https://simplifycpp.org

DRAFT

Software Design with Modern C++

Principles, Architecture, and Engineering for Long-Lived Systems



Prepared by Ayman Alheraki

Software Design with Modern C++

Principles, Architecture, and Engineering for Long-Lived Systems

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

Prepared by Ayman Alheraki

Contents

Fornt Matter **65**

Foreword **65**

Preface: Why Software Design Fails — and How C++ Forces Clarity **74**

How to Read This Book **86**

Target Audience **94**

C++ Standards Covered (C++17 → C++26) **101**

Notation, Terminology, and Conventions **109**

I Foundations of Software Design **119**

1 What Software Design Really Is **120**

1.1 Design vs Coding vs Architecture 120

1.1.1 Three activities, three different levels of responsibility 120

1.1.2 A scale model: what changes when you move between levels 121

1.1.3 What each level produces . . . 122
1.1.4 Why design and architecture fail in real projects . . . 123
1.1.5 How Modern C++ forces clarity . . . 123
1.1.6 A concrete mapping: one requirement, three levels . . . 124
1.1.7 Design expressed as types and contracts (Modern C++ examples) . . . 125
1.1.8 Common misunderstandings and corrections . . . 127
1.1.9 Practical checklist: what to decide before you code . . . 127
1.1.10 Summary . . . 128
1.2 Design as Constraint Management . . . 129
1.2.1 What Are Constraints in Software Systems? . . . 129
1.2.2 Why Constraint Management Is Central to Design . . . 130
1.2.3 Types of Constraints in Software Design . . . 130
1.2.4 Constraints as Design Tools, Not Obstacles . . . 133
1.2.5 Constraint Visibility Through Types . . . 133
1.2.6 Constraint Enforcement at Compile Time vs Runtime . . . 134
1.2.7 Constraint Relaxation and Evolution . . . 134
1.2.8 Tradeoffs and Constraint Conflicts . . . 135
1.2.9 Constraint Management Examples . . . 135
1.2.10 Summary . . . 136
1.3 Design as Decision Making . . . 137
1.3.1 The core premise: everything that matters is a decision . . . 137
1.3.2 Deliberate decisions vs accidental decisions . . . 138
1.3.3 What makes a design decision “good”? . . . 139
1.3.4 The decision surface: where design actually happens . . . 139
1.3.5 Encoding decisions in Modern C++ . . . 141
1.3.6 The economics of decisions: why early clarity is cheaper . . . 143
1.3.7 Decision records: keeping intent alive . . . 144

1.3.8 A practical design-decision checklist 144
1.3.9 Summary 145
1.4 Why Design Matters More Than Language 146
1.4.1 Language is a tool; design is the system 146
1.4.2 What actually breaks long-lived systems 147
1.4.3 Why language debates are attractive but misleading 148
1.4.4 What language *can* do: amplify good design or amplify bad design . . . 148
1.4.5 Design dominates because change dominates 149
1.4.6 Design clarity is created at interfaces, not inside implementations . . . 149
1.4.7 The myth of "language as architecture" 150
1.4.8 Modern C++ as a design stress test 151
1.4.9 A design-first comparison: the same idea, different languages 151
1.4.10 When language matters more (and why it still does not replace design) . 152
1.4.11 Practical guidance: a design-first mindset for C++ systems 153
1.4.12 Summary 153
1.5 Why C++ Exposes Bad Design Faster Than Other Languages 154
1.5.1 C++ makes key engineering realities unavoidable 154
1.5.2 1) Ownership and lifetime: ambiguity becomes failure 155
1.5.3 2) Undefined behavior is a design stress test 156
1.5.4 3) The C++ memory model exposes concurrency mistakes early 157
1.5.5 4) Cost transparency punishes architectural ambiguity 158
1.5.6 5) Physical dependencies reveal coupling 159
1.5.7 6) ABI and binary boundaries punish unstable interfaces 160
1.5.8 7) Tooling and sanitizers make consequences measurable 161
1.5.9 C++ is not "hard"; vague design is hard 161
1.5.10 Design discipline: turning pressure into clarity 162
1.5.11 Summary 162

2 Complexity: The Real Enemy **164**

2.1 Accidental vs Essential Complexity 164

2.1.1 Definitions 164

2.1.2 Why this distinction matters in design 165

2.1.3 Where essential complexity comes from 166

2.1.4 Where accidental complexity comes from 166

2.1.5 A practical diagnostic: the "complexity budget" 167

2.1.6 Modern C++ and the two kinds of complexity 167

2.1.7 Example 1: Essential complexity represented cleanly 168

2.1.8 Example 2: Accidental complexity caused by unclear ownership 170

2.1.9 Example 3: Accidental complexity from "abstraction inflation" 171

2.1.10 Techniques for reducing accidental complexity 172

2.1.11 A decision framework: "Is this complexity essential?" 173

2.1.12 Summary 173

2.2 Scaling Code vs Scaling Teams 174

2.2.1 What it means to scale code 174

2.2.2 What it means to scale teams 175

2.2.3 Why scaling teams is harder than scaling code 176

2.2.4 Interfaces and boundaries: the nexus of code and team scaling 176

2.2.5 Dependency direction and team autonomy 178

2.2.6 Concurrency, teams, and merge contention 178

2.2.7 Build and modularity considerations 180

2.2.8 Onboarding and knowledge transfer 180

2.2.9 Governance, ownership, and team boundaries 181

2.2.10 Summary 182

2.3 Cognitive Load in Large Systems 182

2.3.1 What cognitive load means in software engineering 182

2.3.2 Why cognitive load explodes as systems grow . . . 183
2.3.3 Three kinds of load in code comprehension . . . 184
2.3.4 The cognitive units of a system: concepts, not classes . . . 184
2.3.5 High cognitive load smells in large systems . . . 185
2.3.6 How Modern C++ can reduce cognitive load . . . 186
2.3.7 Local reasoning: the primary cognitive scalability goal . . . 188
2.3.8 Reducing cognitive load in APIs: narrow, explicit, and stable . . . 189
2.3.9 Cognitive load and concurrency in large systems . . . 190
2.3.10 Build structure as cognitive structure . . . 191
2.3.11 Practical design techniques to lower cognitive load . . . 191
2.3.12 A cognitive-load checklist for design reviews . . . 192
2.3.13 Summary . . . 193
2.4 Entropy and Software Decay . . . 193
2.4.1 What “entropy” means in software . . . 193
2.4.2 Why decay is the default outcome . . . 194
2.4.3 The main mechanisms of software decay . . . 195
2.4.4 How to recognize entropy: practical signals . . . 197
2.4.5 Modern C++ as an entropy control tool . . . 198
2.4.6 Entropy is fought with *continuous design work* . . . 200
2.4.7 A practical anti-decay strategy for long-lived C++ systems . . . 201
2.4.8 Summary . . . 203
2.5 Why “It Works” Is Not a Design Argument . . . 203
2.5.1 Working is a baseline, not a justification . . . 203
2.5.2 Design is about properties beyond functionality . . . 204
2.5.3 Why “it works” persists: local success hides global cost . . . 205
2.5.4 C++ makes the gap between “works” and “is designed” painfully real . . . 205
2.5.5 What a design argument should sound like . . . 206

2.5.6 Example 1: “It works” but the ownership contract is undefined 207
2.5.7 Example 2: “It works” but concurrency is undefined 208
2.5.8 Example 3: “It works” but error semantics are incoherent 209
2.5.9 Design requires evidence: tests, measurement, and enforcement 209
2.5.10 A checklist: replacing “it works” with design-level questions 210
2.5.11 Summary 210

3 The Philosophy of Modern C++ Design 212
3.1 Zero-Cost Abstractions 212
3.1.1 What “zero-cost” means (and what it does not mean) 212
3.1.2 Why zero-cost abstractions matter for long-lived systems 213
3.1.3 The enabling mechanisms in Modern C++ 214
3.1.4 A cost model vocabulary: where overhead hides 217
3.1.5 Examples of zero-cost design in practice 217
3.1.6 When abstractions are *not* zero-cost 219
3.1.7 Zero-cost as a design method, not a slogan 220
3.1.8 A pragmatic guideline set for Modern C++ design 222
3.1.9 Summary 223
3.2 Pay-for-What-You-Use Principle 223
3.2.1 Why the principle exists: cost is a design constraint 224
3.2.2 The three main cost dimensions 224
3.2.3 How Modern C++ enables pay-for-what-you-use 226
3.2.4 Common violations: hidden costs disguised as convenience 229
3.2.5 Pay-for-what-you-use at the architecture level 231
3.2.6 A practical discipline: making cost part of design review 232
3.2.7 Summary 232
3.3 Determinism, Explicitness, and Control 233
3.3.1 Determinism: predictable behavior under defined contracts 233

3.3.2 Explicitness: design intent must be visible and checkable 235
3.3.3 Control: intentional tradeoffs and cost visibility 237
3.3.4 How the three pillars reinforce each other 239
3.3.5 What happens when these pillars are violated 239
3.3.6 Practical design techniques that embody the philosophy 240
3.3.7 Summary 241
3.4 RAII as a Design Philosophy 241
3.4.1 What counts as a “resource” 242
3.4.2 RAII: the deterministic contract 242
3.4.3 Why RAII is a design philosophy, not a cleanup trick 243
3.4.4 RAII and exception safety: correctness under non-linear control flow . . 243
3.4.5 RAII and invariants: “valid by construction” 244
3.4.6 RAII and ownership models: move-only design 245
3.4.7 RAII at architecture scale: the “composition root” 246
3.4.8 RAII and concurrency: locking, joining, and cancellation safety 248
3.4.9 Common RAII mistakes that create accidental complexity 249
3.4.10 RAII as a policy: making invalid states unrepresentable 249
3.4.11 A practical RAII checklist for design reviews 250
3.4.12 Summary 250
3.5 Compile-Time as a Design Tool 251
3.5.1 What “compile-time as a design tool” means 251
3.5.2 The compile-time toolbox in Modern C++ 252
3.5.3 1) Compile-time constraints: make requirements explicit 252
3.5.4 2) Compile-time invariants: invalid states become unrepresentable . . . 253
3.5.5 3) Compile-time computation: move work out of runtime 254
3.5.6 4) Policy selection without runtime branching 255
3.5.7 5) `static_assert`: design contracts that cannot be ignored 257

