns: safer APIs

For many designs, you want to pass polymorphic objects around without transferring ownership. Prefer references or observer pointers.

Reference-based interfaces

```
void process(const Base& b); // cannot be null; no ownership transfer
```

Pointer-based observer interfaces

```
void process(const Base* b); // may be null; no ownership transfer
```

Document the ownership rule: “caller owns; callee borrows for duration of call.”

16.4.9 Checklist: safe polymorphic ownership in production C++

1. If deletion through base is possible, the base destructor is **public and virtual**.
2. Prefer `std::unique_ptr<Base>` for ownership; use `std::shared_ptr<Base>` only when shared lifetime is truly required.
3. Never return raw owning pointers from factories; return owning smart pointers.
4. Separate **owning** and **borrowing** in the type system (smart pointer vs reference/pointer).
5. Consider deleting copy operations in polymorphic bases to prevent slicing and accidental value semantics.
6. For plugin / ABI boundaries, ensure **destroy in the creating module** (custom deleter or destroy protocol).
7. Prefer **RAII** and clear lifetime scopes; avoid global ownership and unclear shutdown order.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): delete-expression rules, virtual dispatch for destructors, object lifetime.
- cppreference.com: *destructor*, *virtual*, *delete expression*, *std::unique_ptr*, *std::shared_ptr*, *std::weak_ptr*.
- ISO C++ Core Guidelines: rules on ownership (RAII), expressing ownership in types, and polymorphic base class destructor guidance.
- Microsoft Learn (MSVC documentation): guidance on virtual destructors and smart pointers in modern C++.
- Itanium C++ ABI: practical model of virtual destructors and deletion (ABI perspective; implementation detail).

Chapter 17

Templates: Power with Responsibility

17.1 Compile-time Complexity

17.1.1 Why compile-time complexity matters in a safety handbook

Templates are a cornerstone of modern C++: they enable zero-overhead generic code, strong typing, and extensive compile-time validation. The trade-off is that **work moves from runtime to compile time**. If you are not careful, template-heavy code can cause:

- very slow incremental builds (poor developer feedback loop),
- large memory usage in the compiler frontend,
- long CI times and reduced test coverage (people run fewer tests),
- fragile builds (minor edits trigger huge recompilations),
- hard-to-debug error explosions (deep instantiation stacks).

In safety-critical systems, build-time health is part of engineering health: you want a codebase that can be rebuilt, tested, and audited frequently.

17.1.2 What "compile-time complexity" means for templates

Compile-time cost comes from **how much work the compiler must do** to translate your source code into object code. For templates, that work includes:

1. **Parsing and semantic analysis** of template definitions (often in headers).
2. **Name lookup and overload resolution** (including ADL).
3. **Template argument deduction** and constraint checking.
4. **Instantiation** (substitution + generation of specialization code).
5. **Constant evaluation** (constexpr evaluation of algorithms and data).
6. **Optimization of the generated code** (which can be large due to specialization).

The key point: **templates scale with the number of distinct instantiations**. If the same template is used with many different argument combinations, the compiler may have to generate (and optimize) many versions.

17.1.3 The instantiation model: why templates multiply work

A template definition by itself does not necessarily generate code. Code is produced when a template is **instantiated** with concrete arguments. Therefore, each distinct set of template arguments can produce another body of code and another compilation workload.

The basic growth law

If a template is instantiated across many translation units (TUs), then:

- each TU may need to parse and analyze the template definition (because it is visible in headers),
- each TU may implicitly instantiate the same specialization,
- toolchains typically merge duplicate instantiations at link time or via COMDAT-like mechanisms, but the **compile-time work in each TU still happened**.

This is the origin of the classic issue: **header-only template libraries can increase compile time**, even if link-time deduplication keeps only one copy in the final binary.

17.1.4 Primary drivers of template compile-time blowups

Driver 1: combinatorial instantiations

The most common cause is the **Cartesian product** effect: a generic algorithm is instantiated for many types, and those types themselves contain templated members, producing secondary instantiations.

```
#include <type_traits>

// A toy example: compile-time work grows with the number of unique T.
template<class T>
struct Wrapper {
  static constexpr bool heavy =
    std::is_trivially_copyable_v<T> &&
    std::is_trivially_destructible_v<T>;
};
```

```
template<class T>
void process(T) {
  (void)Wrapper<T>::heavy; // forces Wrapper<T> instantiation
}
```

If `process` is used with 200 distinct types, the compiler sees 200 instantiations (and the work inside each instantiation).

Driver 2: unconstrained templates and overload-set explosion

Unconstrained templates often participate in overload resolution for many calls that they will never actually accept. This can cause the compiler to repeatedly attempt substitution, fail, and continue searching.
Use constraints (Concepts in C++20, or SFINAE in older code) to reduce candidate sets. Constraints can increase compile-time in some cases (more checking), but they often *reduce* overall work by shrinking overload sets and making failures early and clear.

```
#include <concepts>
#include <string>

// Good: the constraint removes this template from consideration early.
template<class T>
requires std::integral<T>
std::string to_text(T x) {
  return std::to_string(x);
}
```

Driver 3: recursive template metaprogramming

Template metaprogramming can be powerful, but deep recursion and large type-level computations can overwhelm compilers (depth limits, huge diagnostics, long times).

Prefer modern, simpler alternatives when possible:

- **constexpr functions** instead of type-level recursion for values,
- standard type-traits and library facilities rather than custom TMP,
- Concepts rather than TMP tricks to emulate constraints.

Driver 4: heavy `constexpr` evaluation

Modern C++ increasingly supports compile-time evaluation. This is useful, but it can turn compilation into an interpreter run of complex algorithms.
Two common compile-time traps:

- large compile-time tables generated in headers for many TUs,
- expensive compile-time parsing/formatting/validation logic.

Use compile-time computation where it gives safety or performance that matters, but measure and keep it bounded.

Driver 5: error cascades and deep instantiation stacks

Even when build time is acceptable on success, template-heavy code can create **pathologically slow failing builds**, where diagnostics become enormous. This is a productivity and safety concern: developers avoid refactoring and avoid improving checks if failures are too painful.
Practical mitigations:

- prefer smaller, focused templates,
- constrain templates so errors point to violated requirements,

- isolate heavy templates in fewer headers,
- use explicit instantiation to centralize failures.

17.1.5 How to measure: make compile-time visible

You cannot fix what you cannot see. Use compiler and IDE tooling to identify hot spots:

GCC/Clang: front-end timing summaries

Many toolchains provide a per-phase or per-component timing report. For example, `-ftime-report` is a supported option in Clang-style driver documentation and produces a timing report at the end of compilation.

```
# Example usage (exact output format is toolchain-dependent)
clang++ -O2 -c file.cpp -ftime-report
g++     -O2 -c file.cpp -ftime-report
```

MSVC / Visual Studio: per-file and template instantiation analysis

For MSVC-based workflows, you can:

- print per-file compilation time (`/Bt+`),
- report detailed frontend time breakdown (`/d1reportTime`),
- use Visual Studio **Build Insights** with the **Templates view** to see which template instantiations contribute most to build time.

```
# Example (MSVC command line options)
cl /c /O2 /Bt+ /d1reportTime file.cpp
```

Template instantiation profilers

For deep investigation, specialized tooling exists to profile template instantiation time and memory. Use it when the standard timing summaries are not granular enough.

17.1.6 Engineering patterns to control compile-time complexity

Pattern 1: Minimize instantiation diversity

If you can reduce the number of distinct template argument combinations, you reduce compile-time and code-size.
Tactics:

- avoid templating on values/types that do not affect behavior,
- move rarely-needed policy parameters out of the type system (runtime config),
- consolidate types using common representations (e.g., `std::span<std::byte>` rather than many container variations) when appropriate.

Pattern 2: Explicit instantiation to centralize work

If a template is used with a known set of types, **explicit instantiation** can reduce redundant instantiations across many translation units.

Step 1: declare extern template in a header (consumer side).

```
// widget.hpp
#pragma once
#include <string>

template<class T>
```

```
class Widget {
public:
  std::string name() const;
};

extern template class Widget<int>;
extern template class Widget<double>;
```

Step 2: define and explicitly instantiate in one .cpp (producer side).

```
// widget.cpp
#include "widget.hpp"

template<class T>
std::string Widget<T>::name() const { return "Widget"; }

template class Widget<int>;
template class Widget<double>;
```

This technique is standard C++ and is widely used in large codebases to improve build times.

Pattern 3: Avoid "templates everywhere" in interfaces

Templates in public headers amplify compilation cost because every consumer TU must process them. Consider isolating templates:

- Use non-templated abstract interfaces when runtime polymorphism is acceptable.
- Use type erasure for "value-like polymorphism" when it reduces exposure.
- Use the PIMPL idiom to keep implementation details (including templates) out of headers.

Pattern 4: Prefer constexpr functions over deep TMP when computing values

When the goal is computing a value (not a type), a `constexpr` function is often clearer, easier for compilers to optimize, and friendlier in diagnostics.

```
#include <array>
#include <cstddef>

constexpr std::size_t fib(std::size_t n) {
  std::size_t a = 0, b = 1;
  for (std::size_t i = 0; i < n; ++i) {
    std::size_t next = a + b;
    a = b;
    b = next;
  }
  return a;
}

static_assert(fib(10) == 55);
```

Pattern 5: Constrain templates to reduce substitution churn

Constrain templates to:

- shrink overload candidate sets,
- provide early, meaningful errors,
- avoid instantiating code paths that are never valid for the type.

```
#include <concepts>
#include <iterator>
```

```
template<class It>
requires std::input_iterator<It>
auto sum(It first, It last) {
  using V = std::iter_value_t<It>;
  V acc{};
  for (; first != last; ++first) acc += *first;
  return acc;
}
```

Pattern 6: Be cautious with header-only "constexpr data factories"

A frequent mistake is generating large compile-time data structures in headers, causing every TU to repeat expensive evaluation.
If the data is large or costly to compute:

- move it into a single TU (or a module interface/implementation),
- or precompute it and embed it as a static data blob,
- or generate it at build time with a code generator (if appropriate).

17.1.7 Safety-focused guidance: balancing compile time and robustness

Compile-time mechanisms can *improve safety* (strong types, constraints, static assertions), but they have operational costs. A safe engineering balance is:

- Use compile-time checks when they prevent real classes of defects.
- Keep compile-time algorithms bounded and measurable.
- Prefer simpler mechanisms (constexpr functions, standard traits, Concepts) over clever TMP recursion.
- Make build performance a tracked metric (local dev + CI).

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): templates, instantiation rules, and program translation model.
- cppreference.com: templates overview and rules about visibility at the point of instantiation.
- ISO C++ Core Guidelines, section *T: Templates and generic programming* (including recommendations to use TMP only when needed and to prefer constexpr facilities where appropriate).
- Microsoft Learn (MSVC documentation): function template instantiation model; Build Insights Templates view for analyzing template instantiation cost.
- LLVM/Clang documentation: compiler driver options including timing reports such as `-ftime-report`.
- GNU Compiler Collection documentation: discussion of template instantiation models and their impact on compilation time and code size.

17.2 Error Readability

17.2.1 Why template errors become unreadable

Template diagnostics often become difficult because a single user-level mistake can trigger a long chain of compile-time events:

- overload resolution considers many candidates (including templates),
- template argument deduction tries substitutions that fail,

- constraints (or SFINAE) accept/reject candidates,
- the compiler reports a deep instantiation backtrace that includes internal library types.

Unreadable errors are not just a productivity issue. In safety-oriented code, they reduce the chance that developers will correctly understand constraints, fix root causes, and keep strict checks enabled.
Your goal is to make errors:

- **localized** (fail early, near the call site),
- **intentional** (clear "what is required" and "what failed"),
- **stable** (small changes do not produce radically different walls of text).

17.2.2 The most effective tool: Concepts and constraints (C++20+)

Constraints are the modern way to express template requirements. They improve error readability by turning "substitution explosions" into a clear statement of intent: *this template is only viable if these conditions hold.*

Prefer named concepts over ad-hoc SFINAE

Bad pattern: SFINAE in the signature (harder diagnostics).

```
#include <type_traits>

template<class T,
         class = std::enable_if_t<std::is_integral_v<T>>>
int bit_width(T x) { /* ... */ return 0; }
```

Better pattern: a named concept (clear diagnostics).

```
#include <concepts>

template<class T>
concept Integer = std::integral<T>;

template<Integer T>
int bit_width(T x) { /* ... */ return 0; }
```

When a call fails, the compiler can point directly to the violated concept.

Put constraints on the *most visible* boundary

Constrain the function template itself rather than relying on failures deep in the body.

```
#include <concepts>
#include <string>

template<class T>
concept StringLike =
  requires(T t) {
    { std::string{t} }; // constructible into std::string
  };

template<StringLike T>
std::string normalize(T&& t) {
  return std::string{std::forward<T>(t)};
}
```

This fails at the call site with a statement about `StringLike`, instead of failing later with internal library substitution errors.

17.2.3 Fail early with `static_assert` and clear messages

Use `static_assert` for invariants and unsupported cases

If your template supports only a subset of cases, it is better to reject explicitly with a message than to allow cryptic failures.
A common technique is a dependent false:

```
template<class>
inline constexpr bool dependent_false_v = false;

template<class T>
struct Serializer {
  static_assert(dependent_false_v<T>,
    "Serializer<T>: unsupported type T. Provide a specialization or a
    ↪  custom serializer.");
};
```

Now, if a user attempts `Serializer<MyType>`, the error is a single, intentional statement, not a cascade.

Prefer constraints *and* `static_assert` for layered clarity

Constraints prevent the overload from being considered; `static_assert` explains why a selected path is invalid. Use both when they serve different purposes.

```
#include <concepts>
#include <type_traits>

template<class T>
concept TriviallyCopyable = std::is_trivially_copyable_v<T>;
```

```
template<TriviallyCopyable T>
void send_raw(const T& x) {
  static_assert(sizeof(T) <= 64,
    "send_raw<T>: T is too large for the raw fast path; use send_buffer()
    ↪  instead.");
  (void)x;
}
```

17.2.4 Reduce overload ambiguity: keep candidate sets small

Large overload sets are a major cause of template error noise. When many templates could match, the compiler must explain why each was rejected.

Use constraints to split overloads cleanly

```
#include <concepts>
#include <string>

template<std::integral T>
std::string format(T x) { return std::to_string(x); }

template<std::floating_point T>
std::string format(T x) { return std::to_string(x); }

std::string format(const char* s) { return std::string{s}; }
```

This approach avoids a single “universal reference” template that participates in far too many calls.

Avoid “catch-all” templates unless you truly mean them

This is a frequent source of unreadable diagnostics:

```
// Avoid as a default API shape: it accepts too many calls
template<class T>
void f(T&&);
```

If you must have a catch-all, constrain it or make it a last resort with clear diagnostics.

17.2.5 Make requirements readable: use `requires` expressions

A `requires` expression can state requirements in a way that reads like a checklist.

```
#include <concepts>
#include <iterator>

template<class R>
concept RangeLike =
  requires(R r) {
    std::begin(r);
    std::end(r);
  };

template<class R>
concept SummableRange =
  RangeLike<R> &&
  requires(R r) {
    typename std::remove_reference_t<decltype(*std::begin(r))>;
    { *std::begin(r) + *std::begin(r) };
  };

template<SummableRange R>
auto sum(R&& r) {
  using V = std::remove_cvref_t<decltype(*std::begin(r))>;
  V acc{};
```

```
  for (auto&& x : r) acc = acc + x;
  return acc;
}
```

When `sum(x)` fails, the compiler can show which required expressions were not satisfied (missing `begin/end`, missing `operator+`, etc.).

17.2.6 Prefer simple type-level APIs: alias templates and traits

Complex nested type expressions make diagnostics hard to read. Provide named aliases.

```
#include <type_traits>

template<class T>
using remove_cvref_t = std::remove_cv_t<std::remove_reference_t<T>>;

template<class T>
using value_type_t = typename remove_cvref_t<T>::value_type; // example
```

Even if an error still occurs, the diagnostic contains your meaningful alias name.

17.2.7 Control instantiation depth: isolate heavy templates

If a template triggers deep instantiation (e.g., metaprogramming utilities, formatters, reflection-like layers), isolate it:

- keep heavy templates out of widely included headers when possible,
- split helpers into smaller templates with well-named concepts,
- use explicit instantiation for common specializations to centralize failures.

This improves both compile-time and error localization.

17.2.8 Improve errors by improving APIs: “value semantics” alternatives

Sometimes error readability is telling you something important: the API surface is too generic.
Consider replacing a very generic template API with:

- a small set of constrained overloads for common cases,
- `std::variant` for a closed set of alternatives,
- type erasure (value-like runtime polymorphism) to reduce template exposure.

Reducing template exposure often produces dramatically simpler diagnostics.

17.2.9 Tooling and build settings that help (portable guidance)

Compilers differ, but these practices are widely useful:

- Ensure your build keeps **full diagnostic context** for template errors (some builds reduce backtraces by default).
- Prefer modern language modes (C++20/23/26 where available) so that Concepts are used rather than SFINAE workarounds.
- Keep standard library and third-party headers up to date; diagnostics and concept-based constraints often improve with library evolution.

17.2.10 Professional checklist: writing templates that fail well

1. Use **Concepts** and **requires** to express requirements at the interface.
2. Prefer **named concepts** over clever type-trait expressions in signatures.

3. Use **static_assert with a message** for unsupported cases and invariants.
4. Avoid unconstrained “catch-all” templates in public APIs.
5. Keep overload sets small and separate them with constraints.
6. Introduce readable aliases for complex type expressions.
7. Isolate heavy templates and consider explicit instantiation for common types.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): templates, constraints (Concepts), requires-expressions, and overload resolution rules.
- cppreference.com: pages on *templates*, *SFINAE*, *constraints and concepts*, *requires expressions*, and *static_assert*.
- ISO C++ Core Guidelines: *T: Templates and generic programming* (including guidance to express requirements clearly and avoid surprising template interfaces).
- *Effective Modern C++* (Scott Meyers): guidance on type deduction, interfaces, and reducing template surprises (conceptually aligned with the standard model).
- Compiler vendor documentation (GCC, Clang/LLVM, MSVC): diagnostics configuration and template/concepts diagnostic behavior.

17.3 Safer Template Design

17.3.1 Why templates need “safety design”

Templates amplify both strengths and mistakes:

- A small interface mistake propagates into many instantiations.
- Undefined behavior and lifetime bugs can become generic and widespread.
- Overly-permissive templates accept unintended types and silently change meaning.
- Overly-clever metaprogramming makes debugging and review harder.

Safer template design means writing generic code that is:

- hard to misuse,
- explicit about requirements,
- robust against lifetime and ownership mistakes,
- predictable in overload resolution and type deduction,
- friendly to diagnostics and auditing.

17.3.2 Design principle 1: constrain templates at the boundary (C++20+)

The safest generic code starts by stating what is allowed. In modern C++, **Concepts** and **requires-clauses** are the preferred way to express requirements.

Avoid unconstrained "accept anything" templates

```
// Risky API: participates in too many calls.
template<class T>
void log(T&& x);
```

Prefer explicit constraints:

```
#include <concepts>
#include <string>
#include <string_view>

template<class T>
concept StringLike =
  requires(T t) {
    { std::string_view{t} } -> std::same_as<std::string_view>;
  };

void log(std::string_view s);  // simplest and safest

template<StringLike T>
void log(T&& t) {
  log(std::string_view{t});
}
```

This pattern shrinks overload sets and makes intent clear at the call site.

Prefer named concepts to ad-hoc trait expressions

```
#include <concepts>
#include <type_traits>

template<class T>
concept TriviallyCopyable = std::is_trivially_copyable_v<T>;

template<TriviallyCopyable T>
void send_raw(const T& x) {
  (void)x;
}
```

A named concept improves error readability and prevents accidental misuse.

17.3.3 Design principle 2: fail early, fail clearly

Templates should reject invalid usage with a single, understandable diagnostic.

Use `static_assert` with a message for invariants

```
#include <type_traits>

template<class T>
void serialize(const T&) {
  static_assert(std::is_trivially_copyable_v<T>,
    "serialize<T>: T must be trivially copyable for the raw serializer. "
    "Provide a custom serializer for non-trivial types.");
}
```

Dependent false for unsupported primary templates

```
template<class>
inline constexpr bool dependent_false_v = false;

template<class T>
struct Hash {
  static_assert(dependent_false_v<T>,
    "Hash<T>: no hash defined for this type; provide a specialization.");
};
```

This creates intentional, stable diagnostics instead of deep template backtraces.

17.3.4 Design principle 3: separate owning from non-owning interfaces

Generic code is especially prone to lifetime bugs when it blurs ownership.

Prefer views for non-owning inputs

```
#include <span>
#include <cstddef>

void process_bytes(std::span<const std::byte> data); // non-owning view
```

Do not accidentally accept temporaries and store views

A classic bug is storing a view (`std::span`, `std::string_view`) that refers to a temporary.

```
#include <string_view>
#include <string>

struct BadStore {
  std::string_view sv;

  template<class T>
  explicit BadStore(T&& t) : sv(t) {} // may bind to temporary ->
  ↪  dangling
};

int main() {
  BadStore x{std::string("temp")}; // dangling view after full-expression
}
```

Safer approach: if you store, you usually must **own**.

```
#include <string>
#include <string_view>

struct SafeStore {
  std::string s;
```

```
  explicit SafeStore(std::string_view sv) : s(sv) {}
};
```

17.3.5 Design principle 4: avoid forwarding-reference traps

"Perfect forwarding" is powerful, but many unsafe APIs begin with an unnecessary `T&&`.

When `T&&` is correct

Use forwarding references when you genuinely forward to another call and do not store references.

```
#include <utility>
#include <vector>

template<class T, class... Args>
void emplace_one(std::vector<T>& v, Args&&... args) {
  v.emplace_back(std::forward<Args>(args)...);
}
```

When `T&&` is unsafe or misleading

Avoid forwarding references for simple inputs where a value or view is better:

```
// Risky: accepts everything; may copy unexpectedly; confuses overloads.
template<class T>
void set_name(T&& t);
```

Prefer explicit stable types:

```
#include <string_view>

void set_name(std::string_view sv); // caller controls lifetime
↪  responsibility
```

17.3.6 Design principle 5: avoid returning dangling references

Templates often return `auto` or `decltype(auto)` and accidentally return references to dead objects.

The common pitfall

```
#include <utility>

template<class T>
decltype(auto) bad_get(T&& x) {
  return x; // if x is a forwarding reference bound to a temporary ->
  ↪  returns dangling reference
}
```

Safer rule of thumb:

- Return by value unless you **intentionally** return a reference with a clear lifetime contract.
- Avoid `decltype(auto)` in public APIs unless you truly need reference preservation.

17.3.7 Design principle 6: avoid implicit narrowing and surprising conversions

Generic code often accepts types that "convert" but lose information.

Prefer explicit conversions in templates

```
#include <type_traits>

template<class To, class From>
```

```
To narrow_cast(From x) {
  // Intentional: not a full narrowing checker, but makes conversion
  ↪  explicit.
  return static_cast<To>(x);
}
```

When narrowing is forbidden, use checked conversions and constraints (or dedicated utilities).

17.3.8 Design principle 7: choose the right generic abstraction

Not every problem needs a template. Safer designs often reduce template exposure.

Prefer standard vocabulary types

- `std::span` for contiguous non-owning ranges
- `std::string_view` for read-only string views (with strict lifetime discipline)
- iterators/ranges concepts for generic algorithms
- `std::variant` for closed-set polymorphism with value semantics
- type erasure (e.g., `std::function`) to hide template complexity behind stable interfaces

Closed-set polymorphism via `std::variant` reduces template misuse

```
#include <variant>
#include <string>

struct Circle { double r; };
struct Square { double a; };
```

```
using Shape = std::variant<Circle, Square>;

std::string describe(const Shape& s) {
  return std::visit([](const auto& x) -> std::string {
    using T = std::decay_t<decltype(x)>;
    if constexpr (std::is_same_v<T, Circle>) return "Circle";
    else return "Square";
  }, s);
}
```

This avoids inheritance pitfalls and keeps errors local.

17.3.9 Design principle 8: control specialization and customization points

Unsafe generic libraries often allow users to “customize” behavior in ways that break invariants.

Prefer explicit customization points

Modern libraries often use a dedicated customization mechanism rather than “specialize anything anywhere”.

- Prefer free functions discovered by ADL only when you carefully control the namespace and naming.
- Prefer explicit tag-invoke / customization point objects (CPOs) in advanced libraries.
- Avoid requiring users to partially specialize templates in the standard namespace (not allowed) or in ways that violate the One Definition Rule.

If you use ADL, constrain and document it

```
#include <concepts>

namespace lib {

struct to_text_tag {};

template<class T>
concept HasToText =
  requires(T t) {
    to_text(to_text_tag{}, t); // found by ADL in T's namespace
  };

template<class T>
requires HasToText<T>
auto to_text(T&& t) {
  return to_text(to_text_tag{}, std::forward<T>(t));
}

} // namespace lib
```

This keeps customization explicit and reduces accidental matches.

17.3.10 Design principle 9: reduce compile-time cost without sacrificing safety

Safety checks are good, but make them efficient and localized.

- Constrain templates so invalid candidates are rejected early.
- Avoid deep recursive metaprogramming when a constexpr function works.

- Prefer smaller templates with readable names over monolithic mega-templates.
- Use explicit instantiation for common specializations when appropriate.

17.3.11 Design principle 10: auditability and maintainability are safety properties

Template code is safer when it is reviewable:

- prefer simple, direct code paths,
- avoid clever SFINAE tricks when Concepts can state the rule,
- keep error messages intentional (named concepts, static_assert messages),
- keep lifetimes explicit (owning vs viewing types),
- keep interfaces stable (avoid surprising deduction and overload ambiguity).

17.3.12 Practical "safe template" patterns (ready to reuse)

Pattern A: constrained overload + clear fallback error

```
#include <concepts>
#include <type_traits>

template<class T>
concept ByteLike = std::is_trivially_copyable_v<T> && (sizeof(T) == 1);

template<ByteLike T>
void write_bytes(const T* data, std::size_t n) {
  (void)data; (void)n;
```

```
}

template<class T>
void write_bytes(const T*, std::size_t) = delete; // hard error for
↪  non-byte types
```

Pattern B: base template rejected, specializations allowed

```
template<class>
inline constexpr bool dependent_false_v = false;

template<class T>
struct Formatter {
  static_assert(dependent_false_v<T>,
    "Formatter<T>: provide a specialization Formatter<YourType>.");
};

// specialization example:
struct MyType {};

template<>
struct Formatter<MyType> {
  static const char* format(const MyType&) { return "MyType"; }
};
```

Pattern C: non-owning input view + owning storage

```
#include <string>
#include <string_view>

struct Name {
  std::string value;
  explicit Name(std::string_view sv) : value(sv) {}
```

```
};
```

This prevents storing dangling views.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): templates, constraints (Concepts), requires-expressions, overload resolution, object lifetimes, and reference binding rules.
- cppreference.com: *templates*, *constraints and concepts*, *requires*, *static_assert*, *std::span*, *std::string_view*, *std::variant*, and type traits.
- ISO C++ Core Guidelines: guidelines for templates and generic programming; guidelines on expressing ownership and lifetime via types.
- *C++ Templates: The Complete Guide* (Vandevoorde, Josuttis, Gregor): template interface design, instantiation behavior, and best practices.
- *Effective Modern C++* (Scott Meyers): guidance on type deduction, forwarding references, and designing clear, safe interfaces.

17.4 Concepts (C++20+)

17.4.1 What Concepts are

Concepts (introduced in C++20) let you place **named, compile-time requirements** on template arguments. A concept is a boolean predicate evaluated during compilation, and it becomes part of the **public interface** of a template.
Concepts improve safety and maintainability because they:

- prevent templates from accepting unintended types,
- make overload resolution more predictable,
- produce clearer and earlier diagnostics than many SFINAE-based designs,
- document constraints directly in code.

17.4.2 The building blocks

1) Defining a concept

A concept is defined with the `concept` keyword and a constraint expression.

```
#include <type_traits>

template<class T>
concept TrivialByte =
  std::is_trivially_copyable_v<T> && (sizeof(T) == 1);
```

Concepts are typically defined in headers, near the templates that use them.

2) Using a concept to constrain templates

There are three common forms.

A) Constrained template parameter

```
#include <concepts>

template<std::integral T>
T gcd(T a, T b) {
  while (b != 0) {
```

```
    T r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

B) Requires-clause

```
#include <concepts>

template<class T>
requires std::integral<T>
T gcd2(T a, T b) { return gcd(a, b); }
```

C) Abbreviated function template (“constrained `auto`”)

```
#include <concepts>

auto twice(std::integral auto x) {
  return x + x;
}
```

All three forms express the same idea: the template is only viable if the constraint is satisfied.

17.4.3 Requires-expressions: describing what must be valid

A **requires-expression** checks whether certain expressions, types, and relations are well-formed for a given type. It is the most precise way to define a concept.

Basic requires-expression

```
#include <concepts>
```

```
template<class T>
concept Addable =
  requires(T a, T b) {
    a + b;              // expression must be valid
  };
```

Typed requirements: require a specific result type

```
#include <concepts>

template<class T>
concept AddToSame =
  requires(T a, T b) {
    { a + b } -> std::same_as<T>;
  };
```

Compound requirements: require noexcept

```
#include <concepts>

template<class T>
concept NoexceptSwappable =
  requires(T a, T b) {
    { a.swap(b) } noexcept;
  };
```

Nested requirements: require a boolean condition

```
#include <concepts>
#include <type_traits>

template<class T>
```

```
concept SmallTrivial =
  requires {
    requires std::is_trivially_copyable_v<T>;
    requires (sizeof(T) <= 16);
  };
```

This is why you sometimes see `requires requires`: the first `requires` introduces a requires-expression, and the second `requires` introduces a nested requirement inside it.

17.4.4 How constraints affect overload resolution (why this improves safety)

Constraints are not only checks; they also guide which overload is selected.

Constrained overload sets become clearer

```
#include <concepts>
#include <string>

template<std::integral T>
std::string classify(T) { return "integral"; }

template<std::floating_point T>
std::string classify(T) { return "floating"; }

std::string classify(const char*) { return "c-string"; }
```

A call like `classify(3)` selects the integral overload. A call like `classify(3.14)` selects the floating overload.

Subsumption: “more constrained” wins

When multiple constrained templates could match, C++ defines a partial ordering based on constraints. Intuitively, the template with the **stronger / more specific** constraint is preferred (when it *subsumes* another). This reduces accidental ambiguity compared to many SFINAE-heavy designs.

17.4.5 Standard library concepts you should use

C++20 also introduced a rich set of **standard concepts**, including:

- Core concepts such as `std::same_as`, `std::derived_from`, `std::convertible_to`, `std::integral`, `std::floating_point`.
- Concepts for iterators and ranges used throughout the Ranges library (e.g., iterator/range-related concepts in `<iterator>` and `<ranges>`).

Using standard concepts is usually safer than inventing local ad-hoc traits, because they encode well-known requirements and integrate cleanly with the standard library design.

17.4.6 Concepts vs SFINAE: what changes and what does not

What changes

- Constraints let the compiler reject invalid candidates **before** instantiating deep template bodies, which often improves diagnostics.
- Requirements are expressed directly in the signature, making interfaces self-documenting.

What does not change

- Constraints are still compile-time logic; they do not eliminate the need for correct semantics, invariants, and testing.
- Concepts do not prevent all misuse: a type can satisfy syntactic requirements while still violating intended meaning. Good concepts aim to model semantic categories, not just "has a plus operator."

17.4.7 Safety-oriented best practices

1) Constrain at the boundary, not deep in the body

Do not rely on failures inside the function body to reject types. Write constraints so misuse fails at the call site.

```
#include <concepts>
#include <string_view>

template<class T>
concept StringViewable =
  requires(T t) { std::string_view{t}; };

std::size_t count_dots(std::string_view sv);

template<StringViewable T>
std::size_t count_dots(T&& t) {
  return count_dots(std::string_view{t});
}
```

2) Prefer named concepts

Named concepts create stable diagnostics and reusable requirements.

```
#include <concepts>

template<class T>
concept Counter =
  std::integral<T> && requires(T x) { ++x; x++; };

template<Counter T>
T bump(T x) { return ++x; }
```

3) Use `static_assert` for business rules

Constraints express type/operation requirements. For policy limits and invariants, a clear `static_assert` message is often better.

```
#include <concepts>

template<std::integral T>
T safe_shift_left(T x, unsigned k) {
  static_assert(sizeof(T) <= 8,
    "safe_shift_left<T>: only supports integral types up to 64-bit.");
  return (k < (8u * sizeof(T))) ? (T)(x << k) : T{};
}
```

4) Avoid unconstrained forwarding references in public APIs

A template like `template<class T> void f(T&&);` participates in too many calls and often causes surprising overload resolution and unreadable errors. Constrain it or replace it with stable vocabulary types (e.g., `std::span`, `std::string_view`) when appropriate.

17.4.8 Common pitfalls

Pitfall 1: creating concepts that are purely syntactic

A concept that checks only syntax (“has operator+”), without modeling meaning, can accept nonsense types. Prefer concepts that reflect semantic categories and document expectations.

Pitfall 2: lifetime hazards hidden behind constrained `auto`

Concepts do not automatically make lifetime safe. For example, accepting `std::string_view` inputs is safe only if the view does not outlive the referenced data. Use owning types when storing data.

Pitfall 3: assuming identical compiler diagnostics and support

Concepts are standardized, but diagnostic quality and some syntax support can vary by toolchain and version. Ensure your CI compilers and IDE toolchains support the subset of Concepts syntax you rely on.

17.4.9 Practical checklist for Concepts adoption

1. Define small, named concepts for your domain vocabulary.
2. Prefer standard library concepts whenever they match your intent.
3. Constrain overload sets to make selection predictable.
4. Keep requires-expressions readable and minimal.
5. Combine constraints with clear `static_assert` messages for policy rules.
6. Treat Concepts as an interface tool: they improve correctness and readability, but do not replace semantic reasoning and tests.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): constraints, concepts, requires-clauses, requires-expressions, and constrained declarations.
- cppreference.com: *Constraints and concepts (since C++20)*; *requires-expression (since C++20)*.
- ISO C++ Core Guidelines: guidance on writing meaningful concepts (semantic categories vs syntactic constraints).
- Compiler/IDE documentation for Concepts support in major toolchains (GCC, Clang, MSVC) when selecting a minimum supported version.

Chapter 18

Type Safety and Strong Types

18.1 Avoiding Primitive Obsession

18.1.1 What “primitive obsession” means

Primitive obsession is the habit of representing important domain concepts using raw primitive types such as `int`, `double`, `bool`, and `std::string`.
Examples:

- representing a **UserId**, **OrderId**, and **SessionId** all as `int`,
- representing **bytes**, **milliseconds**, and **meters** all as `double`,
- representing a **file path** as `std::string` instead of `std::filesystem::path`.

Primitive obsession is a **type-safety** problem: the code compiles even when the meaning is wrong. In safety-oriented systems, this is a frequent source of silent logic errors, security issues, and reliability bugs.

18.1.2 Why primitives are dangerous in safety-critical code

1) Accidental mix-ups compile cleanly

```
using UserId    = int;
using ProductId = int;

void add_to_cart(UserId u, ProductId p);

int main() {
  UserId u = 42;
  ProductId p = 7;

  add_to_cart(p, u); // compiles: swapped arguments, wrong meaning
}
```

The compiler cannot help you because `UserId` and `ProductId` are the same type.

2) Units and scaling bugs become inevitable

```
double timeout = 500;    // is this ms? seconds?
double size    = 64;     // bytes? KiB? MB?
```

Unit confusion is a classic source of outages. Strong types make such mistakes harder.

3) Invariants are not enforced

Primitives cannot enforce domain rules (e.g., “port in [1..65535]” or “non-empty name”). Strong types can validate at construction and centralize invariants.

4) APIs become ambiguous

When every function takes `int` and `std::string`, the signatures stop documenting intent. Strong types turn the signature into documentation that the compiler can enforce.

18.1.3 The goal: make invalid states and invalid calls harder

Avoiding primitive obsession does not mean banning primitives. It means:

- use primitives **internally** when appropriate,
- expose **strong, meaningful types at boundaries** (APIs, interfaces, modules),
- enforce invariants once, near construction, and preserve them.

18.1.4 Technique 1: Prefer standard strong types

The standard library already provides strong types for common concepts.

Time and duration: `std::chrono`

Never represent timeouts, intervals, and timestamps as raw integers/doubles.

```
#include <chrono>

void set_timeout(std::chrono::milliseconds t);

int main() {
  set_timeout(std::chrono::milliseconds{500}); // explicit and
  ↪  unambiguous
}
```

You can accept generic durations safely:

```
#include <chrono>

template<class Rep, class Period>
void set_timeout(std::chrono::duration<Rep, Period> t) {
  auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(t);
  // store and use milliseconds internally
  (void)ms;
}
```

Paths: `std::filesystem::path`

```
#include <filesystem>

void open_file(const std::filesystem::path& p);
```

This avoids fragile ad-hoc string conventions and improves portability.

Non-owning views: `std::span`, `std::string_view` (with lifetime discipline)

Views are strong in meaning but still require correct lifetimes. Use them for input parameters, avoid storing them unless the referred data outlives the view.

```
#include <span>
#include <cstddef>

void hash_bytes(std::span<const std::byte> data);
```

18.1.5 Technique 2: Use `enum class` instead of magic integers/booleans

Avoid boolean parameters that lose meaning

```
// Avoid: what does 'true' mean?
void connect(bool use_tls, bool verify_cert);
```

Prefer expressive enums:

```
enum class TlsMode { off, on };
enum class CertVerification { skip, require };

void connect(TlsMode tls, CertVerification verify);
```

Now the call site is self-explanatory and type-safe.

18.1.6 Technique 3: Strong typedefs (distinct types with the same representation)

In C++, a plain `using UserId = int;` does **not** create a new type. A strong typedef does.

Minimal strong type wrapper

```
#include <cstdint>
#include <compare>

template<class T, class Tag>
class Strong {
public:
  using value_type = T;

  constexpr explicit Strong(T v) : value_(v) {}

  constexpr T value() const noexcept { return value_; }

  friend constexpr auto operator<=>(Strong, Strong) = default;

private:
  T value_;
```

```
};

struct UserIdTag {};
struct ProductIdTag {};

using UserId    = Strong<std::uint32_t, UserIdTag>;
using ProductId = Strong<std::uint32_t, ProductIdTag>;

void add_to_cart(UserId u, ProductId p);

int main() {
  UserId u{42};
  ProductId p{7};

  add_to_cart(u, p);   // OK
  // add_to_cart(p, u); // ERROR: different types
}
```

Keep construction explicit

The `explicit` constructor prevents accidental conversions:

```
UserId u = UserId{42}; // OK
// UserId u2 = 42;     // ERROR: no implicit conversion
```

This single design choice blocks a large class of accidental misuse.

18.1.7 Technique 4: Enforce invariants at construction

Strong types are most valuable when they enforce domain rules.

Non-empty string type

```
#include <string>
#include <stdexcept>
#include <string_view>

class NonEmptyString {
public:
  explicit NonEmptyString(std::string_view sv) : value_(sv) {
    if (value_.empty()) {
      throw std::invalid_argument("NonEmptyString: empty value");
    }
  }

  std::string_view view() const noexcept { return value_; }

private:
  std::string value_; // owns the data
};
```

Now any function receiving `NonEmptyString` can assume non-empty.

Range-limited integer type (example: TCP/UDP port)

```
#include <cstdint>
#include <stdexcept>

class Port {
public:
  explicit Port(std::uint16_t p) : value_(p) {
    if (p == 0) {
      throw std::invalid_argument("Port: 0 is invalid");
    }
```

```
  }

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

private:
  std::uint16_t value_;
};
```

18.1.8 Technique 5: Make units explicit (and hard to mix)

When you cannot use a full units library, you can still create lightweight unit types.

```
#include <cstdint>

struct Bytes {
  std::uint64_t n;
  explicit constexpr Bytes(std::uint64_t v) : n(v) {}
};

struct KiB {
  std::uint64_t n;
  explicit constexpr KiB(std::uint64_t v) : n(v) {}
  constexpr Bytes to_bytes() const noexcept { return Bytes{n * 1024u}; }
};

void allocate(Bytes b);

int main() {
  allocate(Bytes{512});
  allocate(KiB{64}.to_bytes());
}
```

This prevents mixing unrelated quantities by accident.

18.1.9 Technique 6: Strong error types instead of integers

Avoid returning primitive error codes without meaning.

Prefer `std::error_code` / `std::error_condition` or richer result types

```
#include <system_error>

std::error_code do_io(); // expressive and standardized
```

For richer APIs, use a result type (library-provided or project-provided) that carries:

- error category,
- message/context,
- optional payload.

The key is: errors are not “just integers”.

18.1.10 Where to apply strong types first (highest payoff)

In most codebases, the biggest safety wins come from strong types at boundaries:

1. public APIs and module interfaces,
2. security-sensitive decisions (auth, permissions, limits),
3. units and timeouts,
4. identifiers and handles (IDs, file descriptors, OS handles),
5. serialization/deserialization boundaries (parse into strong types early).

18.1.11 Interop strategy: strong types outside, primitives inside

A practical approach for performance and incremental adoption:

- Use strong types in function signatures, data models, and invariants.
- Store the underlying primitive inside the strong type.
- Convert at the boundary; keep inner loops primitive when needed.

This yields safety without forcing templates and wrappers everywhere.

18.1.12 Performance and memory impact

Strong types are usually **zero-overhead**:

- A wrapper around an integer often has the same size and alignment as the integer.
- With inlining and optimization, accessors typically compile away.

The real cost is usually **API churn** and **learning curve**, which is a good trade for fewer production defects in safety-critical systems.

18.1.13 Common pitfalls and how to avoid them

Pitfall 1: using `using` aliases and thinking they are strong types

`using UserId = int;` is not type-safe. Use a wrapper.

Pitfall 2: making the wrapper implicitly convertible

Avoid implicit conversion operators like `operator int()`. They reintroduce primitive obsession through a back door.
Prefer explicit accessors like `value()`.

Pitfall 3: storing views with uncertain lifetime

Do not store `std::string_view` or `std::span` unless you can prove the referred data outlives the view. Use owning types when storing.

Pitfall 4: over-designing hundreds of tiny types

Be strategic. Strong types are most valuable where confusion is costly: IDs, units, and invariants. Use them where they reduce real risk.

18.1.14 Professional checklist: eliminating primitive obsession

1. Replace magic integers/booleans with `enum class`.
2. Replace ambiguous numeric quantities with `std::chrono` durations and time points.
3. Introduce strong ID types for identifiers and handles.
4. Enforce domain invariants at construction; keep strong types valid-by-construction.
5. Keep conversions explicit; avoid implicit conversion operators.
6. Use view types for non-owning parameters, owning types for stored state.
7. Prefer standard vocabulary types (`path`, `span`, `string_view`) where appropriate.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): core language rules for conversions, user-defined types, overload resolution, and `enum class`.

- cppreference.com: `enum class`, `std::chrono`, `std::filesystem::path`, `std::span`, `std::string_view`, `std::error_code`.
- ISO C++ Core Guidelines: rules promoting strong types, avoiding raw `bool` parameters, using `enum class`, and expressing intent in types.
- *Effective Modern C++* (Scott Meyers): guidance on avoiding surprising implicit conversions and designing clear interfaces.
- *C++ API Design* best practices literature: expressing domain meaning through types and enforcing invariants at boundaries.

18.2 `enum class`

18.2.1 What `enum class` is

`enum class` (a **scoped enumeration**) was introduced in C++11 to fix several safety problems of traditional (unscoped) `enum`.
Compared to `enum`, `enum class` provides:

- **Scope**: enumerators are scoped (no global name pollution).
- **Strong typing**: no implicit conversion to integers.
- **Better overload resolution**: avoids accidental matches with arithmetic and boolean APIs.
- **Controllable underlying type**: explicitly choose the storage representation.

In safety-oriented code, `enum class` should be the default form of enumeration.

18.2.2 The safety problems with unscoped `enum`

Problem 1: implicit conversion to integer

Unscoped enums implicitly convert to integers, enabling subtle bugs.