3.5.8 6) `constinit`: preventing static initialization hazards 258
3.5.9 7) Compile-time types as domain language 258
3.5.10 Costs and tradeoffs: compile-time is not free 259
3.5.11 A pragmatic design checklist 260
3.5.12 Summary 260

II Core Design Principles Revisited (Through C++) 261

4 Responsibility, Cohesion, and Boundaries 262
4.1 Single Responsibility Reinterpreted 262
4.1.1 Why the Classic SRP Needs a Modern Restatement 262
4.1.2 Responsibility as a Boundary, Not a Size Constraint 263
4.1.3 A Practical Definition of "One Responsibility" in C++ 263
4.1.4 Cohesion: The Technical Signal Behind SRP 264
4.1.5 The SRP Trap: "One Class, One Job" Is Too Naive 265
4.1.6 SRP Reinterpreted Through Modern C++ Concepts 265
4.1.7 How SRP Fails in the Real World: Common Anti-Patterns in C++ . . . 269
4.1.8 SRP at Multiple Scales: Class, Module, Component, Library 270
4.1.9 Engineering SRP: Boundary Techniques That Work in Modern C++ . . 271
4.1.10 A Decision Checklist: Does This Unit Violate SRP? 273
4.1.11 SRP Does Not Mean "No Collaboration" 274
4.1.12 Summary: SRP as "Axis of Change" for Long-Lived C++ Systems . . . 274
4.2 High Cohesion in Practice 275
4.2.1 Cohesion as the Engineering Signal Behind Good Design 275
4.2.2 Cohesion vs. Coupling: The Non-Negotiable Pair 275
4.2.3 Forms of Cohesion You Can Recognize in Code Reviews 276
4.2.4 The Core Mechanism of High Cohesion in C++: Invariants 277

4.2.5 API Vocabulary: Cohesion at the Interface Level 279
4.2.6 Cohesion at File and Header Level: Dependency Surface as a Design Boundary 280
4.2.7 High Cohesion in Components: “Stable Core, Volatile Edge” 280
4.2.8 Cohesion Under Concurrency: Thread-Safety Is a Responsibility Boundary 281
4.2.9 Measuring Cohesion: Practical Signals and Lightweight Metrics 283
4.2.10 Refactoring Toward High Cohesion: Patterns That Work 283
4.2.11 High Cohesion in Modern C++ Builds: Keep Public Surfaces Small . . 285
4.2.12 Review Checklist: “Is This Unit Cohesive?” 286
4.2.13 Summary: Cohesion as a Boundary Discipline 286
4.3 Responsibility vs Ownership 287
4.3.1 Why This Distinction Matters in Modern C++ 287
4.3.2 Definitions That Scale Beyond Single Classes 287
4.3.3 A Design Rule: Never Mix Policy and Lifetime Accidentally 288
4.3.4 Responsibility Without Ownership: “Rules Over Objects” 288
4.3.5 Ownership Without Responsibility: “RAII as a Mechanical Contract” . 290
4.3.6 Where Confusion Happens: The “Owner” Is Not Always the “Responsible” 291
4.3.7 Responsibility Boundaries: Who Owns the Invariant? 292
4.3.8 Ownership Models and Their Design Meaning 293
4.3.9 The Key Architectural Move: “Ownership at the Edge, Responsibility at the Core” 294
4.3.10 Case Study: Service Composition Root vs Domain Service 295
4.3.11 Design Smells: When Responsibility and Ownership Are Misaligned . 297
4.3.12 Practical Checklist for Reviews 298
4.3.13 Summary 298
4.4 Designing with Lifetimes 299
4.4.1 Lifetimes as a First-Class Design Dimension 299

4.4.2 The Three Questions That Drive Lifetime-Oriented Design 299
4.4.3 Lifetime Vocabulary: Values, Owners, Borrowers, and Views 300
4.4.4 The Default Rule: Prefer Value Semantics 301
4.4.5 RAII as the Lifetime Backbone 302
4.4.6 Owning Pointers: Expressing Exclusive and Shared Lifetime 303
4.4.7 Borrowing and Non-Owning Access: References, Raw Pointers, and Views 304
4.4.8 Lifetime-Aware Interfaces: Make Ownership Transfer Obvious 305
4.4.9 Lifetime at Architecture Scale: Scopes and Roots 307
4.4.10 Lifetimes and Concurrency: Avoid Lifetime Races 308
4.4.11 Aliasing as a Lifetime Risk: Design to Reduce It 310
4.4.12 Lifetime-Oriented Error Safety: The Strong Exception Guarantee as a Design Tool 310
4.4.13 Design Smells That Indicate Lifetime Problems 311
4.4.14 A Practical Design Checklist: Designing with Lifetimes 311
4.4.15 Summary 312

5 Coupling, Dependencies, and Direction 313
5.1 Physical vs Logical Dependencies 313
5.1.1 Two Kinds of Dependency Every Long-Lived C++ System Must Control 313
5.1.2 Logical Dependencies: The “Needs” Relationship 314
5.1.3 Physical Dependencies: The “Must Include/Link/Load” Relationship . 314
5.1.4 The Key Insight: Logical Decoupling Can Be Defeated by Physical Coupling 315
5.1.5 Dependency Direction: Logical Direction vs Physical Direction 316
5.1.6 A Mental Model: “Concept Graph” vs “Build Graph” 316
5.1.7 Common C++ Mechanisms That Create Physical Dependencies 317
5.1.8 Engineering Strategies to Align Physical With Logical Dependencies . 318
5.1.9 Recognizing Misalignment: Practical Smells 321

5.1.10 Design Checklist: Physical vs Logical Dependencies 322
5.1.11 Summary 323
5.2 Dependency Direction and Stability 323
5.2.1 Why Dependency Direction Is an Architectural Control Lever 323
5.2.2 Stability: What It Really Means in Engineering Terms 324
5.2.3 The Core Law: Dependencies Should Point Toward More Stable Code . 325
5.2.4 Two Kinds of Direction in C++: Logical and Physical 326
5.2.5 Stable Core vs Volatile Edge: A Practical Layering Model 326
5.2.6 The Dependency Inversion Move: Make Stable Policy Independent of Volatile Details 327
5.2.7 Stability and Build-Time Architecture: Headers Are Dependencies . . . 328
5.2.8 Stability as “Change Cost”: Fan-In and Fan-Out Intuition 329
5.2.9 Stability Patterns That Work in Modern C++ 329
5.2.10 Anti-Patterns: How Direction and Stability Fail 330
5.2.11 A Concrete Rule Set for C++ Teams 331
5.2.12 Worked Example: Inverting Direction to Stabilize the Core 332
5.2.13 Review Checklist: Direction and Stability 333
5.2.14 Summary 334
5.3 Header Dependencies vs Module Dependencies 334
5.3.1 Why This Comparison Matters in Long-Lived C++ Systems 334
5.3.2 Header Dependencies: Textual Inclusion and Its Consequences 335
5.3.3 Module Dependencies: Import-Based Semantics and Stronger Boundaries 337
5.3.4 Conceptual Mapping: Include Graph vs Import Graph 337
5.3.5 Code Illustration: Headers vs Modules (Same Intent, Different Physics) 338
5.3.6 Where Modules Improve Coupling and Direction 340
5.3.7 Where Headers Remain Dominant (Practical Reality) 341

5.3.8 Design Guidance: How to Engineer Dependency Direction in a Hybrid World . . . 342
5.3.9 Migration Strategy: From Headers to Modules Without Breaking the System . . . 343
5.3.10 Smells and Failure Modes . . . 344
5.3.11 Review Checklist: Header vs Module Dependencies . . . 344
5.3.12 Summary . . . 345
5.4 Designing for Replacement . . . 345
5.4.1 Replacement as the Real Definition of "Maintainability" . . . 345
5.4.2 The Replacement Contract: What Must Remain Stable . . . 346
5.4.3 Two Preconditions for Replaceability . . . 347
5.4.4 A Practical Architecture Pattern: Ports and Adapters (Through C++) . . . 348
5.4.5 Example: Designing a Storage Subsystem for Replacement . . . 348
5.4.6 Preventing Replacement Failure: Do Not Leak Vendor Types . . . 350
5.4.7 Replacement at the Physical Level: Headers, ABI, and Build Graph . . . 351
5.4.8 Designing for Replacement of Cross-Cutting Concerns . . . 351
5.4.9 Replacement Requires Testability: Contract Tests Over Implementation Tests . . . 353
5.4.10 Designing for Replacement Under Concurrency and Lifetimes . . . 354
5.4.11 Replacement Smells: Signals That a Subsystem Cannot Be Swapped . . . 355
5.4.12 A Practical Checklist for "Replaceability by Design" . . . 356
5.4.13 Summary . . . 356

6 Abstraction Without Damage **357**
6.1 Abstraction vs Indirection . . . 357
6.1.1 The Central Confusion: "Abstract" Is Not "Indirect" . . . 357
6.1.2 Abstraction: Compressing Meaning, Not Adding Layers . . . 358
6.1.3 Indirection: A Costly Power Tool . . . 360