```
enum Color { red = 0, green = 1, blue = 2 };

int main() {
  Color c = red;
  int x = c;       // implicit conversion: allowed
  if (c) {         // also allowed: Color converts to int then to bool
    // ...
  }
}
```

This makes it easy to accidentally pass an enum to an integer API or treat it as a boolean condition.

Problem 2: enumerator names leak into the surrounding scope

```
enum Status { ok, error };

enum Mode { ok, safe }; // ERROR or conflict: 'ok' already used in the
↪  same scope
```

18.2.3 How `enum class` solves these problems

Scoped enumerators

Enumerators must be qualified:

```
enum class Status { ok, error };
```

```
enum class Mode   { ok, safe };

int main() {
  Status s = Status::ok;
  Mode   m = Mode::ok;  // no conflict, different scopes
}
```

No implicit conversion to integer or bool

```
enum class Status { ok, error };

void f(int);

int main() {
  Status s = Status::ok;

  // f(s);          // ERROR: no implicit conversion to int
  // if (s) {}      // ERROR: no implicit conversion to bool

  f(static_cast<int>(s)); // explicit conversion is required (and
  ↪  visible)
}
```

Requiring `static_cast` is a feature: it forces the programmer to make conversions explicit and reviewable.

18.2.4 Choosing the underlying type

By default, the underlying type of a scoped enum is an implementation-chosen integral type large enough to represent its enumerators. You can choose it explicitly when you need a specific representation (serialization, ABI, memory layout).

```cpp
#include <cstdint>

enum class ErrorCode : std::uint16_t {
  ok = 0,
  invalid = 1,
  timeout = 2
};
```

When you should specify an underlying type

- When the enum is stored on disk or sent over the network.
- When the enum is part of a stable ABI boundary.
- When memory layout is critical and the set of values is stable.
- When interfacing with C APIs that expect a specific integer width.

When you should *not* specify it

If you do not have a concrete representation constraint, leaving it unspecified can be better:

- the compiler can choose an efficient type,
- you avoid premature ABI commitments,
- future additions may fit without revisiting representation.

18.2.5 Comparisons and ordering

Scoped enums support equality comparisons. Ordering comparisons depend on language rules and what operations are available; in general, treat enums as **labels**, not numbers.

```cpp
enum class State { idle, running, stopped };

int main() {
  State a = State::idle;
  State b = State::running;

  bool same = (a == b); // OK
  bool diff = (a != b); // OK

  // bool less = (a < b); // do not rely on ordering unless your design
  ↪  requires it
}
```

If you need ordering (for arrays, serialization tables, etc.), make it explicit via a mapping function or explicit underlying values.

18.2.6 Switch statements and exhaustive handling

Switch is natural for enums

```cpp
#include <string_view>

enum class Status { ok, error, busy };

std::string_view to_string(Status s) {
  switch (s) {
    case Status::ok:    return "ok";
    case Status::error: return "error";
    case Status::busy:  return "busy";
  }
  return "unknown"; // defensive fallback
}
```

Why the fallback matters

Even with a `switch`, compilers are not required to prove exhaustiveness in all cases (and values can be formed via casts or corrupted input). A defensive fallback improves robustness. For safety-critical code, consider:

- a fallback return or error path,
- logging or assertions in unreachable paths (depending on your policy),
- validating inputs when reading from untrusted sources.

18.2.7 Bitmask enums (flags) with `enum class`

A common need is representing bit flags. `enum class` does not implicitly support bitwise operations; this is intentional to prevent accidental arithmetic.

Define explicit operators for flag enums

```
#include <cstdint>
#include <type_traits>

enum class Perm : std::uint32_t {
  none  = 0,
  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 any(Perm p) noexcept {
  using U = std::underlying_type_t<Perm>;
  return static_cast<U>(p) != 0;
}

int main() {
  Perm p = Perm::read | Perm::write;
  if (any(p & Perm::read)) {
    // has read permission
  }
}
```

Why this pattern is safer

- It keeps accidental integer arithmetic out of the API.
- It limits operations to those you intentionally provide.
- It makes the enum a strongly typed flags set.

18.2.8 Interoperability with C and legacy APIs

When calling C APIs or OS APIs that expect integer codes, you will often need explicit conversion. Keep it local and visible.

```
#include <type_traits>
```

```
enum class Mode : int { a = 1, b = 2 };

int c_api(int mode);

int main() {
  Mode m = Mode::a;
  int r = c_api(static_cast<std::underlying_type_t<Mode>>(m));
  (void)r;
}
```

If the boundary is frequent, consider a dedicated conversion function to centralize it.

18.2.9 Common mistakes and professional recommendations

Mistake 1: using unscoped `enum` by habit

Default to `enum class`. Use unscoped `enum` only when:

- you need implicit conversion intentionally (rare in safe code),
- you interoperate with a legacy interface that depends on it (and you cannot change it).

Mistake 2: using `enum class` but reintroducing implicit conversions

Avoid adding implicit conversion operators or helper functions that turn the enum back into an int everywhere. Keep conversions explicit at the boundary.

Mistake 3: relying on ordinal values as stable IDs

If values must be stable (serialization, logging protocols), assign explicit values and treat them as part of your compatibility contract.

```cpp
#include <cstdint>

enum class WireCode : std::uint8_t {
  ok      = 0,
  denied  = 1,
  timeout = 2
};
```

Mistake 4: using `enum class` for open-ended sets

Enums represent a **closed set** of named values. If your set is open-ended (plugins, user-defined categories), consider stronger alternatives such as:

- strong typedef wrappers around integers/strings,
- registries with validation,
- `std::string` + validation when values come from the outside world.

18.2.10 Checklist: adopting `enum class` in safety code

1. Use `enum class` by default; avoid unscoped enums unless required.
2. Choose an explicit underlying type only when you have a real representation constraint.
3. Avoid implicit conversions; keep boundary conversions explicit.
4. For flags, provide explicit bitwise operators and helper functions.
5. Provide a `to_string` (or equivalent) and handle unknown values defensively.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): enumerations, scoped enumerations, underlying types, conversions, and `switch` semantics.
- cppreference.com: *Enumeration declaration*, *Scoped enumeration*, *std::underlying_type*, and *switch statement*.
- ISO C++ Core Guidelines: guidance recommending `enum class` for type safety and avoiding implicit conversions and name pollution.
- Compiler vendor documentation (GCC, Clang, MSVC): conformance notes and warnings related to `switch` over enums and scoped enum behavior.

18.3 Strong Typedefs

18.3.1 What a "strong typedef" is

A **strong typedef** is a technique to create a **distinct type** that has the **same runtime representation** as an existing type (`int`, `std::uint32_t`, `std::string`, etc.) but is **not implicitly interchangeable** with it.
It solves a common safety problem:

> Two values can have the same physical representation but different meaning. The compiler should prevent you from mixing them by accident.

18.3.2 Why plain `using` is not enough

A simple alias does **not** introduce a new type.

```
using UserId    = std::uint32_t;
using ProductId = std::uint32_t;

void add_to_cart(UserId u, ProductId p);

int main() {
  UserId u = 10;
  ProductId p = 99;

  add_to_cart(p, u); // compiles (bug): same underlying type
}
```

A strong typedef makes this mistake a compile-time error.

18.3.3 Design goals for strong typedefs in safety code

A good strong typedef should:

- be **explicit** to construct (no implicit conversions),
- preserve **zero-overhead** representation when possible,
- support only the **operations that make sense** for the domain,
- provide **clear access** to the underlying value when needed,
- be friendly to generic code (comparisons, hashing, formatting) in a controlled way.

18.3.4 Technique 1: Minimal strong typedef wrapper (“tagged type”)

This is the most common pattern: wrap the underlying type and add a unique tag type so that different strong typedefs never mix.

Minimal implementation (C++11+)

```
#include <utility>
#include <type_traits>

template<class T, class Tag>
class Strong {
public:
  using value_type = T;

  constexpr explicit Strong(const T& v) : value_(v) {}
  constexpr explicit Strong(T&& v) : value_(std::move(v)) {}

  constexpr const T& value() const & noexcept { return value_; }
  constexpr T&       value() &       noexcept { return value_; }
  constexpr T&&      value() &&      noexcept { return std::move(value_);
  ↪  }

private:
  T value_;
};
```

Using it for identifiers

```
#include <cstdint>

struct UserIdTag {};
struct ProductIdTag {};

using UserId    = Strong<std::uint32_t, UserIdTag>;
using ProductId = Strong<std::uint32_t, ProductIdTag>;

void add_to_cart(UserId u, ProductId p);
```

```
int main() {
  UserId u{42};
  ProductId p{7};

  add_to_cart(u, p);     // OK
  // add_to_cart(p, u); // ERROR: different types
}
```

This prevents accidental swaps while keeping the underlying representation small and fast.

18.3.5 Choosing which operations to support (do not overexpose)

A strong typedef is safer when it exposes only meaningful operations.
For example, an ID type typically supports:

- equality and ordering (for containers),
- hashing (for unordered containers),
- formatting / logging,
- explicit access to the underlying value at boundaries.

It usually should **not** support arithmetic like addition and multiplication (unless the domain makes that meaningful).

18.3.6 Technique 2: Add comparisons (C++11+) and optional three-way (C++20+)

C++11-style comparisons

```
template<class T, class Tag>
```

```
class StrongId {
public:
  constexpr explicit StrongId(T v) : value_(v) {}

  friend constexpr bool operator==(StrongId a, StrongId b) noexcept {
    return a.value_ == b.value_;
  }

  friend constexpr bool operator!=(StrongId a, StrongId b) noexcept {
    return !(a == b);
  }

  friend constexpr bool operator<(StrongId a, StrongId b) noexcept {
    return a.value_ < b.value_;
  }

  T value() const noexcept { return value_; }

private:
  T value_;
};
```

C++20 three-way comparison (optional)

```
#include <compare>

template<class T, class Tag>
class StrongCmp {
public:
  constexpr explicit StrongCmp(T v) : value_(v) {}
  friend constexpr auto operator<=>(const StrongCmp&, const StrongCmp&) =
  ↪  default;
```

```
  T value() const noexcept { return value_; }
private:
  T value_;
};
```

Use this only if your project uses C++20 and you want defaulted ordering.

18.3.7 Technique 3: Strong typedefs for units and quantities

Strong typedefs are also excellent for preventing unit mix-ups.

Distinct time/size units

```
#include <cstdint>

struct BytesTag {};
struct KiBTag   {};

using Bytes = Strong<std::uint64_t, BytesTag>;
using KiB   = Strong<std::uint64_t, KiBTag>;

constexpr Bytes to_bytes(KiB k) {
  return Bytes{ k.value() * 1024u };
}

void allocate(Bytes b);

int main() {
  allocate(Bytes{512});
  allocate(to_bytes(KiB{64}));
}
```

When possible, prefer `std::chrono` for time quantities because it is standardized and integrates with the library ecosystem.

18.3.8 Technique 4: Enforcing invariants (valid-by-construction strong types)

The strongest safety benefit comes when the type enforces domain rules.

Example: Non-zero ID

```
#include <cstdint>
#include <stdexcept>

class NonZeroUserId {
public:
  explicit NonZeroUserId(std::uint32_t v) : value_(v) {
    if (v == 0) throw std::invalid_argument("NonZeroUserId: 0 is
    ↪  invalid");
  }

  std::uint32_t value() const noexcept { return value_; }

  friend bool operator==(NonZeroUserId a, NonZeroUserId b) noexcept {
    return a.value_ == b.value_;
  }

private:
  std::uint32_t value_;
};
```

Now functions that accept `NonZeroUserId` can assume it is never zero.

18.3.9 Explicit conversion at boundaries (and only at boundaries)

A strong typedef should not silently convert back to the primitive, because that would reintroduce primitive obsession.

Do not add implicit conversion operators

Avoid patterns like:

```
// Avoid: unsafe, reintroduces implicit mixing.
struct BadId {
  std::uint32_t v;
  operator std::uint32_t() const { return v; }
};
```

Prefer explicit accessors such as `value()` and keep conversions localized.

18.3.10 Interoperability patterns

C APIs and serialization

When calling a C API or writing to disk/network, convert explicitly:

```
#include <cstdint>

struct UserIdTag {};
using UserId = Strong<std::uint32_t, UserIdTag>;

void write_u32(std::uint32_t);

void store(UserId id) {
  write_u32(id.value()); // explicit boundary conversion
}
```

Hashing and unordered containers

A strong typedef can be used as a key in unordered containers if you provide hashing. A safe approach is to define a custom hasher type near the strong typedef.

```
#include <cstddef>
#include <cstdint>
#include <functional>
#include <unordered_map>

struct UserIdTag {};
using UserId = Strong<std::uint32_t, UserIdTag>;

struct UserIdHash {
  std::size_t operator()(const UserId& id) const noexcept {
    return std::hash<std::uint32_t>{}(id.value());
  }
};

int main() {
  std::unordered_map<UserId, int, UserIdHash> m;
  m.emplace(UserId{1}, 123);
}
```

Avoid specializing templates inside `namespace std` unless the standard explicitly allows it (it allows specialization of `std::hash` for user-defined types, but many teams still prefer local hashers to keep ownership clear).

18.3.11 Where strong typedefs deliver the biggest safety gains

Strong typedefs are most valuable for values that are frequently mixed up:

- identifiers: `UserId`, `OrderId`, `SessionId`

- sizes and indices: `ByteCount`, `ElementCount`, `Index`
- units and scaling: `Seconds`, `Milliseconds`, `Meters`
- handles and capabilities: `Fd`, `Handle`, `Token`

They are especially important at API boundaries and in security-sensitive subsystems.

18.3.12 Common pitfalls and safe recommendations

Pitfall 1: overloading too many operators

If you add arithmetic operators, you may accidentally make IDs behave like integers. Only provide operations that match the domain meaning.

Pitfall 2: making construction implicit

Keep constructors **explicit**. If you need ergonomic creation, provide named factories:

```
struct UserIdTag {};
using UserId = Strong<std::uint32_t, UserIdTag>;

constexpr UserId make_user_id(std::uint32_t v) { return UserId{v}; }
```

Pitfall 3: forgetting that strong typedefs do not validate

A strong typedef distinguishes meaning but does not automatically enforce invariants. If invariants matter, enforce them (as shown earlier).

Pitfall 4: scattering conversion logic everywhere

Keep conversions at well-defined boundaries (parsing, serialization, external APIs). Inside your domain logic, stay in strong types.

18.3.13 Professional checklist

1. Use strong typedefs for IDs, units, handles, and boundaries where mix-ups are costly.

2. Prefer **explicit** construction and explicit accessors (`value()`).

3. Provide only domain-meaningful operations (often comparisons and hashing).

4. Enforce invariants at construction when the domain requires it.

5. Convert to primitives only at boundaries, not throughout the code.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): class types, user-defined conversions, explicit constructors, overload resolution, and strong typing semantics.

- cppreference.com: *explicit specifier*, *user-defined conversion function*, *enum class* (related type-safety), and *std::hash* specialization rules.

- ISO C++ Core Guidelines: rules promoting strong types, avoiding primitive obsession, avoiding implicit conversions, and expressing intent in interfaces.

- *Effective Modern C++* (Scott Meyers): guidance on explicitness, type deduction pitfalls, and designing safer interfaces.

- *C++ Templates: The Complete Guide* (Vandevoorde, Josuttis, Gregor): generic design patterns and type-safety practices relevant to tagged types.

18.4 Safer APIs

18.4.1 Why API design is a safety topic

In C++, many serious defects are not caused by complex algorithms. They are caused by **ambiguous interfaces** that allow incorrect use:

- unclear ownership (who deletes? who keeps it alive?),
- unclear nullability (can this be null?),
- unclear bounds (how many elements are valid?),
- unclear units (ms vs s, bytes vs KiB),
- unclear error handling (exceptions? error codes? both?),
- unclear thread-safety and lifetime rules.

A safe API is an API where:

- correct use is easy and natural,
- incorrect use is hard or impossible,
- the type system carries intent (ownership, units, constraints),
- failure is explicit and consistent.

18.4.2 Principle 1: Express ownership in types

Do not return raw owning pointers

Returning `T*` from a factory hides ownership.

```
// Avoid
struct Widget {};
Widget* make_widget(); // who deletes?
```

Prefer owning smart pointers:

```
#include <memory>

struct Widget {};

std::unique_ptr<Widget> make_widget() {
  return std::make_unique<Widget>();
}
```

Prefer `std::unique_ptr` as the default owning type

- single owner is simplest to reason about,
- move-only prevents accidental copies and double delete,
- it composes naturally with RAII.

Use `std::shared_ptr` only when lifetime must be shared

Shared ownership is sometimes necessary, but it adds:

- atomic reference counting overhead in many implementations,

- risk of cycles (leaks) unless `std::weak_ptr` breaks them,
- more complex reasoning about who “really” owns.

18.4.3 Principle 2: Express borrowing and nullability clearly

Use references for non-null borrowed parameters

```
struct Config {};

void apply_config(const Config& cfg); // cannot be null, no ownership
↪  transfer
```

Use pointers for optional borrowed parameters

```
struct Logger {};

void set_logger(Logger* log); // may be null; still non-owning by
↪  convention
```

Avoid “nullable references” via `std::optional<std::reference_wrapper<T>`

This technique can be useful in advanced code, but it is often harder to read than a pointer. Use it only when you need reference semantics specifically.

18.4.4 Principle 3: Make bounds explicit

Buffer overflows and out-of-bounds reads are among the highest-impact bug classes. A safer API makes size and bounds part of the type.

Prefer `std::span` for contiguous ranges

```
#include <span>
#include <cstddef>

void digest(std::span<const std::byte> data);
```

This avoids the fragile pattern `(ptr, length)` being split across parameters.

Prefer `std::string_view` for read-only string input (with lifetime discipline)

```
#include <string_view>

void log_line(std::string_view line);
```

Do **not** store a `std::string_view` unless you can guarantee that the referenced character data outlives the view.

Avoid C-string APIs as your primary interface

```
// Avoid as a primary API
void set_name(const char* s); // ambiguous lifetime, may be null, may not
↪  be terminated
```

If you must support C, wrap it at the boundary and convert to safer types internally.

18.4.5 Principle 4: Prefer strong types over primitives

Primitives hide meaning. Strong types prevent accidental mix-ups.

Avoid boolean parameter confusion

```
// Avoid: what does 'true' mean?
void connect(bool use_tls, bool verify_cert);
```

Prefer `enum class`:

```
enum class TlsMode { off, on };
enum class CertVerification { skip, require };

void connect(TlsMode tls, CertVerification verify);
```

Prefer `std::chrono` for time

```
#include <chrono>

void set_timeout(std::chrono::milliseconds t);
```

Use strong typedefs for IDs and handles

```
#include <cstdint>

template<class T, class Tag>
class Strong {
public:
  explicit constexpr Strong(T v) : v_(v) {}
  constexpr T value() const noexcept { return v_; }
private:
  T v_;
};

struct UserIdTag {};
using UserId = Strong<std::uint32_t, UserIdTag>;
```

```
void load_user(UserId id);
```

18.4.6 Principle 5: Make failure explicit and consistent

C++ supports multiple error-handling models (exceptions, error codes, optional values). A safer API chooses one model consistently per subsystem and documents it.

Avoid ambiguous "sentinel" return values

```
// Avoid: -1 might be a valid value; caller must remember convention.
int find_item(/*...*/);
```

Better alternatives include:

- `std::optional<T>` when "no value" is a normal outcome,
- `std::error_code` when you want non-throwing errors,
- exceptions when failures are exceptional and you can enforce invariants via RAII.

Use `std::optional` for "may be missing"

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

std::optional<std::string> find_env(const std::string& key);
```

Use `std::error_code` for non-throwing system-style APIs

```
#include <system_error>
#include <span>
#include <cstddef>
```

```
std::error_code read_all(std::span<std::byte> out);
```

When using exceptions, keep the strong exception safety story

If a function throws, the API should still preserve invariants:

- no leaks (RAII),
- no partially updated observable state unless documented,
- consistent cleanup paths.

18.4.7 Principle 6: Use const-correctness and value categories wisely

Do not overuse forwarding references in public APIs

A template like `template<class T> void f(T&&);` accepts too much and often surprises overload resolution. Prefer stable vocabulary types and explicit overloads.

Prefer pass-by-value for small, owning inputs you intend to copy

This can be safer and simpler than complex overload sets.

```
#include <string>

void set_title(std::string s) {  // value: caller can move or copy
  // store s
}
```

This is often called the “pass by value, then move” pattern.

18.4.8 Principle 7: Clarify lifetime and threading contracts

Even the best types cannot express everything. A safe API states:

- what must outlive what (views, callbacks, observers),
- whether methods are thread-safe, re-entrant, or require external synchronization,
- which operations are noexcept and what they guarantee.

Callbacks: avoid dangling references

If you store callbacks, prefer owning callables (e.g., `std::function`) or explicit lifetime-managed registrations. Do not store references to temporary lambdas.

```
#include <functional>

class Timer {
public:
  void set_handler(std::function<void()> f) { handler_ = std::move(f); }
  void fire() { if (handler_) handler_(); }
private:
  std::function<void()> handler_;
};
```

18.4.9 Principle 8: Ensure APIs are hard to misuse

Use types and access control to forbid invalid calls

Example: prevent deletion through base pointer if ownership must be controlled.

```
struct NonDeletable {
  virtual void run() = 0;
```

```
protected:
  ~NonDeletable() = default; // cannot delete via NonDeletable*
};
```

Prefer constructors that establish invariants

Make invalid states unrepresentable:

```
#include <string_view>
#include <string>
#include <stdexcept>

class NonEmptyString {
public:
  explicit NonEmptyString(std::string_view sv) : s_(sv) {
    if (s_.empty()) throw std::invalid_argument("empty string");
  }
  std::string_view view() const noexcept { return s_; }
private:
  std::string s_;
};
```

18.4.10 Design patterns for safer APIs (common recipes)

Recipe A: “View in, own inside”

```
#include <string>
#include <string_view>

class Name {
public:
  explicit Name(std::string_view sv) : s_(sv) {}
  std::string_view view() const noexcept { return s_; }
```

```
private:
  std::string s_;
};
```

Recipe B: “Span in” instead of `(ptr, size)`

```
#include <span>
#include <cstddef>

void compress(std::span<const std::byte> in,
              std::span<std::byte> out);
```

Recipe C: “Enum class” instead of boolean flags

```
enum class FlushMode { no, yes };
void write_record(FlushMode flush);
```

Recipe D: “Strong typedef” IDs and handles

```
#include <cstdint>

struct FileIdTag {};
using FileId = Strong<std::uint32_t, FileIdTag>;

void close_file(FileId id);
```

18.4.11 Checklist: auditing an API for safety

1. Ownership is explicit (`unique_ptr`/`shared_ptr`, not raw owning pointers).
2. Borrowing and nullability are explicit (references for non-null, pointers for optional).
3. Bounds are explicit (`span`, size-aware containers, not raw pointers alone).

4. Units are explicit (`chrono`, strong types).
5. Errors are explicit and consistent (no ambiguous sentinel values).
6. Lifetimes of views and callbacks are documented and enforced where possible.
7. Thread-safety expectations are stated.
8. Invariants are established at construction and preserved.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): references, pointers, object lifetimes, exceptions, `enum class`, templates, and standard library vocabulary types.
- cppreference.com: `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`, `std::span`, `std::string_view`, `std::optional`, `std::error_code`, `std::function`.
- ISO C++ Core Guidelines: ownership and lifetime guidelines (RAII, avoid raw owning pointers), interface guidelines (express intent, avoid boolean traps), and bounds safety recommendations.
- Microsoft Learn / GCC / Clang documentation: guidance on diagnosing lifetime/bounds issues and using modern standard types (toolchain support notes).
- *Effective Modern C++* (Scott Meyers): advice on interface design, passing by value vs reference, and avoiding deduction/forwarding surprises.

Part VI

Modern C++ Safety Tools and Language Features

Chapter 19

Sanitizers and Runtime Analysis

19.1 AddressSanitizer

19.1.1 What AddressSanitizer is

AddressSanitizer (ASan) is a compiler-instrumentation and runtime system that detects many classes of **memory safety bugs** at runtime with useful, actionable diagnostics (stack traces, bug type, and the exact access).
ASan is designed for **development and testing**: it intentionally trades performance and memory overhead for high bug-finding power. It is one of the most effective tools for discovering:

- heap buffer overflows and underflows,
- stack buffer overflows and underflows,
- global buffer overflows,

- use-after-free,
- double-free and invalid free,
- certain stack lifetime bugs (use-after-return / use-after-scope),
- and (often) memory leaks via LeakSanitizer integration on supported platforms.

19.1.2 How ASan works (mental model)

At a high level, ASan works by:

- **instrumenting** your code at compile time to check memory accesses,
- using a **shadow memory** mapping to track which regions are valid,
- placing **red zones** around allocations to detect out-of-bounds,
- **quarantining** freed heap memory to increase the chance of catching use-after-free.

You do not need to understand internal details to use it correctly, but this model explains why ASan is effective: it makes invalid accesses fail *at the moment they occur* and points you to the call stack that caused them.

19.1.3 When to use ASan

Use ASan whenever you want to detect memory safety bugs early:

- unit tests and integration tests,
- fuzzing runs (ASan is commonly paired with coverage-guided fuzzers),
- CI “sanitized” builds,
- reproducing production crashes in a controlled test environment.

19.1.4 Building with ASan (Clang / GCC)

Core flags

The essential build switch is:

```
# Compile + link with AddressSanitizer
CXX=clang++   # or g++
CXXFLAGS="-O1 -g -fsanitize=address -fno-omit-frame-pointer"
LDFLAGS="-fsanitize=address"

$CXX $CXXFLAGS main.cpp -o app $LDFLAGS
```

Recommended defaults:

- `-g` for line-level stack traces.
- `-O1` (often a good balance: catches bugs while keeping runtime usable).
- `-fno-omit-frame-pointer` to improve stack traces and profiler output.

Optional: detect more stack lifetime issues

Some toolchains support additional instrumentation and runtime options for stack bugs:

- **use-after-scope**: often enabled via an extra compile flag such as `-fsanitize-address-use-after-scope` in Clang-based toolchains.
- **use-after-return**: commonly controlled as a runtime option (for example via `ASAN_OPTIONS=detect_stack_use_after_return=1`).

These checks can increase overhead, so enable them when you suspect stack lifetime errors or in dedicated CI sanitizer runs.

Static vs shared sanitizer runtime

On some platforms you can choose whether the sanitizer runtime is linked dynamically or statically (e.g., `-shared-libasan` / `-static-libasan` in GCC/Clang-style toolchains). The best choice depends on your deployment and test environment:

- dynamic runtime can reduce binary size and simplify updates,
- static runtime can simplify running tests in minimal containers.

19.1.5 Building with ASan (MSVC on Windows)

MSVC supports AddressSanitizer through the `/fsanitize=address` option.

```
:: MSVC example
cl /Zi /Od /fsanitize=address /EHsc main.cpp
```

Important operational note from Microsoft guidance: ASan is intended for development/testing, not for shipping production binaries.

19.1.6 Running ASan-instrumented programs

ASan behavior is configured primarily at runtime using the environment variable `ASAN_OPTIONS`. The options do not affect compilation; they control the runtime behavior of the instrumented program.

Practical ASAN_OPTIONS you will use often

Control whether the program stops at the first error By default, ASan typically terminates the process on the first detected memory bug. For some test workflows you may want to continue after reporting (when supported), but do not rely on recovery for correctness.

Common knobs include options in the `halt_on_error`/recover family, and compiler-side recovery flags (toolchain-dependent).

Leak detection Leak detection is commonly integrated via LeakSanitizer (LSan) and is often enabled by default on Linux in ASan builds; on macOS it may require enabling via `ASAN_OPTIONS=detect_leaks=1`.

```
# Enable leak detection (commonly needed on macOS)
export ASAN_OPTIONS="detect_leaks=1"
./app
```

Symbolization Good stack traces require symbols:

- compile with `-g`,
- keep debug symbols available for the binary and relevant libraries,
- ensure the environment can symbolize stacks (toolchain/runtime dependent).

Logging Many ASan runtimes support writing reports to files (useful in CI) via options such as a `log_path`-style setting.

19.1.7 Suppressions and ignore lists

Sometimes you need to suppress known, acceptable issues (for example, third-party code you cannot change). Modern Clang-based toolchains support **sanitizer special case lists** (also called ignore lists) that can disable instrumentation or reporting for specific entities. Typical workflow:

- create an ignorelist file that names functions/files to treat specially,

- pass it at compile time using the appropriate flag (toolchain-specific),
- keep suppressions minimal, documented, and regularly reviewed.

Suppressions should be a last resort: prefer fixing the underlying bug or isolating unsafe code.

19.1.8 A small example: ASan catching use-after-free

Buggy code

```
#include <iostream>

int main() {
  int* p = new int(42);
  delete p;

  // Use-after-free
  std::cout << *p << "\n";

  return 0;
}
```

Build and run

```
clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer uaf.cpp -o uaf
./uaf
```

You should see an ASan report identifying a heap-use-after-free, including:

- the invalid read/write location,
- the allocation and deallocation stack traces,
- the access stack trace.

19.1.9 Limitations and correct expectations

ASan is powerful, but it is not a proof of correctness.

What ASan does not guarantee

- **No race detection**: use ThreadSanitizer (TSan) for data races.
- **No uninitialized-read detection**: use MemorySanitizer (MSan) where supported.
- **Not all bugs are detected**: some memory bugs may escape detection depending on execution paths, optimizations, allocator behavior, and platform constraints.

Performance and memory overhead

ASan adds noticeable overhead (runtime slowdown and increased memory usage). This is normal. For safety engineering, the recommended practice is to run:

- normal optimized builds for performance testing,
- dedicated sanitized builds for correctness testing (often in CI and fuzzing).

Do not ignore "only in ASan" failures

If ASan reports a bug, treat it as a real defect. ASan is designed to have extremely low false positive rates for memory safety violations, because it detects actual invalid accesses.

19.1.10 Professional workflow (recommended)

1. Add a dedicated **ASan build configuration** (CMake preset or build profile).
2. Run unit tests and integration tests under ASan in CI on every change.

3. For hard-to-reproduce crashes, build with ASan and run the smallest reproducer.
4. For components that parse untrusted input, add fuzzing with ASan enabled.
5. Keep suppressions rare and justified; track them like technical debt.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): object lifetime rules, undefined behavior, and pointer validity (the language rules ASan helps you enforce in practice).
- LLVM/Clang Sanitizers documentation: AddressSanitizer and LeakSanitizer behavior, supported bug classes, runtime configuration, and ignorelists.
- GCC documentation and libsanitizer notes: AddressSanitizer integration and usage in GCC-based toolchains.
- Microsoft Learn (MSVC): `/fsanitize=address` and AddressSanitizer usage guidance for Visual Studio toolchains.
- Google Sanitizers documentation/wiki: conceptual overview and practical guidance for ASan/LSan usage.

19.2 UndefinedBehaviorSanitizer

19.2.1 What UndefinedBehaviorSanitizer is

UndefinedBehaviorSanitizer (UBSan) is a compiler-based runtime checker that instruments your program to detect many kinds of **undefined behavior (UB)** while the program runs. UB is dangerous in C++ because it means the program has left the rules of the language. After UB happens, the compiler is allowed to assume it never happens, and it may optimize in

ways that make the program behave unpredictably. UBSan helps you catch these issues early by turning many UB cases into precise runtime reports (often with file/line and a stack trace when enabled).
UBSan is intended for **development, testing, and CI**, not as a production safety mechanism.

19.2.2 What UBSan can detect (practical categories)

UBSan can detect many UB families, including:

Integer UB and numeric traps

- signed integer overflow (when enabled),
- invalid shifts (shift by a negative amount or by an amount $\geq$ bit width),
- division problems such as division by zero in checked configurations,
- some problematic conversions (for example, float-to-integer overflow when enabled).

Pointer and object model violations

- dereferencing a null pointer (in instrumented cases),
- misaligned pointer dereference (alignment UB),
- object size and bounds issues where the compiler can instrument the access,
- invalid object member access patterns that violate language rules (varies by toolchain/check).

Type-related UB (when enabled)

- invalid downcasts and type confusion in some modes (for example, `vptr` checks),
- passing values outside the valid range for certain types (for example, `bool` and enums in some configurations).

Notes about coverage

- UBSan does **not** detect every possible UB. It detects what it can instrument.
- The exact set of checks and defaults depends on the compiler version, target, and selected `-fsanitize=` options.

19.2.3 How UBSan works (a useful mental model)

UBSan inserts runtime checks at compile time. When a check fails, the UBSan runtime reports:

- the kind of undefined behavior (e.g., “shift out of bounds”),
- where it occurred (source location),
- and often a stack trace if you enable stack traces and keep debug symbols.

Unlike AddressSanitizer, UBSan typically has:

- **small runtime overhead** for many checks,
- **no change to address space layout** and usually no ABI impact,
- a focus on **language-rule violations** rather than memory redzones.

19.2.4 Building with UBSan (Clang / GCC)

Common, practical build flags

```
# Typical UBSan build (Clang or GCC)
CXX=clang++   # or g++
CXXFLAGS="-O1 -g -fsanitize=undefined -fno-omit-frame-pointer"
LDFLAGS="-fsanitize=undefined"

$CXX $CXXFLAGS main.cpp -o app $LDFLAGS
```

Recommendations:

- Use `-g` for readable reports.
- Use `-O1` as a good baseline for tests (fast enough, still debuggable).
- Keep frame pointers (`-fno-omit-frame-pointer`) for better stack traces.

Selecting checks explicitly

Many teams prefer enabling specific checks rather than relying on broad defaults. Examples (availability may vary by toolchain/version):

```
# Example: enable common UB checks explicitly
clang++ -O1 -g
↪  -fsanitize=shift,integer-divide-by-zero,unreachable,vptr,bounds,null,alignmen
↪  main.cpp -o app
```

Guideline:

- Start with `-fsanitize=undefined` in CI.
- If noise is high, move to a curated list and add checks back gradually.

19.2.5 Running UBSan: controlling behavior with `UBSAN_OPTIONS`

UBSan is configured at runtime using the environment variable `UBSAN_OPTIONS`. Common options include:

Stack traces

By default, some UBSan configurations may not print a full stack trace. Enable stack traces explicitly:

```
export UBSAN_OPTIONS="print_stacktrace=1"
./app
```

Halting vs continuing

Many UBSan configurations *report and continue* by default. For CI, it is often better to stop immediately:

```
export UBSAN_OPTIONS="halt_on_error=1:print_stacktrace=1"
./app
```

Log files

In CI you may prefer log files:

```
export
↪  UBSAN_OPTIONS="log_path=ubsan_report:print_stacktrace=1:halt_on_error=1"
./app
```

19.2.6 Suppressions and targeted disabling

Suppressions are useful when you must tolerate known issues in third-party code, but they should be rare and reviewed regularly.

Runtime suppressions

UBSan supports suppressions via `UBSAN_OPTIONS=suppressions=...` in many environments.

```
export UBSAN_OPTIONS="suppressions=ubsan.supp:print_stacktrace=1"
./app
```

Compile-time ignorelists (special case lists)

Clang-based toolchains support **sanitizer special case lists** (ignorelists) to disable or adjust sanitizer behavior for specific functions/files. This is a controlled way to avoid instrumenting known-problematic code.

Function-level disabling (surgical escape hatch)

For rare cases where a function intentionally relies on behavior that UBSan would report, some compilers provide function attributes to disable specific checks for that function. Use this only when the behavior is well-justified and documented.

19.2.7 A minimal example: shift-out-of-bounds

Buggy code

```
#include <cstdint>
#include <iostream>

int main() {
  std::uint32_t x = 1u;
  int shift = 32;              // UB for a 32-bit type: shift count
  ↪  equals bit width
  std::uint32_t y = x << shift;
```

```
  std::cout << y << "\n";
}
```

Build and run

```
clang++ -O1 -g -fsanitize=undefined -fno-omit-frame-pointer shift.cpp -o
↪  shift_ubsan
./shift_ubsan
```

Expected outcome:

- UBSan reports a “shift out of bounds” (or equivalent) diagnostic,
- identifies the source location,
- optionally includes a stack trace when enabled.

19.2.8 How UBSan fits with other sanitizers

UBSan complements other tools; it does not replace them.

- **AddressSanitizer (ASan)**: best for heap/stack/global memory corruption, use-after-free, buffer overflows.
- **ThreadSanitizer (TSan)**: best for data races and concurrency bugs.
- **MemorySanitizer (MSan)**: best for uninitialized reads (supported mainly on Clang/Linux).

A common CI setup is:

- one ASan job,

- one UBSan job,
- one TSan job (when feasible),
- plus fuzzing with ASan/UBSan.

19.2.9 Platform notes (important for planning)

GCC and Clang

UBSan is supported by both GCC and Clang (with different coverage and defaults depending on version and target).

MSVC

As of many common toolchains, MSVC provides AddressSanitizer support, while UBSan support is not generally available in the same way as Clang/GCC. On Windows, many teams run UBSan using Clang/LLVM-based toolchains (including clang-cl workflows) when they need UBSan.

19.2.10 Professional workflow (recommended)

1. Add a dedicated UBSan build configuration (CMake preset / CI job).
2. Compile with `-O1 -g -fno-omit-frame-pointer`.
3. Run unit + integration tests under UBSan.
4. Set `UBSAN_OPTIONS=print_stacktrace=1:halt_on_error=1` in CI.
5. Treat every UBSan report as a real defect unless proven otherwise.
6. Keep suppressions minimal and track them as technical debt.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): definition and consequences of undefined behavior, and the language rules UBSan helps validate dynamically.
- LLVM/Clang Documentation: *UndefinedBehaviorSanitizer* (supported checks, instrumentation model, runtime behavior).
- LLVM/Clang Documentation: *Sanitizer Special Case List* (ignorelists for UBSan and other sanitizers).
- GCC Documentation: sanitizer options and behavior as implemented by GCC and its libsanitizer integration.
- Microsoft toolchain notes: AddressSanitizer support in MSVC and the practical need for Clang-based toolchains on Windows when UBSan is required.

19.3 ThreadSanitizer

19.3.1 What ThreadSanitizer is

ThreadSanitizer (TSan) is a compiler-instrumentation and runtime system for detecting **data races** (and related concurrency issues) in C/C++ programs while they execute. It is one of the most effective practical tools for finding races that are hard to reproduce in production. A **data race** occurs when:

- two (or more) threads access the same memory location concurrently,
- at least one access is a write,
- and there is no proper synchronization that establishes a happens-before relation.

In C++ terms, a data race is undefined behavior for ordinary (non-atomic) objects. TSan helps you detect the concrete runtime situations that violate the rules.

19.3.2 What TSan detects in practice

Primary target: data races on ordinary memory

TSan reports races such as:

- unsynchronized reads/writes to shared variables,
- races on heap objects shared across threads,
- races on globals and statics,
- races introduced by publishing pointers/references without synchronization.

What TSan understands (and why results are usually meaningful)

TSan is designed to understand common synchronization and atomic operations, so it can distinguish correct synchronization from real races.
However, if your program uses synchronization that TSan cannot see (for example, custom inline assembly fences, unsupported runtime primitives, or exotic lock-free techniques without standard atomics), TSan can produce misleading reports.

A key limitation: TSan only reports what executes

TSan finds races that happen in the observed runs. If a code path does not execute, or the timing does not trigger the conflicting access, there will be no report. This is why TSan is commonly paired with:

- stress tests,

- randomized schedulers and repeated runs,
- fuzzing for concurrent components,
- CI sanitizer jobs.

19.3.3 How TSan works (a useful mental model)

TSan instruments:

- memory accesses,
- synchronization operations (mutexes, condition variables, atomics, etc.),

and builds a dynamic model of happens-before relationships. When it observes two conflicting accesses that are not properly ordered by synchronization, it reports a data race, usually with stack traces of the racing accesses.

19.3.4 Supported platforms (important planning note)

Clang (LLVM) supported platforms

The Clang documentation lists TSan support on multiple 64-bit platforms (Linux, Darwin/macOS, Android, and several BSDs). 32-bit platforms are considered problematic and are not planned.

MSVC status on Windows

As of current Microsoft documentation, MSVC officially documents `/fsanitize=address` (AddressSanitizer). The same documentation and related Microsoft notes indicate other sanitizers (including `/fsanitize=thread`) are not yet provided as first-class MSVC options. Practical implication:

- For TSan on Windows, many teams use Clang/LLVM-based toolchains where supported, or run TSan on Linux/macOS CI even if production is Windows.
- If your project is cross-platform, running sanitizer CI on Linux is often the most reliable path to TSan coverage.

19.3.5 Building with TSan (Clang / GCC)

Core flags

To enable TSan, compile and link with `-fsanitize=thread`:

```
# Clang or GCC
CXX=clang++   # or g++
CXXFLAGS="-O1 -g -fsanitize=thread -fno-omit-frame-pointer"
LDFLAGS="-fsanitize=thread"

$CXX $CXXFLAGS main.cpp -o app $LDFLAGS
```

Recommended defaults:

- `-g` for source locations and stack traces.
- `-O1` as a good balance (more realistic codegen than `-O0`, still debuggable).
- `-fno-omit-frame-pointer` to improve stack traces.

Important: sanitizer incompatibilities

In many real toolchains, you should assume you will run different builds for different sanitizers (ASan build, TSan build, UBSan build). In particular, TSan is commonly run as a separate configuration rather than combined into a single binary. (Build multiple CI jobs rather than trying to enable everything at once.)

19.3.6 Running TSan: `TSAN_OPTIONS`

TSan is configured at runtime using `TSAN_OPTIONS`.