6.1.4 The Rule: Use Abstraction to Remove Detail, Use Indirection to Enable Change . . . 361
6.1.5 Abstraction Mechanisms in Modern C++ (Often Without Indirection) . 362
6.1.6 Indirection Mechanisms in Modern C++ (Choose Intentionally) 363
6.1.7 The "Abstraction Damage" Failure Modes . . . 365
6.1.8 A Practical Decision Matrix (Through C++) . . . 366
6.1.9 Combining Both Safely: Abstraction With Controlled Indirection . . . 366
6.1.10 Review Checklist: Abstraction vs Indirection . . . 367
6.1.11 Summary . . . 367
6.2 Interfaces, Concepts, and Type Erasure . . . 368
6.2.1 Three Ways to Express "A Contract" in Modern C++ . . . 368
6.2.2 Interfaces: Runtime Contracts With Strong Boundary Properties 369
6.2.3 Concepts: Compile-Time Contracts With Zero Runtime Dispatch . . . 370
6.2.4 Type Erasure: Runtime Polymorphism Without Exposing Templates or Inheritance . . . 372
6.2.5 How to Choose: The Engineering Trade-Offs . . . 375
6.2.6 Avoiding Design Damage: Common Failure Modes . . . 376
6.2.7 A Robust Pattern: "Concepts Internally, Type Erasure at the Boundary" 377
6.2.8 Review Checklist: Interfaces, Concepts, Type Erasure . . . 378
6.2.9 Summary . . . 378
6.3 Virtual Dispatch vs Static Polymorphism . . . 379
6.3.1 Why This Choice Decides Whether Abstraction Helps or Hurts 379
6.3.2 Definitions: Two Ways to Substitute Behavior . . . 380
6.3.3 The Real Trade-Off: Runtime Flexibility vs Physical Dependency Cost . 380
6.3.4 Virtual Dispatch: When It Is the Correct Tool . . . 381
6.3.5 Static Polymorphism: When It Is the Correct Tool . . . 382
6.3.6 CRTP: Static Polymorphism Without Virtuals (But With Discipline) . . 384

6.3.7 Performance Reality: The Cost Is Often Not the Call Itself 385
6.3.8 Build and Boundary Reality: C++ Physical Dependencies Decide Scaling 386
6.3.9 Interface Evolution: Static vs Virtual Stability 386
6.3.10 A Strong Pattern: "Static Inside, Virtual Outside" 387
6.3.11 Worked Example: Virtual Boundary With Static Inner Loop 388
6.3.12 Design Checklist: Choosing Between Virtual and Static 389
6.3.13 Summary 390
6.4 When Abstraction Becomes a Liability 390
6.4.1 The Purpose of Abstraction and the Point Where It Breaks 390
6.4.2 Liability Category 1: Premature Abstraction (Wrong Contract Too Early) 391
6.4.3 Liability Category 2: Over-Generalization (The "Lowest Common Denominator" Interface) 392
6.4.4 Liability Category 3: Leaky Abstractions (Clients Still Need the Details) 393
6.4.5 Liability Category 4: Indirection Without Meaning (Layering as a Habit) 395
6.4.6 Liability Category 5: Abstraction That Breaks Local Reasoning (Hidden Global Effects) 396
6.4.7 Liability Category 6: Abstraction That Destroys Performance Predictability 397
6.4.8 Liability Category 7: Abstraction That Explodes Build Times and Physical Coupling 398
6.4.9 Liability Category 8: Abstraction That Freezes the Wrong Responsibility 399
6.4.10 Modern C++ Correctives: How to Build Abstraction Without Liability . 400
6.4.11 A Practical Checklist: "Is This Abstraction a Liability?" 401
6.4.12 Summary 402

7 Composition Over Inheritance (Correctly Applied) 403
7.1 Inheritance: Legitimate Uses Only 403
7.1.1 The Position of Modern C++: Inheritance Is Not Evil, but It Is Expensive 403
7.1.2 Two Kinds of Inheritance in C++: Implementation vs Interface 404

7.1.3 Legitimate Use #1: Pure Interface for Runtime Polymorphism at Boundaries 404
7.1.4 Legitimate Use #2: Non-Virtual Interface (NVI) to Enforce Invariants . 406
7.1.5 Legitimate Use #3: Traits/Mixins via CRTP (Static Polymorphism Inheritance) 407
7.1.6 Legitimate Use #4: Framework-Required Inheritance (Controlled Adapters Only) 408
7.1.7 Legitimate Use #5: Polymorphic Value Semantics (Rare, Carefully Designed) 409
7.1.8 Illegitimate (or High-Risk) Uses: What to Avoid 410
7.1.9 Composition Over Inheritance: The Real Meaning 412
7.1.10 Practical Rules for Teams (C++-Specific) 412
7.1.11 Review Checklist: “Is Inheritance Legitimate Here?” 413
7.1.12 Summary 413
7.2 Composition Patterns in Modern C++ 414
7.2.1 Composition as the Default Engineering Strategy 414
7.2.2 Pattern 1: Constructor Injection With Reference Semantics (Lightweight Composition) 415
7.2.3 Pattern 2: Owning Composition With Value Members (RAII Aggregation) 416
7.2.4 Pattern 3: Owning Composition With `std::unique_ptr` (Stable Addresses / Polymorphism) 418
7.2.5 Pattern 4: Policy-Based Design (Static Composition With Templates) . 419
7.2.6 Pattern 5: Decorator via Composition (Add Behavior Without Inheritance) 421
7.2.7 Pattern 6: Type Erasure as a Composition Boundary (Runtime Flexibility Without Inheritance) 422
7.2.8 Pattern 7: Composition Root (Wiring Dependencies in One Place) . . . 423
7.2.9 Pattern 8: Component + Facade (Contain Third-Party Dependency Spread) 424

7.2.10 Composition Under Modern Constraints: Lifetimes, Concurrency, and Build Graphs . . . 425
7.2.11 Composition Smells and Corrective Actions . . . 426
7.2.12 Review Checklist: Composition Patterns . . . 427
7.2.13 Summary . . . 427
7.3 Policy-Based Design . . . 428
7.3.1 What Policy-Based Design Means in Modern C++ . . . 428
7.3.2 Why Policy-Based Design Belongs in "Composition Over Inheritance" 429
7.3.3 Core Benefits (When Used Correctly) . . . 430
7.3.4 Primary Costs (The Design Must Pay Them Honestly) . . . 431
7.3.5 The Shape of a Policy: A Small, Focused Behavioral Unit . . . 431
7.3.6 Example 1: Logging Policy (No Runtime Dispatch) . . . 432
7.3.7 Example 2: Error Handling Policy (Return vs Throw vs Terminate) . . 433
7.3.8 Using Concepts to Make Policies Explicit and Safe . . . 435
7.3.9 Policy Interactions: Avoiding the "Combinatorial Explosion" . . . 436
7.3.10 Where Policy-Based Design Works Best . . . 436
7.3.11 Where Policy-Based Design Becomes Harmful . . . 437
7.3.12 Integration Pattern: Policy-Based Internals Behind a Stable Boundary . 438
7.3.13 Review Checklist: Policy-Based Design . . . 439
7.3.14 Summary . . . 439
7.4 Mixins, CRTP, and Compile-Time Composition . . . 440
7.4.1 Why Compile-Time Composition Exists . . . 440
7.4.2 Definitions . . . 441
7.4.3 Pattern 1: Basic Mixin for Capability Injection . . . 441
7.4.4 Pattern 2: CRTP as Static Polymorphism (Algorithm Skeleton Without Virtuals) . . . 442
7.4.5 Pattern 3: Stackable Mixins (Multiple Capabilities by Composition) . . 444

7.4.6 Concept-Constrained CRTP and Mixins (Modern C++ Discipline) . . . 445
7.4.7 Compile-Time Composition vs Runtime Composition: The Correct Boundary . . . 446
7.4.8 Failure Modes: When Mixins and CRTP Become a Liability . . . 447
7.4.9 Practical Rules for Safe Use in Production . . . 449
7.4.10 Review Checklist: Mixins and CRTP . . . 449
7.4.11 Summary . . . 449

III Object Model, Ownership, and Lifetime Design 451

8 Value Types vs Identity Types 452
8.1 What Makes an Object a "Value" . . . 452
8.1.1 The Engineering Definition of a Value Object . . . 452
8.1.2 Values vs Identities: The Boundary of Meaning . . . 453
8.1.3 The Canonical Properties of Value Types in Modern C++ . . . 453
8.1.4 A Value is Not "Small"; a Value is "Well-Behaved" . . . 455
8.1.5 Example: A Strong Value Object With an Invariant . . . 456
8.1.6 Non-Examples: When an Object Is Not a Value . . . 457
8.1.7 Value Types and Resource Ownership: The RAII Requirement . . . 458
8.1.8 The Rule of Zero and the Rule of Five: Value-Semantics Guidance . . . 459
8.1.9 Value Objects in Domain Modeling: "Entity vs Value" Mapping . . . 459
8.1.10 A Practical Checklist: "Is This a Value?" . . . 460
8.1.11 Design Guidance: How to Make Values "Strong" in C++ . . . 460
8.1.12 Summary . . . 461
8.2 Identity, Mutability, and Aliasing . . . 462
8.2.1 Why This Triangle Defines Real Software Failures . . . 462
8.2.2 Identity Types: What Makes an Object "The Same" Across Time . . . 463