Stop on the first report (useful for CI)

```
export TSAN_OPTIONS="halt_on_error=1"
./app
```

Write reports to files (useful for CI artifacts)

```
export TSAN_OPTIONS="log_path=tsan_report:halt_on_error=1"
./app
```

Suppress known issues (last resort)

If you must suppress known reports (for example, third-party libraries you cannot fix), TSan supports a suppressions file via `TSAN_OPTIONS`.

```
# Create a suppressions file and point TSAN to it.
export TSAN_OPTIONS="suppressions=tsan.supp:halt_on_error=1"
./app
```

Treat suppressions as tracked technical debt: keep them small, justified, and reviewed.

19.3.7 A minimal example: a real data race

Buggy code

```
#include <thread>
#include <iostream>

int counter = 0;
```

```
void worker() {
  for (int i = 0; i < 100000; ++i) {
    ++counter; // data race: unsynchronized write
  }
}

int main() {
  std::thread t1(worker);
  std::thread t2(worker);
  t1.join();
  t2.join();

  std::cout << counter << "\n";
}
```

Build and run under TSan

```
clang++ -O1 -g -fsanitize=thread -fno-omit-frame-pointer race.cpp -o
↪  race_tsan
TSAN_OPTIONS="halt_on_error=1" ./race_tsan
```

Expected outcome:

- TSan reports a data race,
- identifies the racing accesses and stack traces,
- points to the exact source locations.

19.3.8 Fix patterns (what TSan reports usually mean)

When TSan reports a race, the correct fix is typically one of the following:

1) Protect shared state with a mutex

```
#include <mutex>
#include <thread>
#include <iostream>

int counter = 0;
std::mutex m;

void worker() {
  for (int i = 0; i < 100000; ++i) {
    std::lock_guard<std::mutex> lock(m);
    ++counter;
  }
}

int main() {
  std::thread t1(worker), t2(worker);
  t1.join(); t2.join();
  std::cout << counter << "\n";
}
```

2) Use atomics for simple counters

```
#include <atomic>
#include <thread>
#include <iostream>

std::atomic<int> counter{0};

void worker() {
  for (int i = 0; i < 100000; ++i) {
    counter.fetch_add(1, std::memory_order_relaxed);
```

```
  }
}

int main() {
  std::thread t1(worker), t2(worker);
  t1.join(); t2.join();
  std::cout << counter.load() << "\n";
}
```

3) Fix publication (happens-before) bugs

Many races come from publishing a pointer/reference to an object before its initialization is properly synchronized. Common fixes:

- initialize fully before publishing,
- publish through a mutex,
- publish via atomics with correct ordering,
- use thread-safe initialization patterns (e.g., function-local static initialization).

19.3.9 Common false alarms and how to reason about them

TSan aims to be accurate, but reports can be misleading when synchronization is not visible to the tool. Examples include:

- synchronization implemented in inline assembly,
- custom lock-free primitives without compiler-recognized atomics,
- unsupported runtime primitives.

In these cases, do not immediately suppress the report. First, verify whether your synchronization establishes a real happens-before relation in the C++ memory model.

19.3.10 Performance and usage guidance

TSan introduces significant runtime overhead and can increase memory usage. This is expected. The professional approach is:

- run TSan in dedicated test/CI configurations,
- keep production builds separate,
- use targeted stress tests to maximize race discovery.

19.3.11 Professional workflow (recommended)

1. Add a dedicated **TSan build profile** (CMake preset / CI job).
2. Compile with `-O1 -g -fno-omit-frame-pointer -fsanitize=thread`.
3. Run unit + integration tests under TSan.
4. Add stress tests (repeat runs, randomized scheduling, higher iteration counts).
5. Use `TSAN_OPTIONS` in CI (at least `halt_on_error=1` and `log_path=`).
6. Treat suppressions as rare exceptions; prefer fixing the underlying issue.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): the memory model, data races, atomics, and synchronization (the rules TSan helps validate dynamically).

- LLVM/Clang Documentation: *ThreadSanitizer* (instrumentation model, runtime options, supported platforms).
- Google Sanitizers documentation/wiki: practical guidance and common race patterns; suppressions mechanism.
- Research paper: *ThreadSanitizer: data race detection in practice* (design and behavior of TSan).
- Microsoft Learn: */fsanitize* and AddressSanitizer scope; Microsoft notes on future sanitizer directions.
- Mozilla documentation: TSan accuracy assumptions and unsupported synchronization caveats (useful for interpreting reports).

19.4 When and how to use them

19.4.1 Why sanitizers belong in every safety workflow

Sanitizers are compiler-instrumented runtime checkers. They detect concrete violations while the program runs, and they typically report:

- the exact kind of defect (e.g., heap-use-after-free, data race, shift-out-of-bounds),
- the instruction/location where it happened,
- a call stack for the access and often for the allocation/synchronization context.

They are most effective when used continuously (CI) and early (unit tests, integration tests, fuzzing), because many of these bugs are timing-dependent and can be extremely hard to reproduce later.

19.4.2 Which sanitizer to use (mapping defect class to tool)

Choose sanitizers based on what you are trying to catch:

Memory safety and lifetime bugs

- **AddressSanitizer (ASan)**: out-of-bounds, use-after-free, double-free, and related heap/stack/global memory errors.
- **LeakSanitizer (LSan)** (often integrated with ASan on many platforms): memory leaks in tests and long-running scenarios.

Undefined behavior and language-rule violations

- **UndefinedBehaviorSanitizer (UBSan)**: many forms of undefined behavior such as invalid shifts, some integer overflows (when enabled), misalignment, and type-related UB checks.

Concurrency

- **ThreadSanitizer (TSan)**: data races and related concurrency misuse.

Uninitialized reads (where supported)

- **MemorySanitizer (MSan)**: uses of uninitialized memory (primarily on Clang/Linux in many real-world toolchains).

19.4.3 When to run sanitizers

Always run them in these situations

- Before merging changes (CI sanitizer jobs).

- When refactoring code that touches lifetimes, ownership, or threading.
- When investigating crashes, memory corruption, or rare heisenbugs.
- When parsing untrusted input (pair with fuzzing).

When sanitizer runs pay off the most

- **Nightly or scheduled stress runs**: more time means more interleavings and more coverage.
- **After adding tests**: sanitizers only find what executes.
- **During API design**: sanitizer feedback often exposes unclear ownership/lifetime contracts.

Do not treat sanitizer coverage as a proof

Sanitizers find bugs that occur in the observed runs. They reduce risk dramatically, but they do not prove correctness.

19.4.4 How to build sanitized configurations correctly

Core build principles (all sanitizers)

- Keep symbols: compile with `-g` (or MSVC `/Zi`).
- Use a realistic optimization level: `-O1` is a strong default.
- Keep frame pointers when possible: `-fno-omit-frame-pointer` improves stacks.
- Treat sanitizer builds as separate configurations (do not mix and match randomly).

Typical GCC/Clang command lines

ASan (memory errors)

```
clang++ -O1 -g -fno-omit-frame-pointer -fsanitize=address \
  main.cpp -o app_asan
```

UBSan (undefined behavior)

```
clang++ -O1 -g -fno-omit-frame-pointer -fsanitize=undefined \
  main.cpp -o app_ubsan
```

TSan (data races)

```
clang++ -O1 -g -fno-omit-frame-pointer -fsanitize=thread \
  main.cpp -o app_tsan
```

Important: sanitizer combinations and separation

In professional CI, you typically run separate jobs:

- **ASan job** for memory safety,
- **UBSan job** for UB checks,
- **TSan job** for races,
- optional **MSan job** for uninitialized reads.

Reason: different sanitizers have different instrumentation models and runtime requirements, and combining them can be unsupported or counterproductive. Prefer multiple targeted runs over one “everything” build.

19.4.5 Runtime configuration: make reports actionable

Most sanitizer runtimes are configured through environment variables:

- `ASAN_OPTIONS` for AddressSanitizer,
- `UBSAN_OPTIONS` for UndefinedBehaviorSanitizer,
- `TSAN_OPTIONS` for ThreadSanitizer,
- `LSAN_OPTIONS` for LeakSanitizer (when used explicitly).

Recommended CI defaults

Stop at first error (fail fast) Failing fast makes CI output smaller and easier to act on.

```
export ASAN_OPTIONS="halt_on_error=1"
export UBSAN_OPTIONS="halt_on_error=1:print_stacktrace=1"
export TSAN_OPTIONS="halt_on_error=1"
```

Write logs to files (CI artifacts)

```
export ASAN_OPTIONS="halt_on_error=1:log_path=asan"
export UBSAN_OPTIONS="halt_on_error=1:print_stacktrace=1:log_path=ubsan"
export TSAN_OPTIONS="halt_on_error=1:log_path=tsan"
```

Symbolization (readable stacks)

Sanitizer stack traces are only as good as your symbolization:

- compile with debug information,
- avoid stripping symbols in sanitizer builds,

- ensure the environment can symbolize stacks (toolchain-dependent),
- in some environments you may need to make `llvm-symbolizer` discoverable.

19.4.6 How to interpret reports (and avoid common mistakes)

Treat sanitizer findings as real defects by default

Sanitizers typically report concrete invalid operations that occurred at runtime. The safest posture is:

> Assume it is a real bug until you can prove otherwise.

Understand the three-stack pattern (especially ASan)

Many ASan reports are most useful when you look at:

- the stack trace of the invalid access (where it crashed),
- the stack trace of the allocation (where the object was created),
- the stack trace of the deallocation (where it was freed).

This trio usually points directly to the ownership/lifetime contract violation.

TSan reports require memory-model thinking

A TSan "data race" means the observed interleaving lacked a happens-before relation. Fixes usually fall into one of these categories:

- protect shared state with a mutex,
- use atomics correctly for shared state,

- fix publication and initialization ordering,
- stop sharing the object (ownership redesign).

UBSan reports often reveal silent portability hazards

UBSan frequently catches bugs that "work on my machine" but are illegal C++: misaligned access, invalid shift counts, and signed overflow assumptions. These are exactly the bugs that become security problems later.

19.4.7 Suppressions and ignore lists: use them like a scalpel

Suppressions are sometimes necessary (third-party code, legacy blocks), but they are a risk: they can hide real regressions.

Rules for safe suppression

1. Suppress only what you cannot fix yet, and attach a ticket/owner.
2. Keep suppressions minimal and specific (function/file scope, not global).
3. Review suppressions regularly and delete them when fixed.
4. Never suppress issues in security-critical code paths without a plan.

Example: TSan suppressions

```
export TSAN_OPTIONS="suppressions=tsan.supp:halt_on_error=1"
./app_tsan
```

19.4.8 Where sanitizers fit in a modern CI pipeline

A practical, high-signal CI layout:

- **Fast lane (per commit)**:
  - unit tests under ASan,
  - unit tests under UBSan,
  - selected concurrency tests under TSan (if runtime permits).
- **Deep lane (nightly/scheduled)**:
  - full integration suite under ASan and UBSan,
  - repeated stress runs under TSan,
  - fuzzing with ASan (often plus UBSan).

19.4.9 How to integrate sanitizers with CMake (typical pattern)

Keep a separate preset or build type rather than injecting sanitizer flags everywhere.

```
# Example snippet for a sanitizer-enabled build configuration
set(SAN_FLAGS_COMMON "-O1 -g -fno-omit-frame-pointer")

# Choose one sanitizer per configuration:
#   -fsanitize=address
#   -fsanitize=undefined
#   -fsanitize=thread
set(SAN_FLAGS "${SAN_FLAGS_COMMON} -fsanitize=address")
```

```
add_compile_options(${SAN_FLAGS})
add_link_options(-fsanitize=address)
```

In practice, you create one configuration per sanitizer and run them as separate CI jobs.

19.4.10 Practical “how-to” recipes

Recipe A: reproduce a production crash

1. Reduce the input to a minimal reproducer (if input-driven).
2. Build with ASan (and optionally UBSan).
3. Run the reproducer under the sanitized binary.
4. Use the report to identify the lifetime/ownership mistake.

Recipe B: hard-to-reproduce race

1. Build with TSan.
2. Increase iteration counts; add stress loops.
3. Run multiple times (races are timing-sensitive).
4. Fix by establishing a clear happens-before relation.

Recipe C: “safe by default” test runner

```
#!/usr/bin/env bash
set -euo pipefail

export ASAN_OPTIONS="halt_on_error=1:detect_leaks=1"
```

```
export UBSAN_OPTIONS="halt_on_error=1:print_stacktrace=1"
export TSAN_OPTIONS="halt_on_error=1"

./unit_tests
```

19.4.11 Common pitfalls (and how to avoid them)

Pitfall 1: missing debug symbols

Without symbols, stack traces lose most of their value. Always build sanitized tests with debug info.

Pitfall 2: ignoring "only fails under sanitizer"

Sanitizer failures usually mean a real bug. The instrumentation changes timing and layout, which may expose latent defects. Fix the defect; do not dismiss it.

Pitfall 3: storing views and blaming sanitizers

ASan often catches use-after-scope/use-after-free caused by storing: `std::string_view`, `std::span`, raw pointers, or references. The fix is almost always an ownership/lifetime redesign (own the data or constrain lifetime).

Pitfall 4: using unsupported synchronization under TSan

If you use synchronization primitives that the tool cannot model, TSan may misreport. Prefer standard library synchronization and atomics for best results.

19.4.12 Bottom line recommendations

- Run **ASan + UBSan** on every change in CI (high value, catches many defects).

- Run **TSan** regularly (at least nightly) for concurrency-heavy components.
- Add **fuzzing with ASan** for parsers and input-driven code.
- Keep sanitizer builds **separate** and reproducible (dedicated presets).
- Treat suppressions as **temporary debt**, not a permanent solution.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): object lifetime rules, undefined behavior, and the memory model (data races and happens-before).
- LLVM/Clang Sanitizers documentation: AddressSanitizer, UndefinedBehaviorSanitizer, ThreadSanitizer, and configuration guidance.
- GCC documentation and libsanitizer integration notes: sanitizer enabling flags and runtime environment variables.
- Microsoft Learn: AddressSanitizer and `/fsanitize=address` usage and runtime notes for MSVC toolchains.
- Google Sanitizers documentation/wiki: practical usage patterns, runtime options, and suppressions guidance.

Chapter 20

Static Analysis Tools

20.1 What static analysis can (and cannot) catch

20.1.1 What static analysis is (in one sentence)

Static analysis examines source code (and sometimes build metadata) to find likely bugs and rule violations **without running the program**. It complements sanitizers: static analysis reasons about *possible* executions, while sanitizers report what *actually* happened in a run.

20.1.2 Two major families of static analysis

1) Rule-based checkers (pattern and semantic checks)

These tools match known risky patterns and API misuses, often with C++ semantic awareness. Examples include many **clang-tidy** checks (bugprone, performance, readability, modernize, etc.).
Typical strengths:

- high speed,
- good at enforcing coding rules and preventing common mistakes,
- often low setup cost.

Typical limits:

- not fully path-sensitive by default,
- may miss bugs that require deeper interprocedural reasoning.

2) Path-sensitive analyzers and dataflow engines (deeper reasoning)

These tools attempt to reason about control flow and program states across branches and function calls.

Path-sensitive symbolic execution The **Clang Static Analyzer** is explicitly described as **path-sensitive** and **inter-procedural**, using **symbolic execution**.

Dataflow/taint tracking **CodeQL** provides libraries for local/global dataflow and taint tracking to model how values can flow from sources to sinks across a codebase.
Typical strengths:

- can find issues that require reasoning across multiple branches or calls,
- can detect classes of security issues (taint-to-sink patterns) at scale,
- can provide richer explanations of how a defect is reached (tool-dependent).

Typical limits:

- more configuration and compute cost,

- more sensitive to build configuration and modeling gaps,
- higher risk of false positives/negatives when the model is incomplete.

20.1.3 What static analysis can catch well

1) Many memory and resource management defects (in certain patterns)

Static analysis can often catch:

- resource leaks (forgetting to free/close on some paths),
- null pointer dereferences (in reachable paths),
- use-after-free patterns (when lifetime relationships are visible to the analyzer),
- buffer overruns in patterns the tool models well.

For example, Microsoft C/C++ Code Analysis explicitly lists common reported errors such as **buffer overruns**, **uninitialized memory**, **null pointer dereferences**, and **memory/resource leaks**.

2) API misuse and dangerous patterns (high signal, fast checks)

Rule-based checkers are strong at preventing common mistakes:

- swapped or easily-swappable parameters,
- suspicious comparisons,
- exception and move-misuse patterns,
- incorrect `enable_if` usage,

- unsafe or suspicious `memset`/`memcpy` patterns.

The clang-tidy check catalog demonstrates the breadth of these categories (especially `bugprone-*`).

3) Security patterns via dataflow/taint tracking (when modeled well)

If sources and sinks are defined, dataflow/taint analyses can detect issues like:

- user-controlled input reaching dangerous operations,
- missing validation/sanitization steps,
- sensitive data flowing to logs or network operations.

CodeQL documentation describes local/global dataflow and taint tracking as core mechanisms.

4) Contract and annotation-driven findings

Some ecosystems become much more powerful when you add annotations:

- Windows/MSVC analysis can leverage annotations (for example SAL) to improve detection and reduce noise,
- API contracts can help analyzers reason about nullability, bounds, and ownership.

This is why annotation support is a serious safety investment for large C++ codebases.

20.1.4 What static analysis usually cannot catch (or catches unreliably)

1) Bugs that depend on runtime values, timing, and environment

Static analysis cannot reliably prove properties that depend on:

- real scheduling and thread interleavings,
- allocator layout and memory reuse,
- I/O timing and partial failures,
- OS signals and asynchronous events,
- hardware-specific behavior.

This is why **ThreadSanitizer** exists: dynamic execution is often the only practical way to expose real races in test runs.

2) Many forms of undefined behavior that are heavily data-dependent

Some UB can be detected statically in certain patterns, but many UB cases depend on values and paths. Sanitizers like UBSan are often the fastest way to catch the executed UB reliably.

3) Deep semantic correctness

Static analysis can tell you “this pointer may be null” or “this value may flow to a sink”. It cannot reliably tell you “this algorithm satisfies the business rule” or “this is the correct protocol state machine” unless the rule is encoded explicitly and is checkable.

4) Complete absence of false positives and false negatives

All real static analyzers must trade precision for scalability:

- **False positives** happen when the analyzer cannot precisely model constraints or environment.
- **False negatives** happen when the analyzer does not explore a path, lacks modeling of a library call, or the bug pattern is outside its check set.

The Clang Static Analyzer is explicitly path-sensitive and symbolic, but practical documents and discussions acknowledge precision/scalability limits and false positives as part of real-world usage.

20.1.5 Concrete examples: what the tool can and cannot infer

Example A: easy rule-based catch (swap-prone parameters)

```
#include <cstddef>
#include <cstring>

void copy_bytes(void* dst, const void* src, std::size_t n) {
  std::memcpy(dst, src, n);
}

int main() {
  char a[8]{};
  char b[8]{};
  copy_bytes(b, a, 8);    // correct
  copy_bytes(a, b, 8);    // also correct (but easy to swap in real APIs)
}
```

Many static checkers can warn about functions whose parameters are easily swapped, or about dangerous C-style APIs, because the pattern is recognizable and common.

Example B: path-sensitive bug (leak on one branch)

```
#include <cstdio>

FILE* open_file(const char* path) {
  FILE* f = std::fopen(path, "rb");
  if (!f) return nullptr;
```

```
  // Suppose more code here...
  bool ok = false; // imagine this depends on parsing a header

  if (!ok) {
    return nullptr; // leak: f not closed on this path
  }

  return f; // caller owns FILE*
}
```

A path-sensitive analyzer may catch the leak because it reasons about both branches. However, the result depends on whether the tool models `fopen/fclose` ownership correctly, and on how complex the missing "ok" computation is.

Example C: why runtime tools are still required (timing-dependent race)

```
#include <thread>

int g = 0;

void t1() { for (int i = 0; i < 100000; ++i) ++g; }
void t2() { for (int i = 0; i < 100000; ++i) ++g; }

int main() {
  std::thread a(t1), b(t2);
  a.join(); b.join();
}
```

A static analyzer may warn about unsynchronized global access in simple cases, but reliable race discovery in real systems typically requires dynamic detection under realistic schedules (for example, TSan).

20.1.6 Why static analysis sometimes “misses obvious bugs”

When a tool misses a bug, it is usually due to one of these reasons:

1) The bug is outside the configured check set

Rule-based tools only report enabled checks. Clang-tidy is explicitly a framework with many modular checks.

2) Build configuration mismatch

Templates, macros, and platform-specific code can change meaning drastically. If the analyzer does not see the same compile options, include paths, and defines as the real build, results degrade.

3) Incomplete modeling

If the tool does not model key library functions or custom allocators/synchronization correctly, it cannot reason precisely.

4) Path explosion and scalability limits

Path-sensitive analysis must limit exploration to remain fast. This can cause missed paths (false negatives) or conservative assumptions (false positives).

20.1.7 Practical guidance: how to get the best results

1) Run static analysis in CI with stable configuration

- Use the same compile commands as production builds (compile database is ideal).
- Pin tool versions in CI to avoid surprise rule changes.

2) Treat findings as triage, not as "truth"

- A warning is a lead: verify it.
- Fix high-confidence issues immediately (leaks, null deref, obvious lifetime mistakes).
- For complex findings, reduce the code until the warning becomes obviously right or obviously wrong.

3) Combine static analysis with sanitizers

- Static analysis finds likely issues without tests covering them.
- Sanitizers confirm real invalid behavior along executed paths.

Together, they catch far more than either alone.

4) Use annotations and stronger types to help tools

Annotations (where applicable) and strong C++ types (`span`, `string_view` with lifetime discipline, RAII wrappers, strong typedefs) reduce ambiguity, making both static and dynamic tools more effective.

20.1.8 Summary: the correct expectations

- Static analysis is excellent for **pattern-based bugs**, many **resource/lifetime mistakes**, and **security dataflow patterns**.
- Static analysis is weak for **timing-dependent** behavior, **deep semantic correctness**, and it cannot guarantee absence of bugs.
- The most professional approach is a **toolchain**: static analysis + sanitizers + tests + code review + strong API design.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): the language rules that define undefined behavior, object lifetime, and the concurrency model (what tools try to protect you from).
- LLVM/Clang Documentation: *Clang Static Analyzer* (path-sensitive, interprocedural, symbolic execution).
- LLVM/Clang Documentation: *clang-tidy* and its check catalog (rule-based and lightweight static checks).
- Microsoft Learn: C/C++ Code Analysis overview and warning documentation (typical defect classes and workflow).
- GitHub CodeQL Documentation: dataflow and taint tracking for C/C++ analysis.

20.2 Professional workflows

20.2.1 Goal: make static analysis routine, not heroic

In professional C++ teams, static analysis is most valuable when it is:

- **repeatable** (same tool versions, same configuration),
- **connected to the real build** (same flags, includes, defines),
- **actionable** (clear ownership of warnings, low noise),
- **continuous** (runs on every pull request and on a schedule),
- **tracked over time** (regressions blocked; backlog measured).

Static analysis is not one tool. It is a workflow that typically combines:

- **rule-based checkers** such as clang-tidy (many modular checks),
- **path-sensitive analyzers** such as the Clang Static Analyzer (path-sensitive, interprocedural, symbolic execution),
- **platform/vendor analyzers** such as MSVC Code Analysis and the C++ Core Guidelines checkers (CppCoreCheck),
- **security-focused semantic analysis** such as CodeQL (database + queries; supports CodeQL CLI and GitHub scanning).

20.2.2 Baseline workflow: five stages

Stage 1: Standardize toolchain and configuration

Professional teams pin:

- compiler (and version),
- analyzer tools (and version),
- configuration files (checked-in and code-reviewed).

For clang-tidy, the configuration is typically stored in a `.clang-tidy` file, and checks are explicitly selected from the catalog.
For MSVC code analysis, teams select rule sets and optionally enable Core Guidelines checks.

Stage 2: Connect analysis to the real build

A static analyzer is only as correct as the compilation context it sees. Professional practice is to drive analysis from the build system:

CMake: produce a compile commands database Many clang-based tools are designed to consume a compilation database (commonly `compile_commands.json`).

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

Then run tools using that database (typical pattern is a `-p` argument for the build directory).

Clang Static Analyzer: whole-project invocation The Clang Static Analyzer documentation describes whole-project usage via tools like `scan-build` and CodeChecker.

Stage 3: Define a policy (what is a failure?)

Static analysis becomes manageable when you define a policy:

- **Gate policy:** new warnings in changed code are not allowed.
- **Debt policy:** existing warnings are tracked and reduced over time.
- **Severity mapping:** security and correctness warnings are higher priority than style.

This policy is enforced in CI (blocking) and supported by local developer tooling (fast feedback).

Stage 4: Triage and ownership

A professional triage loop:

1. Confirm the warning is real (minimal reproducer if needed).
2. Fix it or annotate the reason it is safe.
3. If it is a false positive, prefer **narrow suppression** (file/function) over global suppression.
4. Assign ownership and a deadline for any temporary suppression.

Stage 5: Measure and improve

Track:

- number of warnings (by category),
- warnings in changed lines vs legacy code,
- time-to-fix for high-severity findings,
- suppression count (should decrease).

20.2.3 Workflow A: clang-tidy in daily development

clang-tidy is a clang-based lint and static analysis framework with many modular checks.

Recommended usage pattern

- Keep a project-wide `.clang-tidy` configuration.
- Start with a curated set of checks (high-signal first).
- Enable auto-fix only for safe transformations, and review the diff.
- Run clang-tidy:
  - locally on touched files before pushing,
  - in CI on the diff or on the whole project (depending on scale).

Typical invocation driven by a compilation database

```
# Build with compile_commands.json
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
cmake --build build

# Run clang-tidy against a file using the build directory as -p
clang-tidy -p build src/foo.cpp
```

Professional check selection

In safety-focused C++, many teams prioritize:

- `cppcoreguidelines-*` checks (Core Guidelines enforcement),
- `bugprone-*` checks (common correctness issues),
- `modernize-*` checks (safer modern idioms),
- carefully selected `performance-*` checks (avoid accidental slow paths).

Noise control rules

- Do not enable every check at once.
- Prefer "fix the code" over "silence the warning".
- When suppression is necessary, keep it local and documented.

20.2.4 Workflow B: Clang Static Analyzer for deeper bug finding

The Clang Static Analyzer is documented as **path-sensitive** and **interprocedural**, using **symbolic execution**.

When it is most valuable

- complex control-flow with early returns,
- resource-management code (handles, FILE*, sockets),
- error-handling paths and cleanup paths,
- code that is difficult to cover exhaustively with tests.

Whole-project runs

The official documentation discusses driving analysis using tools such as `scan-build` and CodeChecker.
A typical pattern:

- capture the compilation database,
- analyze using default or selected checker sets,
- consume results in a report format suitable for CI and review.

Professional practice: schedule and gate

Because deeper analysis can be slower:

- run a fast subset per PR,
- run full analyzer + CTU/taint configurations on a nightly schedule (when applicable),
- gate only high-confidence checkers in the fast lane; treat others as advisory until tuned.

20.2.5 Workflow C: MSVC Code Analysis and Core Guidelines checkers

Microsoft provides C/C++ code analysis (`/analyze`) and a set of checks aimed at enforcing the C++ Core Guidelines (CppCoreCheck / EspXEngine).

Professional usage pattern

- select a ruleset appropriate for the product,
- enable analysis on build in CI and/or as a dedicated pipeline stage,
- treat “new warnings” as failures,
- tune rulesets gradually rather than enabling all rules immediately.

Why vendor analysis matters

If you ship primarily on Windows, MSVC analysis can catch:

- SAL/annotation-driven contract violations (where used),
- Windows-specific API misuse patterns,
- Core Guidelines-related safety issues surfaced by the MSVC toolchain.

20.2.6 Workflow D: CodeQL for security and variant analysis

CodeQL analysis is based on creating a CodeQL database from the codebase and running query suites against it. GitHub documents CodeQL code scanning as a way to analyze code and surface results as code scanning alerts.

When CodeQL is the right tool

- when you want security-focused queries at scale,
- when you want to find **variants** of a vulnerability (same pattern across the codebase),
- when you need dataflow/taint-style reasoning across functions and files.

Professional usage pattern

1. Run CodeQL scanning in CI for pull requests and main branch merges.
2. Start with the default or security-extended suites, then add custom queries for your frameworks.
3. Treat high-severity alerts as blockers.
4. Use CodeQL results to drive targeted refactoring and to prevent reintroduction.

20.2.7 Putting it together: a practical CI blueprint

A common professional CI layout (static analysis only) looks like this:

Fast lane (every PR / every change)

- clang-tidy on changed files (curated checks; block regressions),
- MSVC `/analyze` (Windows lane, if Windows is a target),
- CodeQL scanning (security baseline; block high severity).

Deep lane (nightly / scheduled)

- Clang Static Analyzer whole-project scan, including heavier checker sets,
- CodeQL full suites + custom queries,
- “debt reduction” reports (trend lines, top categories, suppression audit).

20.2.8 How to keep the workflow stable and low-noise

Rule 1: treat build configuration as part of analysis correctness

Always run analysis against the same compilation flags used for real builds. If the analysis does not match the real build, treat results as unreliable.

Rule 2: prefer type- and API-level fixes

Fixing the same warning repeatedly is a design smell. If a warning class repeats, consider:

- replacing raw pointers with RAII,
- replacing `(ptr, size)` with `std::span`,
- introducing strong typedefs or `enum class` for parameters,
- tightening ownership and lifetime contracts.

Static analysis becomes more effective when the code expresses intent.

Rule 3: suppressions are temporary and audited

If you must suppress:

- keep it local (specific file/function),

- document the reason and add an owner,
- require removal as part of refactoring milestones.

Rule 4: train developers to read reports

A good workflow teaches engineers to:

- identify the core issue category (lifetime, bounds, concurrency, contract),
- reproduce locally with the same tool version,
- reduce the code until the cause is obvious,
- write a regression test or add a rule to prevent recurrence.

20.2.9 Minimal templates you can adapt

Local developer script pattern (clang-tidy)

```
#!/usr/bin/env bash
set -euo pipefail

cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
cmake --build build

# Run on a small set of touched files (example list)
clang-tidy -p build src/foo.cpp src/bar.cpp
```

CMake pattern for toggling analysis (conceptual)

```
option(ENABLE_CLANG_TIDY "Enable clang-tidy during build" OFF)
```

```
if (ENABLE_CLANG_TIDY)
  set(CMAKE_CXX_CLANG_TIDY
      clang-tidy;-p;${CMAKE_BINARY_DIR})
endif()
```

References (authoritative, no links)

- LLVM/Clang Documentation: clang-tidy (modular checks, configuration model).
- ISO C++ Core Guidelines: tool support notes for clang-tidy and Microsoft CppCoreCheck.
- LLVM/Clang Documentation: Clang Static Analyzer (path-sensitive, interprocedural, symbolic execution; project-level usage via scan-build/CodeChecker).
- Microsoft Learn: Code analysis for C/C++ overview and quickstart; Core Guidelines checker usage.
- GitHub Documentation: CodeQL code scanning overview and CodeQL CLI workflows (database creation and analysis).

Chapter 21

Modern Language Features for Safety

21.1 `std::span`

21.1.1 What `std::span` is

`std::span` (C++20, header `<span>`) is a **non-owning view** over a **contiguous sequence** of objects. It represents "pointer + length" as a single type, making APIs clearer and harder to misuse.
A `span`:

- does **not** allocate and does **not** own memory,
- can refer to memory in a built-in array, `std::array`, `std::vector`, and other contiguous storage,
- carries size information so you can iterate and index with less risk than raw pointer + separate length.

21.1.2 Why `std::span` is a safety feature

It removes the "split parameter" bug class

Passing a pointer and a size as separate parameters is easy to get wrong (swapped, mismatched, or computed in different units). A `span` binds them together:

```
#include <cstddef>

void process(const int* data, std::size_t n); // ambiguous: are we sure n
↪  matches data?

// safer
#include <span>
void process(std::span<const int> data);
```

This design makes many bugs harder to write and easier to review.

It supports safer iteration patterns

A `span` has `begin()/end()`, `size()`, and `data()` so you can write range-style code without losing bounds information.

21.1.3 Core concepts: extent, size, and representation

Dynamic extent vs static extent

A `span` can be:

- **dynamic extent**: `std::span<T>` where the length is stored at runtime,
- **static extent**: `std::span<T, N>` where the length is part of the type.

`std::dynamic_extent` is the special constant that means "dynamic size".

A key performance fact

A dynamic-extent `std::span<T>` is conceptually just “pointer + size” (no ownership), so it is typically lightweight to pass by value.

21.1.4 Construction: where spans come from

From built-in arrays and `std::array`

```
#include <array>
#include <span>

void f(std::span<const int> s);

int main() {
  int a[4] = {1,2,3,4};
  f(a); // OK: span over built-in array

  std::array<int, 3> b{10,20,30};
  f(b); // OK: span over std::array
}
```

From `std::vector` and other contiguous containers

```
#include <vector>
#include <span>

void g(std::span<int> s);

int main() {
  std::vector<int> v{1,2,3,4,5};
  g(v); // OK: span over vector's contiguous storage
```

```
}
```

From pointer + length (boundary use)

Sometimes you must accept legacy C interfaces. Convert to span immediately:

```
#include <span>
#include <cstddef>

void process(std::span<const unsigned char> bytes);

extern "C" void legacy(const unsigned char* p, std::size_t n) {
  if (!p && n != 0) return; // defensive: legacy may pass null with
  ↪  nonzero length
  process(std::span<const unsigned char>(p, n));
}
```

21.1.5 The most important rule: a span does not extend lifetime

Span is a view; lifetime must be managed elsewhere

Because `span` is non-owning, the referenced memory must outlive the span. This is the most common misuse pattern:

```
#include <span>
#include <vector>

std::span<int> bad() {
  std::vector<int> v{1,2,3};
  return v; // ERROR in design: returns a span to destroyed data
}
```

Professional rule:

Use `std::span` to *borrow* a contiguous range, never to *store* or *return* a view to short-lived memory.

Invalidation: when a span becomes dangling

For a `span s`, pointers/iterators/references into `s` become invalid whenever the underlying range `[s.data(), s.data() + s.size())` would be invalidated (for example, a vector reallocation).

```
#include <vector>
#include <span>

int main() {
  std::vector<int> v{1,2,3};
  std::span<int> s = v;

  v.push_back(4); // may reallocate -> s may now dangle (do not use s
  ↪  after this)
}
```

21.1.6 Indexing and bounds: what `span` does and does not do

Span carries bounds, but operator[] does not check

`std::span` makes bounds *available* (`size()`), but `operator[]` is not required to perform bounds checking.
Safety practice:

- prefer range-for loops and iterators,
- validate indices before indexing,
- use helper functions such as `subspan`, `first`, `last` to express slicing safely.

21.1.7 Slicing safely: `subspan`, `first`, `last`

These operations let you express “work on a prefix/suffix/window” without recomputing raw pointers:

```
#include <span>
#include <cstddef>

int sum(std::span<const int> s) {
  int r = 0;
  for (int x : s) r += x;
  return r;
}

int main() {
  int a[6] = {1,2,3,4,5,6};
  std::span<const int> s = a;

  auto first3 = s.first(3);
  auto last2  = s.last(2);
  auto mid    = s.subspan(2, 3); // elements [2,5)

  (void)sum(first3);
  (void)sum(last2);
  (void)sum(mid);
}
```

These slicing functions tend to produce clearer code than manual pointer arithmetic.

21.1.8 Working with bytes: `as_bytes` and `as_writable_bytes`

Sometimes you need a view of the **object representation** (byte-wise) for hashing, serialization, or I/O buffers.

```
#include <span>
#include <cstddef>
#include <cstdint>

struct Header {
  std::uint32_t magic;
  std::uint16_t version;
  std::uint16_t flags;
};

std::uint32_t checksum(std::span<const std::byte> bytes);

std::uint32_t header_checksum(const Header& h) {
  std::span<const Header, 1> one(&h, 1);
  return checksum(std::as_bytes(one)); // view as bytes
}
```

`std::as_bytes` and `std::as_writable_bytes` produce byte spans; `as_writable_bytes` only participates when the element type is not const.

21.1.9 API patterns that become safer with `span`

Pattern A: input-only buffer

```
#include <span>
#include <cstddef>

void parse(std::span<const std::byte> input);
```

Pattern B: output buffer provided by caller

```
#include <span>
```

```
#include <cstddef>

std::size_t compress(std::span<const std::byte> in,
                     std::span<std::byte> out);
```

Pattern C: “view in, own inside”

```
#include <span>
#include <vector>

class Packet {
public:
  explicit Packet(std::span<const std::byte> data)
    : bytes_(data.begin(), data.end()) {}

  std::span<const std::byte> view() const noexcept { return bytes_; }

private:
  std::vector<std::byte> bytes_; // owns storage
};
```

This keeps external interfaces safe while ensuring lifetime correctness.

21.1.10 Common mistakes and professional rules

Mistake 1: storing a span as a long-lived member without a lifetime contract

Storing a span is safe only when the pointed-to storage is guaranteed to outlive the object. If you cannot guarantee that, store an owning container instead (`std::vector`, `std::string`, etc.).

Mistake 2: using a span after modifying the underlying container

If a container operation can invalidate pointers (e.g., vector growth), assume the span is invalid. Recreate the span after such operations.

Mistake 3: confusing "safer API" with "automatic bounds checking"

`span` improves safety by making bounds explicit and APIs clearer. It does not automatically make out-of-bounds impossible. You still need:

- correct indexing logic,
- tests,
- sanitizers (ASan) for dynamic enforcement.

21.1.11 Checklist: adopting `std::span` safely

1. Use `std::span` for non-owning parameters that accept contiguous sequences.
2. Prefer `std::span<const T>` for inputs; use `std::span<T>` only when mutation is required.
3. Do not return spans to local or temporary storage.
4. Treat container reallocation as span invalidation; recreate spans after mutations.
5. Prefer slicing functions (`first/last/subspan`) over manual pointer arithmetic.
6. Use `as_bytes` / `as_writable_bytes` for byte-level views when needed.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): `std::span` and the requirements on contiguous sequences (C++20 and later).
- cppreference.com: `std::span` overview, invalidation notes, `<span>` header, `std::dynamic_extent`, and `std::as_bytes`/`std::as_writable_bytes`.
- Microsoft Learn: `span` class documentation and guidance for using `span` as a safer alternative to pointer+index for contiguous sequences.

21.2 `std::optional`

21.2.1 What `std::optional` is

`std::optional<T>` (since C++17, header `<optional>`) represents an object that **may or may not contain a value of type `T`**. It is a **value type** that stores its payload `T` directly (no ownership of external memory is implied by the type itself), and it models the concept of an "optional result" without using sentinel values.
An optional object is always in exactly one of two states:

- **engaged**: it contains a `T`,
- **disengaged**: it contains no value.

21.2.2 Why `std::optional` improves safety

It replaces dangerous sentinels

Many legacy APIs encode "no value" using:

- `-1` or other magic integers,
- null pointers (`T*` where null means absent),
- empty strings or special enum values.

These patterns are easy to misuse (forget to check, confuse valid data with sentinel). `std::optional` forces callers to acknowledge absence explicitly via `has_value()` or an equivalent check.

It makes intent visible in the type system

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

std::optional<std::string> find_user_display_name(int user_id);
```

This signature communicates: "there might be no name". A plain `std::string` return cannot express that without a convention.

21.2.3 Core operations and correct usage

Constructing an empty optional

Use default construction or `std::nullopt`:

```
#include <optional>

std::optional<int> a;                // empty (disengaged)
std::optional<int> b = std::nullopt; // empty
```

`std::nullopt` is the standard "empty" tag object for `std::optional`.

Constructing a value

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

std::optional<int> oi = 42;
std::optional<std::string> os{std::in_place, "hello"}; // construct
↪  in-place
```

`std::in_place` requests in-place construction of the contained `T`.

`has_value()` and `operator bool`

`has_value()` and `operator bool` report whether the optional is engaged.

```
#include <optional>
#include <iostream>

void use(std::optional<int> x) {
  if (x) { // same as x.has_value()
    std::cout << "value: " << *x << "\n";
  } else {
    std::cout << "no value\n";
  }
}
```

21.2.4 Access patterns: safe, safer, and dangerous

`operator*` and `operator->`

`*opt` and `opt->member` assume the optional is engaged. They do not perform a presence check for you; you must ensure the precondition.

`value()` throws on empty

`value()` returns a reference to the contained value if present; otherwise it throws `std::bad_optional_access`.

```
#include <optional>

int must_get(std::optional<int> x) {
  return x.value(); // throws std::bad_optional_access if empty
}
```

Prefer explicit handling: `value_or`

`value_or(default)` returns the contained value if present; otherwise it returns the provided default.

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

std::string name_or_unknown(std::optional<std::string> name) {
  return name.value_or("unknown");
}
```

Professional guidance:

- Use `value()` only when emptiness is a true logic error and throwing is acceptable.
- Use `value_or` when there is a meaningful default.
- Prefer returning `std::optional` (or `std::expected`) to force callers to handle failure explicitly.

21.2.5 Modifying state: `reset` and `emplace`

`reset()` disengages (destroys contained value if any)

`reset()` destroys the contained value if present and leaves the optional empty.

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

void clear(std::optional<std::string>& s) {
  s.reset();
}
```

`emplace()` constructs in-place

`emplace(args...)` constructs a new value in-place; if a value was already present, it is destroyed first.

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

void set_name(std::optional<std::string>& name, const char* p) {
  name.emplace(p); // in-place construct string from p
}
```

21.2.6 C++23 monadic operations: safer composition without nested `if`

C++23 adds monadic operations to `std::optional`: `and_then`, `transform`, and `or_else`.
These functions reduce error-prone manual unwrapping and make intent clearer.

`and_then`: chain functions that return `optional`

If a value is present, call `f(value)` and return its `optional` result; otherwise return empty.

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

std::optional<int> parse_int(const std::string& s);
std::optional<int> safe_div10(int x);

std::optional<int> pipeline(const std::optional<std::string>& input) {
  return input.and_then(parse_int)
              .and_then(safe_div10);
}
```

`transform`: map value to a new type

If a value is present, apply `f` and return an `optional` containing the result; otherwise return empty.

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

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

std::optional<std::size_t> name_length(int id) {
  return get_user_name(id).transform([](const std::string& s) {
    return s.size();
  });
}
```

`or_else`: provide a fallback optional

If a value exists, return it; otherwise call `f()` and return its optional result.

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

std::optional<std::string> read_from_cache();
std::optional<std::string> read_from_disk();

std::optional<std::string> load_config() {
  return read_from_cache().or_else(read_from_disk);
}
```

21.2.7 Design guidance: choosing `optional` vs alternatives

Use `std::optional<T>` when absence is normal

Good fits:

- lookup may fail (not found),
- parsing may fail (invalid),
- configuration value may be missing (not specified),
- optional parameter in API design.

Prefer `std::expected` (C++23) when you need error details

If you need to explain *why* something failed (error codes, messages), `std::expected<T, E>` is usually a better fit than `optional<T>`. (Use `optional` when you only need presence/absence.)

Avoid `std::optional<T&>` and use references deliberately

`std::optional` cannot hold a reference type directly in standard C++. If you need an optional reference, common patterns include:

- return a pointer (`T*`) where null means absent,
- use `std::optional<std::reference_wrapper<T>>` (with clear lifetime rules).

21.2.8 Common pitfalls and professional rules

Pitfall 1: calling `value()` on an empty optional

This throws `std::bad_optional_access`.
Professional rule:

> Do not use `value()` unless you have already proven presence in the same logical scope, or emptiness is a genuine logic error and exceptions are part of your error strategy.

Pitfall 2: assuming moved-from optionals become empty

After move, an object is valid but in an unspecified state unless specified otherwise. Do not write code that depends on a moved-from optional being empty; treat moved-from values as unspecified. (This is the general C++ rule for moved-from objects.)

Pitfall 3: using `optional` as a "nullable owning pointer"

`optional<std::unique_ptr<T>>` can be valid, but it is often redundant:

- `unique_ptr` already models "may be null",

- layering `optional` on top usually adds complexity without extra safety.

Prefer one clear representation of absence.

Pitfall 4: ignoring size/performance constraints

`optional<T>` stores a `T` plus bookkeeping for engagement state. This is usually cheap and stack-friendly, but in hot paths you should still:

- pass `optional` by value only when appropriate (it is typically small for small `T`),
- prefer `optional<T&>` alternatives for very large `T` (but enforce lifetimes carefully),
- measure if the code is performance critical.

21.2.9 Checklist: safe adoption of `std::optional`

1. Use `std::optional` instead of sentinel values to represent "maybe" results.
2. Prefer `if (opt)` or `has_value()` before dereferencing.
3. Prefer `value_or()` for defaults; avoid `value()` unless justified.
4. Use `reset()` to clear; use `emplace()` to construct in-place.
5. In C++23 code, prefer `and_then/transform/or_else` to avoid manual unwrapping.
6. If you need rich error information, prefer `std::expected` over `optional`.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): `std::optional` (`<optional>`), `std::nullopt`, `std::bad_optional_access`, and C++23 monadic operations.
- cppreference.com: `std::optional` overview and members (`has_value`/`operator bool`, `value`, `value_or`, `reset`, `emplace`, comparisons, and C++23 monadic operations).
- WG21 paper P0798R8: rationale and specification direction for monadic operations added in C++23.
- Microsoft Learn: `optional` class reference (behavior of `has_value`, `value`, `value_or`, `reset`).

21.3 `std::expected`

21.3.1 What `std::expected` is

`std::expected<T, E>` (since C++23, header `<expected>`) is a vocabulary type for **explicit, exception-less error handling**. It represents *either*:

- an **expected value** of type `T` (success), or
- an **unexpected error** of type `E` (failure).

An important guarantee is that `std::expected` is **never valueless**: it always contains either a value or an error.
This makes failure explicit in the type system:

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

std::expected<int, std::string> parse_port(std::string_view s);
```

21.3.2 Why `std::expected` is a safety feature

It forces callers to handle failure

Compared to sentinel returns or out-parameters, `std::expected` makes failure part of the function contract. This reduces:

- silently ignored error codes,
- partial initialization bugs,
- ambiguous “special values” like `-1` or empty strings.

It preserves error information

`std::optional<T>` can represent “maybe no value”, but it cannot explain *why* it is absent. `std::expected<T, E>` carries a structured error `E`.

21.3.3 Key associated types and tags

C++23 introduces a small set of related facilities in `<expected>`:

- `std::unexpected<E>`: a wrapper representing an error value,
- `std::unexpect` (and `std::unexpect_t`): a tag for constructing the error state,
- `std::bad_expected_access<E>` (and specialization `<void>`): exception type thrown by `expected::value()` when the object holds an error.

21.3.4 Core operations (the minimal set you must master)

Checking state: `has_value()` / `operator bool`

`has_value()` (and `operator bool`) tells you whether a value is present.

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

std::expected<int, std::string> get();

void use() {
  auto r = get();
  if (r) {
    std::cout << "ok: " << *r << "\n";
  } else {
    std::cout << "err: " << r.error() << "\n";
  }
}
```

Accessing the value: `*`, `->`, `value()`

- `*r` and `r->member` access the value when present (precondition: value exists).
- `r.value()` returns the value, but if an error is present it throws `std::bad_expected_access<E>` (or `<void>`).

Professional rule:

> Use `value()` only when throwing on failure is an acceptable policy. Otherwise, branch on `has_value()` and handle `error()` explicitly.

Accessing the error: `error()`

`r.error()` returns a reference to the error object `E` when the expected holds an error.

Providing defaults: `value_or`

`value_or(default)` returns the success value if present; otherwise returns the given default value.

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

std::expected<int, std::string> parse_int(std::string_view);

int parse_with_default(std::string_view s) {
  return parse_int(s).value_or(0);
}
```

21.3.5 Construction patterns (safe and idiomatic)

Return success by returning `T`

```
#include <expected>
#include <string_view>

std::expected<int, int> parse_digit(std::string_view s) {
  if (s.size() != 1 || s[0] < '0' || s[0] > '9')
    return std::unexpected(1); // error code 1
  return int(s[0] - '0');      // success
}
```

Return failure using `std::unexpected`

`std::unexpected<E>` is the standard wrapper for creating the error state.

In-place construction of value or error

For expensive types, in-place construction avoids extra moves/copies:

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

std::expected<std::string, std::string> make(bool ok) {
  if (ok)  return std::expected<std::string, std::string>(std::in_place,
  ↪  "READY");
  return std::expected<std::string, std::string>(std::unexpect, "failed
  ↪  to build");
}
```

21.3.6 Monadic operations (C++23): composing safely without nested `if`

C++23 includes monadic operations for `std::expected` such as `and_then`, `transform`, and `or_else`. These operations were standardized to make composition consistent with the similar C++23 optional monadic style.

`and_then`: chain functions that return `expected`

If a value exists, call the next function; otherwise propagate the error unchanged.

```
#include <expected>
#include <string_view>

enum class Err { empty, bad_digit };
```

```
std::expected<int, Err> parse_digit(std::string_view s) {
  if (s.empty()) return std::unexpected(Err::empty);
  if (s.size() != 1 || s[0] < '0' || s[0] > '9') return
  ↪  std::unexpected(Err::bad_digit);
  return int(s[0] - '0');
}

std::expected<int, Err> twice(int x) { return x * 2; }

std::expected<int, Err> pipeline(std::string_view s) {
  return parse_digit(s).and_then(twice);
}
```

`transform`: map the value, keep the same error

```
#include <expected>
#include <string_view>

enum class Err { bad };

std::expected<int, Err> parse(std::string_view);

std::expected<long, Err> as_long(std::string_view s) {
  return parse(s).transform([](int x) { return long(x); });
}
```

`or_else`: recover or translate errors

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

std::expected<int, std::string> from_cache();
std::expected<int, std::string> from_disk();
```

```
std::expected<int, std::string> load() {
  return from_cache().or_else([](const std::string&) {
    return from_disk();
  });
}
```

21.3.7 `expected<void, E>`: success without a value

C++23 provides a specialization for `expected<void, E>` to represent operations that either succeed (no payload) or fail with an error.

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

std::expected<void, std::string> write_config();

void run() {
  auto r = write_config();
  if (!r) {
    // No value to read, only the error:
    // r.error()
  }
}
```

21.3.8 Choosing the error type `E` (professional guidance)

The quality of `std::expected` usage depends heavily on `E`. Prefer:

- a small `enum class` for closed sets of errors,
- a small error struct (code + message + context) when you need more detail,

- a domain error type shared across modules when errors must propagate far.

Avoid:

- using plain integers without meaning (hard to maintain),
- using heavyweight error objects in hot paths without measuring.

21.3.9 Common misuse patterns and how to avoid them

Mistake 1: calling `value()` in exception-free code

`value()` throws `std::bad_expected_access` when an error is present. If your codebase forbids exceptions, prefer:

- `if (r) {...} else {...}`,
- monadic composition (`and_then/transform/or_else`),
- explicit propagation helpers.

Mistake 2: discarding the error

Do not convert to `bool` and ignore `error()`. If failure matters, the error is the point. If failure does not matter, you probably want `optional` (absence without explanation).

Mistake 3: mixing unrelated error types across layers

If every layer invents a new `E`, long-distance propagation becomes painful. Professional strategy:

- translate errors at module boundaries,
- or standardize on a shared error type for a subsystem.

Mistake 4: using `expected` where exceptions are the right tool

For failures that are truly exceptional, rare, and benefit from stack unwinding, exceptions can be simpler. For frequent, local, expected failures (parsing, validation, I/O that can fail), `expected` is often clearer and more predictable.

21.3.10 Practical API patterns (recommended)

Pattern A: parsing and validation

```
#include <expected>
#include <string_view>

enum class ParseErr { empty, invalid };

std::expected<int, ParseErr> parse_int(std::string_view s);
```

Pattern B: I/O with structured error

```
#include <expected>
#include <string>
#include <vector>
#include <cstdint>

struct IoErr {
  int code;
  std::string message;
};

std::expected<std::vector<std::uint8_t>, IoErr> read_file_bytes(const
↪  std::string& path);
```

Pattern C: boundary translation (error mapping)

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

enum class DomainErr { not_found, denied, unknown };

std::expected<int, std::string> low_level();

std::expected<int, DomainErr> api() {
  return low_level().transform([](int x) { return x; })
                    .or_else([](const std::string& msg) ->
                     ↪  std::expected<int, DomainErr> {
                      if (msg == "ENOENT") return
                       ↪  std::unexpected(DomainErr::not_found);
                      if (msg == "EACCES") return
                       ↪  std::unexpected(DomainErr::denied);
                      return std::unexpected(DomainErr::unknown);
                    });
}
```

21.3.11 Checklist: adopting `std::expected` safely

1. Use `std::expected<T, E>` when callers must handle failure and you want to preserve error information.

2. Prefer a meaningful `E` (`enum class` or small error struct).

3. In exception-free code, avoid `value()`; branch on state or use monadic operations.

4. Translate errors at module boundaries to keep public APIs stable.

5. Add tests for both success and failure paths; expected makes failure paths explicit and testable.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): `<expected>`, `std::expected`, `std::unexpected`, `std::unexpect`, and `std::bad_expected_access` (C++23).
- cppreference.com: `std::expected` overview, guarantee of never being valueless, specializations (including `expected<void, E>`), observers (`has_value`, `value`, `error`), and the `<expected>` header contents.
- WG21 papers: P0323 (standardization of `std::expected`) and P2505 (monadic functions for `std::expected` in C++23).
- Microsoft C++ Team Blog: motivation and guidance for using `std::expected` as exception-less error handling.

21.4 `constexpr` for correctness

21.4.1 Why `constexpr` matters for safety

`constexpr` is not primarily a performance feature. In safety-focused C++, its main value is **correctness by construction**:

- You can move logic from runtime to compile time, where it is checked deterministically.
- You can validate invariants early using `static_assert`.

- You can prevent invalid configurations from compiling at all.

The key rule is simple:

> A `constexpr` function *can* run at compile time. It *will* run at compile time only when the call is used in a constant-evaluated context and its inputs allow it.

21.4.2 Mental model: compile-time evaluation vs runtime evaluation

A `constexpr` function has two modes

A `constexpr` function is an ordinary function that is *also* eligible for constant evaluation. If you call it with non-constant inputs, it runs at runtime like any other function.

```
#include <cstdint>

constexpr std::uint32_t rotl(std::uint32_t x, unsigned r) {
  return (x << r) | (x >> (32u - r));
}

constexpr std::uint32_t A = rotl(0x12345678u, 8); // compile-time
↪  eligible

std::uint32_t runtime(std::uint32_t x, unsigned r) {
  return rotl(x, r); // same function, runtime call
}
```

If you need "must be compile time": use `consteval`

C++20 adds `consteval` to declare an **immediate function**: every potentially-evaluated call must produce a compile-time constant expression.

```cpp
#include <cstdint>

consteval std::uint32_t make_mask(unsigned bits) {
  return (bits == 32) ? 0xFFFFFFFFu : ((1u << bits) - 1u);
}

constexpr std::uint32_t M = make_mask(12); // OK
// auto x = make_mask(n);                  // ERROR if n is not a
↪  constant expression
```

Use `consteval` when a runtime call would be meaningless or unsafe.

21.4.3 How `constexpr` improves correctness

1) Compile-time invariants with `static_assert`

`static_assert` turns correctness requirements into compilation rules.

```cpp
#include <array>
#include <cstddef>

constexpr std::size_t required_alignment = 16;

template <class T>
constexpr bool ok_alignment() {
  return alignof(T) >= required_alignment;
}

struct Good { alignas(16) int x; };
struct Bad  { int x; };

static_assert(ok_alignment<Good>(), "Good must meet alignment
↪  requirement");
```

```
static_assert(!ok_alignment<Bad>(), "Bad should fail"); // demonstration
```

This prevents unsafe builds and configuration drift.

2) Safer configuration: reject invalid parameters early

A common safety pattern is to compute and validate parameters in constant evaluation.

```
#include <cstddef>

constexpr std::size_t checked_block_size(std::size_t n) {
  // Example policy: must be power of two and within bounds
  if (n == 0 || n > 4096) return 0;
  if ((n & (n - 1)) != 0) return 0;
  return n;
}

constexpr std::size_t Block = checked_block_size(1024);
static_assert(Block != 0, "Invalid block size");
```

This turns a runtime misconfiguration into a compile error.

3) Eliminate sentinel values and undefined states

`constexpr` often enables safer alternatives to ad-hoc runtime code:

- precomputed tables with verified size and content,
- verified encodings and bit layouts,
- validated protocol constants.

```cpp
#include <array>
#include <cstdint>
#include <cstddef>

constexpr std::array<std::uint8_t, 256> make_reverse_table() {
  std::array<std::uint8_t, 256> t{};
  for (std::size_t i = 0; i < t.size(); ++i) {
    std::uint8_t x = static_cast<std::uint8_t>(i);
    std::uint8_t r = 0;
    for (int b = 0; b < 8; ++b) { r = (r << 1) | (x & 1u); x >>= 1; }
    t[i] = r;
  }
  return t;
}

constexpr auto Reverse = make_reverse_table();
static_assert(Reverse[0b00000001] == 0b10000000);
```

The table is correct by construction, and the assertion guarantees at least one known property.

4) Safer parsing/validation in constant evaluation (design-time tests)

When inputs are compile-time strings (or compile-time data), parsing can become a compile-time check.

```cpp
#include <array>
#include <cstddef>

constexpr bool is_hex(char c) {
  return ('0' <= c && c <= '9') || ('a' <= c && c <= 'f') || ('A' <= c &&
  ↪  c <= 'F');
}
```

```
template <std::size_t N>
constexpr bool all_hex(const char (&s)[N]) {
  // Expects null-terminated string literal
  for (std::size_t i = 0; i + 1 < N; ++i)
    if (!is_hex(s[i])) return false;
  return true;
}

static_assert(all_hex("DEADBEEF"));
static_assert(!all_hex("DEAD BEEF")); // space is not hex
```

This is a powerful technique for compile-time protocol constants and identifiers.

21.4.4 Modern `constexpr`: what changed from C++11 to C++23

Restrictions were gradually relaxed

C++11 introduced `constexpr` with significant restrictions; later standards relaxed them. Today, many more constructs are permitted in constant evaluation than in early C++11, and the definition of constant expressions and eligibility is precisely specified.

C++20: more `constexpr` in the library, and constexpr-friendly containers

C++20 significantly expanded `constexpr` usage, including support for more library functionality and enabling more code to be constant-evaluated. A key enabling direction for library containers was described in WG21 work on "more `constexpr` containers".

C++20: `consteval` (immediate functions)

`consteval` provides a stronger guarantee than `constexpr`: calls must be compile-time. This is ideal for enforcing that safety checks cannot be bypassed at runtime.

C++23: richer functional-style composition in some vocabulary types

C++23 added "monadic" operations to types like `std::optional` and `std::expected`, which often compose well with constant evaluation for configuration and validation code. (Use these where they help readability; do not force the style.)

21.4.5 Correctness patterns you can standardize in a codebase

Pattern A: compile-time checked tags and IDs

```
#include <cstdint>

consteval std::uint32_t fourcc(char a, char b, char c, char d) {
  // Enforce ASCII range at compile time
  auto ok = [](unsigned char x) { return x >= 0x20 && x <= 0x7E; };
  if (!ok((unsigned char)a) || !ok((unsigned char)b) ||
      !ok((unsigned char)c) || !ok((unsigned char)d)) {
    // Force failure: no runtime fallback
    throw "Invalid FourCC"; // ill-formed in constant evaluation context
  }
  return (std::uint32_t)(unsigned char)a
       | ((std::uint32_t)(unsigned char)b << 8)
       | ((std::uint32_t)(unsigned char)c << 16)
       | ((std::uint32_t)(unsigned char)d << 24);
}

constexpr std::uint32_t TAG = fourcc('P','N','G',' ');
```

This ensures invalid IDs cannot compile.

Pattern B: compile-time validated array sizes

```
#include <array>
```

```cpp
#include <cstddef>

template <std::size_t N>
constexpr std::array<int, N> make_array() {
  static_assert(N > 0, "N must be positive");
  static_assert(N <= 1024, "N too large");
  std::array<int, N> a{};
  for (std::size_t i = 0; i < N; ++i) a[i] = (int)i;
  return a;
}

constexpr auto A = make_array<64>();
```

The size policy becomes a compile-time rule.

Pattern C: constexpr “unit tests” for pure functions

```cpp
#include <cstdint>

constexpr std::uint32_t hash32(std::uint32_t x) {
  x ^= x >> 16;
  x *= 0x7FEB352Du;
  x ^= x >> 15;
  x *= 0x846CA68Bu;
  x ^= x >> 16;
  return x;
}

static_assert(hash32(0u) != hash32(1u));
static_assert(hash32(123u) == hash32(123u));
```

For deterministic functions, these checks prevent accidental regressions.

21.4.6 What `constexpr` cannot guarantee

1) It does not automatically make code “safe”

A `constexpr` function can still be logically wrong. It only moves evaluation earlier and makes some violations detectable earlier.

2) It cannot replace runtime checks for runtime data

If input comes from files, networks, devices, or users, you still need runtime validation. `constexpr` is for what you can know at compile time.

3) Compile-time execution still follows language rules

Constant evaluation obeys the rules of constant expressions. Not every expression is eligible for constant evaluation.

21.4.7 Costs and pitfalls (professional guidance)

Pitfall 1: compile-time complexity

Large constexpr computations can slow compilation and increase memory usage.
Professional practice:

- use constexpr to validate and precompute only what brings real safety value,
- keep compile-time tables modest unless you have measured the impact,
- prefer small, local `static_assert` checks over heavy compile-time metaprograms.

Pitfall 2: confusing `constexpr` with `const`

`const` prevents mutation through that name. `constexpr` means eligible for constant evaluation and usable in constant-expression contexts.

Pitfall 3: using `constexpr` when you need `constinit`

If you want to guarantee **compile-time initialization** of a variable with static storage duration, the relevant tool is `constinit` (C++20), not `constexpr`. (`constinit` is about initialization; `constexpr` is about constant evaluation and constant expressions.)

Pitfall 4: relying on compile-time evaluation where it is not required

A `constexpr` function call is not guaranteed to be compile-time unless it is used in a constant-evaluated context. If runtime calls are unacceptable, use `consteval`.

21.4.8 Checklist: `constexpr` as a correctness tool

1. Mark pure, deterministic helpers `constexpr` so they can be checked at compile time.
2. Use `static_assert` to enforce invariants and policies (sizes, ranges, alignments).
3. Use `consteval` when runtime evaluation would be unsafe or meaningless.
4. Keep compile-time work proportional to safety value; measure build impact.
5. Combine constexpr correctness checks with runtime testing and sanitizers for full coverage.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): rules for constant expressions, constant evaluation, and the meaning of `constexpr`.
- cppreference.com: `constexpr` specifier and constant expression rules.
- cppreference.com: `consteval` (immediate functions) and its semantics.
- WG21 paper P1073R3: immediate functions (`consteval`) wording and model.
- WG21 paper P0784R7: "More constexpr containers" (direction enabling broader constant evaluation in library code).

21.5 Safer initialization patterns

21.5.1 Why initialization is a safety topic

Many serious C++ defects start as **initialization defects**:

- using an object before it has a valid value (used-before-set),
- constructing an object in an invalid state (broken invariants),
- global/static initialization ordering bugs (the static initialization order fiasco),
- partial initialization on error paths,
- thread-unsafe lazy initialization.

Professional safety work treats initialization as a **design contract**:

> An object should be usable immediately after construction, and it should never exist in a “half-valid” state.

The C++ Core Guidelines explicitly recommend always initializing objects to avoid used-before-set and maintainability problems.

21.5.2 Rule 1: Always initialize (and do it where the variable is defined)

Prefer initialization over assignment

Initialization builds the object into a valid state in one step. Assignment often means the object is temporarily invalid.

```
int x;         // uninitialized: unsafe
x = 42;        // assignment later

int y = 42;    // initialized: safe
```

This rule is strongly recommended by the C++ Core Guidelines (“Always initialize an object”).

For class members: prefer in-class member initializers

In-class member initializers make the default state explicit and reduce the risk of forgetting a member in one constructor.

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

class Config {
public:
  Config() = default;
```

```
  explicit Config(std::string name) : name_(std::move(name)) {}

private:
  std::string name_ = "default";
  int retries_ = 3;
  bool verbose_ = false;
  std::vector<int> ports_{}; // clearly empty
};
```

This pattern also improves correctness when constructors are added later.

21.5.3 Rule 2: Prefer braced initialization {} when it improves correctness

Braces reduce narrowing mistakes

List-initialization ({}) has special rules and is designed to prevent certain unsafe conversions (notably narrowing).

```
int a = 3.14;  // allowed (narrowing happens)
int b{3.14};   // error: narrowing (safer)
```

Prefer explicit intent for zero / empty initialization

When you mean “zero” or “empty”, make it visible:

```
int count{};                  // 0
double avg{};                 // 0.0
std::string s{};              // empty
std::vector<int> v{};         // empty
```

C++ initialization rules are subtle; {} provides a consistent, explicit form.

Know the key exception: initializer-list constructors

Braces may prefer `std::initializer_list` overloads, which can surprise you. Treat this as a readability issue and use parentheses when you want “count/value” style constructors.

```
#include <vector>

// vector has an initializer_list constructor
std::vector<int> a{10, 20}; // two elements: 10, 20
std::vector<int> b(10, 20); // ten elements, each 20
```

This is still safe when you know your intent, but it must be written clearly.

21.5.4 Rule 3: Use constructors to enforce invariants

Make invalid states unrepresentable

If a value must satisfy constraints, validate in the constructor and store only valid data.

```
#include <stdexcept>

class Port {
public:
  explicit Port(int p) : value_(p) {
    if (p < 1 || p > 65535) throw std::out_of_range("invalid port");
  }
  int value() const noexcept { return value_; }

private:
  int value_;
};
```

This prevents the “default-constructed but invalid” object problem.

Prefer delegating constructors to avoid duplicated initialization logic

```
#include <string>

class User {
public:
  User() : User("guest") {}                // delegate
  explicit User(std::string name) : name_(std::move(name)) {}

private:
  std::string name_;
};
```

Duplicated initialization code is a common source of inconsistent object states.

21.5.5 Rule 4: Prefer RAII and factory functions for owning objects

Avoid two-phase initialization with raw `new`

Two-phase initialization (allocate, then "remember to initialize fully") is fragile. Prefer `std::make_unique` (C++14+) and `std::make_shared` to create owning smart pointers in one expression. `std::make_unique` is specified as constructing `T` and wrapping it in a `std::unique_ptr`.

```
#include <memory>
#include <string>

struct Session {
  explicit Session(std::string host, int port);
};

auto s = std::make_unique<Session>("example.com", 443);
```

This improves safety by:

- preventing leaks on early returns/exceptions,
- making ownership explicit,
- ensuring the object is fully constructed before exposure.

Prefer “build then publish”

For shared objects used by multiple threads:

- build the object fully in a local scope,
- then publish it atomically (e.g., store into a shared pointer under a mutex).

This avoids exposing partially initialized state to other threads.

21.5.6 Rule 5: Represent “may be missing” explicitly

Many initialization mistakes come from pretending something always exists.

Use `std::optional` when absence is normal

`std::optional<T>` models “value or no value” explicitly (C++17).

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

std::optional<std::string> read_env(const char* name);

void configure() {
  auto v = read_env("LOG_PATH");
```

```
  if (v) {
    // use *v
  } else {
    // fallback
  }
}
```

Use `std::expected` when you need error details

`std::expected<T,E>` (C++23) models "value or error" and is never valueless.

```
#include <expected>
#include <string_view>

enum class ParseErr { empty, invalid };

std::expected<int, ParseErr> parse_port(std::string_view);
```

These types prevent silent failure paths and make initialization outcomes explicit.

21.5.7 Rule 6: Make static and global initialization safe

Understand the static initialization order fiasco

Across translation units, the order of dynamic initialization of non-local objects is not reliable. This is the classic static initialization order fiasco.

Prefer "construct on first use" for non-trivial globals

The recommended idiom is a function-local static:

```
#include <string>
```

```cpp
const std::string& global_name() {
  static const std::string s = "service"; // initialized on first call
  return s;
}
```

This avoids cross-TU initialization order issues and is a standard mitigation.

Thread-safe initialization of function-local statics (C++11+)

Block-scope variables with static storage duration are initialized the first time control passes through their declaration. In C++11 and later, this initialization is required to be thread-safe; Microsoft documents this behavior and notes re-attempt after an exception during initialization.

Prefer `constinit` for globals that must be initialized at compile time

`constinit` (C++20) enforces that a variable has no dynamic initialization; if it would, the program is ill-formed.

```cpp
#include <cstddef>

constinit int g_limit = 256; // must be constant-initialized (no dynamic
↪  init allowed)
```

Use this when:

- you want to avoid static initialization order problems,
- you want to guarantee initialization is completed before runtime begins.

21.5.8 Rule 7: Prefer simple, explicit lazy initialization patterns

Prefer function-local static over `std::call_once` when possible

`std::call_once` and `std::once_flag` provide one-time initialization. However, cppreference notes that initialization of function-local statics is guaranteed to occur only once even with multiple threads and *may be more efficient* than an equivalent `call_once` pattern.

```
#include <mutex>
#include <string>

std::string& config_path_call_once() {
  static std::once_flag flag;
  static std::string path;

  std::call_once(flag, []{
    path = "/etc/service.conf";
  });
  return path;
}
```

Use `call_once` when you need:

- explicit control over the initialization procedure across multiple functions,
- or to coordinate initialization that cannot be expressed as a single local-static initializer.

21.5.9 Rule 8: Use designated initializers for aggregates when it reduces mistakes (C++20)

For aggregate types, designated initializers can reduce errors from positional initialization and document intent. C++20 designated initializers are only for aggregates, and designators must appear in member order.

```
struct Limits { int min; int max; int step; };

Limits lim{ .min = 0, .max = 100, .step = 5 }; // clear and less
↪  error-prone
```

This is especially helpful when an aggregate has many fields or similar field types.

21.5.10 A compact professional checklist

1. Initialize every object at definition (Core Guidelines).

2. Prefer in-class member initializers for stable defaults.

3. Prefer braces when they prevent narrowing and improve clarity; use parentheses when you need to avoid initializer-list ambiguity.

4. Enforce invariants in constructors; avoid invalid default states.

5. Prefer RAII and factory functions (`make_unique`, `make_shared`) over raw `new`.

6. Represent absence explicitly (`optional`) and failures explicitly (`expected`).

7. Avoid fragile global initialization: use construct-on-first-use or `constinit` where appropriate.

8. Prefer function-local statics for lazy singletons; consider `call_once` only when it is the clearer tool.

9. Use designated initializers for aggregates to avoid positional field mistakes (C++20).

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): initialization rules, constant initialization, thread-safe initialization of block-scope statics (C++11+), `constinit` (C++20), and designated initializers for aggregates (C++20).
- C++ Core Guidelines: ES rules recommending always-initialize and safer declaration/initialization practices.
- cppreference.com: initialization and list-initialization rules; static initialization order fiasco and construct-on-first-use; `constinit`; `std::call_once`/`std::once_flag`; aggregate/designated initialization; `std::make_unique`.
- Microsoft Learn: C++11 thread-safe initialization semantics for local statics (`/Zc:threadSafeInit` documentation describing the standard behavior).

Part VII

Professional Engineering Practices

Chapter 22

Designing Safe APIs

22.1 Expressing ownership in interfaces

22.1.1 Why ownership belongs in the type, not in comments

Most lifetime bugs happen at *API boundaries*:

- the callee assumes the argument stays alive, but the caller passes a temporary,
- the caller assumes the callee takes ownership, but the callee only borrows,
- both sides assume the other one frees the resource.

A professional C++ API makes ownership **explicit in the signature** using well-known vocabulary types (`unique_ptr`, `shared_ptr`, `span`, `string_view`, `optional`, `expected`). The C++ Core Guidelines explicitly treat smart pointers as the primary way to represent ownership.

22.1.2 Ownership vocabulary: four common intent categories

1) The callee does not own (borrows, non-owning access)

Use:

- `T&` / `const T&` for mandatory objects that must exist,
- `T*` for optional objects (nullable) *without ownership*,
- `std::span<T>` for contiguous ranges (pointer + size as one unit),
- `std::string_view` for read-only string views (same idea for strings).

2) The callee takes ownership exclusively

Use `std::unique_ptr<T>` to express exclusive ownership transfer.

3) The callee shares ownership

Use `std::shared_ptr<T>` to express shared ownership (reference-counted lifetime).

4) The callee returns ownership

Return:

- `std::unique_ptr<T>` for exclusive ownership,
- `std::shared_ptr<T>` for shared ownership.

22.1.3 Rule set: parameter types that communicate ownership

Borrowing: prefer references or views

Borrow a single object If the function requires a valid object and does not store the pointer beyond the call:

```
struct Widget { void do_work() const; };

void use(const Widget& w) { // borrower, non-null by construction
  w.do_work();
}
```

Use `T*` when the parameter is optional:

```
void maybe_use(const Widget* w) { // borrower, nullable, non-owning
  if (w) w->do_work();
}
```

Borrow a range: use `std::span` `std::span` is a non-owning view of a contiguous sequence and carries size together with the pointer, which removes an entire bug class from APIs.

```
#include <span>
#include <cstddef>

int sum(std::span<const int> s) {
  int r = 0;
  for (int x : s) r += x;
  return r;
}
```

Important lifetime rule: a `span` does not extend lifetime; it is invalidated when the underlying storage is invalidated.

Taking exclusive ownership: accept `std::unique_ptr<T>` by value

`std::unique_ptr` owns and disposes its managed object when it goes out of scope. It is move-only; move assignment transfers ownership.
When the callee should assume ownership, the clearest signature is by value:

```
#include <memory>

struct Job { virtual ~Job() = default; virtual void run() = 0; };

class Scheduler {
public:
  void submit(std::unique_ptr<Job> j) { // takes ownership
    jobs_.push_back(std::move(j));
  }
private:
  std::vector<std::unique_ptr<Job>> jobs_;
};
```

Caller code makes transfer explicit:

```
auto j = std::make_unique<MyJob>();
sched.submit(std::move(j)); // j becomes empty after move
```

This matches the meaning of `unique_ptr` ownership transfer in its specification and operations.

Do not accept `unique_ptr<T>&` to "take ownership" Taking `unique_ptr<T>&` often hides the transfer and complicates call sites. Prefer by-value when ownership is transferred, and prefer borrowing `T&` or `T*` when it is not.

Sharing ownership: accept `std::shared_ptr<T>` by value only when you truly share

`shared_ptr` retains shared ownership; the object is destroyed when the last owning `shared_ptr` is destroyed or reset. Copying a `shared_ptr` participates in that ownership sharing.
Signature meaning:

- `shared_ptr<T>` parameter by value: callee **shares** ownership.
- `const shared_ptr<T>&`: callee **observes** shared ownership without extending it.

```
#include <memory>

struct Service { void handle(); };

void run_async(std::shared_ptr<Service> svc);          // shares ownership
↪  (extends lifetime)
void run_sync(const std::shared_ptr<Service>& svc);    // observes; does
↪  not extend lifetime
```

Do not use `use_count()` for correctness `shared_ptr::use_count()` is explicitly documented as approximate and unsafe for correctness decisions in multithreaded code.

22.1.4 Return types: how to communicate ownership back to the caller

Return `std::unique_ptr<T>` for exclusive ownership

```
#include <memory>

struct Base { virtual ~Base() = default; };

std::unique_ptr<Base> make_object();
```

This means: caller owns the returned object exclusively. That aligns with the definition of `unique_ptr` as an owning smart pointer.

Return `std::shared_ptr<T>` only when multiple owners are required

```
#include <memory>

std::shared_ptr<const Config> global_config(); // shared, long-lived
```

Be mindful that `shared_ptr` supports advanced behaviors such as the aliasing constructor, where the stored pointer differs from the managed pointer while sharing ownership control. This is useful but can confuse API consumers if not documented.

22.1.5 Factory functions: build-then-publish is safer than partially-initialized objects

A safe initialization pattern is:

1. fully construct a valid object,
2. then publish ownership via the return type.

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

struct Error { int code; std::string msg; };

class Database {
public:
  static std::expected<std::unique_ptr<Database>, Error> open(std::string
  ↪  conn);
```

```
  // ...
};
```

This avoids exposing “half-valid” objects and makes failure explicit.

22.1.6 Where raw pointers still appear (and how to keep them safe)

Borrowing raw pointers is fine if the rule is clear

Raw pointers are appropriate for:

- optional parameters (nullable borrower),
- interacting with C APIs,
- non-owning graph relationships (observer edges).

If you must express owning raw pointers, the Core Guidelines discuss annotation helpers such as `owner<T*>` as documentation aids, but prefer real RAII ownership whenever possible.

Never return owning raw pointers from modern C++ APIs

Returning `T*` without ownership semantics invites leaks and double frees. Return `unique_ptr` or `shared_ptr` instead.

22.1.7 Polymorphism and ownership: the destructor rule

If you own derived objects through a base pointer, the base must have a virtual destructor. `unique_ptr<Base>` and `shared_ptr<Base>` will delete through `Base*`. Without a virtual destructor, that is undefined behavior.

```cpp
#include <memory>

struct Base {
  virtual ~Base() = default; // required for polymorphic deletion
};

struct Derived : Base { ~Derived() override = default; };

std::unique_ptr<Base> make() { return std::make_unique<Derived>(); }
```

22.1.8 A compact decision table you can apply consistently

Parameters (incoming)

- **Must exist, borrowed:** `T&` / `const T&`
- **Optional, borrowed:** `T*` / `const T*`
- **Borrow a contiguous range:** `std::span<T>`
- **Take exclusive ownership:** `std::unique_ptr<T>` (by value)
- **Share ownership:** `std::shared_ptr<T>` (by value)

Returns (outgoing)

- **Return owned object:** `std::unique_ptr<T>` or `std::shared_ptr<T>`
- **Return optional result:** `std::optional<T>`
- **Return value-or-error:** `std::expected<T,E>`

22.1.9 Checklist: professional ownership clarity

1. Every pointer-like parameter must have a clear ownership meaning.

2. Use `unique_ptr` to represent exclusive ownership; take it by value to transfer ownership.

3. Use `shared_ptr` only for real shared ownership; do not use `use_count()` for correctness.

4. Use views (`span`, `string_view`) for non-owning range/string parameters; do not store them without a lifetime contract.

5. Prefer returning ownership (smart pointers) over returning raw owning pointers.

6. For polymorphic ownership, ensure a virtual destructor in the base type.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): object lifetime, smart pointers, and destruction rules for polymorphism.

- ISO C++ Core Guidelines: resource management and expressing ownership using smart pointers.

- cppreference.com: `std::unique_ptr` ownership semantics and move transfer; `std::shared_ptr` shared ownership and `use_count` caveats; `std::span` as a non-owning contiguous range view and invalidation rules.

22.2 Minimizing misuse

22.2.1 Why “misuse resistance” is an API requirement

A safe API is not only correct when used perfectly. It is also:

- difficult to call incorrectly,
- hard to misunderstand during code review,
- explicit about preconditions and lifetime rules,
- friendly to static analysis and sanitizers,
- stable under refactoring.

The C++ Core Guidelines emphasize that interfaces are a contract between caller and callee, and that making expectations precise is essential to avoid error-prone usage.

22.2.2 Principle 1: Put the contract in the type system

Express ownership and lifetime directly

Prefer vocabulary types that encode intent:

- **borrowing a range:** `std::span<T>` (non-owning view of contiguous elements),
- **borrowing a string:** `std::string_view` (non-owning view; lifetime must outlive the view),
- **optional value:** `std::optional<T>`,
- **value-or-error:** `std::expected<T,E>` (C++23),

- **ownership transfer:** `std::unique_ptr<T>`, **shared ownership:** `std::shared_ptr<T>`.

Also follow the guideline: *never transfer ownership via a raw pointer*. Use smart pointers (or clear ownership annotations) so that ownership is explicit and tool-checkable.

Avoid "pointer + length" split parameters

```
void hash(const unsigned char* p, std::size_t n); // easy to misuse

#include <span>
#include <cstddef>
#include <cstdint>

void hash(std::span<const std::byte> bytes); // size travels with the
↪  pointer
```

`std::span` describes a view over a contiguous sequence, and invalidation follows the underlying storage rules.

Treat `std::string_view` as a borrowing type, not a storage type

`string_view` is efficient for parameters, but it is easy to misuse if stored beyond the lifetime of the original string. The standard library documentation explicitly states that the programmer must ensure the view does not outlive the referenced character array.

```
#include <string>
#include <string_view>

struct BadStore {
  std::string_view v; // can dangle
};
```

```
struct GoodStore {
  std::string s; // owns the characters
};
```

22.2.3 Principle 2: Make invalid states unrepresentable

Strong types instead of primitive obsession

If a value has constraints or domain meaning, encode that in a type. This prevents swapping parameters accidentally and centralizes validation.

```
#include <stdexcept>

class Port {
public:
  explicit Port(int p) : value_(p) {
    if (p < 1 || p > 65535) throw std::out_of_range("invalid port");
  }
  int get() const noexcept { return value_; }
private:
  int value_;
};

class TimeoutMs {
public:
  explicit TimeoutMs(int ms) : value_(ms) {
    if (ms < 0) throw std::out_of_range("invalid timeout");
  }
  int get() const noexcept { return value_; }
private:
  int value_;
```

```
};

void connect(Port port, TimeoutMs timeout);
```

Now `connect(TimeoutMs{...}, Port{...})` is a compile error.

Prefer `enum class` for closed sets of options

Use scoped enums so you do not silently mix unrelated integers and flags.

```
enum class Mode { read_only, read_write };
enum class Format { json, binary };

void load(Mode m, Format f);
```

22.2.4 Principle 3: Remove ambiguous or dangerous call patterns

Avoid boolean parameters in public APIs

Booleans do not self-document at call sites.

```
void save(bool force, bool atomic); // unclear

enum class SaveMode { normal, force };
enum class Atomicity { nonatomic, atomic };

void save(SaveMode mode, Atomicity atomicity); // self-documenting
```

Prefer “options object” over long parameter lists

Long lists encourage mis-ordering and accidental defaults.

```
#include <string>

struct UploadOptions {
  bool compress = true;
  bool verify_checksum = true;
  int retry_count = 3;
};

void upload(std::string path, UploadOptions opt);
```

This reduces misuse while keeping backward compatibility when new options are added.

Use `explicit` to prevent unintended conversions

Converting constructors are a common misuse vector. Make them explicit unless implicit conversion is clearly safe and intended.

```
#include <string>

class UserId {
public:
  explicit UserId(std::string s) : value_(std::move(s)) {}
private:
  std::string value_;
};
```

22.2.5 Principle 4: Make ignoring results difficult

Use `[[nodiscard]]` for results that must be checked

`[[nodiscard]]` (since C++17) lets the compiler warn when a return value is discarded.

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

enum class Err { invalid, io };

[[nodiscard]] std::expected<int, Err> parse_and_open(std::string_view
↪  path);
```

Guidance:

- mark functions returning `expected`/error codes as `[[nodiscard]]`,
- consider `[[nodiscard("reason")]]` to explain what the caller must do (supported since C++20).

22.2.6 Principle 5: Enforce non-null and bounds expectations

Prefer references when null is not a valid state

If null is never valid, prefer `T&` / `const T&` for parameters.

Use `not_null` for non-null pointers (when a pointer is required)

The Core Guidelines include the rule: declare a pointer that must not be null as `not_null`. If you adopt the Guidelines Support Library (GSL) (or a compatible subset), you can document and enforce this at the API boundary.

```
#include <gsl/gsl> // or a GSL-compatible implementation

void process(gsl::not_null<const char*> p);
```

Avoid raw “array as pointer” interfaces

Passing arrays as a single pointer loses size information and is easy to misuse. The Core Guidelines warn against this pattern. Prefer `std::span` to keep bounds attached to the data.

22.2.7 Principle 6: Make preconditions explicit and checkable

Document preconditions in the signature and enforce at runtime

Use:

- type choices to encode constraints (strong types, spans, not_null),
- `assert`/exceptions/returned errors for runtime validation,
- `static_assert` for compile-time constraints.

```
#include <span>
#include <cassert>

int at(std::span<const int> s, std::size_t i) {
  assert(i < s.size()); // precondition
  return s[i];
}
```

Know the direction of standard Contracts (C++26 work)

WG21 continues work on a Contracts facility aimed at a minimal viable product for C++26, intended to make preconditions/postconditions more explicit and tool-friendly. Even before a standardized feature is widely available, design your interfaces so contract checks can be added without changing call sites.

22.2.8 Principle 7: Reduce error-prone conversions

Prefer brace initialization to catch narrowing

`T{...}` prevents many narrowing conversions that silently lose information.