8.2.3 Mutability: The Operational Reason for Identity 463
8.2.4 Aliasing: The Real Problem Most Systems Fail to Model 464
8.2.5 Identity + Mutability + Aliasing: The Most Dangerous Combination . . 465
8.2.6 Design Goal: Make Aliasing Explicit and Constrained 465
8.2.7 Aliasing and Ownership in Modern C++: What the Types Mean 466
8.2.8 Concrete Failure Mode 1: Shared Ownership Used as a Convenience . 466
8.2.9 Concrete Failure Mode 2: Borrowed Views Stored as If They Were Owned 468
8.2.10 Concrete Failure Mode 3: Container Invalidation and Aliasing 469
8.2.11 Identity Types: How to Model Them Safely in Modern C++ 470
8.2.12 A Safe Architectural Pattern: Identity Behind Handles 472
8.2.13 Practical Checklist: Designing Identity Types Without Aliasing Damage 474
8.2.14 Summary 474
8.3 Designing Value-Centric APIs 475
8.3.1 Why Value-Centric APIs Are the Default in Modern C++ 475
8.3.2 Core Principles of Value-Centric API Design 476
8.3.3 Practical API Shapes That Promote Value Semantics 477
8.3.4 Use `std::optional` and `std::expected`-style Results Instead of Sentinel Values 480
8.3.5 Strong Types: Make APIs Speak the Domain 482
8.3.6 Snapshot and Immutable Data Transfer Objects 483
8.3.7 Minimizing Hidden Aliasing in API Surfaces 484
8.3.8 Value-Centric APIs and Concurrency: Transfer, Do Not Share 484
8.3.9 Value-Centric APIs at Boundaries: Keep the Core Clean 485
8.3.10 Checklist: Designing a Value-Centric API 486
8.3.11 Summary 486
8.4 When Identity Is Unavoidable 487
8.4.1 Identity Is Not a Preference; It Is a Constraint 487

8.4.2 The Concrete Conditions That Force Identity 487
8.4.3 The First Rule of Identity: Do Not Pretend It Is a Value 491
8.4.4 Identity-Friendly C++ Design: The Standard Patterns 491
8.4.5 Identity and Concurrency: Choose One of Three Models 497
8.4.6 Smells That Indicate Identity is Uncontrolled 498
8.4.7 A Checklist: "Identity is Unavoidable, Is It Designed Correctly?" . . . 498
8.4.8 Summary 499

9 Ownership as a First-Class Design Concept 500
9.1 Ownership vs Lifetime 500
9.1.1 Why This Distinction Matters 500
9.1.2 Definitions: Two Different Questions 501
9.1.3 The Relationship: Ownership Influences Lifetime, But Does Not Define It 501
9.1.4 The Four Fundamental Roles in an Ownership Design 502
9.1.5 Ownership Models in Modern C++ 502
9.2 Unique Ownership Models 503
9.2.1 What "Unique Ownership" Means in System Design 503
9.2.2 The Core Unique-Ownership Representations in Modern C++ 504
9.2.3 Model 1: Unique Ownership by Value (The Default Model) 505
9.2.4 Model 2: Move-Only Resource Wrappers (Unique Ownership for Handles) 506
9.2.5 Model 3: `std::unique_ptr` (Exclusive Ownership of Dynamic Objects) 508
9.2.6 Model 4: Unique Ownership with Borrowed Access (Owner + Borrowers) 510
9.2.7 Model 5: Unique Ownership at the Composition Root 511
9.2.8 Unique Ownership and API Design: How to Make It Unambiguous . . 512
9.2.9 Common Pitfalls in Unique Ownership Models 513
9.2.10 Checklist: Is a Unique Ownership Model Correct Here? 513
9.2.11 Summary 514
9.3 Shared Ownership (And Its Real Cost) 515

9.3.1 Shared Ownership Is a Lifetime Policy, Not a Design Shortcut 515
9.3.2 When Shared Ownership Is Actually Appropriate 515
9.3.3 The Real Cost of Shared Ownership 516
9.3.4 The Two Meanings of "Shared": Do Not Mix Them 518
9.3.5 Safe Pattern: Shared Ownership of Immutable (or Effectively Immutable) State 519
9.3.6 Safe Pattern: Weak Observers to Break Cycles and Prevent Ownership Pollution 520
9.3.7 Danger Pattern: Shared Ownership as "Default Return Type" 520
9.3.8 Shared Ownership with Mutable Identity Objects: The High-Risk Case 521
9.3.9 Architecture Strategy: Confine `shared_ptr` to "Async Edges" 522
9.3.10 Alternatives to Shared Ownership (Often Better) 522
9.3.11 Checklist: Using Shared Ownership Without Losing Control 523
9.3.12 Summary 524
9.4 Ownership Graphs in Large Systems 524
9.4.1 Why "Ownership Graph" Is the Correct Mental Model 524
9.4.2 Two Graphs You Must Not Confuse 525
9.4.3 The Composition Root: Where Ownership Must Be Anchored 526
9.4.4 The Ideal Shape: A Tree With Borrowed Cross-Edges 526
9.4.5 A Practical Ownership Taxonomy for Large Systems 527
9.4.6 Visualizing Ownership Graphs as Edges and Contracts 529
9.4.7 Example: A Healthy Ownership Graph (Tree + Borrowing) 530
9.4.8 Failure Shape 1: Ownership Distributed by Convenience (`shared_ptr` Everywhere) 531
9.4.9 Failure Shape 2: Borrowers Escape Their Owners (Dangling Lifetimes) 532
9.4.10 Large-System Strategy: "Simple Ownership, Rich Access" 533
9.4.11 How to Engineer Ownership Graphs Deliberately 534

9.4.12 Checklist: Ownership Graph Quality in a Large System 535
9.4.13 Summary 536

10 RAII as a System-Level Design Tool **537**
10.1 RAII Beyond Memory 537
10.1.1 RAII as a General Resource Protocol 537
10.1.2 Why RAII Is a System-Level Design Tool 538
10.1.3 The Anatomy of a Correct RAII Type 539
10.1.4 RAII for Files and OS Handles 540
10.1.5 RAII for Locking and Concurrency Control 542
10.1.6 RAII for Transactions (Commit, Rollback, and Failure Semantics) . . . 543
10.1.7 RAII for Temporary State Changes (Restore-On-Exit) 544
10.1.8 RAII for Threads and Asynchronous Work 546
10.1.9 RAII and Exception Safety: The Architectural Guarantee 547
10.1.10 RAII at the Boundary: Adapters Around Unsafe APIs 547
10.1.11 Common RAII Design Mistakes (System-Level View) 548
10.1.12 Checklist: RAII Beyond Memory 549
10.1.13 Summary 549
10.2 Resources as Design Units 550
10.2.1 From “Objects” to “Resources” as the Primary Design Primitive 550
10.2.2 What Counts as a Resource in Modern C++ System Design 550
10.2.3 Why “Resources” Are Better Units Than “Classes” 551
10.2.4 The Resource Contract: Five Questions Every Resource Type Must Answer552
10.2.5 RAII as the Universal Mechanism for Resource Units 553
10.2.6 Designing a Resource Unit: The Minimal RAII Skeleton 553
10.2.7 Resources With Protocol: “Acquire/Use/Release” Is Not Enough 555
10.2.8 Resources and Boundaries: Where to Place the Resource Units 557
10.2.9 Resource Units and Ownership Graphs 557

10.2.10 Resource Units and Error Handling Strategy 558
10.2.11 The Hidden Resource: Invariants and “Permission to Use” 558
10.2.12 Common Mistakes When Resources Are Not Treated as Design Units . 559
10.2.13 Checklist: Resource-Unit Design Review 560
10.2.14 Summary 560
10.3 Deterministic Cleanup in Complex Systems 561
10.3.1 Why Deterministic Cleanup Is a System Requirement 561
10.3.2 The Three Cleanup Dimensions: Local, Subsystem, System 562
10.3.3 The Deterministic Cleanup Contract 563
10.3.4 Staged Teardown: Stop, Drain, Release 564
10.3.5 Encoding Teardown Stages in Types 565
10.3.6 Deterministic Cleanup Under Partial Initialization 566
10.3.7 Deterministic Cleanup and Callbacks: The Hidden Leak 568
10.3.8 Deterministic Cleanup and Deadlock Avoidance 570
10.3.9 Deterministic Cleanup and Shutdown Order (The Ownership Graph Problem) 571
10.3.10 Idempotent Cleanup: The Production-Grade Requirement 572
10.3.11 Checklist: Deterministic Cleanup in Complex Systems 572
10.3.12 Summary 573
10.4 RAII vs Garbage Collection Design Tradeoffs 573
10.4.1 Two Different Resource Models, Two Different System Behaviors . . . 573
10.4.2 What RAII Guarantees That GC Does Not 574
10.4.3 What GC Provides That RAII Does Not 576
10.4.4 The Core Tradeoff: Determinism vs Reachability-Based Reclamation . 577
10.4.5 Latency and Throughput: The System-Level Perspective 577
10.4.6 Memory Fragmentation and Locality 578
10.4.7 Non-Memory Resources: The Major Architectural Difference 579

10.4.8 Failure Modes and Operational Risk 580
10.4.9 Concurrency: Sharing vs Transfer 581
10.4.10 Expressing Intent: “Owning” vs “Borrowing” 582
10.4.11 Hybrid Architectures: RAII at the Core, GC at the Edge 582
10.4.12 Concrete Design Guidance: Choosing the Model 582
10.4.13 A C++ Lens: RAII Does Not Mean “Manual Everything” 583
10.4.14 Illustration: RAII for a Non-Memory Resource vs GC-Style Explicit Close 583
10.4.15 Checklist: Evaluating RAII vs GC for a Long-Lived System 585
10.4.16 Summary 586

IV Interfaces, APIs, and Contracts 587

11 Designing Stable Interfaces 588
11.1 API vs ABI 588
11.1.1 Why the Distinction Is Not Academic 588
11.1.2 Definitions 589
11.1.3 The Practical Matrix: API Compatibility vs ABI Compatibility 590
11.1.4 What Typically Belongs to the ABI Surface in C++ 591
11.1.5 Common ABI Breaks That Look “Harmless” at the Source Level 591
11.1.6 The C Boundary: Why C APIs Are Often Used for ABI Stability 593
11.1.7 C++ ABI Stability Strategy: Hide Layout, Expose Behavior 594
11.1.8 Versioning: You Cannot Avoid It 596
11.1.9 Where ABI Stability Matters Most 597
11.1.10 A Practical C++ Rule Set for Stable Interfaces 597
11.1.11 Checklist: “Is This an API Change or an ABI Change?” 599
11.1.12 Summary 599
11.2 Binary Compatibility Considerations 600

11.2.1 Binary Compatibility Is an Operational Contract 600
11.2.2 What “Binary Compatibility” Actually Depends On 600
11.2.3 Binary Compatibility Surfaces: What You Expose Becomes Your ABI . 601
11.2.4 The Most Common Binary Breaks in C++ Libraries 601
11.2.5 Binary Compatibility Design Strategies 604
11.2.6 A Binary Compatibility Policy: What You Must Freeze 607
11.2.7 Toolchain and Build-Mode Compatibility (Often Overlooked) 607
11.2.8 Symbol Visibility and ODR: Controlling What Becomes ABI 608
11.2.9 Exception and Error-Model Rules at Binary Boundaries 608
11.2.10 Ownership Rules Across Binary Boundaries 609
11.2.11 Designing for Additive Evolution Without ABI Break 609
11.2.12 Checklist: Binary Compatibility Review 610
11.2.13 Summary 611
11.3 Header-Only vs Compiled Interfaces 612
11.3.1 The Real Choice: “Where Does the Code Live?” 612
11.3.2 Header-Only Interfaces: What They Really Mean 613
11.3.3 Compiled Interfaces: What They Really Mean 613
11.3.4 The Key Engineering Dimensions 614
11.3.5 Concrete Example: Header-Only Utility vs Compiled Library Core . . 617
11.3.6 Where Header-Only Shines (Legitimate Use Cases) 618
11.3.7 Where Compiled Interfaces Win (Long-Lived System Reality) 619
11.3.8 The Hidden Cost of Header-Only: “Behavior Is Frozen into Clients” . . 619
11.3.9 A Mature Hybrid Pattern: Thin Headers, Compiled Core 620
11.3.10 Modern C++ Modules and Interface Strategy (Conceptual View) 620
11.3.11 Decision Checklist: Choosing Header-Only vs Compiled 621
11.3.12 Summary 621
11.4 Designing for Long-Term Evolution 622

11.4.1 Long-Term Evolution Is a Design Constraint, Not a Hope 622
11.4.2 The Three Surfaces You Must Manage 623
11.4.3 Principle: Prefer Additive Evolution Over Mutating Change 624
11.4.4 Make the Default Path Stable and Predictable 625
11.4.5 Design for Extension Points Up Front 625
11.4.6 Hide Representation; Expose Behavior 627
11.4.7 Stability Requires Versioning Rules 628
11.4.8 Contracts: Specify What Must Never Change 629
11.4.9 Avoid "Semantic Drift" Through Explicit Behavior Guarantees 629
11.4.10 Design the Error Boundary as a Stability Boundary 630
11.4.11 Prefer Value-Centric Interfaces for Stability 630
11.4.12 Minimize "Surface Area" and "Policy Exposure" 631
11.4.13 Evolution-Friendly Techniques in Modern C++ 631
11.4.14 Checklist: Is This Interface Designed to Evolve for 10+ Years? 632
11.4.15 Summary 633

12 Error Handling as Design **634**

12.1 Error Codes, Exceptions, and Expected 634
12.1.1 Error Handling Is Part of the Interface Contract 634
12.1.2 Three Models, Three Different Contracts 635
12.1.3 A Design Taxonomy of Failures 635
12.1.4 Error Codes: Strengths, Weaknesses, and Proper Use 637
12.1.5 Exceptions: Strengths, Weaknesses, and Proper Use 638
12.1.6 `std::expected`: Strengths, Weaknesses, and Proper Use 640
12.1.7 Choosing Between the Three: A Practical Policy 642
12.1.8 Performance and Latency: What Actually Matters 643
12.1.9 Preserving Context Without Breaking Stability 644
12.1.10 Compositional Patterns With `expected` 644

12.1.11 Designing No-Fail APIs: When `noexcept` Is a Contract 645
12.1.12 A Boundary Rule for Long-Lived Systems 646
12.1.13 Worked Example: One Operation, Three Error Models 646
12.1.14 Checklist: Selecting the Error Model for an Interface 647
12.1.15 Summary 648
12.2 Error Propagation Strategies 649
12.2.1 Propagation Is the Real Problem 649
12.2.2 The Propagation Contract: Identity, Context, and Actionability 650
12.2.3 Layering: Who Detects, Who Enriches, Who Handles? 650
12.2.4 Strategy 1: Return Status / Error Codes With Explicit Propagation . . . 651
12.2.5 Strategy 2: `std::expected` (Value-or-Error) Propagation 653
12.2.6 Strategy 3: Exceptions With Boundary Translation 655
12.2.7 Strategy 4: Error Mapping (From Low-Level to Domain Errors) 657
12.2.8 Strategy 5: Cause Chains Without Heap-Heavy Exceptions 658
12.2.9 Propagation and Logging: Log Once, With Full Context 659
12.2.10 Propagation and Retry Policy 659
12.2.11 Propagation Across Concurrency Boundaries 660
12.2.12 Propagation and ABI Boundaries 661
12.2.13 A Practical Unified Policy for Large Systems 661
12.2.14 Checklist: Error Propagation Design Review 662
12.2.15 Summary 663
12.3 Designing Failure-Aware APIs 663
12.3.1 Failure-Aware Means the API Makes Failure Explicit 663
12.3.2 The Failure-Aware Contract: Five Required Properties 664
12.3.3 Principle 1: Make Failure Visible in the Type System 664
12.3.4 Principle 2: Separate Programmer Errors from Operational Failures . . 666
12.3.5 Principle 3: Define Post-Failure Object State and Guarantees 666

12.3.6 Principle 4: Encode Ownership and Lifetime Rules for Failure Paths . . 667
12.3.7 Principle 5: Make Retry Semantics Explicit (Idempotency Matters) . . 668
12.3.8 Principle 6: Avoid "Boolean APIs" for Rich Failure Domains 669
12.3.9 Principle 7: Design Error Identity as Stable Domains and Codes 670
12.3.10 Principle 8: "Noexcept at Boundaries" as a Deployment Tool 671
12.3.11 Principle 9: "Handle Once, Propagate Many" 671
12.3.12 Worked Example: Failure-Aware Parse + Open + Serve 672
12.3.13 Checklist: Failure-Aware API Design Review 674
12.3.14 Summary 675
12.4 Making Errors Unignorable 675
12.4.1 Why "Unchecked Errors" Are a System Bug 675
12.4.2 Two Kinds of "Ignorable" Failures 676
12.4.3 Mechanism 1: `[[nodiscard]]` as a Contract Signal 677
12.4.4 Mechanism 2: Prefer `expected` Over Sentinel Values 678
12.4.5 Mechanism 3: Make the Success Path Ergonomic Without Allowing Silent Failure 679
12.4.6 Mechanism 4: Strengthen the Type System With Narrow Result Types . 680
12.4.7 Mechanism 5: Enforce Checking Through Project-Wide Policy 681
12.4.8 Mechanism 6: Design "Must-Use" RAII Guards for Critical Operations 682
12.4.9 Mechanism 7: Split Best-Effort APIs From Must-Succeed APIs 683
12.4.10 Mechanism 8: Prevent Partial Success From Looking Like Success . . 683
12.4.11 Mechanism 9: Use Explicit Boundary Functions for Handling 684
12.4.12 Checklist: Making Errors Unignorable 685
12.4.13 Summary 686

13 Contracts, Invariants, and Preconditions 687
13.1 Expressing Assumptions Explicitly 687
13.1.1 Assumptions Are Hidden Contracts 687

13.1.2 Three Places Where Assumptions Must Be Expressed 688
13.1.3 Assumptions as Preconditions, Postconditions, and Invariants 689
13.1.4 The Core Rule: “Make Invalid States Unrepresentable” 690
13.1.5 Expressing Assumptions in Function Signatures 691
13.1.6 Concepts: Expressing Compile-Time Assumptions in Generic Code . . 692
13.1.7 Runtime Checks: Defensive Validation at Boundaries 693
13.1.8 Assertions: Expressing Internal Assumptions and Invariants 694
13.1.9 Designing Types Around Invariants 695
13.1.10 Explicit Assumptions About Concurrency 697
13.1.11 Performance Assumptions as Contracts 698
13.1.12 An Interface Checklist for Explicit Assumptions 698
13.1.13 Summary 699
13.2 Runtime vs Compile-Time Contracts 699
13.2.1 Contracts Are Executable Assumptions 699
13.2.2 What Compile-Time Contracts Can Guarantee 700
13.2.3 What Runtime Contracts Can Guarantee 702
13.2.4 Runtime Checking Tools: Validation vs Assertions vs Termination . . . 704
13.2.5 Choosing Compile-Time vs Runtime: A Design Matrix 705
13.2.6 A Combined Example: Buffer Write Contract 706
13.2.7 Runtime Contracts and Performance Policy 707
13.2.8 Contracts Across ABI Boundaries 708
13.2.9 Checklist: Contract Placement Review 708
13.2.10 Summary 708
13.3 Defensive Design Without Paranoia 709
13.3.1 The Goal: Robustness With Clear Responsibility 709
13.3.2 A Practical Mental Model: Trust Zones 710
13.3.3 Defensive Design Starts With Clear Contracts 712