```
int safe_int(double x) {
  // int y = x;   // narrowing, allowed
  int y{x};       // error if narrowing
  return y;
}
```

Use named narrowing helpers when policy requires it

The Guidelines ecosystem commonly uses `gsl::narrow` / `gsl::narrow_cast` to make narrowing explicit and reviewable.

22.2.9 A practical review checklist for "misuse-minimized" APIs

1. Can a caller misunderstand ownership or lifetime from the signature alone? If yes, switch to vocabulary types (smart pointers, `span`, `string_view`) and document lifetime constraints.

2. Do pointer parameters allow null when null is invalid? Prefer references or `not_null`.

3. Are there split parameters (`T*` + length), unscoped enums, magic integers, or boolean flags? Replace with `span`, `enum class`, and option types.

4. Can callers ignore errors without noticing? Use `[[nodiscard]]` and return `expected`/`optional` where appropriate.

5. Are preconditions visible, testable, and enforceable? Encode constraints in types; add checks at boundaries; keep contracts stable.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): language rules for references/pointers, initialization, attributes (including `[[nodiscard]]`), and standard library vocabulary types.
- C++ Core Guidelines: Interfaces (contract-focused API design) and ownership rules (avoid ownership transfer via raw pointers; prefer explicit ownership vocabulary).
- cppreference.com: `std::span` semantics and invalidation notes; `std::string_view` lifetime responsibilities; `[[nodiscard]]` attribute definition and evolution.
- WG21: Contracts rationale and safety discussion (C++26 direction and interaction with safety and analysis).
- GSL ecosystem (Guidelines Support Library compatible implementations) to express non-null and narrowing policies in code.

22.3 Fail-fast design

22.3.1 What “fail-fast” means in C++ safety engineering

Fail-fast means: **detect violations of the program’s contract as early as possible and stop the current operation immediately** (often by terminating in debug/test builds, or by returning a structured error at API boundaries).
Fail-fast is not the same as “never handle errors”. The professional goal is:

- **Fail fast on programmer bugs** (violated preconditions, broken invariants, impossible states).
- **Handle recoverable failures explicitly** (I/O failure, invalid user input, network timeouts).

This separation aligns with the Core Guidelines contract mindset (preconditions/postconditions, and avoiding ambiguous interfaces).

22.3.2 A decision rule: bug vs recoverable failure

Fail-fast candidates (programmer bugs)

Use fail-fast for:

- violated preconditions: null where forbidden, out-of-range index, wrong state machine step,
- violated invariants: “size must match buffer length”, “mutex must be held”,
- UB-adjacent situations: deleting through non-virtual base, data races, lifetime bugs,
- internal logic contradictions: “this branch is unreachable”.

These are best treated as **defects**, not “errors to be handled”.

Recoverable failures (expected in production)

Return structured failures for:

- invalid external input (files, CLI, network packets),
- resource acquisition failure (cannot open file, cannot allocate, permission denied),

- timeouts and transient network problems,
- user mistakes (bad configuration).

A robust API often uses `std::expected<T,E>` (C++23) to express recoverable failure explicitly.

22.3.3 Fail-fast tools in modern C++

1) Compile-time fail-fast: `static_assert`, `constexpr`, `consteval`

If a property can be checked at compile time, do it there. `static_assert` makes the program ill-formed if the condition is false.

```
#include <type_traits>
#include <cstddef>

template <class T>
struct PacketView {
  static_assert(std::is_trivially_copyable_v<T>,
                "PacketView requires trivially copyable payload");
  const T* p;
  std::size_t n;
};
```

When runtime evaluation must be forbidden, use `consteval` (immediate functions): calls must be constant-evaluated.

```
#include <cstdint>

consteval std::uint32_t make_tag(char a, char b, char c, char d) {
  return (std::uint32_t)(unsigned char)a
```

```
        | ((std::uint32_t)(unsigned char)b << 8)
        | ((std::uint32_t)(unsigned char)c << 16)
        | ((std::uint32_t)(unsigned char)d << 24);
}

constexpr auto TAG = make_tag('P','K','T','1'); // compile-time only
```

2) Debug-time fail-fast: `assert`

`assert(expr)` is a macro intended for catching contract violations. If `NDEBUG` is defined at the point `<cassert>` is included, `assert` does nothing; otherwise, a failing `assert` prints diagnostics and calls `std::abort`.

```
#include <cassert>
#include <span>

int at(std::span<const int> s, std::size_t i) {
  assert(i < s.size());     // catches programmer misuse in debug/test
  return s[i];
}
```

Professional guidance:

- Use `assert` for **internal invariants** and **preconditions** that should never be violated by correct code.
- Do not use `assert` as your only defense against **hostile/untrusted input**, because it may compile out.

3) Production fail-fast: `std::terminate` / `std::abort`

`std::terminate` calls the currently installed `terminate_handler`; the default handler calls `std::abort`.

```
#include <exception>

[[noreturn]] void fail_fast() noexcept {
  std::terminate();
}
```

Use production termination for violations that imply corrupted state or undefined behavior risk. It is often safer to stop than to continue with potentially compromised invariants.

4) Enforced contracts via GSL-style `Expects`/`Ensures`

The Core Guidelines define a contract style with preconditions and postconditions, and the Guidelines Support Library (GSL) provides facilities used by the Guidelines.
Many GSL implementations treat `Expects`/`Ensures` as **fail-fast** checks (commonly terminating on violation), which is consistent with the Guidelines intent for contract violation detection.

```
#include <gsl/gsl>

int divide(int a, int b) {
  Expects(b != 0);      // precondition: caller must not pass zero
  return a / b;
}
```

Professional guidance:

- Use contract checks at **API boundaries inside your system** where misuse must be detected immediately.
- Ensure your build and diagnostics pipeline makes fail-fast failures actionable (stack traces, crash dumps, logs).

5) `noexcept` as a fail-fast boundary

A key fail-fast mechanism is `noexcept`. If an exception escapes a `noexcept` function, the runtime calls `std::terminate`.

```
#include <stdexcept>

void commit() noexcept {
  // If something throws past this boundary -> terminate
}
```

Use `noexcept` to mark **non-recoverable boundaries** where propagation is forbidden:

- destructors that must not throw,
- low-level cleanup paths,
- error-reporting code where throwing would cause recursion or loss of diagnostics.

But do not misuse `noexcept`:

> Mark a function `noexcept` only when you can actually uphold the contract, or when termination on violation is the intended design.

22.3.4 Design patterns that minimize misuse and amplify fail-fast

Pattern A: Validate at the boundary, assume inside

Validate untrusted input once, then rely on invariants internally.

```
#include <expected>
#include <string_view>
```

```
enum class ParseErr { empty, non_digit };

std::expected<int, ParseErr> parse_digit(std::string_view s) {
  if (s.empty()) return std::unexpected(ParseErr::empty);
  if (s.size() != 1 || s[0] < '0' || s[0] > '9')
    return std::unexpected(ParseErr::non_digit);
  return int(s[0] - '0');
}

int internal_use(int d) {
  // internal contract: d in [0,9]
  // fail-fast here is reasonable, because callers should have validated
  return d * 7;
}
```

Pattern B: Make invalid states unrepresentable

If a parameter must satisfy constraints, wrap it in a type that enforces them.

```
#include <stdexcept>

class Port {
public:
  explicit Port(int p) : v(p) {
    if (p < 1 || p > 65535) throw std::out_of_range("invalid port");
  }
  int get() const noexcept { return v; }
private:
  int v;
};

void connect(Port p);
```

Now many “misuses” become compile errors or immediate failures at construction.

Pattern C: Prefer `span` to keep bounds attached

This reduces misuse and makes range contracts explicit.

```
#include <span>
#include <cstddef>

void fill(std::span<unsigned char> out, unsigned char value) {
  for (auto& b : out) b = value;
}
```

22.3.5 Fail-fast and debugging: making termination useful

Termination is only productive if you can diagnose it. Make sure your engineering workflow captures:

- stack traces (debug symbols in test builds),
- core dumps / crash dumps,
- logs that include the violated contract and context.

For `assert`, the standard requires diagnostics to include the expression text and file/line info (and `__func__` since C++11) before abort.

22.3.6 Contracts in the language: direction (C++26 work)

WG21 is actively standardizing a Contracts facility aimed at C++26, with a central proposal (P2900) and rationale documents discussing design tradeoffs (assertion kinds, evaluation, violation handling).
Design your APIs so that:

- preconditions/postconditions are explicit and local,
- checks can be introduced or strengthened without changing call sites,
- contract violations are treated as bugs (fail-fast) rather than recoverable errors.

22.3.7 Checklist: a practical fail-fast policy for safe APIs

1. **Define what is a bug vs a recoverable error** (and encode it in types: `expected` for recoverable, contracts/assertions for bugs).
2. **Fail at compile time** when possible using `static_assert` and `consteval`.
3. **Use `assert` for internal contracts** and remember it may compile out under `NDEBUG`.
4. **Use `std::terminate`/`std::abort`** when continuing risks UB or corrupted state.
5. **Use `noexcept` as a deliberate fail-fast boundary**; violations lead to `std::terminate`.
6. **Prefer contract-style checks** (`Expects`/`Ensures`) where you want consistent fail-fast behavior across builds.
7. **Make failures diagnosable**: require stack traces and crash artifacts in test/CI.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): `noexcept` semantics and termination on violation; exception handling and program termination rules.

- cppreference.com: `assert` (interaction with `NDEBUG`, abort on failure, required diagnostics); `std::terminate` and `std::terminate_handler`; `std::abort`; `noexcept` behavior.
- ISO C++ Core Guidelines: interface contracts and the Guidelines philosophy on expressing assumptions clearly.
- Microsoft GSL repository: GSL as a Guidelines-supporting library used to express contract-style checks and safer vocabulary utilities.
- WG21 Contracts papers: P2900 (Contracts for C++), and the 2025 rationale document discussing design and safety tradeoffs.

Chapter 23

Error Handling Strategies

23.1 Exceptions vs. error codes

23.1.1 The real question: what kind of failure is this?

Professional C++ error handling starts by classifying failures into two groups:

- **Recoverable conditions** that are expected in normal operation (file not found, parse failure, network timeout, permission denied). These should be reported in a way that callers are likely to handle correctly.
- **Bugs and contract violations** (violated preconditions, broken invariants, out-of-bounds access, use-after-free). These are not recoverable by the caller in a meaningful way; continuing may be unsafe. These should fail fast (assert/contracts/terminate).

The C++ Core Guidelines recommend adopting an explicit error-handling strategy early and

emphasize systematic error handling (typically exceptions + RAII, with alternatives when exceptions are not usable).

23.1.2 What exceptions provide (and what they cost)

What exceptions are good at

Exception handling is a language mechanism that transfers control up the call stack to a handler. That has practical benefits:

- **Non-local propagation:** deep call stacks can report failure without manually threading error checks through every layer.
- **RAII + unwinding:** when an exception propagates, destructors for fully-constructed objects are run during stack unwinding, which is why RAII is central to exception safety.
- **Cleaner "happy path" code:** normal logic is not interleaved with repetitive error-checking at every call site.
- **Uniform reporting:** exceptions can carry rich information (exception object type + message + error codes).

The ISO C++ FAQ summarizes the typical intent: exceptions signal errors that cannot be handled locally (e.g., failure to acquire a resource in a constructor). Microsoft's modern C++ guidance similarly describes exceptions as the preferred reporting mechanism in many scenarios, especially when the handler is far from the detection point.

What exceptions cost

Exceptions also introduce real engineering costs:

- **Build modes and platforms:** some environments disable exceptions (embedded, freestanding, some game engines, kernel-like code, ABI boundaries).
- **Control-flow transparency:** you must know what can throw, where it can throw from, and what invariants must hold if it throws.
- **Testing failure paths:** exception safety requires systematically testing and reasoning about rollback/commit behavior.
- **`noexcept` boundaries:** if an exception escapes a `noexcept` function, the program calls `std::terminate`.

23.1.3 What error codes provide (and what they cost)

Error codes done professionally: `std::error_code` and friends

Modern C++ does not mean returning magic integers. For OS and low-level failures, the standard library provides:

- `std::error_code`: platform-dependent error value + category.
- `std::error_condition`: platform-independent condition used for portable comparisons.
- `std::system_error`: an exception type that carries a `std::error_code`.

This gives you a coherent vocabulary for failure without inventing ad-hoc numeric codes.

The core advantage of error codes

- **Explicit local control-flow:** every call site chooses what to do.

- **Works where exceptions are forbidden:** kernels/freestanding builds, certain ABIs, certain performance-critical policies.
- **Predictability:** no hidden control transfer, easier to reason about in some low-level code.

The core disadvantage of error codes

- **Error propagation boilerplate:** without help, code becomes repetitive and callers may ignore checks.
- **Lost context:** integer codes often lose meaning unless paired with structured error objects.
- **Partial initialization hazards:** if you rely on manual cleanup on every return path, you re-introduce resource leaks unless you use RAII consistently.

23.1.4 A modern compromise: result types (`expected`)

C++23 adds `std::expected<T,E>` to represent either a value or an error, and it is never valueless. This supports an "error codes, but structured and hard-to-ignore" approach:

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

enum class ParseErr { empty, invalid };

std::expected<int, ParseErr> parse_port(std::string_view s) {
  if (s.empty()) return std::unexpected(ParseErr::empty);
  int v = 0;
  for (char c : s) {
```

```
    if (c < '0' || c > '9') return std::unexpected(ParseErr::invalid);
    v = v * 10 + (c - '0');
  }
  if (v < 1 || v > 65535) return std::unexpected(ParseErr::invalid);
  return v;
}
```

`expected` keeps error handling explicit while making propagation and composition easier than raw integer returns.

23.1.5 Safety tradeoffs in practice

Exception safety vs. manual cleanup

With exceptions enabled, RAII is the default safety net: resources acquired in constructors or factories are automatically released during unwinding.
Without exceptions, RAII is still essential, but you must be stricter about propagating errors immediately and consistently.

Library design: do not surprise users

A library should choose one main strategy and expose it consistently:

- **Exception-based API:** operations that cannot complete throw; callers catch at subsystem boundaries.
- **Result-based API:** operations return `expected` or `error_code`; callers check and propagate.

Mixing both in the same layer (sometimes throw, sometimes return codes) tends to confuse users and causes missed checks. The Core Guidelines explicitly call for defining an error-handling strategy early and being consistent.

ABI boundaries and C APIs

Across C boundaries, exceptions must not escape. The common safe pattern is:

- catch all exceptions at the boundary,
- translate into an error code or `expected` result,
- keep the exception-based implementation internal.

```
#include <exception>
#include <system_error>

extern "C" int api_do_work(/*...*/) noexcept {
  try {
    // internal C++ code that may throw
    return 0;
  } catch (const std::system_error& e) {
    // translate to stable error code contract
    return e.code().value();
  } catch (...) {
    return -1; // generic failure code
  }
}
```

`noexcept` here is intentional: if an exception would otherwise escape, termination would occur.

23.1.6 Choosing between exceptions and error codes: a decision guide

Prefer exceptions when

- the failure is **rare** and the normal path should stay clean,

- the handler is **far** from the failure site,
- you benefit from **automatic unwinding** with RAII,
- your environment supports exceptions and your ABI policy allows them.

This matches both the ISO C++ FAQ and Microsoft's modern guidance describing exceptions as the preferred reporting mechanism in many scenarios.

Prefer error codes / `expected` when

- exceptions are unavailable or forbidden (freestanding, some embedded, some performance/footprint policies),
- failure is **common** and you want it in the normal control flow (parsing, validation, "not found" lookups),
- you need a stable C-compatible surface (plugins, FFI, C ABI),
- you require explicit and predictable control-flow for low-level subsystems.

Use `std::error_code` for OS-facing errors, and `std::expected` for typed domain errors.

23.1.7 How to avoid the two biggest failure modes

Failure mode A: ignored error codes

If you return error codes, design the API so results are hard to ignore:

- use `[[nodiscard]]` on error-returning functions,
- return a structured result (`expected`) rather than a naked `int`,
- centralize translation at boundaries and propagate uniformly.

Failure mode B: exception-unsafe code

If you use exceptions, enforce exception safety rules:

- never let destructors throw,
- maintain class invariants across throws (either commit fully or roll back safely),
- use RAII for every resource,
- use `noexcept` deliberately as an architectural boundary (and only when termination is acceptable on violation).

23.1.8 A practical hybrid strategy used in many professional codebases

A common, safe layering model:

- **Core low-level layer (platform/FFI/driver):** return `std::error_code` or `expected` (no exceptions cross boundaries).
- **Domain layer:** use `expected` for common domain failures (validation, parsing, optional resources).
- **Application orchestration layer:** optionally use exceptions for rare failures that must unwind a large stack, with a single top-level boundary that catches and reports.

This reduces boilerplate in high-level code while keeping boundaries stable and explicit.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): exception handling model, stack unwinding, `noexcept`, and termination behavior.

- C++ Core Guidelines, *E: Error handling*: strategy and consistency guidance; emphasis on systematic error handling (typically exceptions + RAII) with alternatives where needed.
- ISO C++ FAQ: *Exceptions and Error Handling* (intent and usage guidance).
- cppreference.com: exception handling overview and stack unwinding; `noexcept` semantics and `std::terminate`; `std::error_code`, `std::error_condition`, `std::system_error`; `std::expected` (C++23).
- Microsoft documentation: modern C++ best practices for errors and exception handling.

23.2 RAII and exceptions

23.2.1 Why RAII and exceptions fit together

RAII (Resource Acquisition Is Initialization) binds the lifetime of a resource (memory, file handle, socket, mutex lock, thread handle, etc.) to the lifetime of an object. When the object is constructed, it acquires the resource; when it is destroyed, it releases it.
Exceptions change control flow: they can leave a scope early. The safety problem is obvious: if control leaves early, who releases the resources acquired in that scope?
RAII is the standard C++ answer:

> If resources are owned by objects with correct destructors, stack unwinding caused by exceptions still runs those destructors, so resources are released on every path.

This idea underpins the basic exception-safety guarantees (no resource leaks, invariants preserved) described in the standard C++ exception model and its widely used formulation (nothrow/strong/basic guarantees).

23.2.2 The three practical exception-safety guarantees

A useful professional vocabulary is:

- **Nothrow (nofail) guarantee:** the operation never lets an exception propagate. Destructors are expected to meet this, and since C++11 destructors are `noexcept` by default.
- **Strong guarantee:** if the operation throws, the program state is unchanged (transactional behavior).
- **Basic guarantee:** if the operation throws, the program remains valid: no leaks, invariants intact, but some state may have changed.

RAII is what makes the basic guarantee realistic in normal C++ code.

23.2.3 Stack unwinding: what actually happens

When an exception propagates out of a scope, the runtime destroys (unwinds) fully constructed automatic objects in that scope, running their destructors. This is why RAII works: owned resources are released even on exceptional exits.

23.2.4 The destructor rule: destructors must not throw

Why throwing from a destructor is dangerous

If a destructor throws while another exception is already propagating, the program must terminate (there is no safe way to continue). This is the well-known C++ rule highlighted in the ISO C++ FAQ.
In modern C++, destructors are implicitly `noexcept` by default, so an uncaught exception escaping a destructor triggers `std::terminate`.

The correct pattern: catch inside destructors

If cleanup code can fail, catch failures inside the destructor and convert them into a safe action: log, set a flag, store an error for later observation, or fail-fast if continuing is unsafe.

```
#include <cstdio>
#include <utility>

class File {
public:
  explicit File(const char* path, const char* mode)
    : f_(std::fopen(path, mode)) {}

  File(File&& other) noexcept : f_(std::exchange(other.f_, nullptr)) {}
  File& operator=(File&& other) noexcept {
    if (this != &other) {
      close_noexcept();
      f_ = std::exchange(other.f_, nullptr);
    }
    return *this;
  }

  File(const File&) = delete;
  File& operator=(const File&) = delete;
```

```
  ~File() noexcept { close_noexcept(); }

  std::FILE* get() const noexcept { return f_; }

  // explicit close for callers that need to detect failure
  bool close() noexcept {
    if (!f_) return true;
    int rc = std::fclose(f_);
    f_ = nullptr;
    return rc == 0;
  }

private:
  void close_noexcept() noexcept {
    if (!f_) return;
    // Destructors must not throw; we cannot report errors reliably here.
    // Best-effort: attempt close and ignore failure or record it
    ↪  elsewhere.
    (void)std::fclose(f_);
    f_ = nullptr;
  }

  std::FILE* f_{nullptr};
};
```

This design reflects a key practical reality: destructors cannot reliably report failure. The API provides an explicit `close()` for callers that need error reporting, while the destructor is best-effort and non-throwing. (This is also a recurring discussion point in modern guideline work: “failable cleanup” must be designed explicitly, not hidden in destructors.)

23.2.5 RAII building blocks you should standardize on

Owning memory: `std::unique_ptr` and `std::vector`

Owning types automatically clean up during normal exits and during unwinding. This is the core of exception-safe resource management.

```
#include <memory>
#include <vector>

struct Node { int x; };

std::unique_ptr<Node> make_node() {
  return std::make_unique<Node>(Node{42});
}

std::vector<int> build() {
  std::vector<int> v;
  v.push_back(1);
  v.push_back(2);
  return v; // strong exception safety provided by the standard library
   ↪  where applicable
}
```

Locks: `std::lock_guard` / `std::unique_lock`

The lock is released reliably even if code throws.

```
#include <mutex>

std::mutex m;

void update() {
```

```
  std::lock_guard<std::mutex> lock(m); // releases in destructor
  // work that might throw
}
```

General cleanup: scope guards

Scope guards are RAII for “do this on scope exit”. They reduce cleanup duplication and improve correctness.

```
#include <utility>

template <class F>
class ScopeExit {
public:
  explicit ScopeExit(F f) noexcept : f_(std::move(f)), active_(true) {}
  ScopeExit(ScopeExit&& other) noexcept
    : f_(std::move(other.f_)), active_(std::exchange(other.active_,
    ↪  false)) {}
  ~ScopeExit() noexcept { if (active_) f_(); }
  void release() noexcept { active_ = false; }
private:
  F f_;
  bool active_;
};

template <class F>
ScopeExit<F> scope_exit(F f) noexcept { return
↪  ScopeExit<F>(std::move(f)); }

void work() {
  int* p = new int[128];
  auto guard = scope_exit([&] { delete[] p; }); // cleanup even on throw
```

```
  // ...
  guard.release(); // if ownership transferred elsewhere
}
```

23.2.6 Designing RAII types for exception safety

Acquire resources in constructors or factories

If an object cannot become valid, it should not be constructed into an unusable half-state. A common recommendation is to signal construction failure with an exception (when exceptions are enabled) rather than leaving an incomplete object.

```
#include <stdexcept>

class Socket {
public:
  Socket(/*params*/) {
    // acquire resource; if fail:
    // throw std::runtime_error("cannot open socket");
  }
  ~Socket() noexcept {
    // release resource (must not throw)
  }
};
```

If your environment cannot throw, keep the same RAII principle but use a factory returning a result type (e.g., `std::expected`) and construct the RAII object only on success. The Core Guidelines explicitly discuss simulating RAII and using error codes systematically when exceptions are not available.

Make moves `noexcept` when feasible

Strong exception safety in containers and generic code often depends on operations like move construction not throwing. The general exception-safety guidance emphasizes that providing a no-throw guarantee is only reliable when all operations used are no-throw (or exceptions are contained).

```
class Handle {
public:
  Handle(Handle&&) noexcept = default;
  Handle& operator=(Handle&&) noexcept = default;
  ~Handle() noexcept = default;
};
```

23.2.7 Writing exception-safe operations: commit/rollback

RAII helps with leaks, but you must still protect invariants and partially-updated state.

Basic guarantee: keep invariants and avoid leaks

```
#include <vector>

class Buffer {
public:
  void add(int x) {
    // std::vector provides strong/basic guarantees for many operations;
    // regardless, class invariants remain intact if an exception is
    ↪  thrown.
    data_.push_back(x);
  }
private:
  std::vector<int> data_;
```

```
};
```

Strong guarantee: update via “build then swap”

A standard technique for strong exception safety is:

1. build the new state in a temporary object,
2. commit by swapping.

This is consistent with the strong guarantee description (rollback to the previous state on throw).

```
#include <string>
#include <utility>

class Config {
public:
  void set_name(std::string s) {
    std::string tmp = std::move(s); // may allocate; if it throws, *this
    ↪  unchanged
    name_.swap(tmp);                // commit (no-throw for std::string
    ↪  swap in practice)
  }
private:
  std::string name_;
};
```

23.2.8 `noexcept` boundaries and termination

If an exception escapes a `noexcept` function, the runtime calls `std::terminate`. Use this deliberately:

- Destructors should be non-throwing (default `noexcept`) because they may run during unwinding.
- Low-level boundaries (C ABI, thread entry points) should be `noexcept` and catch/translate all exceptions to avoid unexpected termination.

```
#include <exception>

extern "C" int api_entry() noexcept {
  try {
    // internal C++ code may throw
    return 0;
  } catch (...) {
    // translate to error code/log; never let exceptions cross the
    ↪  boundary
    return -1;
  }
}
```

23.2.9 Checklist: RAII + exceptions done professionally

1. Own every resource with an object (RAII), so unwinding releases resources automatically.
2. Design destructors to never throw; catch internally if needed.
3. For failable release, provide an explicit operation (e.g., `close()`) that reports errors; keep the destructor best-effort and `noexcept`.
4. Decide which functions offer basic vs strong guarantees; implement strong guarantee with commit/rollback patterns when needed.

5. Use `noexcept` intentionally as an architectural boundary; remember violation calls `std::terminate`.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): exception handling, stack unwinding, destructor semantics, and `noexcept`/termination rules.
- cppreference.com: RAII overview; exception handling and exception-safety guarantees; `noexcept` and `std::terminate`.
- ISO C++ FAQ (isocpp.org): destructors must not throw during stack unwinding and recommended handling strategies.
- Microsoft Learn: guidance on designing for exception safety and the relationship between no-throw guarantees and callees.
- Bjarne Stroustrup, *Exception Safety: Concepts and Techniques*: formalization of basic/strong guarantees and techniques for exception-safe design.
- SEI CERT C++ Coding Standard: guidance on preventing resource leaks when handling exceptions and avoiding incomplete object states.
- C++ Core Guidelines: guidance for error handling strategies when exceptions are not available (RAII simulation, failing fast, systematic error codes).

23.3 Designing failure-safe code

23.3.1 What “failure-safe” means in C++

Failure-safe code continues to behave predictably when operations fail:

- it does not leak resources,
- it does not break invariants,
- it does not leave partially-initialized objects visible to callers,
- it makes failures explicit (either by propagating an exception or by returning a structured error),
- it fails fast on contract violations rather than attempting unsafe recovery.

In modern C++, failure-safety is largely built on two pillars:

1. **RAII** for cleanup on all paths, especially during exception unwinding.
2. **Exception-safety guarantees** (basic, strong, nothrow) as an explicit engineering contract.

23.3.2 The minimum bar: the Basic exception-safety guarantee

cppreference summarizes the three standard guarantees used in practice:

- **Nothrow (nofail) guarantee:** the operation never lets exceptions propagate.
- **Strong guarantee:** if the operation throws, program state is rolled back to the state before the call (transactional behavior).
- **Basic guarantee:** if the operation throws, no resources are leaked and all invariants remain intact (the program is still valid).

For safety engineering, the rule is:

> Every operation that can fail must provide at least the **basic guarantee**.

23.3.3 Rule 1: Use RAII everywhere to prevent leaks

SEI CERT explicitly warns that exceptions can bypass manual cleanup and lead to resource leaks, and recommends RAII to reclaim resources automatically when objects go out of scope.

Bad: manual cleanup that is skipped on failure

```
#include <cstdio>
#include <stdexcept>

void write_bad(const char* path) {
  std::FILE* f = std::fopen(path, "wb");
  if (!f) throw std::runtime_error("open failed");

  // If anything throws below, fclose might never run:
  // ...
  throw std::runtime_error("something failed");

  std::fclose(f);
}
```

Good: RAII wrapper ensures cleanup during unwinding

```
#include <cstdio>
#include <stdexcept>

class File {
public:
  explicit File(const char* path, const char* mode)
    : f_(std::fopen(path, mode)) {
    if (!f_) throw std::runtime_error("open failed");
```

```
  }

  File(const File&) = delete;
  File& operator=(const File&) = delete;

  ~File() noexcept { if (f_) std::fclose(f_); } // must not throw

  std::FILE* get() const noexcept { return f_; }

private:
  std::FILE* f_;
};

void write_good(const char* path) {
  File f(path, "wb");
  // ...
  throw std::runtime_error("something failed"); // file still closes
}
```

This pattern directly addresses the CERT rule about not leaking resources when exceptions occur.

23.3.4 Rule 2: Keep invariants true at all times

Failure-safe code is not only leak-free. It also preserves invariants under partial execution. That is the heart of the basic guarantee.

Pattern A: do work on a temporary, then commit (strong guarantee)

The strong guarantee is typically implemented with commit/rollback semantics. The Core Guidelines explicitly discuss that the swap technique can provide a stronger guarantee than writing directly into the target state.

```cpp
#include <vector>
#include <utility>

class SafeList {
public:
  void replace_all(std::vector<int> v) {
    // If copying/moving into 'v' fails before this call, *this is
    ↪  unchanged.
    data_.swap(v); // commit step
  }

private:
  std::vector<int> data_;
};
```

Pattern B: localize mutation and keep objects valid (basic guarantee)

If strong guarantee is too expensive, still ensure:

- invariants remain true,
- partially applied effects do not create undefined behavior later.

```cpp
#include <vector>

class Buffer {
public:
  void push(int x) {
    // If push_back throws (e.g., allocation failure), vector remains
    ↪  valid.
    data_.push_back(x);
  }
```

```
private:
  std::vector<int> data_;
};
```

The standard library containers are designed with exception-safety guarantees, and the guarantee vocabulary above is the common framework used to reason about them.

23.3.5 Rule 3: Never let destructors throw

cppreference notes that the nothrow guarantee is expected of destructors and functions that may be called during stack unwinding, and destructors are `noexcept` by default since C++11.
Therefore:

- Destructors should be `noexcept` and must not allow exceptions to escape.
- If cleanup can fail, provide an explicit `close()`/`commit()` that reports failure; keep the destructor best-effort.

```
class Tx {
public:
  void commit();            // may report failure (exception or error
  ↪  result)
  ~Tx() noexcept {          // must not throw
    if (!committed_) rollback_noexcept();
  }
private:
  void rollback_noexcept() noexcept { /* best-effort */ }
  bool committed_{false};
};
```

23.3.6 Rule 4: Build-then-publish for initialization safety

A common failure mode is exposing partially initialized objects. CERT explicitly mentions partially initialized states as a risk in exception scenarios and points toward RAII as a safer design.

Exception-enabled: constructors can throw before publication

```
#include <memory>
#include <stdexcept>

class Service {
public:
  Service(); // may throw; if it throws, no object escapes
};

std::unique_ptr<Service> make_service() {
  return std::make_unique<Service>();
}
```

Exception-free boundary: return a result type

C++23 `std::expected<T,E>` represents either a value or an error and is never valueless.

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

enum class Err { config_missing, invalid };

class Service {
public:
```

```
  static std::expected<std::unique_ptr<Service>, Err> create(std::string
  ↪  cfg);
private:
  Service() = default;
};
```

This style makes failure explicit while still preventing half-constructed objects from leaking into the system.

23.3.7 Rule 5: Make failure propagation hard to ignore

Prefer structured propagation over ad-hoc integers

For domain failures:

- use `expected` for "value or error",
- use `optional` for "value or nothing",
- use `std::error_code` for OS-facing failures (file operations, system calls).

Compose failure-safe steps cleanly

C++23 provides combinators such as `and_then` for `expected` to chain operations while propagating errors without manual branching.

```
#include <expected>
#include <string_view>

enum class Err { parse, range };

std::expected<int, Err> parse_int(std::string_view);
std::expected<int, Err> check_range(int);
```

```
std::expected<int, Err> parse_and_validate(std::string_view s) {
  return parse_int(s).and_then(check_range);
}
```

This reduces boilerplate and makes the failure path explicit and uniform.

23.3.8 Rule 6: Decide where to fail fast

Failure-safe code also means **not pretending** you can recover from a broken contract. Use fail-fast (assert/contracts/terminate) when continuing could lead to undefined behavior or corrupted invariants.
This complements (not replaces) structured error reporting for recoverable conditions.

23.3.9 A short implementation checklist

1. Every function that can fail must state its guarantee (basic/strong/nothrow).
2. Own every resource with RAII so cleanup happens during unwinding and early returns.
3. Keep invariants true even under partial execution (at least basic guarantee).
4. Use commit/rollback (build-then-swap) when you need the strong guarantee.
5. Destructors never throw; provide explicit “failable” operations separately.
6. At boundaries, return structured results (`expected`/`error_code`) and make them hard to ignore.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): exception handling model, stack unwinding, destructor `noexcept` semantics (C++11+), and library guarantees.
- cppreference.com: exception-safety guarantees (nothrow/strong/basic) and guidance that destructors are expected to be nothrow; C++23 `std::expected` and its "never valueless" property; `expected::and_then`.
- SEI CERT C++ Coding Standard (ERR57-CPP): do not leak resources when handling exceptions; recommends RAII and warns about bypassed cleanup and partially initialized objects.
- C++ Core Guidelines: discussion of stronger guarantees via swap/commit techniques and general contract-based design guidance.
- Microsoft Learn: modern C++ guidance recommending selecting and documenting one of the exception-safety guarantees for functions that might throw.

Chapter 24

Testing for Memory Safety

24.1 Unit testing limitations

24.1.1 Why unit tests matter, and why they are not enough

Unit tests are excellent for verifying *intended behavior* on representative inputs:

- correct outputs for known cases,
- regression prevention during refactoring,
- documenting expected semantics at the API level.

But memory safety failures are often *not behavioral bugs*. They are **semantic violations** (undefined behavior, lifetime misuse, data races) whose visible effects can be delayed, non-deterministic, platform-dependent, or optimized away.
A critical C++ fact drives many limitations:

> Compilers are not required to diagnose undefined behavior, and the program is not required to do anything meaningful once UB occurs.

Therefore, a unit test suite can be 100% green and still ship serious memory safety bugs.

24.1.2 Limitation 1: Unit tests sample executions, not the full state space

Finite tests vs. infinite behaviors

A unit test executes only a tiny subset of:

- input combinations,
- runtime states (heap layout, allocator behavior),
- timing/interleavings (thread scheduling),
- platform/compiler/optimization combinations.

Memory safety defects often require a specific heap layout or timing window to manifest. If the triggering conditions are not hit, the test passes.

24.1.3 Limitation 2: Undefined behavior can pass tests

UB may appear "stable" until one day it is not

Because UB has no required diagnostic and no required runtime outcome, it may:

- "work" in debug builds and fail in release builds,
- fail only with a different compiler or optimization,
- disappear when logging is added (Heisenbugs),
- corrupt memory silently and explode later in an unrelated test.

Example: out-of-bounds write that often goes unnoticed

```
#include <array>
#include <cstddef>

int buggy_sum(const int* p, std::size_t n) {
  std::array<int, 4> tmp{};          // small buffer
  for (std::size_t i = 0; i < n; ++i) {
    tmp[i] = p[i];                   // UB when n > 4 (out-of-bounds write)
  }
  int s = 0;
  for (int x : tmp) s += x;
  return s;
}
```

Many unit tests will pass if they never call `buggy_sum` with `n > 4`. Even worse, the program may still appear correct for some time, because UB can do *anything*.

24.1.4 Limitation 3: Memory corruption is often delayed and misattributed

The crash usually happens far from the bug

Classic memory errors (heap buffer overflow, use-after-free) can corrupt allocator metadata or nearby objects. The process may continue and crash later, causing:

- misleading stack traces,
- failures in unrelated unit tests,
- intermittent flakiness that is hard to reproduce.

This is why dynamic instrumentation (sanitizers) is routinely paired with unit tests: it detects the error *at the moment it happens*, not when it finally crashes.

24.1.5 Limitation 4: Concurrency bugs evade unit tests

Data races depend on scheduling

Data races are inherently timing-dependent: the same test can pass 10,000 times and fail once. Even a "correct" unit test may hide concurrency bugs by running with a different threading model, by mocking, or by adding synchronization that does not exist in production. ThreadSanitizer exists because unit tests alone are not sufficient to explore interleavings. It also has known overhead and practical constraints (memory/virtual address space usage, no static linking of certain runtimes).

Example: a race that may never show up in unit tests

```
#include <thread>
#include <atomic>

int x = 0;              // non-atomic shared variable

void writer() { x = 42; }
void reader() {
  if (x == 42) {
    // This branch may or may not be observed.
    // The race itself is the bug.
  }
}

void test_like() {
  std::thread t1(writer);
```

```
  std::thread t2(reader);
  t1.join();
  t2.join();
}
```

This code has a data race; the result is undefined behavior under the C++ memory model. A unit test might pass forever and still be wrong.

24.1.6 Limitation 5: Sanitizer coverage depends on how you build and run tests

Even when you *do* run sanitizers, limitations remain:

- Sanitizers only detect what actually executes under the instrumentation.
- Some instrumentation can be missing if parts of the program (or third-party libraries) are not compiled/linked with the sanitizer runtime, which can lead to missed detections.

For example, Clang documents AddressSanitizer limitations and overhead characteristics. MSVC also documents cases where incorrect linking/overrides can reduce ASan effectiveness and cause missed detections.
Therefore, “unit tests + sanitizers” is stronger than unit tests alone, but it is still not a proof of memory safety.

24.1.7 Limitation 6: Tests can accidentally introduce UB themselves

Test code is still C++ code. If fixtures, mocks, or test utilities contain UB, the test outcome becomes unreliable. Real-world test frameworks even document certain patterns as undefined behavior (for example, expectation ordering rules in mocking frameworks), which means a failing or passing test might not reflect the correctness of the product code at all.

24.1.8 Limitation 7: Unit tests validate behavior, not contracts

Many memory safety rules are **contracts**:

- ownership and lifetime rules,
- non-null and bounds requirements,
- thread-safety requirements (which locks must be held),
- "this API must not be called after shutdown" rules.

Unit tests can demonstrate correct usage, but they do not automatically enforce contracts across all call sites. Contracts must be enforced by design (types, RAII, ownership vocabulary), by runtime checks in debug/CI, and by tooling.

24.1.9 What to do professionally: treat unit tests as one layer

A practical memory-safety workflow

Keep unit tests, but recognize their limits and systematically add layers:

1. **Run the full test suite under sanitizers**:
    - ASan for spatial/temporal memory errors and related issues (with documented limitations and overhead),
    - UBSan for a broad set of UB classes (only when the UB executes),
    - TSan for data races (also with overhead/limitations).
2. **Use multiple build modes** (debug/release, different optimizations) because UB and races are sensitive to optimization and layout.

3. **Use static analysis and guideline checks** to catch issues without needing the exact runtime trigger.

4. **Add fuzzing and property-based testing** for parser/decoder/input-heavy components where corner cases dominate.

5. **Add stress tests** (long-running, high-concurrency, randomized scheduling) to widen interleaving coverage beyond unit tests.

24.1.10 A concise checklist: when unit tests are likely to miss memory bugs

Unit tests are likely to miss a memory safety bug when:

- the failure requires a rare heap layout or timing window,
- the bug is UB and appears "fine" in one build configuration,
- concurrency interleavings are the trigger,
- key code paths are not executed by tests (low coverage),
- sanitizer instrumentation is incomplete or mis-linked.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): undefined behavior model; concurrency and data race rules; object lifetime rules.
- cppreference.com: undefined behavior (no diagnostic required; behavior unconstrained); "no diagnostic required" concept.

- LLVM/Clang documentation: AddressSanitizer limitations; UndefinedBehaviorSanitizer overview; ThreadSanitizer limitations.
- Microsoft documentation: AddressSanitizer known issues (cases where ASan may miss detections if not linked/configured correctly).
- ISO C++ Core Guidelines: tooling and guideline enforcement ecosystem (clang-tidy / CppCoreCheck) as complementary layers to tests.

24.2 Stress and fuzz testing

24.2.1 Why memory safety needs more than unit tests

Many memory-safety defects are **trigger-dependent**:

- they require rare inputs (edge-case parsing, unusual encodings),
- they require specific heap layouts (use-after-free, buffer overflow),
- they require specific interleavings (data races, atomic misuse).

Two complementary techniques target these failure modes:

- **Fuzz testing:** explore huge input spaces automatically.
- **Stress testing:** explore runtime states and timing (load, concurrency, duration).

In professional C++ safety practice, these are usually combined with sanitizers so that bugs are detected at the moment they occur, not later. OSS-Fuzz explicitly describes pairing fuzzers with sanitizers and supports multiple engines.

24.2.2 Fuzz testing fundamentals (coverage-guided fuzzing)

What fuzzing is actually doing

A modern fuzzer does not generate random bytes blindly. Coverage-guided fuzzers:

- start from an initial **corpus** (seed inputs),
- mutate inputs in many ways,
- run the target repeatedly,
- measure which code regions were reached,
- keep mutations that increase coverage or trigger new behaviors.

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine that links directly into your target and uses coverage instrumentation (SanitizerCoverage) to guide mutations.

What fuzzing finds well

Fuzzing is especially effective at finding:

- out-of-bounds reads/writes,
- use-after-free,
- integer overflows that lead to dangerous sizes/indices,
- undefined behavior that manifests on unusual paths,
- parser/decoder mistakes that unit tests rarely cover.

In practice, libFuzzer-based workflows explicitly aim to trigger crashes, memory bugs, and undefined behavior in the code under test.

24.2.3 Writing a good fuzz target (harness)

The minimal libFuzzer harness

A libFuzzer target is a function with a stable signature that consumes a byte buffer. The engine repeatedly calls this function with mutated inputs.

```
#include <cstddef>
#include <cstdint>
#include <string_view>

// Example "library" API under test:
int parse_message(std::string_view bytes);

extern "C" int LLVMFuzzerTestOneInput(const std::uint8_t* data,
↪  std::size_t size) {
  // Convert to a view without copying.
  std::string_view sv(reinterpret_cast<const char*>(data), size);

  // Call into the component under test.
  // Any crash, sanitizer finding, or UB is a bug to fix.
  (void)parse_message(sv);

  return 0;
}
```

Harness rules that prevent false failures

A high-quality harness must:

- be deterministic for a given input (no dependency on current time or external I/O),
- avoid global state leakage across iterations (reset state),

- avoid input rejection too early (do not discard most inputs instantly),
- avoid undefined behavior inside the harness itself.

When running in persistent-style loops, it is especially important that the harness resets the system-under-test state after each execution, otherwise one iteration can influence the next.

24.2.4 Engine choices and where each fits

LibFuzzer: best for in-process library fuzzing

LibFuzzer is in-process and can be very fast because it avoids per-test process startup overhead; it uses coverage to guide mutations.
Use it when:

- you fuzz a library or component callable in-process,
- you want very high throughput,
- you can make the harness deterministic and self-contained.

AFL++: excellent for binaries and persistent mode speedups

AFL++ documentation strongly encourages persistent mode because it can increase speed significantly by running many test cases per process.
Use AFL++ when:

- you fuzz command-line programs or binary interfaces,
- you want robust forkserver-based isolation,
- you can benefit from persistent mode for higher throughput.

OSS-Fuzz: continuous fuzzing infrastructure for open source

OSS-Fuzz supports multiple fuzzing engines (including libFuzzer and AFL++) together with sanitizers and a distributed execution environment.
Use OSS-Fuzz when:

- you maintain an open-source project and want continuous fuzzing at scale,
- you want standardized crash reporting and long-running coverage growth.

24.2.5 Fuzzing + sanitizers: the standard best practice

Coverage-guided fuzzing becomes dramatically more valuable when combined with sanitizers:

- fuzzing finds the triggering input,
- sanitizers explain the root cause precisely at the point of violation.

OSS-Fuzz is explicitly built around fuzzers in combination with sanitizers.
Practical guidance:

- Use ASan for heap/stack/global out-of-bounds and use-after-free.
- Use UBSan to catch many undefined behaviors when executed.
- Use TSan for concurrency-heavy fuzz targets only when feasible (higher overhead).

24.2.6 Stress testing fundamentals

What stress testing is actually doing

Stress testing tries to provoke bugs by exploring runtime states:

- heavy load (many iterations, long duration),
- large data sizes,
- many threads and frequent context switches,
- forced contention (locks, atomics, queues),
- randomized schedules or injected delays.

Stress testing is particularly important for concurrency correctness, where bugs may depend on rare interleavings.

Stress testing is not the same as benchmarking

Benchmarking tries to measure performance under controlled conditions. Stress testing tries to **break assumptions**:

- saturation (CPU, memory),
- failure injection (timeouts, partial reads),
- repetition until rare states appear.

24.2.7 Practical stress tests for memory safety and concurrency

Pattern A: Long-running randomized state testing

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

void apply_operation(std::vector<int>& v, int op);
```

```
void stress_vector_ops() {
  std::mt19937 rng(12345);
  std::uniform_int_distribution<int> dist(0, 9);

  std::vector<int> v;
  for (std::size_t i = 0; i < 10'000'000; ++i) {
    int op = dist(rng);
    apply_operation(v, op);
  }
}
```

Run this under sanitizers to turn rare heap/lifetime defects into actionable reports.

Pattern B: Concurrency stress with forced contention

```
#include <atomic>
#include <thread>
#include <vector>
#include <chrono>

struct Counter {
  std::atomic<int> x{0};
  void inc() { x.fetch_add(1, std::memory_order_relaxed); }
};

void stress_counter(Counter& c) {
  constexpr int threads = 8;
  constexpr int iters   = 2'000'000;

  std::vector<std::thread> ts;
  ts.reserve(threads);
  for (int t = 0; t < threads; ++t) {
```

```
    ts.emplace_back([&]{
      for (int i = 0; i < iters; ++i) {
        c.inc();
        if ((i & 1023) == 0) std::this_thread::yield(); // increase
        ↪  interleavings
      }
    });
  }
  for (auto& th : ts) th.join();
}
```

ThreadSanitizer as a stress companion

TSan detects data races with significant overhead and practical limitations (memory overhead and large virtual address space mapping; static linking of libc/libstdc++ not supported). This means:

- you may need smaller stress configurations under TSan than in native runs,
- you should design a dedicated TSan stress suite (fewer threads/iterations, but many repetitions).

24.2.8 Combining fuzzing and stress testing

Hybrid 1: Fuzz to generate inputs, then stress the state machine

Use fuzzing to discover “interesting” inputs and save them to a corpus. Then run a stress test that replays those inputs repeatedly, under different threading and allocator conditions.

Hybrid 2: Differential fuzzing

If you have two implementations (new vs old, optimized vs reference), fuzz both with the same input and compare outputs. This is excellent for detecting subtle logic differences in parsers and decoders that may later trigger memory safety issues.

24.2.9 What these techniques cannot guarantee

- Fuzzing and stress testing increase the chance of finding bugs; they do not prove the absence of bugs.
- Coverage-guided fuzzing covers what your harness can reach. If the harness rejects inputs too early or does not model the real API usage, coverage and bug-finding drop.
- Persistent-mode fuzzing and long-running stress loops require careful state reset; otherwise, you get false positives, flaky behavior, and irreproducible crashes.

24.2.10 A compact professional workflow

1. Write a minimal deterministic harness (`LLVMFuzzerTestOneInput` for libFuzzer).
2. Seed with a small corpus of valid/interesting inputs.
3. Run fuzzing under ASan/UBSan first; add TSan for concurrency-heavy targets as feasible.
4. Enable persistent mode where applicable for throughput (AFL++ guidance).
5. When a crash occurs: minimize and reproduce, fix, and keep the reproducer in the regression corpus.

6. Add stress tests that replay the corpus under load, different thread counts, and different build modes.

References (authoritative, no links)

- LLVM Documentation: *LibFuzzer* (in-process, coverage-guided evolutionary fuzzing; harness entrypoint; sanitizer coverage guidance).
- Google OSS-Fuzz Documentation: supported engines (libFuzzer, AFL++, etc.) and fuzzing with sanitizers at scale.
- AFL++ Documentation: *Fuzzing in Depth* and persistent mode guidance and rationale.
- LLVM/Clang Documentation: *ThreadSanitizer* limitations and overhead considerations for stress testing.

24.3 Concurrency testing

24.3.1 Why concurrency bugs are uniquely hard to test

Concurrency failures are often:

- **non-deterministic**: they depend on timing and scheduling,
- **rare**: they require a specific interleaving,
- **catastrophic**: a single data race is undefined behavior in C++ and can lead to memory corruption.

This means a test can pass thousands of times and still be wrong.

24.3.2 What you must test for

1) Data races

A data race occurs when two threads access the same memory location concurrently, at least one access is a write, and there is no ordering that establishes a happens-before relationship. In C++ (since C++11), a data race produces undefined behavior.

2) Deadlocks and livelocks

- **Deadlock**: threads block forever waiting for each other (often due to inconsistent lock ordering).
- **Livelock**: threads keep running but make no progress (e.g., repeated retries and backoffs that collide).

Some tooling (including ThreadSanitizer) can detect certain lock-order issues and deadlock patterns, but you should still design tests that try to force these conditions.

3) Atomic misuse and memory-order bugs

Atomics prevent data races only when used correctly, and memory-order rules can be subtle. Many “it works on my machine” atomic bugs manifest only on different CPUs or under different compiler optimizations.

24.3.3 A professional concurrency testing stack

Layer A: deterministic structure in tests

Even without special tools, you can make tests more effective by:

- using barriers/latches so threads start at the same time,

- injecting yields/sleeps to widen the interleaving space,
- repeating the same test many times,
- minimizing external noise (I/O, logging, timing dependencies).

Example: turning a “maybe race” into a frequent race

```
#include <atomic>
#include <thread>
#include <vector>
#include <cassert>

static int g = 0; // intentionally non-atomic for demonstration

void race_example() {
  g = 0;
  std::atomic<bool> go{false};

  std::thread t1([&]{
    while (!go.load(std::memory_order_acquire)) {}
    for (int i = 0; i < 1'000'000; ++i) {
      g++;                    // data race: UB
      if ((i & 1023) == 0) std::this_thread::yield();
    }
  });

  std::thread t2([&]{
    while (!go.load(std::memory_order_acquire)) {}
    for (int i = 0; i < 1'000'000; ++i) {
      g++;                    // data race: UB
      if ((i & 1023) == 0) std::this_thread::yield();
    }
```

```
  });

  go.store(true, std::memory_order_release);
  t1.join();
  t2.join();
}

int main() {
  // Repeat to increase the chance of observable failure without tools.
  for (int r = 0; r < 200; ++r) race_example();
}
```

This may or may not visibly fail without instrumentation, but it is still undefined behavior because it has a data race.

Layer B: ThreadSanitizer (TSan) in CI

ThreadSanitizer is a dynamic data race detector implemented as compiler instrumentation plus a runtime library. It is one of the most practical tools for concurrency testing because it reports the race at the moment it occurs (with stack traces).

What TSan costs (plan your CI accordingly) Clang's documentation provides typical overhead ranges:

- slowdown: about 5x–15x,
- memory overhead: about 5x–10x.

And it documents important limitations:

- it maps a lot of virtual address space (which can interact unexpectedly with `ulimit`),
- libc/libstdc++ static linking is not supported.

Minimal Clang build/run pattern (conceptual)

```
# Example only; adjust for your build system:
clang++ -O1 -g -fsanitize=thread -fno-omit-frame-pointer test.cpp -o
↪  test_tsan
./test_tsan
```

Layer C: stress testing for interleavings

Stress testing tries to expand the explored scheduling space:

- high thread counts,
- long runs,
- frequent yields,
- deliberate contention on shared data structures,
- randomized delays (but keep test determinism when you can reproduce seeds).

A common production-quality approach is:

1. run a smaller stress suite under TSan (because overhead is high),
2. run a larger stress suite without TSan for longer duration,
3. run both repeatedly in CI and nightly pipelines.

Layer D: combine sanitizers where it makes sense

Concurrency bugs frequently cause memory corruption. So, in addition to TSan:

- run ASan+UBSan builds for memory errors and UB on single-threaded and lightly threaded suites,

- use TSan builds for race detection on dedicated concurrency suites.

(You generally do not combine TSan and ASan in the same binary; treat them as different instrumented builds.)

Layer E: platform/toolchain reality (Windows note)

On Windows with MSVC, AddressSanitizer is supported (Visual Studio 2019 16.9+), and Microsoft documents the supported `/fsanitize` options accordingly. However, ThreadSanitizer support in MSVC is not generally available as of late 2025; there are active requests for `/fsanitize=thread`.
Practical implication: for Windows-focused projects, a common workflow is:

- run ASan with MSVC (or clang-cl where appropriate) for memory issues,
- run TSan on Linux/macOS builds with Clang for data races.

24.3.4 Testing patterns for common concurrency hazards

Pattern 1: lock-order testing (deadlock prevention)

If your design requires a lock order, encode it in tests:

- create two threads that acquire locks in opposite order,
- make acquisition timing overlap using a barrier,
- the test should either prove the design avoids deadlock (e.g., uses `std::scoped_lock`) or fail quickly.

```
#include <mutex>
#include <thread>
```

```cpp
#include <atomic>

std::mutex a, b;

void safe_lock_both() {
  // scoped_lock locks both without deadlock (uses a deadlock-avoidance
  ↪  algorithm)
  std::scoped_lock lock(a, b);
  // critical section
}

void test_locking() {
  std::atomic<bool> go{false};

  std::thread t1([&]{ while (!go.load()) {} safe_lock_both(); });
  std::thread t2([&]{ while (!go.load()) {} safe_lock_both(); });

  go.store(true);
  t1.join();
  t2.join();
}
```

Pattern 2: invariants under concurrency

Test that invariants remain true when many operations interleave:

- push/pop correctness in queues,
- reference counting correctness,
- “no lost updates” in counters,
- ordering assumptions (when required).

Run these tests:

- repeatedly,
- with randomized operation mixes,
- under TSan to catch hidden races.

24.3.5 Interpreting results correctly

A passing suite is not a proof

Even with stress + TSan:

- you only check interleavings that occurred,
- you only check code paths that executed,
- you may miss bugs that require different environments, CPUs, or loads.

But in practice, a pipeline that includes repeated stress runs and TSan catches large classes of concurrency defects that unit tests miss.

Treat any data race report as critical

Because C++ defines data races as undefined behavior, a reported race should be fixed (or proven to be a false positive with a clear justification).

24.3.6 Checklist: a practical concurrency-testing policy

1. **Define the concurrency contract** (which functions are thread-safe; which locks protect what).

2. **Make tests interleaving-friendly**: barriers, yields, repetition.
3. **Run TSan in CI** on a dedicated build (expect 5x–15x slowdown).
4. **Run stress tests** (short under TSan; longer without TSan).
5. **Use multiple platforms/toolchains** when possible; do not rely on a single compiler/OS.
6. **Treat races as release blockers**: data races are UB in C++.

References (authoritative, no links)

- ISO/IEC 14882 (C++ Standard): concurrency, happens-before, data races, and undefined behavior rules.
- cppreference.com: data race implies undefined behavior; multithread and UB summaries.
- LLVM/Clang documentation: ThreadSanitizer overview, typical overhead, and limitations (virtual address space mapping; static linking unsupported).
- Google Sanitizers documentation: TSan as a data race detector and its relationship to the C++11 memory model.
- Microsoft Learn: MSVC AddressSanitizer support and `/fsanitize` options; current gap for `/fsanitize=thread`.

Part VIII

Modern C++ up to C++26

Chapter 25

The Evolution of Safety in C++

25.1 From C++11 to C++26

25.1.1 What changed, and why it matters for safety

Between C++11 and C++26, the language evolved from "you can build safe systems if you are careful" into a toolset that increasingly helps you **state** safety properties (types, constraints, contracts) and **check** them (compile time, runtime tooling, and hardened libraries).
This chapter summarizes the most important safety-relevant milestones, focusing on:

- memory and lifetime safety (ownership, bounds, initialization),
- concurrency correctness (data races, cancellation, structured threading),
- type safety (stronger types, safer interfaces),
- diagnosability (better errors, better debugging tools).

25.1.2 C++11: the safety foundation

C++11 is the turning point where C++ gained a formal memory model and modern "vocabulary" tools for safer code.

1) Concurrency becomes a defined model

C++11 introduced the memory model, atomics, and the core rules for multithreaded execution and data races. If your program has a data race, the behavior is undefined.

2) RAII becomes practical by default: move + smart pointers

`std::unique_ptr` (C++11) expresses unique ownership and reliably destroys the resource when leaving scope.

```
#include <memory>
#include <cstdio>

struct File {
  std::FILE* f{};
  explicit File(const char* path) : f(std::fopen(path, "rb")) {}
  ~File() { if (f) std::fclose(f); }
  File(const File&) = delete;
  File& operator=(const File&) = delete;
  File(File&&) = default;
  File& operator=(File&&) = default;
};

std::unique_ptr<File> open_file(const char* path) {
  auto p = std::make_unique<File>(path); // C++14 for make_unique, or
  ↪  allocate directly in C++11
  if (!p->f) return nullptr;
```

```
  return p;
}
```

Even when you use a raw handle internally, the lifetime becomes explicit and exception-safe.

3) Type safety improvements you feel every day

C++11 made several "make wrong code harder" features mainstream, including strongly scoped enums, `nullptr`, `noexcept`, and compile-time assertions (`static_assert`).

```
enum class Mode { read_only, read_write }; // no implicit conversion to
↪  int

void configure(Mode m);

static_assert(sizeof(void*) == 8, "This build expects 64-bit pointers");
```

25.1.3 C++14: small changes that reduce error surface

C++14 is a "polish" release, but a few changes are directly safety-relevant.

1) `std::make_unique`: safer ownership construction

C++14 standardized `std::make_unique`. It reduces repetition, improves exception safety patterns, and helps avoid mismatched `new`/`delete`. Compiler-support tables list `std::make_unique` as a C++14 library feature.

```
#include <memory>

struct Node { int x; };

auto p = std::make_unique<Node>(); // safer than raw new in most
↪  codebases
```

25.1.4 C++17: vocabulary types and safer interfaces

C++17 strengthened the standard library with types that prevent entire classes of bugs by making intent explicit.

1) `std::optional` and friends: "absence" without sentinel hacks

When a value may be missing, `std::optional<T>` makes the possibility explicit, instead of forcing sentinel values or out-parameters.

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

std::optional<int> parse_int(const std::string& s);

if (auto v = parse_int("42")) {
  // safe: value exists
}
```

2) `std::string_view`: safer non-owning string parameters

`std::string_view` encourages APIs that avoid unnecessary allocations while clearly signaling non-ownership (you still must respect lifetime rules).

3) `[[nodiscard]]`: make ignored errors noisy

`[[nodiscard]]` encourages callers to handle returned status values.

```
[[nodiscard]] bool save() noexcept;

void f() {
  save(); // compilers may warn: ignoring a nodiscard result
}
```

25.1.5 C++20: constraints and structured concurrency tools

C++20 delivered major features (concepts, ranges, coroutines, modules) and important concurrency utilities.

1) Concepts: safer templates by expressing requirements

Concepts allow templates to state constraints directly, improving correctness and error messages by rejecting invalid types earlier (at the right abstraction level).

```
#include <concepts>

template <std::integral T>
T add_checked(T a, T b) {
  return a + b;
}
```

2) `std::span`: safer views over contiguous memory

`std::span<T>` is a lightweight view that bundles pointer + length, which helps avoid "pointer without size" APIs and makes bounds reasoning easier.

```
#include <span>
#include <cstddef>

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

3) `std::jthread` and stop tokens: structured thread lifetime

`std::jthread` joins on destruction and supports cooperative cancellation via `std::stop_token`.

```
#include <thread>
#include <stop_token>

void worker(std::stop_token st);

void run() {
  std::jthread t(worker); // auto-joins (RAII for threads)
  // request stop via t.request_stop() when needed
}
```

This reduces “forgot to join/detach” bugs and encourages orderly shutdown.

25.1.6 C++23: safer error propagation and better diagnostics

C++23 continues the pattern of adding vocabulary types for correctness and making failures easier to observe.

1) `std::expected`: explicit success-or-error return

`std::expected<T,E>` standardizes a structured alternative to raw error codes for recoverable failures, avoiding ambiguous sentinel returns.

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

enum class Err { bad_format, out_of_range };
```

```
std::expected<int, Err> parse_int(std::string_view s);

auto r = parse_int("123");
if (!r) {
  // handle r.error()
} else {
  int value = *r;
}
```

2) `<stacktrace>`: standard stack traces for debugging

C++23 introduces the diagnostics header `<stacktrace>` with `std::stacktrace` facilities.
This improves post-failure investigations and reduces dependence on platform-specific APIs.

25.1.7 C++26: contracts, hardening, and safety-oriented library additions

C++26 is the next standard in progress; many features are already adopted into the working draft and tracked by cppreference.

1) Contracts: stating preconditions, postconditions, and assertions

WG21 adopted Contracts for C++26 (P2900) into the working draft.
Contracts let you write:

- **preconditions** (requirements on inputs/state),
- **postconditions** (guarantees on outputs/state),
- **`contract_assert`** statements (general assertions).

A key safety detail: contract evaluation semantics are **implementation-defined** and can vary (including during constant evaluation).

```
int divide(int a, int b)
  pre(b != 0)
{
  return a / b;
}

void g(int* p)
{
  contract_assert(p != nullptr);
  *p = 1;
}
```

C++26 also defines how contract violations are handled, including a program-level handler named `::handle_contract_violation` with a `std::contracts::contract_violation` argument.

2) Standard library hardening (first use of contracts)

C++26 includes an initial portable standard library hardening effort adopted for C++26 (as reported by WG21 meeting trip reports), turning some common library undefined behaviors into contract violations under the hardening model.

3) Concurrency and reclamation support: hazard pointers and RCU

cppreference lists new C++26 headers `<hazard_pointer>` and `<rcu>`, which target safe memory reclamation patterns in concurrent programs.
This matters because many high-performance lock-free structures are not “hard” due to atomics alone, but due to safe reclamation of nodes.

4) Execution control library and other safety-adjacent facilities

cppreference lists an **Execution control library (C++26)** and several new headers that are relevant to robust systems work, including: `<contracts>`, `<debugging>`, `<inplace_vector>`, `<simd>`, `<linalg>`, and C compatibility headers like `<stdchkint.h>`.

5) A small but important library building block: `std::is_within_lifetime`

cppreference lists `std::is_within_lifetime` as a C++26 library feature. This reflects a broader direction: make object lifetime reasoning more expressible and analyzable (while still respecting that C++ is not a fully memory-safe language).

25.1.8 Cross-version safety trend: from "discipline" to "expressed intent"

Across these standards, you can see a consistent direction:

1) Make ownership and lifetime explicit

- C++11 smart pointers and move semantics establish the ownership vocabulary.
- C++20 `std::span` improves non-owning interface safety.
- C++26 adds more library support for lifetime- and concurrency-related correctness.

2) Detect errors earlier

- C++17 `[[nodiscard]]` reduces ignored-result bugs.
- C++20 concepts improve template correctness at compile time.

- C++26 contracts allow expressing correctness properties that can be observed/enforced under implementation-defined semantics.

3) Improve diagnosability and operational safety

- C++23 adds `std::expected` for clearer error handling and `<stacktrace>` for better debugging.
- C++26 adds `<debugging>` and contract-related reporting pathways.

25.1.9 What to standardize on today (C++11 → C++26 mindset)

A practical, modern safety baseline looks like this:

- **Ownership:** default to RAII and smart pointers; avoid raw owning pointers.
- **Interfaces:** prefer `std::span`, `std::string_view`, and strong types; avoid "pointer + implicit size" APIs.
- **Errors:** prefer `std::expected` for recoverable errors where exceptions are not desired; apply `[[nodiscard]]` to important return values.
- **Threads:** prefer `std::jthread` for RAII thread lifetime; test under race detectors.
- **Forward-looking:** adopt contracts and library hardening when your toolchain supports them and your build modes define a clear policy for enforcement vs observation.

25.2 Problems that were solved

25.2.1 Overview: safety progress as problem elimination

From C++11 to C++26, C++ did not become a fully memory-safe language, but it *did* remove (or dramatically reduce) several long-standing categories of practical safety failures by:

- standardizing safer vocabulary types (ownership, absence, views),
- formalizing concurrency rules and providing safer threading utilities,
- adding language mechanisms to express intent (constraints, contracts),
- making misuse harder to compile or easier to diagnose.

This section lists concrete problems that were effectively solved (or made substantially less common) by modern C++ features and professional guidance.

25.2.2 Problem 1: "Who owns this pointer?" (ownership ambiguity)

What was common before

Large codebases frequently used raw pointers for ownership transfer and lifetime management, which created:

- leaks on early returns/exceptions,
- double deletes,
- use-after-free due to unclear ownership boundaries.

What modern C++ solved

- **Unique ownership** is expressed directly with `std::unique_ptr` (C++11), which owns and disposes of the object when leaving scope.

- The C++ Core Guidelines made the rule explicit: owning raw pointers/references are dangerous, and owners should be identified using ownership types and conventions.

```
#include <memory>

struct Image { /* ... */ };

// Clear ownership: caller owns the result.
std::unique_ptr<Image> load_image();

void f() {
  auto img = load_image(); // auto cleanup on all paths
  // ...
} // img destroyed here, resource released
```

This does not magically prevent all lifetime bugs, but it solves a major class of ownership ambiguity by making ownership part of the type.

25.2.3 Problem 2: “Error ignored” bugs (unchecked results)

What was common before

APIs returning error indicators were often ignored accidentally, especially in large systems where reviews miss a single unchecked return value.

What modern C++ solved

C++17 introduced `[[nodiscard]]` to discourage discarding important results by making it diagnosable at compile time.

```
[[nodiscard]] bool write_all() noexcept;
```

```
void g() {
  write_all(); // compilers may warn: nodiscard result ignored
}
```

This does not force correctness, but it makes a common class of silent failures visibly wrong.

25.2.4 Problem 3: "Sentinel values" and unsafe out-parameters for failures

What was common before

Functions used:

- special sentinel returns (e.g., `-1`, `nullptr`),
- output parameters with partial initialization on failure,
- ad-hoc error code conventions that were easy to misuse.

These patterns frequently caused:

- missed error handling,
- partially initialized outputs,
- confusing contracts (what does `-1` mean here?).

What modern C++ solved

- C++17 `std::optional<T>` standardizes "value may be absent".
- C++23 `std::expected<T,E>` standardizes "value or error", and is *never valueless*.

```
#include <expected>
#include <string_view>

enum class ParseErr { bad_format, out_of_range };

std::expected<int, ParseErr> parse_int(std::string_view s);

void h() {
  auto r = parse_int("123");
  if (!r) {
    // handle r.error()
  } else {
    int v = *r;
    (void)v;
  }
}
```

This directly eliminates many ambiguous error conventions by encoding success/failure in the return type.

25.2.5 Problem 4: "Pointer without size" APIs (bounds mistakes)

What was common before

A very common pre-modern interface looked like:

```
void process(const int* p); // size lost; caller and callee must
↪  coordinate implicitly
```

This style frequently led to:

- out-of-bounds reads/writes,
- mismatched length assumptions,

- fragile APIs that are hard to review.

What modern C++ solved

C++20 `std::span<T>` is a standard view over a contiguous sequence, bundling pointer + length into a single parameter.

```
#include <span>

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

This does not guarantee bounds safety by itself (you can still index wrongly), but it solves the interface-level loss of size information that historically enabled many bounds bugs.

25.2.6 Problem 5: "Forgot to join" threads and unsafe thread lifetime

What was common before

Managing `std::thread` manually often caused:

- threads left unjoined on early returns/exceptions,
- accidental `detach()` and lifetime hazards,
- shutdown code that is difficult to reason about.

What modern C++ solved

C++20 introduced `std::jthread`:

- it automatically joins on destruction,
- it supports cooperative cancellation in certain situations.

```
#include <thread>
#include <stop_token>

void worker(std::stop_token st) {
  while (!st.stop_requested()) {
    // do work
  }
}

void run() {
  std::jthread t(worker);    // joins automatically on scope exit
  t.request_stop();          // cooperative cancellation request
} // no forgotten join
```

This solves a real production class of thread-lifetime bugs by making thread lifetime RAII-managed.

25.2.7 Problem 6: “Template misuse is discovered too late”

What was common before

Before concepts, templates often failed with:

- errors far from the actual misuse,
- long, low-signal diagnostics.

What modern C++ solved

C++20 concepts allow templates to state requirements explicitly, rejecting invalid types earlier (at the right abstraction boundary). This is a major reduction in template misuse cost and a practical correctness improvement in generic libraries.

```
#include <concepts>

template <std::integral T>
T add(T a, T b) {
  return a + b;
}
```

While error readability still depends on toolchains, the core problem (requirements are implicit and discovered deep in instantiation) is fundamentally addressed by language support for constraints.

25.2.8 Problem 7: "Contracts live only in comments"

What was common before

Preconditions and postconditions were often informal:

- stated in documentation,
- enforced by debug-only asserts,
- inconsistently checked across builds.

This led to:

- unsafe recovery attempts after invalid input,

- silent propagation of corrupted state,
- inconsistent behavior between debug and release.

What modern C++ solved

C++26 contracts (as specified by the accepted Contracts proposal and reflected in cppreference) introduce a standard way to write preconditions, postconditions, and contract assertions.

```
int divide(int a, int b)
  pre(b != 0)
{
  return a / b;
}

void set_nonnull(int* p) {
  contract_assert(p != nullptr);
  *p = 1;
}
```

Contracts do not guarantee memory safety, and their enforcement model is implementation-defined, but they solve a long-standing problem: expressing contracts in the language instead of only in comments and ad-hoc macros.

25.2.9 What remains unsolved (and why that matters)

Even with these improvements:

- C++ still permits undefined behavior,
- raw pointers still exist and can be misused,

- safe concurrency still requires careful design,
- tools (sanitizers, static analysis) are still essential.

So the correct conclusion is not “C++ is safe now”, but:

> Modern C++ removed many historically common safety traps by making intent explicit, making misuse diagnosable earlier, and standardizing safer vocabulary types.

25.3 Problems that remain

25.3.1 Context: modern C++ is safer, not automatically safe

C++11–C++26 added many tools that *help* write safer systems, but the language still permits constructs that can trigger undefined behavior, lifetime violations, and concurrency failures. The remaining problems are largely structural: C++ still prioritizes performance, compatibility, and expressive power, so many safety properties are *not enforced by default*. This section summarizes the key safety problems that remain even in high-quality C++26-era code.

25.3.2 1) Undefined behavior still exists (and can invalidate testing)

Undefined behavior (UB) remains a core reality: when UB occurs, the implementation is not required to diagnose it, and the program is not required to do anything meaningful.
UB is not just “a crash risk”; it breaks reasoning:

- a program may appear correct under tests and still be wrong,
- optimization can transform code based on the assumption that UB never occurs,

- the symptom may appear far from the bug.

```
#include <cstddef>

int read_bad(const int* p, std::size_t n) {
  // UB if n == 0: dereference past the valid range (even if it "seems to
  ↪  work").
  return p[n - 1];
}
```

The language does not promise a warning or a predictable failure; correctness must be engineered.

25.3.3 2) Lifetime safety is still a human responsibility

Modern C++ gives strong tools (RAII, smart pointers, standard containers), but lifetime errors are still possible whenever raw pointers/references outlive what they refer to.
The C++ Core Guidelines explicitly call out multiple problem areas that remain sources of serious failures (crashes and security violations), including casts, array decay, and range errors.

```
#include <string_view>
#include <string>

std::string_view dangling_view() {
  std::string s = "hello";
  return std::string_view(s); // view refers to s's buffer
} // s destroyed here -> returned view dangles (UB if used)
```

Key point: safer vocabulary types reduce risk, but they do not remove the need to reason about lifetimes, especially for non-owning views.

25.3.4 3) Bounds safety is still not guaranteed

C++ still allows:

- pointer arithmetic,
- unchecked indexing,
- array-to-pointer decay,
- implicit assumptions about sizes and ranges.

These remain major sources of memory safety vulnerabilities. The Core Guidelines explicitly list array decay and range errors among the major problem areas.

```
#include <vector>

int sum_bad(const std::vector<int>& v) {
  int s = 0;
  for (std::size_t i = 0; i <= v.size(); ++i) { // off-by-one: UB when i
  ↪  == size()
    s += v[i];
  }
  return s;
}
```

Even with `std::span`, you can still index incorrectly; the language does not automatically enforce bounds safety.

25.3.5 4) Data races are still UB, and testing cannot enumerate interleavings

C++11 formalized the memory model and made a hard rule: if a data race occurs, the behavior is undefined.

This remains a central risk because:

- races can be rare and non-deterministic,
- a passing test suite is not proof of race-freedom,
- one missing synchronization edge can corrupt memory.

```
#include <thread>

int shared = 0;

void f() { shared++; } // data race when run concurrently

void run() {
  std::thread t1(f), t2(f), t3(f);
  t1.join(); t2.join(); t3.join(); // UB: shared++ is not atomic
}
```

Tooling like ThreadSanitizer helps, but the language itself still allows race-prone designs.

25.3.6 5) Atomics remain easy to misuse

Even when you avoid data races by using atomics, correctness is not automatic:

- the choice of memory order can be wrong,
- correct publication patterns are subtle,
- performance-driven “relaxed” code can silently break invariants.

The standard memory-order model is powerful, but the complexity remains.

```
#include <atomic>

std::atomic<int*> gptr{nullptr};
int data = 0;

void publish_bad() {
  data = 42;
  gptr.store(&data, std::memory_order_relaxed); // may not safely publish
  ↪ 'data'
}

int read_bad() {
  int* p = gptr.load(std::memory_order_relaxed);
  return p ? *p : 0; // can observe pointer without proper
  ↪ synchronization guarantees
}
```

The existence of `std::atomic` does not eliminate the need for careful, documented memory-order reasoning and review.

25.3.7 6) Contracts help express intent, but they do not make C++ memory-safe

C++26 contracts provide a standard way to write preconditions, postconditions, and contract assertions, and define a contract-violation handler mechanism.
However, important limitations remain:

- contract evaluation and enforcement modes are **implementation-defined** (policy varies by toolchain/build),
- contracts cannot automatically prevent UB caused by misuse of low-level operations unless the program actually checks and responds,

- contracts do not remove the need for careful API design (ownership, lifetime, bounds).

```
int at(int* p, int i, int n)
  pre(p != nullptr)
  pre(0 <= i && i < n)
{
  return p[i]; // still depends on caller respecting lifetime and correct
   ↪  'n'
}
```

Contracts improve clarity and diagnosability, but they are not a complete safety system.

25.3.8 7) Unsafe escape hatches remain necessary (and risky)

C++ still requires (and continues to standardize support for):

- low-level casts,
- type punning and object representation access in carefully constrained forms,
- manual memory and custom allocators,
- ABI boundaries and "extern C" interoperability.

The Core Guidelines highlight unions and casts as problem areas precisely because real systems code needs these mechanisms, but they are also a major source of safety issues. These escape hatches are not "bad"; they are *power tools* that require strict discipline, code review, and tooling support.

25.3.9 8) Legacy code and ecosystem constraints do not disappear

A major practical problem is not the C++26 feature set, but reality:

- large systems have legacy code written with older idioms,
- some platforms cannot enable sanitizers everywhere,
- some domains forbid exceptions or RTTI,
- third-party libraries may not follow modern safety practices.

As the ISO C++ committee chair has emphasized in public discussion, C++ safety must be considered in the broader context of real-world software and deployment constraints, not as a single feature switch.

25.3.10 Practical conclusion: the remaining problems define the safety workflow

Because these problems remain, professional safety practice in C++ requires:

- strong ownership and lifetime conventions (RAII, clear interfaces),
- aggressive testing under sanitizers and race detectors where possible,
- static analysis and guideline enforcement,
- careful concurrency design and review (especially memory order),
- well-defined build-mode policies for contracts/assertions.

Modern C++ reduces the error surface, but it still demands engineering rigor.

Chapter 26

Writing Future-Proof C++ Code

26.1 Avoiding Deprecated Idioms

26.1.1 Why deprecation matters in safety-focused C++

Deprecation is an explicit signal that a feature or facility is on a path toward removal, or is considered harmful enough that new code should not depend on it. For a safety handbook, the practical reasons are simple:

- **Security and correctness**: many deprecated facilities exist because they are error-prone, underspecified, or easily misused.
- **Portability**: different standard library implementations and compilers phase out deprecated behavior at different speeds.
- **Future toolchains**: code that compiles cleanly today may fail with "-std=c++2c"/"-std=c++26" (or newer) when removals take effect.

26.1.2 What “deprecated” means in C++ (and what it does not)

Standard deprecation

A facility can be marked deprecated by the C++ Standard (typically in the deprecation annex and/or in normative wording). Compilers and standard libraries commonly diagnose usage with warnings when warnings are enabled.

Removed vs. deprecated

- **Deprecated**: still present (for now), but using it is discouraged and may emit warnings.
- **Removed**: no longer part of the Standard; it may stop compiling (or require non-standard compatibility modes).

“Deprecated idiom” in practice

Engineers also use the term to mean: *constructs that are not necessarily formally deprecated by the Standard, but are obsolete for professional, safe, modern C++*. In this section, we cover both categories, while clearly distinguishing between them.

26.1.3 C++26 removals you should avoid *now*

Even if your current toolchain still accepts these patterns, new code should not introduce them. Treat them as already gone.

Comparisons of C-style arrays (removed after C++20 deprecation)

C++20 deprecated comparing arrays directly (e.g., two arrays with ==), and C++26 removes that deprecated behavior .

Bad (historically compiled, now removed)

```
int a[3] = {1,2,3};
int b[3] = {1,2,3};

bool same = (a == b); // deprecated in C++20, removed in C++26
```

Good: compare elements explicitly

```
#include <algorithm>
#include <array>

std::array<int,3> a{1,2,3};
std::array<int,3> b{1,2,3};

bool same = (a == b); // well-defined, value comparison
// or: std::equal(a.begin(), a.end(), b.begin());
```

Good: compare views if you must keep raw arrays

```
#include <algorithm>
#include <span>

int a[3] = {1,2,3};
int b[3] = {1,2,3};

bool same = std::ranges::equal(std::span{a}, std::span{b});
```

Arithmetic conversions involving enumerations (removed in C++26)

Implicit arithmetic conversions from an enumeration to arithmetic types were deprecated and are removed in C++26 . Relying on them is a common source of bugs (especially when enum values are used as bitmasks or indices without explicit intent).

Bad: implicit conversion

```
enum Color { red = 1, green = 2, blue = 4 };

int x = red + 1; // obsolete style; removed path in C++26
```

Good: be explicit about representation

```
#include <type_traits>

enum class Color : int { red = 1, green = 2, blue = 4 };

int x = static_cast<std::underlying_type_t<Color>>(Color::red) + 1;
```

Better: avoid arithmetic on enums entirely when it is not a true numeric domain If the enum represents a set of named states, do not treat it as a number. Use a separate integer index type, or a dedicated bitmask type.

Deprecated volatile semantics removed from C++26

Several deprecated features related to `volatile` are removed in the C++26 time frame . This matters because `volatile` is still widely misunderstood: **it is not a concurrency primitive** and does not establish inter-thread ordering.

Bad: using volatile for cross-thread communication

```
volatile bool ready = false;
int data = 0;

// Thread A:
data = 42;
ready = true;
```

```
// Thread B:
while (!ready) { }    // does not create a correct happens-before
↪  relationship
int x = data;         // data race / undefined behavior
```

Good: atomics for signaling

```
#include <atomic>

std::atomic<bool> ready{false};
int data = 0;

// Thread A:
data = 42;
ready.store(true, std::memory_order_release);

// Thread B:
while (!ready.load(std::memory_order_acquire)) { }
int x = data; // safe (with this pattern)
```

Rule

- Use `volatile` for **memory-mapped I/O** and similar hardware-facing constraints.
- Use `std::atomic` / mutexes / condition variables for **concurrency**.

Library removals in C++26: update your code before the toolchain forces you

Unicode conversion facets in `<codecvt>` removed The `<codecvt>` header is deprecated since C++17 and removed in C++26 . If your code uses `std::codecvt_utf8`, `wstring_convert`, or similar legacy facilities, plan a migration.

Practical guidance (no single standard replacement) Unicode text handling is inherently complex. For future-proof code:

- Keep text in UTF-8 (`std::string` / `std::u8string`) end-to-end where possible.
- Convert at system boundaries using your platform APIs (e.g., OS services), and keep conversions localized and well-tested.
- Avoid ad-hoc conversions sprinkled throughout the codebase.

`std::allocator<T>::is_always_equal` removed The deprecated typedef `std::allocator<T>::is_always_equal` is removed in C++26 .

Migration Use `std::allocator_traits<Alloc>::is_always_equal` instead:

```
#include <memory>
#include <type_traits>

template<class Alloc>
constexpr bool alloc_always_equal_v =
  std::allocator_traits<Alloc>::is_always_equal::value;
```

`basic_string::reserve()` overload with no arguments removed The no-argument overload `s.reserve()` was deprecated (C++20) and removed in C++26 . It historically implied “shrink or fit” behavior in some implementations, which is exactly the kind of ambiguity modern C++ tries to avoid.

Migration Use intent-revealing operations:

```
#include <string>

std::string s = "big data";

// Want capacity for growth:
s.reserve(4096);

// Want to request shrinking:
s.shrink_to_fit();
```

26.1.4 C++23 deprecations still common in codebases

Some facilities are not removed (yet) but are formally deprecated and should be avoided in new code.

`std::aligned_storage` and `std::aligned_union` (deprecated in C++23)

These are deprecated in C++23 . They frequently appear in “manual storage” patterns that are easy to get wrong (lifetime, alignment, exception safety).

Preferred alternatives

- Use `std::variant` for type-safe storage of one-of-many object types.
- Use `std::optional<T>` for “maybe present” storage of a single type.
- Use `std::byte` buffers + placement new only inside carefully reviewed low-level components, with explicit lifetime management.

`std::numeric_limits::has_denorm` (deprecated in C++23)

Deprecated in C++23 . Relying on it for numeric behavior tends to be non-portable across targets and FP modes.

Guidance When denormals matter for performance or correctness, handle it as a platform/FP-environment policy, not as a single boolean trait.

26.1.5 Safety-driven "deprecated idioms" even when the Standard does not deprecate them

This subsection focuses on patterns that modern C++ safety guidance treats as obsolete, because safer and clearer alternatives exist. These are strongly aligned with the C++ Core Guidelines and professional code analysis tools .

Owning raw pointers and manual `new`/`delete`

Bad: unclear ownership and leaks on early returns/exceptions

```
struct Widget { /* ... */ };

Widget* make_widget() {
  Widget* p = new Widget();
  // more code...
  return p; // Who deletes? Easy to leak, easy to double-delete.
}
```

Good: express ownership directly

```
#include <memory>
```

```cpp
struct Widget { /* ... */ };

std::unique_ptr<Widget> make_widget() {
  return std::make_unique<Widget>();
}
```

Rule

- Use `std::unique_ptr<T>` for unique ownership.
- Use `std::shared_ptr<T>` only when you truly need shared lifetime and can justify the overhead/complexity.
- Use raw pointers or references for **non-owning** access, and document the lifetime contract.

C-style casts

C-style casts can silently perform a `static_cast`, `const_cast`, and even `reinterpret_cast`, and are harder for tools to reason about.

Bad

```cpp
double d = 3.14;
int x = (int)d; // narrowing, silently truncates
```

Good

```cpp
double d = 3.14;
int x = static_cast<int>(d); // explicit intent, tool-friendly
```

Best for safety: avoid narrowing conversions

```
#include <cstddef>
#include <limits>
#include <stdexcept>

int safe_double_to_int(double d) {
  if (d < static_cast<double>(std::numeric_limits<int>::min()) ||
      d > static_cast<double>(std::numeric_limits<int>::max())) {
    throw std::out_of_range("narrowing conversion");
  }
  return static_cast<int>(d);
}
```

`NULL` and integer-based null conventions

Bad

```
void f(int);
void f(const char*);

f(NULL); // can bind to int overload in some cases; ambiguous intent
```

Good

```
void f(int);
void f(const char*);

f(nullptr); // always pointer null
```

Preprocessor macros as constants or functions

Macros bypass the type system, scoping, and debugging support.

Bad

```
#define BUF_SIZE 4096
#define SQUARE(x) ((x)*(x))
```

Good

```
#include <cstddef>

inline constexpr std::size_t buf_size = 4096;

template<class T>
constexpr T square(T x) noexcept { return x * x; }
```

`volatile` as “thread-safety” (again)

Because this confusion keeps resurfacing in real systems code, it is worth stating as a policy:

> **If the goal is inter-thread synchronization, `volatile` is the wrong tool.**
> Use `std::atomic`, locks, and condition variables.

26.1.6 Migration patterns you can apply mechanically

Replace deprecated facilities with clear intent

- `s.reserve()` → `s.shrink_to_fit()` or `s.reserve(n)` depending on intent .
- `Alloc::is_always_equal` → `std::allocator_traits<Alloc>::is_always_equal` .
- `<codecvt>` → localized, boundary-only conversion strategy .

- raw array comparisons → `std::array` or `std::span` + `std::ranges::equal`.

Prefer "types that carry the rule"

A future-proof refactor often succeeds when you choose types that encode constraints:

- `std::span<T>` for non-owning views with bounds.
- `std::optional<T>` for "maybe present".
- `std::variant<...>` for safe unions.
- `std::unique_ptr<T>` for owning pointers.

26.1.7 Enforcement: keep deprecated usage out of the codebase

Compile-time diagnostics

Enable deprecation warnings and make them actionable in CI:

- Treat deprecation warnings as errors in new/modified code (policy-based).
- Keep a short-term escape hatch for legacy modules during migration, but track it.

Static analysis and guideline checks

Use rule sets aligned with the C++ Core Guidelines to continuously detect obsolete and unsafe idioms . In practice, the most valuable checks are those that:

- flag raw owning pointers and manual lifetime management,
- detect narrowing conversions,

- enforce modern casts and null usage,
- prefer RAII-based cleanup for resources and locks.

Feature-test macros for forward compatibility

When you must write portable code across multiple standard library versions, prefer standard feature-test macros to detect availability rather than guessing by compiler version . This keeps code correct as vendors ship features on different schedules.