13.3.4 Principle 1: Validate at the Boundary, Normalize Once 712
13.3.5 Principle 2: Use Types to Eliminate Entire Categories of Checks 713
13.3.6 Principle 3: Separate Programmer Errors From Operational Failures . . 714
13.3.7 Principle 4: Avoid Redundant Checks That Hide Real Bugs 715
13.3.8 Principle 5: "Make Errors Unignorable" Instead of "Check More" . . . 716
13.3.9 Principle 6: Provide Checked Public APIs and Keep Unchecked Internals Private 716
13.3.10 Principle 7: Defensive Concurrency Requires Explicit Synchronization Contracts 718
13.3.11 Principle 8: "Fail Fast" Only Where Continuing Is Unsafe 718
13.3.12 Operational Discipline: Observability Over Overchecking 719
13.3.13 Checklist: Defensive Without Paranoia 719
13.3.14 Summary 720
13.4 Designing for Correctness 720
13.4.1 Correctness Is an Architectural Property 720
13.4.2 The Correctness Stack: From Types to Runtime to Process 721
13.4.3 Principle 1: Make Invalid States Unrepresentable 722
13.4.4 Principle 2: Express Ownership and Lifetime in the API 724
13.4.5 Principle 3: Separate Trusted Core From Untrusted Boundaries 724
13.4.6 Principle 4: Choose a Single, Coherent Error Model Per Boundary . . . 725
13.4.7 Principle 5: Define Postconditions and Failure Guarantees 726
13.4.8 Principle 6: Invariants by Construction and Encapsulation 727
13.4.9 Principle 7: Concurrency Correctness Requires Explicit Policy 728
13.4.10 Principle 8: Deterministic Resource Management as a Correctness Tool 729
13.4.11 Principle 9: Make Observability Part of Correctness 730
13.4.12 Principle 10: "Simple Core, Strict Boundary" Beats "Defensive Everywhere" 731

13.4.13 Checklist: Designing for Correctness 731
13.4.14 Summary 732

V Compile-Time Design and Metaprogramming 733

14 Type Systems as Design Languages 734
14.1 Types as Constraints 734
14.1.1 From “Type Checking” to “Design Checking” 734
14.1.2 Kinds of Constraints Expressible with Types 735
14.1.3 Strong Types: Domain Constraints by Construction 735
14.1.4 Types as Units: Constraint Composition 737
14.1.5 Concepts and Requires: Protocol Constraints with First-Class Language Support 738
14.1.6 Constraints as an Overload-Resolution Language 740
14.1.7 Constraints and Error Messages: Designing Diagnostics 742
14.1.8 Constraints vs Runtime Validation 743
14.1.9 Policy Types: Constraints on Configuration Space 744
14.1.10 Constraints and Performance Guarantees 745
14.1.11 The Metaprogramming View: Constraints as Compile-Time Proofs . . . 746
14.1.12 Practical Design Rules 746
14.1.13 Summary: Types as an Architectural Medium 747
14.2 Encoding Rules in Types 747
14.2.1 Why Encode Rules in Types? 747
14.2.2 Rule Categories: What Types Can Encode Reliably 748
14.2.3 Encoding Domain Rules with Strong Types 749
14.2.4 Encoding Unit Rules with Dimensional Types 750
14.2.5 Encoding State Rules with Typestate 751

14.2.6 Encoding Ownership Rules with RAII and Move-Only Types 754
14.2.7 Encoding “Valid-Only” Rules with Validated Types 755
14.2.8 Encoding Protocol Rules with Concepts 756
14.2.9 Encoding Capability Rules with Iterator and Callable Constraints . . . 757
14.2.10 Encoding Configuration Rules with Policy Types 757
14.2.11 Encoding “Non-Nullable” Rules in Interfaces 759
14.2.12 Encoding Error-Handling Rules with Type-Based Results 759
14.2.13 Tradeoffs: Where Type-Encoding Can Go Wrong 760
14.2.14 Design Checklist: Encoding Rules in Types 761
14.2.15 Summary 761
14.3 Preventing Invalid States 762
14.3.1 The Design Objective 762
14.3.2 A Practical Taxonomy of Invalid States 763
14.3.3 Make Illegal Combinations Impossible: Strong Types 763
14.3.4 Prevent Partially-Initialized Objects: Construction as a Gate 765
14.3.5 Prevent Wrong-Phase Calls: Typestate for Protocols 768
14.3.6 Prevent Null-Related Invalid States: Model Non-Null Explicitly 770
14.3.7 Prevent Lifetime Invalid States: RAII and Move-Only Ownership . . . 771
14.3.8 Prevent Invalid Values: Encode Ranges and Formats 772
14.3.9 Prevent Incoherent Template Configurations: Constrain Policies with Concepts 774
14.3.10 Prevent Silent Failure: Make Errors Non-Ignorable 775
14.3.11 When Invalid States Cannot Be Fully Removed 775
14.3.12 Engineering Checklist 776
14.3.13 Summary 776
14.4 Expressive APIs with Strong Types 777
14.4.1 Strong Types as an API Design Instrument 777

14.4.2 What “Expressive” Means in C++ APIs . . . 777
14.4.3 A Minimal Strong Type Wrapper . . . 778
14.4.4 Eliminating “Same-Representation” Bugs . . . 779
14.4.5 Named Parameters Without Language Support . . . 780
14.4.6 Strong Types for Units and Time . . . 781
14.4.7 Encoding Validation: “Trusted” Values Only . . . 782
14.4.8 Ownership and Nullability: Strong Types in Resource APIs . . . 784
14.4.9 Constrained Templates: Expressive Generic APIs . . . 785
14.4.10 Interoperation: Exposing the Underlying Representation Safely . . . 786
14.4.11 Design Guidelines for Strong-Type APIs . . . 786
14.4.12 Summary . . . 787

15 Templates as Architectural Tools **788**
15.1 Template Abstraction Layers . . . 788
15.1.1 Templates as Architecture, Not Syntax . . . 788
15.1.2 The Layering Model . . . 789
15.1.3 Layer 1: The Minimal Concept Layer . . . 790
15.1.4 Layer 2: Policy-Based Design (Compile-Time Configuration) . . . 790
15.1.5 Layer 3: Traits and Customization Points (Extensibility Without Inheritance) . . . 792
15.1.6 Layer 4: Static Polymorphism (CRTP) for Architectural Roles . . . 794
15.1.7 Layer 5: Algorithm Layers Over Views (Ranges and `std::span`) . . . 795
15.1.8 Where Template Layers Must Stop: Type Erasure Boundaries . . . 796
15.1.9 Composing Layers: A Worked Architectural Example . . . 798
15.1.10 Engineering Pitfalls and How to Avoid Them . . . 798
15.1.11 Design Checklist: Building Template Abstraction Layers . . . 799
15.1.12 Summary . . . 799
15.2 Generic Programming vs Metaprogramming . . . 800

15.2.1 Why This Distinction Matters Architecturally 800
15.2.2 Core Definitions 801
15.2.3 The Shared Mechanism: Templates, Different Goals 802
15.2.4 Generic Programming: Writing Algorithms Over Requirements 802
15.2.5 Metaprogramming: Shaping Programs at Compile Time 804
15.2.6 Where They Meet: "Metaprogramming in Service of Generic Programming" 806
15.2.7 Architectural Decision Guidance 807
15.2.8 Cost Model: Performance, Build Times, and Complexity 808
15.2.9 A Design Checklist for Teams 809
15.2.10 Summary 809
15.3 Concepts as Design Documentation 810
15.3.1 Concepts Are "Executable Documentation" 810
15.3.2 From Accidental Genericity to Intended Genericity 810
15.3.3 The Vocabulary Layer: Naming Architectural Roles 811
15.3.4 Concepts Clarify API Boundaries 812
15.3.5 Concepts Improve Diagnostics and Reduce Support Burden 813
15.3.6 Layering Requirements: Minimal, Composable Concepts 814
15.3.7 Documenting Performance Intent with Concepts 814
15.3.8 Documenting Safety and Exception Guarantees 815
15.3.9 Concepts as Architectural Guards on Policy Parameters 816
15.3.10 Avoiding Bad Concepts: Common Documentation Failures 817
15.3.11 Practical Guidelines for Using Concepts as Documentation 817
15.3.12 Summary 818
15.4 Reducing Template Complexity 818
15.4.1 Template Complexity Is an Architectural Cost 818
15.4.2 Common Sources of Template Complexity 819

15.4.3 Prefer Concepts Over SFINAE and Ad-Hoc Traits 820
15.4.4 Make Templates Shallow: Layer and Localize Genericity 821
15.4.5 Reduce Instantiation Count: Control the Combinatorics 822
15.4.6 Prefer "Normal C++" Over Template Cleverness 824
15.4.7 Avoid Specialization Networks as Control Flow 824
15.4.8 Keep Error Messages Human: Design for Diagnostics 825
15.4.9 Compilation Firebreaks: Separate Interface and Implementation 826
15.4.10 A Practical Strategy for Teams 827
15.4.11 Summary 828

16 Compile-Time Execution and Design Validation 829
16.1 constexpr and consteval Design 829
16.1.1 Compile-Time Execution as a Design Tool 829
16.1.2 Mental Models and Guarantees 830
16.1.3 Design Targets for Compile-Time Execution 830
16.1.4 Compile-Time Validation with `static_assert` 831
16.1.5 A `constexpr` Parser: Dual-Use Rule Checking 831
16.1.6 `consteval` for Non-Negotiable Rules 832
16.1.7 Compile-Time Tables and Design Consistency Checks 833
16.1.8 Using `constexpr` to Encode Deterministic Identifiers 835
16.1.9 `constexpr` Objects and Invariants 835
16.1.10 Architectural Pattern: Compile-Time Policy Validation 836
16.1.11 Guidelines: Choosing Between `constexpr` and `consteval` 837
16.1.12 Failure Modes and How to Avoid Them 838
16.1.13 Design Checklist 838
16.1.14 Summary 839
16.2 Compile-Time State Machines 839
16.2.1 Why State Machines Belong in the Type System 839