26.1.8 Summary checklist

- Do not add new code that depends on features removed in C++26 (array comparisons, enum arithmetic conversions, removed volatile features, removed library APIs) .
- Treat formal deprecations (e.g., C++23 deprecations) as "must migrate" in actively maintained code .
- Prefer modern types that encode safety and intent: `std::array`, `std::span`, `std::optional`, `std::variant`, `std::unique_ptr`.
- Use CI + warnings + guideline-based static analysis to prevent regressions .

26.2 Designing for evolution

26.2.1 What it means to design for evolution

Future-proof C++ is not "guessing the next Standard." It is designing software that can *change safely* when requirements, toolchains, platforms, and the C++ Standard evolve.
In practice, designing for evolution means:

- **Change is localized**: a change in one subsystem does not force changes everywhere.
- **Interfaces are explicit**: ownership, lifetime, threading, and error contracts are clear.
- **Build and runtime behavior are controlled**: you can adopt new features without breaking old deployments.
- **Upgrades are gradual**: you can mix old and new components and migrate incrementally.

The C++26 feature set and direction strongly reinforces this mindset: the committee has completed the feature freeze for C++26 and is moving into fit-and-finish, so "designing for evolution" should assume real-world C++26 deployment across multiple compiler/library timelines .

26.2.2 Architectural levers that keep change safe

Separate stable interfaces from volatile implementations

A stable boundary allows internal refactoring without forcing rebuilding or rewriting users of the boundary. In C++, this is both an **API** issue (source compatibility) and often an **ABI** issue (binary compatibility).

Guideline-based principle Good interfaces are the most important organizing force in a codebase; they are contracts between users and implementations .

Technique: Pimpl for ABI stability The Pimpl (pointer-to-implementation) idiom hides representation details behind an opaque pointer, keeping the interface stable even when the implementation changes .

```
#include <memory>
#include <string>

class Logger {
public:
  Logger();                          // stable ABI surface
  ~Logger();                         // out-of-line
  Logger(Logger&&) noexcept;
  Logger& operator=(Logger&&) noexcept;

  void info(std::string_view msg);
  void set_level(int level);

private:
  struct Impl;                       // incomplete type
  std::unique_ptr<Impl> p_;          // opaque pointer
};
```

Practical notes:

- Keep destructor and move operations *out-of-line* in a .cpp to avoid exposing `Impl` details.
- Use Pimpl mainly for libraries with ABI concerns. For internal code, prefer simpler designs unless you truly need ABI stability.

Depend on abstractions, not concretions (but keep them testable)

Evolution is easier when you can swap components. In modern C++ this usually means:

- stable pure interfaces for cross-module boundaries,
- value types for local composition,

- dependency injection where it improves testability (not everywhere).

Prefer minimal interfaces Smaller interfaces evolve more safely. When you must grow an interface, do so via *new* functions/types rather than changing meanings of old ones.

26.2.3 Compatibility strategy: source, ABI, and behavioral stability

Define your compatibility target explicitly

When you promise "compatibility," be concrete:

- **Source compatibility**: existing code recompiles.
- **Binary compatibility (ABI)**: old binaries keep running with a new library.
- **Behavioral compatibility**: observable behavior stays compatible, including error handling and concurrency semantics.

If ABI stability matters, design your public types to minimize layout changes:

- avoid exposing data members,
- avoid inline functions that bake behavior into user binaries unless that behavior is truly stable,
- prefer non-virtual public APIs with internal polymorphism (often via Pimpl), or carefully versioned virtual interfaces.

Versioning: prefer additive change

Additive change is the safest:

- add new overloads (without breaking overload resolution),

- add new types,
- add new enumerators only if you control all switch sites (otherwise provide a default case).

Example: forward-compatible enum handling

```
enum class Mode { fast, safe, balanced /* future modes may appear */ };

int compute(Mode m) {
  switch (m) {
    case Mode::fast:     return 1;
    case Mode::safe:     return 2;
    case Mode::balanced: return 3;
  }
  // Defensive: future-proof against new enumerators arriving from newer
  ↪  headers.
  return 2; // choose a conservative default
}
```

This is a deliberate tradeoff: you lose "must handle all cases" enforcement, but gain resilience when code and headers can be out of sync.

26.2.4 Design contracts that can evolve safely

Document preconditions/postconditions today, adopt contracts as toolchains mature

Even before you use a language-level Contracts facility, treat interface rules as *first-class*:

- preconditions (what callers must guarantee),
- postconditions (what the function guarantees on success),

- failure modes (errors, exceptions, partial effects),
- concurrency rules (thread-safe? re-entrant? requires external synchronization?).

C++26 adopts a Contracts facility proposal that provides syntax for preconditions, postconditions, and assertions, plus a range of runtime semantics . Designing for evolution means your contracts should be *stable statements of intent* even as the enforcement mechanism evolves (comments → debug checks → contracts → tooling/auditing).

A practical migration-friendly style

```
#include <cassert>
#include <span>
#include <cstddef>

std::size_t sum_nonneg(std::span<const int> xs) {
  // precondition: all elements are >= 0
  for (int v : xs) { assert(v >= 0); }

  std::size_t sum = 0;
  for (int v : xs) { sum += static_cast<std::size_t>(v); }
  // postcondition: sum >= 0 (always true for size_t), but could express
  ↪  range guarantees here
  return sum;
}
```

Later, such checks can be mapped to contract syntax while preserving the logical contract.

Design for “profiles” and safety modes

C++ safety direction increasingly emphasizes *enforceable profiles* (bounds/lifetime/type-style constraints) that can be enabled and checked by tools and (in some proposals) implementations . To design for evolution:

- keep unsafe low-level code behind narrow boundaries,
- make safe-by-default APIs (spans, views, RAII, explicit ownership),
- keep opt-in “unsafe escape hatches” isolated and reviewable.

26.2.5 Error handling that survives change

Make failure explicit at boundaries

Future-proof error handling avoids “hidden conventions” (magic values, global error state, partially initialized outputs). Instead, choose a boundary policy:

- exceptions for exceptional failure in higher-level code,
- explicit error returns for low-level or performance-critical boundaries,
- never mix both without a documented rule.

Prefer typed results for explicit failures

C++23 standardizes `std::expected` to return either a value or an error, which improves clarity and makes evolution easier than ad-hoc status codes .

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

enum class ParseErr { empty, invalid };

std::expected<int, ParseErr> parse_int(std::string_view s) {
  if (s.empty()) return std::unexpected(ParseErr::empty);
  int v = 0;
```

```
  for (char c : s) {
    if (c < '0' || c > '9') return std::unexpected(ParseErr::invalid);
    v = v * 10 + (c - '0');
  }
  return v;
}
```

Evolution benefit: you can extend the error type (or wrap it) without changing the "success path" signature, and without inventing new sentinel values.

26.2.6 Concurrency design that can adopt new standards incrementally

Avoid baking thread models into every interface

If you expose threads, mutexes, or scheduling details everywhere, evolution becomes expensive. Instead:

- keep concurrency policies at subsystem boundaries,
- pass executors/schedulers (or handles) rather than raw threads,
- express cancellation and lifetimes explicitly.

C++26 introduces a standard concurrency framework commonly referred to as senders/receivers under `std::execution`. Designing for evolution today means you should avoid interface patterns that make adopting structured async and cancellation impossible later.

Migration-friendly async boundary

```
#include <functional>
#include <string>
```

```
// Do: define a boundary in terms of "schedule work" not "create
↪  threads".
using Task = std::function<void()>;

struct Scheduler {
  virtual ~Scheduler() = default;
  virtual void schedule(Task t) = 0;
};
```

Later, the implementation can evolve to use newer standard execution facilities while keeping the boundary stable.

26.2.7 Build and module boundaries that tolerate toolchain evolution

Use feature detection, not guesses

When adopting new language/library features across a range of compilers, prefer:

- standard feature-test macros where available,
- narrowly scoped `#ifdef` fences around *implementation details*,
- a single compatibility layer instead of scattered conditional compilation.

```
#include <version>

#if defined(__cpp_lib_expected) && __cpp_lib_expected >= 202211L
  #include <expected>
  template<class T, class E>
  using expected_t = std::expected<T, E>;
#else
  // fallback: your local polyfill or alternative error strategy
#endif
```

Design for gradual adoption of reflection-based generation

C++26 static reflection is being standardized with a reduced initial feature set proposal . Even before you use reflection, you can prepare by:

- keeping repetitive boilerplate generation in one place (macros or scripts),
- isolating serialization/inspection code behind internal adapters,
- ensuring your types expose clear invariants and validation hooks (so reflection does not bypass safety checks).

The goal is not “use reflection everywhere,” but “when reflection becomes available, adoption is localized.”

26.2.8 Deprecation policy as an evolution tool

Evolution is not only adding features; it is also removing old ones safely. A professional policy usually includes:

- **announce**: document the replacement and the migration path,
- **deprecate**: keep old API working but noisy (compile-time warnings),
- **remove**: after a defined time window (or major version).

This mirrors how the Standard itself evolves (deprecate then remove) and aligns well with modern compiler diagnostics and CI gates.

26.2.9 Practical checklist for evolution-ready C++ APIs

- **Interfaces first**: treat APIs as contracts; keep them small and explicit .

- **Localize volatility**: hide implementation details (Pimpl where ABI matters) .
- **Encode safety in types**: spans/views for non-owning ranges; RAII for resources; explicit ownership.
- **Make failure explicit**: prefer typed error results at boundaries (e.g., `std::expected`) .
- **Prepare for safety enforcement**: keep unsafe code behind narrow boundaries; design for profiles and contract-style enforcement .
- **Avoid toolchain lock-in**: use feature detection and compatibility layers.
- **Plan concurrency evolution**: avoid threading details in public APIs; keep scheduling/cancellation abstract .

26.3 Long-term maintainability

26.3.1 Maintainability is a safety feature

In long-lived C++ systems, many failures are not caused by "hard" bugs in algorithms, but by **maintenance-induced defects**: a correct component becomes incorrect after years of small changes.
Long-term maintainability, in a safety handbook, means engineering the codebase so that:

- changes are easy to review and hard to misuse,
- invariants are explicit and continuously checked,
- concurrency and ownership contracts remain readable years later,
- upgrades to compilers, libraries, and the C++ standard can be done incrementally.

The C++ Core Guidelines explicitly frame their purpose around writing modern C++ code that is simpler, more reliable, and statically safer, with emphasis on interfaces, resource management, memory management, and concurrency .

26.3.2 Design principles that keep code maintainable for a decade

Minimize the number of concepts a reader must hold

A maintainable design is one where *local reasoning* is possible: a developer can understand a function and its correctness without scanning the whole codebase.
Practical levers:

- prefer small, focused components,
- avoid hidden global state,
- keep invariants close to the data they constrain,
- keep error handling consistent and predictable.

Make interfaces explicit and stable

Interfaces are long-lived contracts. They should communicate: ownership, lifetime, nullability, thread-safety, and failure modes.
The Core Guidelines place interfaces at the center of good design because they determine the structure and long-term readability of the system .

Example: express non-owning range parameters with `std::span` `std::span` is a standard non-owning view of a contiguous sequence . It makes the “pointer + length” contract explicit and reduces parameter mismatches.

```
#include <span>
#include <cstddef>

std::size_t checksum(std::span<const std::byte> data) noexcept {
  std::size_t s = 0;
  for (std::byte b : data) s = (s * 131) + static_cast<unsigned>(b);
  return s;
}
```

Maintainability benefit: the function signature documents the contract (contiguous, non-owning, length-known), which reduces future misuse compared to `const void* + size_t`.

Encode invariants in types whenever practical

When a concept is important to correctness, try to represent it directly in the type system:

- use strong types for IDs/handles instead of raw integers,
- prefer value types that keep invariants internally,
- avoid “boolean parameters” that are hard to read at call sites.

Example: avoid ambiguous flags

```
enum class Validation { none, strict };

struct Options {
  Validation validation = Validation::strict;
  int max_items = 1024;
};

void process(Options opt);
```

26.3.3 Resource and lifetime discipline (the foundation of sustainable code)

RAII everywhere, especially at boundaries

RAII is the standard C++ discipline for resource safety: acquire in constructors, release in destructors, so cleanup is automatic under all control paths (including exceptions) .

Bad: manual cleanup that will eventually be forgotten

```
#include <cstdio>

void write_log(const char* path) {
  std::FILE* f = std::fopen(path, "w");
  if (!f) return;

  // many lines later...
  std::fputs("hello\n", f);

  // easy to forget fclose on early returns or new error paths
  std::fclose(f);
}
```

Good: RAII wrapper for reliable cleanup

```
#include <cstdio>
#include <memory>

struct FileCloser {
  void operator()(std::FILE* f) const noexcept { if (f) std::fclose(f); }
};
```

```
using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file(const char* path, const char* mode) {
  return unique_file{ std::fopen(path, mode) };
}

void write_log(const char* path) {
  auto f = open_file(path, "w");
  if (!f) return;
  std::fputs("hello\n", f.get());
}
```

This style scales: future edits do not need to remember cleanup rules.

Ownership should be visible in APIs

Long-lived systems degrade when ownership is implicit. A maintainable codebase uses:

- `std::unique_ptr<T>` for unique ownership,
- `std::shared_ptr<T>` only when shared lifetime is justified,
- raw pointers/references/`std::span` for non-owning access.

This aligns with the Core Guidelines' emphasis on resource safety and avoiding common lifetime mistakes .

26.3.4 Error handling that remains readable under change

Choose one dominant error model per layer

Maintainability improves when each architectural layer has a clear rule:

- high-level application services: exceptions can be appropriate,

- low-level libraries / boundary APIs: explicit error returns can be appropriate,
- do not mix both without a documented convention.

Prefer typed results over ad-hoc conventions

C++23 standardizes `std::expected` as a vocabulary type that holds either a value or an error . It improves maintainability by keeping the success path and error path explicit and type-checked.

```
#include <expected>
#include <string_view>

enum class ParseErr { empty, invalid };

std::expected<int, ParseErr> parse_port(std::string_view s) {
  if (s.empty()) return std::unexpected(ParseErr::empty);

  int v = 0;
  for (char c : s) {
    if (c < '0' || c > '9') return std::unexpected(ParseErr::invalid);
    v = v * 10 + (c - '0');
  }
  if (v < 0 || v > 65535) return std::unexpected(ParseErr::invalid);
  return v;
}
```

Maintainability benefit: adding new error cases usually changes only the error type, not every call site pattern.

26.3.5 Concurrency maintainability

Make thread-safety contracts explicit

Long-term concurrency safety depends on *documented contracts*:

- thread-safe vs not thread-safe,
- which operations require external synchronization,
- whether callbacks may re-enter,
- cancellation/shutdown rules.

Prefer structured lifetime for threads

C++ offers facilities that encourage structured concurrency (e.g., RAII-managed thread objects). This reduces shutdown races and "dangling thread" failure modes, and matches RAII principles .

26.3.6 Contracts and profiles: designing code that can be checked more deeply over time

Write invariants so tools can enforce them later

Even before enabling advanced checking, write code so that invariants can be validated:

- minimize raw owning pointers,
- avoid type-punning and unchecked casts in high-level code,
- keep bounds/lifetime assumptions local and explicit.

C++ is moving toward stronger, checkable specifications in key areas:

- **Contracts** proposal for C++26 provides syntax for preconditions, postconditions, and assertions, with defined evaluation/handling modes .
- **Profiles** are defined as sets of guarantees (e.g., type safety, no range errors) typically implemented by banning risky features and enabling checking .

Maintainability mindset Write today as if your code will be checked tomorrow. When stronger checking becomes available in your toolchain, you want adoption to be configuration, not a rewrite.

26.3.7 Tooling and process: how maintainability is sustained

Make the compiler an active reviewer

A maintainable codebase compiles cleanly at high warning levels and treats warnings seriously. This is where modern toolchains and guideline-based checkers pay back continuously.
Microsoft's documentation for the C++ Core Guidelines checkers describes their goal: emphasize static type safety and resource safety, and reduce the most error-prone parts of the language .

Use guideline-based static analysis as a long-term guardrail

Policy recommendation:

- run static analysis in CI,
- gate new changes on "no new warnings" for your chosen rules,
- track and burn down existing issues as a planned effort.

Deprecation and migration policy

Maintaining code for years requires controlled change:

- announce a replacement API and provide a migration path,
- mark old APIs deprecated (warnings) with a defined timeline,
- remove only at major version boundaries or after a defined window.

This mirrors the Standard's own evolution model, which helps align your codebase with future toolchains .

26.3.8 Preparing for language evolution up to C++26

Adopt new features behind stable boundaries

C++26 is at feature freeze and moving through standardization finalization steps . That implies:

- features will appear in compilers/libraries on different schedules,
- your code should allow gradual adoption.

Practical pattern:

- isolate feature-dependent code in a small compatibility layer,
- keep public APIs stable,
- use feature-test macros (`<version>`) where applicable.

Reflection readiness (without depending on it)

Static reflection proposals for C++26 aim to enable compile-time inspection and generation with a reduced initial feature set . Maintainability opportunity:

- reduce boilerplate (serialization, logging, mapping),
- improve consistency (generated code follows one rule).

Maintainability risk:

- generated code can bypass invariants if not designed carefully.

Design rule:

> Even if reflection generates code, invariants must still live in the types (constructors/validators), not in scattered call sites.

26.3.9 Long-term maintainability checklist

- **Interfaces**: explicit ownership, lifetime, bounds, error model, and thread-safety .
- **RAII**: no manual cleanup paths; resources released automatically .
- **Safer parameter types**: `std::span` for non-owning contiguous ranges .
- **Typed failures**: `std::expected` for explicit success/error results .
- **Contracts and profiles readiness**: invariants written so tools can enforce them later .
- **Tooling**: continuous static analysis and guideline checkers in CI .
- **Controlled evolution**: deprecate with a migration path; avoid abrupt breaking changes .

Conclusion

C++ is safe when used professionally

The honest conclusion

C++ is a powerful systems language. It gives you direct control over memory, object lifetime, concurrency, and performance. That power is exactly why it can be safe *and* dangerous. The key conclusion of this handbook is simple:

> **C++ is safe when it is engineered with professional discipline: explicit contracts, explicit ownership, explicit concurrency rules, and continuous verification by tools.**

Modern C++ (C++11 through C++26) provides strong building blocks for safety: RAII, smart pointers, stronger type systems and library vocabulary types, safer concurrency primitives, and a rapidly improving ecosystem of static analyzers and sanitizers. The C++ Core Guidelines summarize this direction as a practical rule set focused on resource safety, lifetime safety, and concurrency correctness.

What “professional use” means in practice

Professional C++ safety is not one feature. It is a **system of constraints** that makes unsafe behavior hard to write, easy to detect, and cheap to prevent.

1) Ownership and lifetime are explicit

Most memory bugs come from broken lifetime assumptions: dangling pointers, use-after-free, double-delete, and accidental ownership transfer.
Professional C++ makes ownership visible:

- **Own with types**: `std::unique_ptr` for unique ownership; `std::shared_ptr` only when shared lifetime is required.
- **Borrow with types**: references, raw pointers (non-owning), and views like `std::span`.
- **Never hide ownership in a raw pointer return value.**

Bad: ownership is implicit

```
struct Image { /* ... */ };

Image* load_image(const char* path); // Who deletes? When?
```

Good: ownership is explicit and exception-safe

```
#include <memory>

struct Image { /* ... */ };

std::unique_ptr<Image> load_image(const char* path);
```

The Core Guidelines emphasize RAII and resource handles as the primary way to prevent leaks and lifetime bugs.

2) Bounds and preconditions are part of the interface

Many vulnerabilities are simple: reading/writing out of bounds, passing mismatched pointer/length pairs, or assuming a buffer is large enough.
Modern C++ provides vocabulary types that reduce these mistakes: `std::span` expresses "non-owning contiguous range with known length" directly in the signature.

Good: safer range parameter

```
#include <span>
#include <cstddef>

std::size_t count_zeros(std::span<const unsigned> xs) noexcept {
  std::size_t n = 0;
  for (auto v : xs) if (v == 0) ++n;
  return n;
}
```

3) Concurrency safety is engineered, not assumed

C++ concurrency is safe when you follow the memory model and avoid data races. Data races are undefined behavior: correctness and security cannot be guaranteed.
Professional practice:

- use `std::mutex`/`std::lock_guard` for shared mutable state,
- use atomics only with a clear correctness argument,
- document thread-safety contracts (thread-safe? externally synchronized?).

Correct signaling: release/acquire

```
#include <atomic>

std::atomic<bool> ready{false};
int data = 0;

void producer() {
  data = 42;
  ready.store(true, std::memory_order_release);
}

int consumer() {
  while (!ready.load(std::memory_order_acquire)) { }
  return data; // safe under this pattern
}
```

The standard library specifies how memory ordering works via `std::memory_order`.

Maintainable thread lifetime: cooperative stop When you need cancellation, prefer structured, cooperative designs that keep shutdown rules explicit. C++20 introduced `std::jthread` and `std::stop_token` for this pattern.

```
#include <thread>
#include <stop_token>

void worker(std::stop_token st) {
  while (!st.stop_requested()) {
    // do work...
  }
}

int main() {
```

```
  std::jthread t(worker); // joins automatically
  // ...
  // request stop by destroying t or calling request_stop() where
  ↪  appropriate
}
```

4) Error handling is consistent and visible

Safety includes predictable failure behavior. In long-lived systems, inconsistent error conventions become a reliability hazard.
C++23 provides `std::expected` as a vocabulary type to return either a value or an error explicitly.

```
#include <expected>
#include <string_view>

enum class Err { empty, invalid };

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

5) Tools are part of the language you actually use

Professional C++ safety assumes that humans will miss things. So you build a system where mistakes are caught early.

Static analysis and guideline enforcement The Core Guidelines explicitly discuss tool support and guideline profiles. Microsoft documents its Core Guidelines checkers (CoreCheck) as a ruleset aligned with guideline enforcement.

Sanitizers for runtime detection Sanitizers are compiler/runtime instrumentation tools that detect classes of bugs during execution:

- AddressSanitizer finds many memory safety bugs (use-after-free, out-of-bounds, etc.).
- UndefinedBehaviorSanitizer detects many forms of undefined behavior at runtime.
- ThreadSanitizer detects data races, which C++11 and later treat as undefined behavior.

The professional rule is:

> **If it is not tested under sanitizers (and checked by analysis), it is not fully verified.**

Why this is stronger than "C++ is unsafe"

It is true that C++ allows writing unsafe code. It is also true that modern C++ provides a large set of mechanisms to *prevent* unsafe code, and a mature tool ecosystem to *detect* bugs that escape review.
This is exactly why the committee continues investing in safety and correctness work up to C++26: feature freeze for C++26 is completed and the standard is moving through finalization. Contracts and profiles work explicitly targets better, checkable correctness and safety properties.

The professional safety rule set (final checklist)

If you apply the following consistently, modern C++ is a safe engineering environment:

- **RAII everywhere**: no manual cleanup paths for resources.
- **Ownership is explicit**: `unique_ptr`/`shared_ptr` for ownership; raw pointers/references for non-owning.
- **Bounds are explicit**: prefer `std::span` and range-based APIs.
- **Concurrency is disciplined**: no data races; correct synchronization and memory ordering.
- **Errors are consistent**: typed results where appropriate (`std::expected`).
- **Tooling is mandatory**: static analysis + sanitizers in CI.
- **Prepare for evolution**: contracts/profiles direction and gradual adoption up to C++26.

Closing statement

C++ rewards professionalism. When you make invariants explicit, encode intent in types, apply disciplined concurrency rules, and continuously verify with tools, you get: **high performance and strong safety at the same time**.
That is not an accident. It is the engineering culture that modern C++ was designed to support.

Discipline beats restrictions

The central claim

Many safety discussions frame the problem as a choice between two worlds:

- **Restrictions**: “ban enough features and safety will happen.”

- **Freedom**: “let engineers do anything and rely on skill.”

Professional C++ safety is neither extreme.

> **Restrictions help, but discipline is what makes safety reliable.**

Restrictions are a tool. Discipline is the system that makes tools effective: clear contracts, clear ownership, consistent concurrency rules, continuous verification, and code review culture. This is the same philosophy behind the C++ Core Guidelines: make good practices systematic, enforceable, and tool-supported.

Why restrictions alone do not guarantee safety

Restrictions cannot cover the whole problem

A “restricted subset” can reduce risk, but it cannot automatically ensure correctness:

- Logic bugs still happen.
- Concurrency bugs still happen if the synchronization design is wrong.
- API misuse still happens if ownership and lifetime are unclear.
- Performance constraints can force “escapes” back into low-level code.

A restricted language or restricted coding subset may lower the probability of some errors, but it does not replace engineering discipline.

Restrictions often create “escape hatches”

In real production systems, teams eventually meet requirements that push beyond the subset: platform boundaries, ABIs, legacy integration, specialized performance work, or low-level I/O. When this happens, a restricted environment tends to produce:

- undocumented workarounds,
- unreviewed “unsafe islands,”
- inconsistent patterns.

That failure mode is not theoretical. It is a common operational reality in long-lived systems.

What discipline means (in safety terms)

Discipline is not vague. It is a concrete set of professional habits that make unsafe behavior hard to write and easy to detect.

1) Make ownership explicit and enforce it

Restriction approach “Never return raw pointers.”

Discipline approach Make ownership part of the type system and code review rules:

- `std::unique_ptr<T>` expresses unique ownership.
- Borrowing is expressed via references, raw pointers (non-owning), or views like `std::span`.
- Ownership transfer is explicit and testable.

Bad: ownership is unclear (unsafe by default)

```
struct Conn { /* ... */ };
Conn* open_connection();      // who deletes? when?
void use_connection(Conn*);  // does this store it? does it own it?
```

Good: ownership and borrowing are explicit

```cpp
#include <memory>

struct Conn { /* ... */ };

std::unique_ptr<Conn> open_connection();
void use_connection(const Conn&); // non-owning
```

This aligns with the Core Guidelines emphasis on resource and lifetime safety through explicit ownership and RAII.

2) Prefer RAII so cleanup cannot be forgotten

Restriction approach "Do not call `new` or `delete`."

Discipline approach Require RAII for *all* resources: memory, locks, file handles, sockets, transactions, temporary state changes. RAII is not a style preference; it is an engineering mechanism that makes correct cleanup inevitable.

Bad: manual cleanup will break during maintenance

```cpp
#include <mutex>

std::mutex m;

void f() {
  m.lock();
  // ... someone adds an early return here later ...
  m.unlock();
}
```

Good: RAII makes maintenance safe

```
#include <mutex>

std::mutex m;

void f() {
  std::lock_guard<std::mutex> lock(m);
  // ... any return or exception still unlocks ...
}
```

3) Treat concurrency as a contract

Restriction approach "Do not use atomics."

Discipline approach Use a small number of approved patterns, require a correctness argument for each shared state, and document thread-safety at the API boundary. The standard memory model requires that programs avoid data races; the engineering goal is to make that property durable under change.

Bad: unsynchronized shared state (undefined behavior)

```
int shared = 0;

void inc() { ++shared; }  // data race if called concurrently
```

Good: disciplined synchronization

```
#include <mutex>

int shared = 0;
std::mutex mx;
```

```
void inc() {
  std::lock_guard<std::mutex> lock(mx);
  ++shared;
}
```

Discipline is not “always use a mutex.” It is: choose a clear rule, keep it consistent, and make it reviewable.

4) Make preconditions checkable (today) and evolvable (tomorrow)

C++ is moving toward stronger, checkable specifications with Contracts. Even before a Contracts facility is universally deployed, disciplined code:

- states preconditions and postconditions,
- checks critical assumptions in debug/test builds,
- keeps invariants close to the data they constrain.

```
#include <cassert>
#include <span>

int find_first_positive(std::span<const int> xs) {
  // precondition: xs is a valid view; length is known by span
  for (int v : xs) {
    if (v > 0) return v;
  }
  return 0;
}
```

The key maintainability property is that the contract is readable and can later be upgraded to stronger enforcement.

5) Verification is part of the workflow

Discipline assumes humans miss problems. Therefore, professional C++ safety uses tools continuously:

- static analysis and guideline checks in CI,
- sanitizers in testing,
- warnings treated as signals, not noise.

Microsoft documents the C++ Core Guidelines checkers as tool support for a subset of guideline rules. This is an example of discipline institutionalized: the rules do not depend on a single expert remembering everything.

Where restrictions *do* help (as part of discipline)

The right relationship is:

> **Restrictions are guardrails. Discipline is the driving system.**

Profiles: structured restrictions with explicit guarantees

The emerging concept of *profiles* is precisely about making restrictions meaningful: a profile is a set of guarantees (e.g., type safety, no leaks, no range errors) typically enforced by banning risky features and enabling checking.
Profiles are valuable because they transform “restrictions” into:

- explicit goals (what guarantee is intended),
- explicit mechanism (what is restricted and what is checked),
- tool support (what enforces it).

The professional model: safe-by-default, with audited escape hatches

In real C++ systems, some low-level work is unavoidable. Discipline makes this safe by design:

- keep unsafe operations behind narrow interfaces,
- isolate and document them,
- require review and tests specifically for those regions,
- prefer safer vocabulary types at the boundary (RAII handles, `std::span`, `std::expected`).

Why this conclusion is aligned with modern C++ evolution up to C++26

C++26 is feature-frozen and on track through finalization milestones. The major safety direction is not "remove power." It is:

- make correctness more expressible (e.g., Contracts work),
- make safety enforceable via defined rule sets (profiles and guideline-driven checking),
- strengthen the tool-supported culture around modern C++.

This reinforces the conclusion: **professional discipline is the stable strategy** regardless of which features the next standard adds.

A practical "discipline" checklist

The following checklist is intentionally operational. If a team does these consistently, C++ is a safe production environment:

- **Ownership rules**: ownership must be expressed in types; borrowing must be documented.
- **RAII everywhere**: no manual cleanup for locks, resources, or state changes.
- **Thread-safety contracts**: every shared component states its synchronization rule.
- **Error model consistency**: one dominant error strategy per layer; no hidden conventions.
- **Tooling as policy**: static analysis and guideline checks in CI.
- **Profiles as guardrails**: adopt restriction sets that deliver explicit guarantees.
- **Escapes are audited**: narrow unsafe boundaries; high scrutiny; dedicated tests.

Closing statement

Restrictions can reduce risk, but restrictions alone do not create trustworthy systems. What creates trustworthy systems is **discipline**: the consistent, enforceable practice of expressing intent, protecting invariants, and verifying behavior continuously.
That is why, in professional C++, **discipline beats restrictions**.

Modern C++ as an engineering contract

The idea: C++ safety is a contract, not a promise

Modern C++ is often described as a language of freedom. That is true. But modern C++ safety is better understood as an **engineering contract**:

> **C++ gives you performance and control *if* you honor explicit contracts: object lifetime contracts, ownership contracts, bounds contracts, and concurrency contracts.**

This is not marketing. It is how the language is specified and how professional C++ is practiced:

- the standard defines strict rules for lifetime and for the memory model,
- violating those rules leads to undefined behavior (no safety guarantees),
- professional C++ therefore treats correctness constraints as first-class design artifacts.

The C++ Core Guidelines capture this philosophy as a practical set of rules and best practices, emphasizing resource safety, lifetime safety, interfaces, and concurrency correctness.

What a contract looks like in real C++ systems

In a professional codebase, contracts appear in four places:

1. **In the types** (ownership and intent encoded in the signature),
2. **In the API rules** (documented preconditions/postconditions and thread-safety),
3. **In the runtime checks** (assertions, validation, defensive programming at boundaries),
4. **In the tooling** (warnings, static analysis, sanitizers, CI gates).

A codebase is “safe” to the extent that these contracts are explicit, consistent, and enforced.

Contract 1: Lifetime and ownership

Ownership must be visible at the boundary

When ownership is implicit, maintenance eventually breaks the system. Professional C++ avoids “mystery ownership” by using ownership-carrying types.

Bad: ownership and lifetime are not expressed

```
struct Session { /* ... */ };

Session* open_session();   // who owns it?
void close_session(Session*); // who must call this and when?
```

Good: the contract is encoded in the type

```
#include <memory>

struct Session { /* ... */ };

std::unique_ptr<Session> open_session();   // caller owns
void use_session(const Session&);          // caller borrows (non-owning)
```

The Core Guidelines repeatedly emphasize RAII and explicit ownership to prevent leaks and lifetime bugs.

RAII is the enforcement mechanism for cleanup contracts

A lifetime contract is only reliable when cleanup is automatic under *every* control path (return, exception, early exit).

Bad: manual cleanup is a time bomb

```
#include <mutex>

std::mutex m;

void update() {
  m.lock();
  // ... future edits add early returns or exceptions ...
  m.unlock();
}
```

Good: RAII makes the contract maintenance-proof

```
#include <mutex>

std::mutex m;

void update() {
  std::lock_guard<std::mutex> lock(m);
  // ...
}
```

Contract 2: Bounds and validity

Make “pointer + length” unbreakable

A large class of bugs comes from mismatched pointers and lengths. Modern C++ provides vocabulary types that make the contract explicit.

`std::span` is a non-owning view of a contiguous sequence with a known length. This turns a fragile convention into a typed interface.

```
#include <span>
#include <cstddef>

std::size_t checksum(std::span<const std::byte> data) noexcept {
  std::size_t s = 0;
  for (std::byte b : data) s = (s * 131) + static_cast<unsigned>(b);
  return s;
}
```

The maintainability benefit is decisive: the signature itself communicates the contract, and callers cannot accidentally pass the wrong length.

Contract 3: Concurrency and the memory model

The contract: no data races

In modern C++, **data races are undefined behavior**. If two threads access the same object, and at least one access is a write, and there is no synchronization, the program has no correctness guarantee.
Professional practice therefore treats concurrency as a contract:

- shared mutable state must have an explicit synchronization rule,
- atomics must be used with a correctness argument (not intuition),
- the memory order must match the intended happens-before relationship.

A minimal contract pattern: release/acquire

`std::memory_order` specifies ordering constraints around atomic operations.

```
#include <atomic>
```

```
std::atomic<bool> ready{false};
int data = 0;

void producer() {
  data = 42;
  ready.store(true, std::memory_order_release);
}

int consumer() {
  while (!ready.load(std::memory_order_acquire)) { }
  return data; // safe under this pattern
}
```

The safety is not “because atomics exist,” but because the code honors a precise contract: the acquire load observes the effects sequenced-before the release store.

Contract 4: Failure behavior and recovery

A professional contract has explicit failure modes

Safety includes predictable failure: callers must know what happens on error, what is rolled back, and what remains valid.
C++23 standardized `std::expected` as a vocabulary type to represent either a value or an error. This makes error contracts explicit and type-checked.

```
#include <expected>
#include <string_view>

enum class Err { empty, invalid };

std::expected<int, Err> parse_positive(std::string_view s) {
```

```
  if (s.empty()) return std::unexpected(Err::empty);
  int v = 0;
  for (char c : s) {
    if (c < '0' || c > '9') return std::unexpected(Err::invalid);
    v = v * 10 + (c - '0');
  }
  return v;
}
```

This is an engineering contract: the caller must handle the error branch, and the function guarantees it is never "valueless" in an ambiguous state.

Contracts are becoming more explicit in the language direction

Language-level Contracts work in the C++26 era

C++ has long relied on documentation, assertions, tests, and tools to express contracts. The Contracts proposal work (P2900 series) aims to provide standard syntax for preconditions, postconditions, and assertions, along with defined evaluation/handling semantics.
Even before universal deployment, the design lesson is immediate:

> **Write APIs with clear, stable preconditions and postconditions today, so enforcement can strengthen tomorrow without redesigning the system.**

Profiles: making restrictions meaningful

A profile is described as *a set of guarantees*, typically implemented by banning risky features and enabling checking. This aligns perfectly with the "engineering contract" framing: restrictions are not arbitrary; they exist to deliver explicit guarantees.

The enforcement layer: tools are part of the contract

Tool support turns contracts into continuous reality

Professional safety assumes humans miss things. Therefore:

- compile warnings are treated as actionable,
- static analysis runs in CI,
- guideline checking is applied systematically.

The C++ Core Guidelines explicitly note tool support such as clang-tidy guideline checks and Microsoft guideline enforcement tooling. Microsoft documents how Visual Studio supports a subset of Core Guidelines rules via its checkers.

Why this conclusion is timely up to C++26

C++26 has completed feature freeze and is moving through standardization finalization stages. That matters for engineering organizations:

- new facilities will appear across compilers and standard libraries on different schedules,
- contracts/profiles/tooling momentum continues,
- the winning long-term strategy is stable: write code whose safety is a contract and whose enforcement is automated.

Final takeaway

Modern C++ safety is not “C++ magically became safe.” Modern C++ safety is:

A professional engineering contract: types express intent, APIs state rules, concurrency is disciplined, failure is explicit, and tools enforce the rules continuously.

That is how C++ remains both powerful and safe in real systems.

Appendices

Appendix A — Common Memory Bugs Checklist

How to use this checklist

This appendix is designed as a practical, review-friendly checklist for the most common memory bugs in C++. It focuses on **what to look for**, **why it is dangerous**, **typical symptoms**, and **professional countermeasures**.

Two categories cover most memory failures

- **Temporal safety bugs**: using memory at the wrong time (use-after-free, use-after-scope, double-free).
- **Spatial safety bugs**: using the wrong memory location (out-of-bounds, buffer overflow, misaligned access).

Tooling principle Many of these bugs are systematically detectable during testing with sanitizers (ASan, UBSan, TSan) and guideline-based analysis .

A. Temporal safety (lifetime) bugs

A.1 Use-after-free (dangling pointer/reference)

What to look for

- Raw pointers or references that outlive the object they refer to.
- Pointers stored in containers/caches without a clear ownership policy.
- Returning pointers/references to objects owned by a local `unique_ptr` or a temporary container.

Why it is dangerous

It can corrupt memory, cause crashes, or create exploitable vulnerabilities.

Typical symptoms

- crashes far from the bug site,
- data corruption that appears “random”,
- intermittent failures that depend on allocator behavior.

Professional countermeasures

- Express ownership explicitly (`std::unique_ptr`, `std::shared_ptr` only when justified).
- Prefer borrowing views (`T&`, `std::span`) only when lifetime is guaranteed by the caller.
- Keep object lifetimes structured (RAII; avoid manual `new/delete` in most code).
- Run AddressSanitizer in test builds (detects use-after-free) .

Bad example

```
#include <vector>

int* leak_dangling_ptr() {
  std::vector<int> v = {1,2,3};
  return &v[0]; // dangling: v is destroyed at return
}
```

Good example

```
#include <vector>
#include <memory>

std::unique_ptr<std::vector<int>> make_vec() {
  return
  ↪  std::make_unique<std::vector<int>>(std::initializer_list<int>{1,2,3});
}
```

A.2 Use-after-scope / use-after-return (stack lifetime bugs)

What to look for

- Returning pointers/references to local variables.
- Capturing references to locals in callbacks/threads that may run later.
- Storing `std::string_view` or `std::span` into long-lived objects when the underlying storage is temporary.

Detection AddressSanitizer can detect stack-use-after-return / use-after-scope patterns on many toolchains .

Bad example: reference captured beyond scope

```
#include <functional>

std::function<int()> make_bad() {
  int x = 7;
  return [&]() { return x; }; // dangling reference after return
}
```

Good example: capture by value

```
#include <functional>

std::function<int()> make_good() {
  int x = 7;
  return [x]() { return x; }; // safe
}
```

A.3 Double-free and invalid free

What to look for

- Two owners think they must free the same pointer.
- Manual `delete` plus smart pointer ownership on the same object.
- Freeing memory not allocated by the matching allocator.

Professional countermeasures

- Use single ownership (`unique_ptr`) where possible; avoid “shared by default.”
- Do not mix raw `delete` with smart-pointer-managed objects.
- Prefer factory functions (`make_unique`/`make_shared`) to prevent accidental mismatched ownership.

Bad example: double delete

```
#include <memory>

int main() {
  auto p = std::make_unique<int>(42);
  int* raw = p.get();
  delete raw;           // invalid: p will delete it again
} // double-free (undefined behavior)
```

A.4 Iterator and reference invalidation

What to look for

- Keeping iterators/references/pointers into a container across operations that can reallocate or invalidate.
- Returning iterators/references to internal container elements from public APIs without documenting invalidation rules.

Professional countermeasures

- Prefer indices or stable handles when mutation and growth are expected.
- Document invalidation rules (especially for library APIs).
- Keep container mutations local; avoid exposing internal storage.

Bad example: vector reallocation invalidates pointers

```
#include <vector>

int main() {
  std::vector<int> v;
```

```
  v.push_back(1);
  int* p = &v[0];
  v.push_back(2);    // may reallocate
  *p = 9;            // p may now be dangling
}
```

B. Spatial safety (bounds) bugs

B.1 Out-of-bounds read/write (buffer overflow)

What to look for

- Manual pointer arithmetic, especially when lengths are separate parameters.
- Loops that use <= where < is intended.
- Indexing without validating bounds.

Detection AddressSanitizer detects out-of-bounds accesses to heap, stack, and globals .

Bad example: off-by-one

```
#include <cstddef>

void fill(int* a, std::size_t n) {
  for (std::size_t i = 0; i <= n; ++i) { // bug: writes one past end
    a[i] = 0;
  }
}
```

Good example: range-based interface

```cpp
#include <span>

void fill(std::span<int> a) {
  for (int& x : a) x = 0;
}
```

B.2 Unchecked C-style string operations

What to look for

- Any API that writes to a buffer without carrying its length.
- Assumptions that inputs are null-terminated.

Professional countermeasures

- Prefer `std::string`/`std::string_view` and size-aware APIs.
- Prefer `std::span<char>` for mutable buffers with explicit bounds.

B.3 Pointer provenance confusion and invalid pointer arithmetic

What to look for

- Arithmetic that moves pointers outside their original object/array.
- “Re-basing” pointers via integer casts and back.
- Manual memory layout tricks outside tightly-audited low-level code.

Risk This can lead to undefined behavior even if the pointer is never dereferenced in some cases, and it is extremely difficult to review safely.

C. Initialization and object representation bugs

C.1 Uninitialized reads

What to look for

- Primitive members not initialized in all constructors.
- Arrays/buffers used before being filled.
- Conditionals that skip initialization paths.

Professional countermeasures

- Prefer in-class member initializers.
- Prefer constructors that fully establish invariants.
- Use `std::optional` when “maybe initialized” is a real state.

Bad example

```
struct S {
  int x;      // uninitialized
  int y = 0; // initialized
};
```

Good example

```
struct S {
  int x = 0;
  int y = 0;
};
```

C.2 Object lifetime violations with placement new

What to look for

- Placement new into a buffer without correct destruction.
- Accessing an object through a pointer/reference after its lifetime ended.
- Type-punning or aliasing tricks that bypass lifetime rules.

Professional countermeasures

- Prefer `std::variant`/`std::optional` over raw storage tricks in application code.
- If placement new is necessary, isolate it in a small, well-tested component and document invariants.

D. Allocation and deallocation mismatches

D.1 new/delete mismatch and array mismatch

What to look for

- `new T` paired with `delete[]`, or `new T[]` paired with `delete`.
- Allocation in one module and deallocation in another with different runtime/CRT/allocator assumptions.

Detection ASan can catch allocation/deallocation mismatches in many configurations .

D.2 Mixing allocation APIs

What to look for

- `malloc/free` mixed with `new/delete`.
- Custom allocators where ownership is unclear.

Professional countermeasures

- Use a single allocation strategy per subsystem.
- Wrap allocation behind RAII handles so the correct deallocation is automatic.

E. Concurrency-driven memory corruption

E.1 Data races on shared state

What to look for

- Shared mutable objects accessed from multiple threads without consistent locking/atomic discipline.
- "It works in tests" arguments without a defined happens-before relationship.

Why it is a memory bug Data races can cause memory corruption and crashes, not only logical errors. In C++11 and later, data races are undefined behavior .
Detection ThreadSanitizer detects data races during execution .

Bad example

```
int shared = 0;
void inc() { ++shared; } // data race if called concurrently
```

Good example

```
#include <mutex>

int shared = 0;
std::mutex mx;

void inc() {
  std::lock_guard<std::mutex> lock(mx);
  ++shared;
}
```

E.2 Publishing pointers without synchronization

What to look for

- One thread initializes an object and stores a pointer for others without proper synchronization.
- “Volatile” used as if it provides thread-safety (it does not).

Professional countermeasures

- Use mutexes or `std::atomic` with correct memory ordering.
- Prefer higher-level thread-safe designs that avoid shared mutable state.

F. Review checklist (print-friendly)

Use this as a quick pass during review. Treat any “yes” as requiring justification and/or refactoring.

F.1 Ownership and lifetime

- ☐ Are there raw owning pointers in application code?
- ☐ Do any functions return raw pointers/references to internal or temporary storage?
- ☐ Are `string_view`/`span` stored beyond the lifetime of the source?
- ☐ Are callbacks/threads capturing references to locals or short-lived objects?
- ☐ Are iterators/pointers kept across container mutations that may invalidate them?

F.2 Bounds and indexing

- ☐ Do we pass `pointer + length` pairs instead of `span` or a container/view?
- ☐ Are any loops suspicious for off-by-one (<= vs <)?
- ☐ Is indexing done without clear bounds checks or proven invariants?
- ☐ Do we rely on null-termination assumptions for external data?

F.3 Allocation discipline

- ☐ Is `new/delete` used directly where RAII factories could be used instead?
- ☐ Is `new[]/delete[]` used correctly and consistently?
- ☐ Are allocation and deallocation occurring across ABI/module boundaries with different runtimes?
- ☐ Are allocation APIs mixed (`malloc/free` with `new/delete`)?

F.4 Initialization and representation

- ☐ Are there members/locals that could be read before initialization?
- ☐ Are there manual lifetime tricks (placement new, raw storage) outside isolated low-level modules?
- ☐ Are there casts or type-punning patterns that are hard to prove correct?

F.5 Concurrency and shared state

- ☐ Is any shared mutable state accessed without a single, consistent synchronization rule?
- ☐ Are atomics used without a clearly documented memory-ordering argument?
- ☐ Are pointers/objects published to other threads without synchronization?

G. Minimal “professional verification” recipe

For long-lived C++ systems, the following is a widely effective baseline:

- Run **AddressSanitizer** regularly in tests to catch use-after-free and out-of-bounds .
- Run **UndefinedBehaviorSanitizer** to catch many classes of undefined behavior (e.g., misaligned/null pointer dereference, signed overflow) .
- Run **ThreadSanitizer** on multithreaded tests to catch data races .
- Apply guideline-based analysis aligned with lifetime/resource rules .

26.4 Appendix B — RAII Design Checklist

Purpose and scope

RAII (Resource Acquisition Is Initialization) is the core C++ technique for making resource management **automatic, exception-safe, and maintainable**. The idea is to bind acquisition and release of a resource to the lifetime of an object with automatic storage duration . The C++ Core Guidelines explicitly recommend managing resources automatically using RAII . This appendix is a practical checklist for designing:

- RAII resource **handles** (types that own and release a resource),
- RAII **guards** (types that perform cleanup/rollback at scope exit),
- safe interfaces and invariants around these types.

B.1 The RAII contract (what your type must guarantee)

A correct RAII type must guarantee all of the following:

- **Acquire in construction**: the object is either fully constructed with a valid resource, or construction fails (typically by throwing).
- **Release in destruction**: when the object is destroyed, the resource is released exactly once.
- **No leaks on early exits**: returns and exceptions must not bypass cleanup (this is the whole point of RAII) .

B.2 Design checklist (print-friendly)

Use this checklist during design, code review, and refactoring.

B.2.1 Ownership, invariants, and API surface

- ☐ **Does the type clearly own something?** (If not, it may be a view, not a handle.)
- ☐ **Is the "valid" state well-defined?** (e.g., pointer not null; handle not equal to invalid sentinel.)
- ☐ **Is the invalid/empty state representable and safe?** (default construction optional, but if allowed it must be harmless.)
- ☐ **Is it obvious who owns the resource from the type?** (Prefer a dedicated handle type over raw pointers/ints.)
- ☐ **Are borrowing accessors non-owning?** (e.g., `get()`, `operator*`, `operator->` return non-owning access.)

B.2.2 Destructor rules (the most important RAII correctness point)

- ☐ **Destructor releases exactly once** (including after moves).
- ☐ **Destructor is non-throwing** (practically: do not let exceptions escape).
- ☐ **Destructor is safe on the empty state** (if empty state exists).
- ☐ **Release operation is idempotent at the object level** (calling `reset()` twice is safe; double-close is prevented).

B.2.3 Copy and move semantics (Rule of Zero / Rule of Five)

- ☐ **Is copying meaningful?** Most owning RAII handles should be **move-only**.
- ☐ **If copy is supported, is ownership shared intentionally?** (Prefer explicit shared ownership designs rather than accidental copying.)

- ☐ **Move leaves the source empty and destructible** (moved-from objects must be safe to destroy).
- ☐ **Move operations are `noexcept` when possible** (critical for container usability and strong exception safety in algorithms).

B.2.4 Reset, release, and swap (control operations)

- ☐ **Provide `reset()`** to replace the owned resource safely (release old, then adopt new).
- ☐ **Provide `release()`** only if transfer of ownership to a raw handle is required (and document who must release afterwards).
- ☐ **Provide `swap()`** for cheap, safe exchanges of ownership.

B.2.5 Interoperability and boundaries

- ☐ **Cross-module boundaries are explicit** (avoid allocating in one runtime/CRT and freeing in another unless it is guaranteed safe).
- ☐ **C APIs are wrapped immediately** (wrap raw handles into RAII at the boundary).
- ☐ **No raw owning pointers in application code** (ownership should be in a handle type or smart pointer) .

B.2.6 Verification and tool enforcement

- ☐ **Run guideline-based checks** that encourage scoped objects and RAII usage (example: MSVC rule support for preferring scoped objects) .
- ☐ **Test with AddressSanitizer** to catch leaks and many lifetime mistakes earlier in development .

B.3 Reference implementation patterns

This section provides small, reusable patterns that embody the checklist.

B.3.1 Prefer standard RAII types first (Rule of Zero)

The best RAII design is often: *do not design one*. If the resource can be represented using standard library handles, prefer those:

- `std::unique_ptr` for unique ownership (including custom deleters) ,
- `std::lock_guard` / `std::unique_lock` for mutex ownership,
- `std::vector`, `std::string` for memory ownership.

Example: `std::unique_ptr` is a canonical RAII handle

```
#include <memory>

struct Widget { /* ... */ };

std::unique_ptr<Widget> make_widget() {
  return std::make_unique<Widget>();
} // ownership is explicit; destruction is automatic
```

B.3.2 Custom deleter: wrap a C handle safely

Many system APIs return opaque handles that must be closed/freed with a matching function. Wrap them immediately.

Pattern: handle wrapper using `std::unique_ptr` with a deleter

```
#include <cstdio>
#include <memory>

struct FileCloser {
  void operator()(std::FILE* f) const noexcept {
    if (f) std::fclose(f);
  }
};

using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file(const char* path, const char* mode) {
  return unique_file{ std::fopen(path, mode) };
}
```

This pattern satisfies the RAII contract: acquire in construction, release in destruction, no leaks on early returns .

B.3.3 Write a dedicated move-only handle type (when you need a strong type)

If the resource is not pointer-like, or you want stronger typing than `unique_ptr` provides, create a small handle class.

Example: move-only handle wrapper (single-owner)

```
#include <utility>

class SocketHandle {
public:
  using native_handle_type = int;
  static constexpr native_handle_type invalid = -1;
```

```
SocketHandle() noexcept = default;

explicit SocketHandle(native_handle_type h) noexcept : h_(h) {}

SocketHandle(const SocketHandle&) = delete;
SocketHandle& operator=(const SocketHandle&) = delete;

SocketHandle(SocketHandle&& other) noexcept :
 ↪  h_(std::exchange(other.h_, invalid)) {}

SocketHandle& operator=(SocketHandle&& other) noexcept {
  if (this != &other) {
    reset(); // release current
    h_ = std::exchange(other.h_, invalid);
  }
  return *this;
}

~SocketHandle() { reset(); }

native_handle_type get() const noexcept { return h_; }
explicit operator bool() const noexcept { return h_ != invalid; }

void reset(native_handle_type nh = invalid) noexcept {
  if (h_ != invalid) {
    close_native(h_); // must not throw
  }
  h_ = nh;
}

native_handle_type release() noexcept {
```

```
    return std::exchange(h_, invalid);
  }

private:
  native_handle_type h_ = invalid;

  static void close_native(native_handle_type) noexcept {
    // Call the real OS/API close function here (must not throw).
  }
};
```

Checklist mapping:

- move-only ownership (no accidental double close),
- destructor is safe and does not throw,
- moved-from object is empty and safe to destroy,
- `reset()` and `release()` provide controlled lifetime operations.

B.3.4 Guard pattern: rollback/cleanup at scope exit (without owning the resource)

Sometimes you need a scope guard to restore state, unlock something non-standard, or roll back a partial operation. The Core Guidelines model is the same: cleanup must be automatic and exception-safe .

Minimal scope guard

```
#include <utility>

template<class F>
class scope_guard {
```

```
public:
  explicit scope_guard(F f) noexcept : f_(std::move(f)), active_(true) {}
  scope_guard(scope_guard&& other) noexcept
    : f_(std::move(other.f_)), active_(std::exchange(other.active_,
    ↪  false)) {}

  scope_guard(const scope_guard&) = delete;
  scope_guard& operator=(const scope_guard&) = delete;

  ~scope_guard() noexcept {
    if (active_) f_();
  }

  void release() noexcept { active_ = false; }

private:
  F f_;
  bool active_;
};

template<class F>
scope_guard<F> make_scope_guard(F f) noexcept { return
↪  scope_guard<F>(std::move(f)); }
```

Usage

```
#include <cstdio>

void write_two_steps(std::FILE* f) {
  auto rollback = make_scope_guard([&] {
    std::fputs("ROLLBACK\n", f);
  });
```

```
  std::fputs("STEP1\n", f);
  std::fputs("STEP2\n", f);

  rollback.release(); // commit
}
```

The guard is not “ownership”; it is a **cleanup contract** tied to scope.

B.4 Failure modes to audit (common RAII mistakes)

B.4.1 Destructors that can throw

If a destructor throws during stack unwinding, the program will typically terminate. Therefore, the professional rule is: **do not let exceptions escape destructors**. Use logging that cannot throw, or swallow errors in destructors and expose explicit `close()` APIs for error-reporting.

B.4.2 Hidden ownership and double-release hazards

- returning raw owning pointers from factories,
- storing the same raw handle in multiple owners,
- copying an owning handle type.

These are the fastest routes to double-free/double-close.

B.4.3 Incomplete move support

A move-only RAII type must ensure:

- moved-from object is empty,

- moved-to object owns the resource,
- no double-release occurs.

B.5 Short RAII review checklist (one page)

If you need a very fast review pass, use this:

☐ Does construction fully establish the invariant or fail?

☐ Does the destructor release exactly once and never throw?

☐ Is ownership move-only by default (unless shared ownership is explicitly intended)?

☐ Are `reset`/`release` semantics correct and documented?

☐ Is borrowing access clearly non-owning?

☐ Are cross-boundary allocation/deallocation rules safe?

☐ Do tools and CI enforce RAII discipline (guideline checks, sanitizers)?

26.5 Appendix C — Smart Pointer Decision Guide

Purpose

Smart pointers are not “better pointers.” They are **lifetime contracts encoded in types**. This guide helps you choose the correct pointer type (or no pointer at all) and avoid the most common ownership mistakes.
This appendix follows two professional principles emphasized by modern C++ practice and the C++ Core Guidelines:

- **Prefer RAII and clear ownership**: resource lifetime must be explicit and automatically managed.
- **Prefer the simplest correct ownership model**: unnecessary sharing makes systems harder to reason about.

C.1 The first decision: do you need dynamic allocation at all?

Before choosing any smart pointer, ask:

- **Can it be a value?** Prefer direct members, return-by-value, and standard containers.
- **Can it be optional?** Use `std::optional<T>` for "maybe present" without dynamic allocation.
- **Is it a polymorphic interface or large/expensive-to-move object?** Then dynamic allocation may be justified.

Prefer values and containers when possible

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

struct Widget { std::string name; };

Widget make_widget() { return Widget{"ok"}; }        // value return is
↪  normal modern C++
std::vector<Widget> make_many();                      // container owns
↪  elements
```

C.2 The ownership map (what each pointer means)

C.2.1 `std::unique_ptr<T>`: single owner

Use `std::unique_ptr` when:

- exactly one entity owns the object,
- ownership may transfer (move),
- you want the strongest default safety and clarity.

Canonical factory

```
#include <memory>

struct Engine { /* ... */ };

std::unique_ptr<Engine> make_engine() {
  return std::make_unique<Engine>();
}
```

C.2.2 `std::shared_ptr<T>`: shared ownership

Use `std::shared_ptr` only when:

- multiple independent owners must extend lifetime,
- no single owner can be clearly defined,
- the object must outlive the creating scope and is consumed by multiple subsystems with different lifetimes.

Shared ownership is a design decision with cost and risk:

- control block allocation and reference count management,
- harder reasoning about object lifetime,
- risk of reference cycles (leaks) if `shared_ptr` references form a cycle.

Example: shared ownership is explicit

```
#include <memory>

struct Cache { /* ... */ };

std::shared_ptr<Cache> make_cache() {
  return std::make_shared<Cache>();
}
```

C.2.3 `std::weak_ptr<T>`: non-owning observer of shared ownership

Use `std::weak_ptr` when:

- you must reference an object managed by `shared_ptr` without extending its lifetime,
- you need to break cycles (parent/child graphs, observers, caches),
- you need "check if still alive" semantics via `lock()`.

Breaking a cycle

```
#include <memory>
#include <vector>

struct Node {
```

```
  std::vector<std::shared_ptr<Node>> children;
  std::weak_ptr<Node> parent; // weak breaks cycle
};
```

C.2.4 Raw pointer / reference: non-owning borrow

Use raw pointers or references when:

- you do **not** own the object,
- lifetime is guaranteed by the caller or by a higher-level owner,
- the relationship is temporary (typical function parameters).

This is often the most maintainable interface: ownership stays where it belongs, and borrowing stays lightweight.

Borrowing API

```
struct Service { /* ... */ };

void use_service(Service& s);       // cannot be null
void maybe_use(Service* s);         // may be null (document it)
```

C.3 The decision guide (practical rules)

C.3.1 Default choice

> If dynamic allocation is needed and ownership is clear: **use `std::unique_ptr`**. If ownership is unclear: redesign until it becomes clear. Only then consider `std::shared_ptr`.

C.3.2 Decision checklist

- **Q1: Is there exactly one owner?** ⇒ `std::unique_ptr<T>`.
- **Q2: Do multiple independent components need to keep it alive?** ⇒ `std::shared_ptr<T>`.
- **Q3: Do you need a reference to a shared object without extending lifetime?** ⇒ `std::weak_ptr<T>`.
- **Q4: Is it just a temporary borrow?** ⇒ raw pointer/reference (non-owning).

C.4 Interface design rules (the most common professional mistakes)

C.4.1 Do not return a raw owning pointer

Bad: ownership is ambiguous

```
struct Job { /* ... */ };
Job* make_job(); // Who deletes? When?
```

Good: ownership is explicit

```
#include <memory>

struct Job { /* ... */ };
std::unique_ptr<Job> make_job();
```

C.4.2 Do not take `shared_ptr` by value unless you mean to share ownership

Passing `shared_ptr` by value increments/decrements the reference count and expresses that the callee participates in ownership.

Borrow without sharing

```
#include <memory>

struct Image { /* ... */ };

void draw(const Image& img); // best: callee borrows
```

If the function must keep a copy beyond the call, take `shared_ptr`

```
#include <memory>

struct Image { /* ... */ };

class Viewer {
public:
  void set_image(std::shared_ptr<Image> img) { img_ = std::move(img); }
  ↪  // ownership shared intentionally
private:
  std::shared_ptr<Image> img_;
};
```

C.4.3 Prefer `make_unique` and `make_shared`

Use factory helpers to avoid manual `new`, reduce exception-safety hazards, and improve clarity.

```
#include <memory>

struct T { T(int, int) {} };

auto p = std::make_unique<T>(1, 2);
auto q = std::make_shared<T>(1, 2);
```

C.5 Advanced patterns (useful, but easy to misuse)

C.5.1 Custom deleters (matching the allocating API)

When an object must be freed by a specific function (C APIs, OS handles), use a custom deleter.

```
#include <memory>
#include <cstdio>

struct FileCloser {
  void operator()(std::FILE* f) const noexcept { if (f) std::fclose(f); }
};

using unique_file = std::unique_ptr<std::FILE, FileCloser>;

unique_file open_file(const char* path, const char* mode) {
  return unique_file{ std::fopen(path, mode) };
}
```

C.5.2 Arrays: prefer containers; otherwise use the correct form

- Prefer `std::vector<T>` or `std::array<T,N>` for arrays.
- If you must manage a dynamic array with `unique_ptr`, use `std::unique_ptr<T[]>`.

```
#include <memory>
#include <cstddef>

std::unique_ptr<int[]> make_buffer(std::size_t n) {
  return std::unique_ptr<int[]>(new int[n]{}); // value-initialize
   ↪  elements
}
```

C.5.3 `enable_shared_from_this`: only when necessary

`std::enable_shared_from_this` allows an object already owned by a `shared_ptr` to safely produce another `shared_ptr` to itself. It is easy to misuse if the object is not actually owned by `shared_ptr`.

Safe usage pattern

```
#include <memory>

struct Session : std::enable_shared_from_this<Session> {
  std::shared_ptr<Session> self() { return shared_from_this(); }
};

auto s = std::make_shared<Session>();
auto s2 = s->self(); // OK: object is owned by shared_ptr
```

Rule:

- Use it only if the type *must* hand out shared ownership of itself.
- Otherwise, prefer passing a `shared_ptr` from the owning context.

C.5.4 Aliasing constructor of `shared_ptr`: advanced and risky

`shared_ptr` has an aliasing constructor that lets you share ownership of one object while pointing to a subobject. This can be useful for views into a larger allocation but can easily confuse reviewers. Use only with strong justification and clear comments.

C.6 Concurrency and smart pointers (what is safe, what is not)

C.6.1 Smart pointers do not replace synchronization

- Smart pointers manage **lifetime**.
- Mutexes/atomics manage **concurrent access to shared mutable state**.

Even if an object is kept alive by `shared_ptr`, its internal state can still be raced if not synchronized.

C.6.2 `shared_ptr` reference counting and thread safety

The reference count operations are designed for safe ownership management across threads, but that does not automatically make the pointed-to object thread-safe. Treat these as separate contracts.

C.7 Common failure modes and how to prevent them

C.7.1 Cycles with `shared_ptr`

Symptom: memory leak that persists until program exit.
Fix: replace one direction with `weak_ptr`.

C.7.2 "Shared by default" designs

Symptom: unclear ownership, hard-to-reason lifetime, accidental retention of large graphs.
Fix: start with unique ownership and explicit transfers; introduce sharing only at carefully chosen boundaries.

C.7.3 Storing raw pointers beyond the owner's lifetime

Symptom: use-after-free in caches, callbacks, background tasks.
Fix: store `weak_ptr` (for shared ownership graphs) or redesign to store stable IDs/handles.

C.8 One-page smart pointer cheat sheet

- **Use no pointer**: value / container / `optional` whenever possible.
- **Borrow temporarily**: `T&` or `T*` (non-owning).
- **Single owner**: `std::unique_ptr<T>` (default for owned dynamic objects).
- **Multiple owners**: `std::shared_ptr<T>` (only when necessary).
- **Observe shared object**: `std::weak_ptr<T>` (break cycles, avoid accidental retention).
- **Create safely**: `make_unique`, `make_shared`.
- **Never confuse lifetime with thread-safety**: smart pointers are not synchronization.

26.6 Appendix D — Debugging Memory Issues in Practice

Why memory bugs feel "impossible"

Most memory failures are **non-local**. The crash happens far from the mistake because memory corruption breaks invariants silently and only becomes visible later. The professional approach is therefore:

> **Do not "debug the crash." Instrument the program to catch the first invalid memory event.**

In practice, that means: reproduce reliably, compile with debug symbols, enable sanitizers, and let reports point to the first bad access.

D.1 Triage: classify the failure before you chase it

Start by classifying the symptom into one of these categories (each has a different best tool).

D.1.1 Temporal safety: wrong time

- Use-after-free
- Use-after-scope / use-after-return
- Double-free / invalid free

D.1.2 Spatial safety: wrong place

- Out-of-bounds read/write
- Heap buffer overflow / stack buffer overflow / global overflow
- Misaligned access (often reported as undefined behavior)

D.1.3 Initialization/UB and concurrency

- Uninitialized reads (often looks like randomness)
- Undefined behavior (signed overflow, invalid shifts, null/misaligned dereference)
- Data races (can manifest as memory corruption)

D.2 The professional workflow (repeatable, fast)

D.2.1 Step 1: get a reliable reproducer

- Reduce to the smallest input that fails.

- Run it repeatedly (10–100 times). If it is nondeterministic, it is often a race or an uninitialized read.
- Log only what you need. Excess logging can hide timing bugs.

D.2.2 Step 2: build for diagnosis

General rule: **debug symbols always**, and avoid optimizations that erase stack frames while diagnosing.

Clang/GCC: baseline diagnostic build

```
# Baseline: debug symbols + frame pointers help stack traces
CXX=clang++
CXXFLAGS="-O1 -g -fno-omit-frame-pointer"
$CXX $CXXFLAGS -std=c++23 -o app main.cpp
```

MSVC: enable AddressSanitizer when appropriate

```
REM Developer Command Prompt
cl /std:c++20 /Zi /Od /fsanitize=address main.cpp
```

D.2.3 Step 3: enable the right sanitizer first

A strong default order:

1. **ASan** for memory safety (use-after-free, out-of-bounds, etc.)
2. **UBSan** for undefined behavior
3. **TSan** for data races (separately, because it is heavier)

D.3 AddressSanitizer in practice (ASan)

ASan is designed to catch common memory safety errors: out-of-bounds accesses and use-after-free. It can also optionally detect **stack-use-after-return** (a frequent and subtle bug).

D.3.1 Build and run (Clang/GCC)

```
# AddressSanitizer + useful diagnostics
clang++ -O1 -g -fno-omit-frame-pointer \
  -fsanitize=address \
  -o app_asan main.cpp

# Optional: stronger stack-use-after-return detection (if supported)
clang++ -O1 -g -fno-omit-frame-pointer \
  -fsanitize=address -fsanitize-address-use-after-return=runtime \
  -o app_asan_uar main.cpp

# Run with runtime options (example)
ASAN_OPTIONS=detect_stack_use_after_return=1:abort_on_error=1
↪  ./app_asan_uar
```

D.3.2 Build and run (MSVC)

```
REM Build with AddressSanitizer
cl /std:c++20 /Zi /O2 /fsanitize=address main.cpp

REM Run (options via environment variable)
set ASAN_OPTIONS=abort_on_error=1
app.exe
```

D.3.3 Reading ASan reports (what matters)

When ASan reports an error, focus on:

- **The first stack trace at the invalid access** (the bug site).
- **The allocation/free stack trace** (who created/freed the object).
- **The access type** (READ vs WRITE) and size (1, 4, 8, 16 bytes).

D.3.4 Classic bug: use-after-free

Bad example

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

int main() {
  auto* p = new int(7);
  delete p;
  std::cout << *p << "\n"; // use-after-free
}
```

Fix strategy

- Replace raw ownership with RAII (`std::unique_ptr`).
- Ensure pointers do not outlive owners (clear contracts; avoid caching raw pointers).
- Prefer borrowing views only when lifetime is guaranteed (`T&`, `std::span`, `std::string_view`).

D.3.5 Classic bug: out-of-bounds (off-by-one)

```
#include <cstddef>

void fill(int* a, std::size_t n) {
  for (std::size_t i = 0; i <= n; ++i) { // bug: one past end
    a[i] = 0;
  }
}
```

Fix strategy Prefer range-based interfaces:

```
#include <span>

void fill(std::span<int> a) {
  for (int& x : a) x = 0;
}
```

D.4 UndefinedBehaviorSanitizer in practice (UBSan)

UBSan detects many classes of undefined behavior at runtime (typically with lower overhead than ASan). Examples include:

- null or misaligned pointer dereference,
- signed integer overflow,
- invalid casts and other UB categories (depending on enabled checks).

D.4.1 Build and run

```
# UBSan (common baseline). Prefer -O1/-O2 with -g for realistic behavior
↪  + symbols.
```

```
clang++ -O1 -g -fno-omit-frame-pointer \
  -fsanitize=undefined \
  -o app_ubsan main.cpp

# Run with typical runtime options
UBSAN_OPTIONS=halt_on_error=1:print_stacktrace=1 ./app_ubsan
```

D.4.2 Classic UB: signed overflow

```
#include <limits>

int main() {
  int x = std::numeric_limits<int>::max();
  int y = x + 1; // signed overflow: undefined behavior
  (void)y;
}
```

Fix strategy

- Use wider types where overflow is possible.
- Use checked arithmetic where overflow must be detected.
- Use unsigned arithmetic only if you truly want modulo semantics (and document it).

D.5 ThreadSanitizer in practice (TSan)

TSan is a dynamic data-race detector. A data race (two threads accessing the same memory concurrently, at least one write, without synchronization) can lead to crashes and memory corruption.

D.5.1 Build and run

```
# TSan is usually used alone (not combined with ASan in typical
↪  workflows).
clang++ -O1 -g -fno-omit-frame-pointer \
  -fsanitize=thread \
  -o app_tsan main.cpp

TSAN_OPTIONS=halt_on_error=1 ./app_tsan
```

D.5.2 Reading TSan reports

Focus on:

- the two conflicting accesses (READ/WRITE) and their stack traces,
- the synchronization that *should* exist but does not (missing mutex, wrong atomic ordering),
- whether the variable is actually shared (or accidentally shared via aliasing).

Typical fix patterns

- protect shared mutable state with a mutex and a clear locking policy,
- use atomics only with a correctness argument (release/acquire or stronger when needed),
- avoid sharing state by design (message passing, thread confinement, immutable snapshots).

D.6 A practical decision tree (what to try next)

D.6.1 If the program crashes

- Enable ASan first.
- If ASan is clean but behavior is wrong, enable UBSan.
- If the failure is intermittent and disappears with logging, run TSan (race suspicion).

D.6.2 If memory usage grows (leak suspicion)

- Audit ownership (unique ownership by default; shared only when required).
- Look for `shared_ptr` cycles (break with `weak_ptr`).
- Use sanitizer-driven leak detection where supported by your toolchain, and ensure clean shutdown paths.

D.6.3 If failures appear only in Release builds

- Suspect undefined behavior (UBSan) or uninitialized reads.
- Build a *release-like* sanitizer configuration: keep optimizations but retain symbols.
- Remove strict-aliasing hacks and lifetime violations first; they often "work" in Debug and fail in Release.

D.7 Making reports actionable (symbolization and reproducibility)

D.7.1 Always compile with debug symbols

Without symbols, stack traces are often not actionable. Use:

- `-g` on Clang/GCC,
- `/Zi` on MSVC.

D.7.2 Keep frame pointers while diagnosing

Frame pointers improve stack traces (especially on optimized builds). Prefer `-fno-omit-frame-pointer` on Clang/GCC.

D.7.3 Minimize and isolate

Once a sanitizer points to a suspect function:

- isolate it into a small test,
- remove unrelated concurrency,
- shrink the input,
- make the failure deterministic if possible.

D.8 The fastest fixes are usually design fixes

Memory bugs often disappear permanently when you improve design contracts:

- Replace manual `new/delete` with RAII ownership (`unique_ptr`, containers).
- Replace pointer+length with `std::span`.
- Replace “maybe-null” conventions with explicit types (`optional`, clear references).
- Replace uncontrolled sharing with explicit ownership boundaries.

D.9 One-page operational checklist

- ☐ Reproducer is minimal and repeatable.
- ☐ Built with symbols (`-g` or `/Zi`) and readable stack traces.
- ☐ ASan run first for memory errors (and stack-use-after-return if suspected).
- ☐ UBSan run for undefined behavior.
- ☐ TSan run for intermittent failures and suspected races.
- ☐ Fix addresses the root contract (ownership/bounds/concurrency), not just the symptom.
- ☐ Add a regression test and keep sanitizers in CI.

26.7 Appendix E — Glossary of Modern C++ Safety Terms

How to use this glossary

This glossary defines safety-related terms as they are used in modern professional C++ (C++11 through C++26-era practice). Definitions prioritize **precise engineering meaning**: what the term implies for correctness, lifetime, concurrency, interfaces, and tooling.

E.1 Memory, lifetime, and ownership terms

RAII (Resource Acquisition Is Initialization) A C++ technique that binds the lifetime of a resource (memory, file handle, socket, mutex lock, thread, etc.) to the lifetime of an object, so the resource is acquired during initialization and released in the destructor . RAII makes cleanup reliable under early returns and exceptions.

Resource handle An object that owns a resource and releases it in its destructor (an RAII type). Examples include `std::unique_ptr`, `std::lock_guard`, and user-defined handle wrappers .

Scoped object An object whose lifetime is governed by scope (automatic storage duration) so destruction is deterministic at scope exit. Scoped objects are the foundation that makes RAII reliable in practice .

Ownership A responsibility for ending a resource's lifetime (freeing, closing, unlocking, stopping). In professional C++, ownership is expected to be explicit in types and APIs (e.g., `std::unique_ptr` expresses unique ownership) .

Unique ownership Exactly one owner exists at a time. Transfer of ownership is explicit (move). The canonical tool is `std::unique_ptr` .

Shared ownership Multiple owners can keep an object alive independently. The canonical tool is `std::shared_ptr`. Shared ownership increases complexity and can create cycles; it must be a deliberate design choice .

Non-owning pointer A pointer that does not manage lifetime. In the Core Guidelines vocabulary, a raw pointer `T*` is non-owning by default and should not be used to express ownership .

Borrowing Temporary access to an object without owning it. Borrowing is typically expressed via references (`T&`) and sometimes raw pointers (`T*` when null is meaningful), or via views such as `std::span` and `std::string_view` .

Dangling pointer / dangling reference A pointer or reference that refers to an object whose lifetime has ended. Using it is a lifetime violation and often results in use-after-free or use-after-scope.

Use-after-free (UAF) Using memory after it has been released (e.g., after `delete` or after an owning handle releases it). This is a temporal memory safety bug. AddressSanitizer is designed to detect such bugs during testing .

Use-after-scope / use-after-return Using an object after its stack lifetime ended (e.g., returning a pointer to a local variable, or capturing a reference to a local in an asynchronous callback). Often detectable with AddressSanitizer configurations .

Double-free / invalid free Freeing the same allocation twice, or freeing memory not allocated by the corresponding allocator API. A classic symptom of ambiguous ownership. ASan commonly reports these errors .

Spatial safety Correctness of *where* memory is accessed (bounds). Typical failures are out-of-bounds reads/writes, buffer overflows, and indexing mistakes.

Temporal safety Correctness of *when* memory is accessed (lifetime). Typical failures are dangling pointers, use-after-free, and use-after-scope.

Out-of-bounds access Reading or writing outside the bounds of an array/object. This is a spatial safety bug; ASan is designed to detect many such cases .

Buffer overflow A write that exceeds the allocated bounds of a buffer. In C++ this is typically out-of-bounds write, and it can corrupt adjacent memory.

Object lifetime rules The rules that define when an object exists, when it can be accessed, and when it is destroyed. Violations often lead to undefined behavior.

Undefined behavior (UB) A situation where the C++ standard imposes no requirements on program behavior. UB is not a runtime event; it is a property of the program once it executes a forbidden operation (e.g., out-of-bounds, data race, double-free). Cppreference summarizes UB and its implications .

Indeterminate value / uninitialized read Using the value of an object that was never initialized (common with primitive types). Symptoms are often nondeterministic behavior and crashes. (Some categories of uninitialized use are also UB.)

Strict aliasing / type punning Rules about accessing an object through a different type than its actual type. Violations can trigger optimizer-dependent miscompilations. This is a common class of "works in Debug, fails in Release" UB.

Provenance The idea that a pointer's validity is tied to its origin (the object/allocation it came from). "Pointer-as-integer" tricks can break provenance assumptions and can lead to UB. In safety-critical C++ code, such patterns are typically isolated and heavily audited.

E.2 Range, view, and interface vocabulary

`std::span` A non-owning view over a contiguous sequence, carrying pointer + length as a single safe parameter object . Useful for bounds safety and clearer APIs.

`std::string_view` A non-owning view of a character sequence. Like all views, it does not extend lifetime; storing a string_view beyond the lifetime of the referenced string is a common lifetime bug.

Extent (static vs dynamic) For `std::span`, the extent is the number of elements: either encoded in the type (static extent) or stored at runtime (dynamic extent) .

Iterator invalidation Some container operations (reallocation, erase, insert) can invalidate iterators, pointers, and references into the container. Using invalidated iterators is a lifetime bug in practice.

Value semantics A design style where objects behave like values: copy/move creates independent objects (or well-defined shared states), and ownership is clear. Value semantics tends to reduce lifetime hazards.

Rule of Zero A design rule: prefer composing types that already manage resources correctly (standard containers, smart pointers) so you do not need to implement custom destructor/copy/move. This reduces lifetime bugs and makes exception safety easier.

Exception safety A guarantee about program state in the presence of exceptions:

- **Basic guarantee**: invariants preserved, no leaks.
- **Strong guarantee**: operation is atomic (commit or no effect).
- **No-throw guarantee**: operation never throws.

RAII is the core tool that makes these guarantees practical .

E.3 Concurrency and memory model terms

Data race Two conflicting accesses to the same memory location from different threads, at least one write, with no synchronization establishing ordering. In C++11 and later, if a data race occurs, the program's behavior is undefined .

Race condition A broader concept than data race: behavior depends on timing/order of events. Race conditions can exist even in data-race-free programs (e.g., higher-level protocol bugs).

Happens-before A relationship that provides ordering guarantees between evaluations in different threads. If one operation happens-before another, effects of the first are visible to the second in the required ways. Cppreference discusses happens-before in the context of the C++ memory model .

Synchronizes-with A specific relationship created by certain atomic operations (and mutex operations) that establishes happens-before between threads. For atomics, release/acquire is the common building block.

Atomic operation An operation that is indivisible with respect to other threads (in the model) and participates in the memory ordering rules. Atomics are not "faster mutexes"; they require precise correctness arguments.

`std::memory_order` An enumeration that specifies ordering constraints for atomic operations (relaxed, acquire, release, acq_rel, seq_cst) .

Release-acquire A common atomic synchronization pattern: a `store(..., release)` in one thread and a `load(..., acquire)` in another can establish ordering and visibility guarantees when the load observes the stored value.

Sequential consistency (SC) The strongest standard ordering (`memory_order_seq_cst`) providing a single total order of SC operations consistent with program order. Often easiest to reason about, sometimes higher cost than weaker orderings.

Mutex A mutual exclusion primitive (e.g., `std::mutex`) used to protect shared mutable state. Locks establish synchronization and avoid data races when used consistently.

Deadlock A liveness failure where threads wait forever due to cyclic lock dependencies. Deadlock is not a memory bug, but it is a critical correctness bug often intertwined with concurrency design.

Thread confinement A design pattern: an object is used by exactly one thread (or one thread at a time under a protocol), so no synchronization is needed internally. This is often the simplest way to avoid data races.

Immutable snapshot A concurrency pattern: share read-only data, replace by publishing new immutable versions rather than mutating shared state.

E.4 Error-handling and result vocabulary

`std::expected<T,E>` A vocabulary type that holds either a value `T` or an error `E`. It is never valueless . Used to make error contracts explicit without relying solely on exceptions.

`std::unexpected` A wrapper used to construct the error state of an `expected` .

Exception-neutral code Code that does not mishandle exceptions and maintains invariants and cleanup. RAII enables exception neutrality by ensuring cleanup via destructors.

Fail-fast A policy where violations of critical invariants abort execution rather than continuing in a corrupted state. Fail-fast can reduce security exposure and debugging time for safety-critical systems.

E.5 Tooling and verification terms

Static analysis Analysis of code without executing it, to detect likely bugs and enforce rules (including lifetime/ownership guidelines).

Sanitizer A compiler/runtime instrumentation tool that detects classes of bugs during execution of tests.

AddressSanitizer (ASan) A sanitizer designed to detect memory safety errors such as out-of-bounds accesses and use-after-free . It reports the invalid access and often the allocation/free stack traces.

UndefinedBehaviorSanitizer (UBSan) A sanitizer designed to detect many forms of undefined behavior at runtime (e.g., misaligned access, signed overflow), helping catch issues that may only appear under optimization .

ThreadSanitizer (TSan) A sanitizer designed to detect data races in multithreaded programs. Data races in C++11 and later imply undefined behavior .

Symbolization Turning raw addresses in crash/sanitizer reports into file/line/function information using debug symbols. Without symbolization, reports are often not actionable.

Core Guidelines Checker Tooling that enforces a subset of the C++ Core Guidelines rules (e.g., Microsoft C++ Code Analysis support) .

Clang-Tidy guideline checks Clang-Tidy includes checks with the `cppcoreguidelines-*` prefix aligned with guideline enforcement .

Regression test A test added after fixing a bug to ensure it cannot silently reappear. For memory safety, regression tests should be run under sanitizers.

E.6 Contracts, profiles, and evolution terms

Contract (engineering contract) A documented and enforceable statement of preconditions, postconditions, and invariants. In modern C++ practice, contracts are expressed via types, documentation, assertions, tests, and tools.

Language Contracts (P2900 family) A standardized facility proposal for expressing preconditions, postconditions, and assertions using dedicated syntax and semantics . (Deployment and compiler/library support vary by toolchain and version; treat it as an evolving ecosystem feature.)

Profile (Core Guidelines profile) A defined set of guidelines aimed at achieving particular guarantees (for example, bounds safety rules). The Core Guidelines use “profile” to describe applying a coherent subset of rules for a goal and tool enforcement .

Bounds safety profile A ruleset focused on preventing out-of-bounds and unsafe raw pointer usage. Microsoft documents certain checkers as belonging to bounds safety profile rules .

Future-proof code Code designed to remain correct and maintainable as compilers, standards, and platforms evolve. In safety terms, future-proofing means expressing intent in types, isolating low-level unsafe code, and relying on standard vocabulary types and tool-enforced rules.

E.7 Minimal code vocabulary examples (for quick recall)

E.7.1 RAII cleanup at scope exit

```
#include <mutex>

std::mutex m;

void f() {
  std::lock_guard<std::mutex> lock(m); // acquires in ctor, releases in
  ↪  dtor
  // safe under exceptions and early returns
}
```

E.7.2 Bounds-aware parameter with `std::span`

```
#include <span>
#include <cstddef>
```

```
std::size_t count_zeros(std::span<const int> xs) noexcept {
  std::size_t n = 0;
  for (int v : xs) if (v == 0) ++n;
  return n;
}
```

E.7.3 Explicit error result with `std::expected`

```
#include <expected>
#include <string_view>

enum class Err { empty, invalid };

std::expected<int, Err> parse_u32(std::string_view s) {
  if (s.empty()) return std::unexpected(Err::empty);
  int v = 0;
  for (char c : s) {
    if (c < '0' || c > '9') return std::unexpected(Err::invalid);
    v = v * 10 + (c - '0');
  }
  return v;
}
```

References

International Standards

- ISO/IEC 14882:2011(E) — Programming Languages — C++.
- ISO/IEC 14882:2014(E) — Programming Languages — C++.
- ISO/IEC 14882:2017(E) — Programming Languages — C++.
- ISO/IEC 14882:2020(E) — Programming Languages — C++.
- ISO/IEC 14882:2024(E) — Programming Languages — C++ (C++23 standard, published by ISO in 2024).

ISO C++ Committee (WG21) Papers and Reports

- Herb Sutter — Trip Report: June 2025 ISO C++ Standards Meeting (Sofia, Bulgaria). Notes completion of the C++26 feature freeze.
- WG21 N5016 — Minutes of the ISO C++ Committee Meeting, Sofia 2025.
- Tobias Doumler et al. — P2900R14: Contracts for C++.

- Tobias Doumler et al. — P2899R1: Contracts for C++ — Rationale.
- Bjarne Stroustrup — P3704R0: What Are Profiles?

Core Guidelines and Language Design

- Bjarne Stroustrup, Herb Sutter, and contributors — The C++ Core Guidelines (living document, July 2025 revision).

C++ Reference Documentation (cppreference)

- RAII (Resource Acquisition Is Initialization).
- Undefined Behavior in C++.
- Multi-threaded Execution and Data Races (since C++11).
- `std::memory_order` and the C++ Memory Model.
- `std::unique_ptr`.
- `std::shared_ptr`.
- `std::weak_ptr`.
- `std::make_unique`.
- `std::make_shared`.
- `std::enable_shared_from_this`.
- `std::span`.

- `std::string_view`.
- `std::expected` and `std::unexpected` (C++23).

Compiler Tooling and Sanitizers

- LLVM Project — AddressSanitizer (ASan) Documentation.
- LLVM Project — UndefinedBehaviorSanitizer (UBSan) Documentation.
- LLVM Project — ThreadSanitizer (TSan) Documentation.
- LLVM Project — Sanitizer Special Case Lists.
- Google Sanitizers Project — AddressSanitizer Overview.
- Google Sanitizers Project — ThreadSanitizer C++ Manual.

Microsoft C++ Toolchain Documentation

- Microsoft — AddressSanitizer for MSVC.
- Microsoft — Using the C++ Core Guidelines Checkers.
- Microsoft — C++ Core Guidelines Checker Reference (CoreCheck).

Secure Coding Standards

- SEI CERT — C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems in C++.