16.2.2 Two Compile-Time Approaches 840
16.2.3 Typestate: Enforcing Protocol Order by Construction 840
16.2.4 Ergonomics: Avoiding Typestate Friction 845
16.2.5 Compile-Time Validation of a State Machine Definition 846
16.2.6 Using the Validated Table for Runtime Dispatch 848
16.2.7 When Compile-Time State Machines Are Worth It 849
16.2.8 Engineering Guidelines 849
16.2.9 Summary 850
16.3 Static Reflection and Design Introspection 850
16.3.1 Reflection as an Architectural Capability 850
16.3.2 Modern C++ Reflection Model (C++26 Draft Direction) 851
16.3.3 Reflection Values and the `std::meta::info` Handle 852
16.3.4 Core Operations: Reflect, Query, and Splice 852
16.3.5 A First Example: "Access a Member by Reflected Name" 853
16.3.6 Design Introspection Use Cases 854
16.3.7 Introspection with Structural Rules: Compile-Time Validation 856
16.3.8 Controlled Generation: From Introspection to Synthesized Code 857
16.3.9 Annotations and Intent-Driven Introspection 858
16.3.10 Reflection vs Traditional Metaprogramming: Complexity and Clarity . 859
16.3.11 Practical Constraints and Engineering Discipline 859
16.3.12 Summary 860
16.4 Eliminating Runtime Errors by Design 861
16.4.1 The Architectural Goal 861
16.4.2 A Taxonomy of Runtime Errors You Can Eliminate 861
16.4.3 Technique 1: Strong Types to Eliminate Domain Confusion 862
16.4.4 Technique 2: Encode Units and Time in Types 863
16.4.5 Technique 3: Prevent Invalid States (Typestate Protocols) 864

16.4.6 Technique 4: Make Construction a Validation Gate 866
16.4.7 Technique 5: Eliminate Ownership Bugs with RAII and Move-Only Handles 867
16.4.8 Technique 6: Make Errors Non-Ignorable (Typed Error Channels) . . . 868
16.4.9 Technique 7: Constrain Generic APIs with Concepts 869
16.4.10 Technique 8: Compile-Time Validation with `constexpr` and `consteval` 870
16.4.11 What Cannot Be Eliminated (And How to Contain It) 871
16.4.12 A Unified Design Checklist 871
16.4.13 Summary 872

VI Concurrency and Parallel Design 873

17 Concurrency as a Design Problem 874
17.1 Concurrency vs Parallelism 874
17.1.1 Why This Distinction Is a Design Requirement 874
17.1.2 Definitions 874
17.1.3 A Useful Mental Model 875
17.1.4 Concurrency Without Parallelism: The Event Loop 876
17.1.5 Parallelism Without Much Concurrency: Data-Parallel Work 877
17.1.6 Where the Design Difficulty Actually Lives 878
17.1.7 A Practical Comparison of Design Concerns 879
17.1.8 Implications for C++ Design (Memory Model and Data Races) 880
17.1.9 Design Patterns That Separate Concerns 880
17.1.10 When You Need Concurrency, When You Need Parallelism 883
17.1.11 Engineering Checklist 884
17.1.12 Summary 884
17.2 Data Races as Design Failures 885

17.2.1 Why Data Races Are Architectural, Not Accidental 885
17.2.2 The C++ Consequence: Undefined Behavior 885
17.2.3 Definition by Example: What a Data Race Looks Like 886
17.2.4 The Root Cause: Shared Mutable State 887
17.2.5 Design Principle 1: Confinement (Single-Thread Ownership) 887
17.2.6 Design Principle 2: Make Sharing Explicit and Narrow 889
17.2.7 Design Principle 3: Use Atomics for Coordination, Not for General Sharing 890
17.2.8 Design Principle 4: Publication and Initialization Are Protocols 891
17.2.9 Design Principle 5: Data Races Are Often API Design Bugs 892
17.2.10 Thread-Safety Levels as Design Contracts 892
17.2.11 Detecting Races Is Not Preventing Them 893
17.2.12 A Concurrency Design Checklist (Race Prevention) 893
17.2.13 Summary 894
17.3 Ownership and Synchronization 894
17.3.1 Why Ownership Is the Real Concurrency Primitive 894
17.3.2 A Working Vocabulary 895
17.3.3 The Core Rule: No Shared Mutable State Without a Protocol 896
17.3.4 Ownership Models and Their Synchronization Implications 896
17.3.5 Safe Publication: Making Objects Visible Across Threads 900
17.3.6 Synchronization Strategies Derived From Ownership 901
17.3.7 Lock Ordering and Deadlocks as Ownership Violations 903
17.3.8 Read-Mostly Ownership: Immutable Snapshots 904
17.3.9 API Design Rules: Make Thread-Safety a Contract 905
17.3.10 Engineering Checklist 905
17.3.11 Summary 906
17.4 Designing Thread-Safe APIs 907
17.4.1 Thread-Safety Is a Public Contract 907

17.4.2 Define the Thread-Safety Level Explicitly 907
17.4.3 Choose a Concurrency Model Before Writing the API 908
17.4.4 Rule 1: Keep Invariants Behind a Single Synchronization Boundary . . 909
17.4.5 Rule 2: Avoid Leaking References, Iterators, and Raw Pointers to Internals 910
17.4.6 Rule 3: Never Call User Code While Holding a Lock 911
17.4.7 Rule 4: Make Blocking Behavior Explicit 912
17.4.8 Rule 5: Separate Lifetime Safety from Data Safety 914
17.4.9 Rule 6: Define Atomicity at the API Level 915
17.4.10 Rule 7: Prefer Message Passing for Complex Shared Invariants 916
17.4.11 Rule 8: Establish Lock Ordering and Keep Lock Scope Small 918
17.4.12 Testing and Verification Are Part of the API Design 918
17.4.13 A Practical Thread-Safe API Checklist 918
17.4.14 Summary 919

18 Modern C++ Concurrency Models 920
18.1 Message Passing vs Shared State 920
18.1.1 The Architectural Choice That Determines Everything 920
18.1.2 Definitions 921
18.1.3 The Cost Model: What You Pay For 921
18.1.4 Shared State in Practice: A Thread-Safe Shared Cache 922
18.1.5 Message Passing in Practice: A Confined Cache Actor 923
18.1.6 Design Trade-offs: Correctness vs Complexity 926
18.1.7 Choosing the Model: A Practical Decision Framework 927
18.1.8 Hybrid Architectures (Common in Production) 928
18.1.9 Backpressure: The Missing Design Requirement in Message Passing . 928
18.1.10 API Design Rules for Both Models 930
18.1.11 Summary 931
18.2 Futures, Tasks, and Executors 931

18.2.1 Why These Three Concepts Form a Concurrency Model 931
18.2.2 Futures and Promises: The Typed Completion Channel 932
18.2.3 `std::async`: Convenient Tasks With a Hidden Executor 933
18.2.4 Tasks: Defining Work Units Correctly 934
18.2.5 Blocking Semantics: `get()`, `wait()`, and Timeouts 935
18.2.6 Continuations: The Missing Piece in `std::future` 936
18.2.7 Executors: Making "Where It Runs" a First-Class Policy 936
18.2.8 A Simple Thread-Pool Executor 937
18.2.9 Tasks + Futures on an Executor: Building a Result-Producing API . . . 940
18.2.10 Cancellation and Cooperative Stop 941
18.2.11 Design Rules for Futures, Tasks, and Executors 942
18.2.12 Summary 943
18.3 Lock-Free Design Principles 943
18.3.1 Lock-Free Is a Performance Tool, Not a Default 943
18.3.2 Progress Guarantees: Blocking, Lock-Free, Wait-Free 944
18.3.3 The Memory Model Foundation You Must Respect 944
18.3.4 Principle 1: Keep the Shared State Minimal and Atomic 945
18.3.5 Principle 2: Prefer Acquire/Release to Express Publication 946
18.3.6 Principle 3: Design for Contention and Backoff 947
18.3.7 Principle 4: Define Linearization Points 948
18.3.8 Principle 5: Memory Reclamation Is Part of the Algorithm 949
18.3.9 Principle 6: Prefer Compositional Designs Over "One Big Lock-Free Structure" 950
18.3.10 Principle 7: Do Not Use Lock-Free to Avoid Thinking About Locks . . 951
18.3.11 A Concrete Lock-Free Example: Treiber Stack (Core Idea) 951
18.3.12 Practical Lock-Free Design Checklist 953
18.3.13 Summary 953

18.4 Structured Concurrency 954
18.4.1 What “Structured” Means in Concurrency 954
18.4.2 The Architectural Objectives 955
18.4.3 The Canonical Anti-Pattern: Detached Work 955
18.4.4 A Practical C++ Foundation: Scope-Bound Joining 956
18.4.5 Structured Concurrency Needs Cancellation 958
18.4.6 A Structured Task Scope (Join + Cancellation) 959
18.4.7 Structured Concurrency as a Tree 961
18.4.8 The Role of Executors 961
18.4.9 Time Bounds: Deadlines and Timeouts 962
18.4.10 Avoiding Concurrency Leaks in API Design 962
18.4.11 Testing Advantages 963
18.4.12 Engineering Checklist 963
18.4.13 Summary 964

19 Performance-Driven Design **965**
19.1 Latency vs Throughput 965
19.1.1 Why This Trade-off Is Architectural 965
19.1.2 Precise Definitions 965
19.1.3 The Queueing Reality: Why Throughput Can Harm Latency 967
19.1.4 Concurrency and the Latency/Throughput Dial 967
19.1.5 Work Granularity: Too Small vs Too Large 968
19.1.6 The Core Design Patterns 968
19.1.7 Synchronization Costs: When a Lock Becomes a Latency Source . . . 972
19.1.8 CPU Caches and False Sharing 972
19.1.9 Measurement That Matches the Goal 973
19.1.10 A Decision Framework for Architects 974
19.1.11 Engineering Checklist 976

19.1.12 Summary 976
19.2 Cache-Aware Design 977
19.2.1 Why Caches Are a Design Constraint 977
19.2.2 The Practical Cache Model for Designers 977
19.2.3 The Two Golden Rules 978
19.2.4 Data Layout: AoS vs SoA 978
19.2.5 Pointer Chasing vs Streaming Access 980
19.2.6 Working Set Size and Hot/Cold Splitting 981
19.2.7 False Sharing: The Silent Throughput Killer 981
19.2.8 Cache Coherence and Shared Mutable State 983
19.2.9 Prefetching and Access Regularity 984
19.2.10 Allocation Strategy and Locality 984
19.2.11 NUMA Awareness (When It Matters) 985
19.2.12 A Practical Cache-Aware Design Checklist 985
19.2.13 Summary 986
19.3 Memory Layout as Architecture 986
19.3.1 The Architectural Claim 986
19.3.2 The Design Surface of Memory Layout 987
19.3.3 Hot/Cold Splitting as an Architectural Pattern 988
19.3.4 Value Layout vs Reference Layout 989
19.3.5 AoS vs SoA Is an Architectural API Decision 990
19.3.6 Alignment, Padding, and the Cost of “Accidental Coupling” 991
19.3.7 False Sharing as an Architectural Bug 992
19.3.8 Lifetime Grouping: Allocation Strategy as Architecture 993
19.3.9 APIs That Respect Layout: Views and Spans 994
19.3.10 Concurrency Model Follows Layout 995
19.3.11 Design Checklist: Layout-Driven Architecture 995

19.3.12 Summary 996
19.4 Measurement-Driven Decisions 997
19.4.1 Performance Without Measurement Is Fiction 997
19.4.2 Define the Performance Contract First 997
19.4.3 Measure What the User Experiences 998
19.4.4 Latency: Percentiles, Not Averages 999
19.4.5 Throughput: Sustain Under Load, Not Burst 1000
19.4.6 The Overload Curve: Measure Across Utilization Levels 1001
19.4.7 Design Decisions That Require Measurement 1001
19.4.8 Benchmarking Methodology: Avoid Self-Deception 1002
19.4.9 Instrumentation: Knowing Where Time Went 1004
19.4.10 A Practical Profiling Strategy (Architecture-Level) 1004
19.4.11 A/B and Regression Testing for Performance 1005
19.4.12 From Measurements to Decisions: A Decision Table 1005
19.4.13 Engineering Checklist 1007
19.4.14 Summary 1007

VII Architectural Styles in C++ 1008

20 Layered and Modular Architectures 1009
20.1 Traditional Layering Revisited 1009
20.1.1 Why Layering Still Matters (When Done Correctly) 1009
20.1.2 What a Layer Is (Architecturally) 1010
20.1.3 Classical Layering (and Why It Breaks in C++) 1010
20.1.4 Layering in Modern C++: The Updated Rules 1011
20.1.5 A Layered Example With Explicit Boundaries 1013
20.1.6 Layering vs Performance: The Modern Resolution 1015

20.1.7 Layering vs Concurrency: Make the Model Explicit 1016
20.1.8 Revisiting “Strict” Layering: Practical Exceptions 1016
20.1.9 Modern Layering Tools in C++ 1017
20.1.10 Layer Health Metrics (What to Watch Over Time) 1017
20.1.11 Engineering Checklist 1018
20.1.12 Summary 1018
20.2 Dependency Inversion in Practice 1019
20.2.1 The Architectural Problem Dependency Inversion Solves 1019
20.2.2 What Dependency Inversion Is (and Is Not) 1020
20.2.3 The Most Important Practical Principle: Own the Interface in the Stable Layer 1020
20.2.4 A Complete Example (Domain/Application Own the Contract) 1021
20.2.5 Composition Root: Where the System Is Wired Together 1024
20.2.6 Dependency Inversion and Header/Build Hygiene 1025
20.2.7 When Virtual Interfaces Are Not Desired 1026
20.2.8 Concurrency and Dependency Inversion 1027
20.2.9 Common Failure Modes (and How to Avoid Them) 1028
20.2.10 Engineering Checklist 1029
20.2.11 Summary 1029
20.3 C++ Modules as Architectural Boundaries 1030
20.3.1 Why Modules Change Architecture (Not Just Build Speed) 1030
20.3.2 The Mental Model: From Files to “Packages” 1031
20.3.3 Core Building Blocks You Must Know 1032
20.3.4 A Minimal Named Module as a Boundary 1033
20.3.5 Facade Modules: A Practical Architectural Pattern 1034
20.3.6 Partitions as Layering Inside a Module 1034
20.3.7 How Modules Enforce Architectural Hygiene 1036

20.3.8 Build System Reality: Modules Require Dependency Scanning 1037
20.3.9 Common Architectural Pitfalls (And How to Avoid Them) 1038
20.3.10 Migration Strategy for Long-Lived Codebases 1040
20.3.11 Engineering Checklist 1041
20.3.12 Summary 1041
20.4 Preventing Architecture Erosion 1042
20.4.1 Architecture Erosion: The Default Outcome if You Do Nothing 1042
20.4.2 Designing for Non-Erosion: The Three Defense Layers 1043
20.4.3 Principle 1: Make Boundaries Explicit and Small 1043
20.4.4 Principle 2: Enforce Dependency Direction in the Build System 1044
20.4.5 Principle 3: Treat Headers as Architecture (Because They Are) 1045
20.4.6 Principle 4: Dependency Inversion Is the “Erosion Firewall” 1046
20.4.7 Principle 5: Make Concurrency Contracts Part of the Architecture . . . 1047
20.4.8 Principle 6: Institutionalize Architectural Decisions 1047
20.4.9 Automated Defense: Executable Architectural Rules 1048
20.4.10 Human Defense: Code Review With Structural Questions 1050
20.4.11 The Three Classic Erosion Traps (and Their Repairs) 1050
20.4.12 Modules and Targets as Long-Term Boundary Reinforcement 1052
20.4.13 A Practical Anti-Erosion Program 1052
20.4.14 Summary 1053

21 Component-Based and Plugin Architectures 1054
21.1 Runtime Extensibility 1054
21.1.1 What Runtime Extensibility Means 1054
21.1.2 The Three Axes of Extensibility 1055
21.1.3 Runtime Extensibility vs Compile-Time Extensibility 1056
21.1.4 The Plugin Boundary: Define a Minimal Stable Contract 1057
21.1.5 ABI Reality: Why C Interfaces Are Common for Plugins 1057

21.1.6 A Practical Minimal Plugin ABI 1058
21.1.7 Plugin Implementation: C ABI Outside, C++ Inside 1060
21.1.8 Host Loader: Discovery, Loading, Validation 1062
21.1.9 Versioning Strategy: Compatibility Is a Design Feature 1063
21.1.10 Lifecycle: Ownership and Deterministic Unload 1064
21.1.11 Threading and Concurrency Contracts for Plugins 1065
21.1.12 Security and Trust: Plugins Are Code Injection by Default 1065
21.1.13 Designing Extension Points: Stable Core, Flexible Edges 1066
21.1.14 Common Failure Modes (and Their Architectural Fixes) 1067
21.1.15 Engineering Checklist 1067
21.1.16 Summary 1068
21.2 ABI-Safe Plugin Design 1069
21.2.1 Why ABI Safety Is the Core Problem 1069
21.2.2 The Two Golden Strategies 1069
21.2.3 ABI Stability Principles 1070
21.2.4 A Reference ABI Contract (C ABI, Versioned, Extensible) 1072
21.2.5 Plugin-Side Implementation (C ABI Outside, Modern C++ Inside) . . . 1075
21.2.6 Host-Side Safety Rules 1078
21.2.7 Cross-Platform ABI Pitfalls and Defenses 1079
21.2.8 Designing the ABI Surface: Minimal and Stable 1080
21.2.9 Feature Negotiation Without ABI Breaks 1081
21.2.10 Threading and Reentrancy Contracts 1081
21.2.11 Testing ABI Safety 1082
21.2.12 Engineering Checklist 1083
21.2.13 Summary 1083
21.3 Versioning and Compatibility 1084
21.3.1 Compatibility Is a First-Class Architectural Feature 1084

21.3.2 Terminology: Distinguish the Four Kinds of Versioning 1084
21.3.3 The Compatibility Promise: Define What You Guarantee 1086
21.3.4 The Practical Rule: ABI Versions Change Only on Breaking Changes . 1086
21.3.5 The Foundation: Negotiation on Load 1087
21.3.6 A Robust Pattern: Size-Tagged Structs and Function Tables 1087
21.3.7 Semantic Versioning: Useful, but Not Sufficient 1090
21.3.8 Feature Negotiation: Compatibility Without Rebuilds 1090
21.3.9 Multi-Version Interfaces: Supporting Old and New Simultaneously . . 1091
21.3.10 Compatibility of Data Formats: Versioned Serialization 1093
21.3.11 Deprecation and Removal: Managing the Compatibility Window 1093
21.3.12 Failure Handling: Compatibility Must Fail Safe 1094
21.3.13 Practical Compatibility Checklist for C++ Plugin Ecosystems 1094
21.3.14 Summary 1095
21.4 Dependency Isolation 1096
21.4.1 What Dependency Isolation Means in Plugin Systems 1096
21.4.2 The Four Dependency Planes You Must Isolate 1096
21.4.3 The Stable SDK: A Hard Boundary Between Host and Plugins 1098
21.4.4 Preventing Compile-Time Leakage 1099
21.4.5 Link-Time Isolation: Avoid Linking Plugins to Host Implementation . . 1100
21.4.6 Load-Time Isolation: Preventing Dependency Collisions 1101
21.4.7 Runtime Isolation: Preventing Global State Contamination 1103
21.4.8 The “No Shared CRT” and “No Mixed Allocator” Rules 1104
21.4.9 Avoiding ODR-Style Collisions Across Components 1105
21.4.10 Isolation Between Plugins: Preventing Plugin-to-Plugin Coupling . . . 1105
21.4.11 Deployment Isolation: Packaging and Reproducibility 1106
21.4.12 Testing Dependency Isolation 1107
21.4.13 Engineering Checklist 1107

21.4.14 Summary 1108

22 Data-Oriented Design 1109

22.1 Object-Oriented vs Data-Oriented Thinking 1109

22.1.1 The Real Difference: What You Optimize For 1109

22.1.2 Object-Oriented Thinking: The Classical Mental Model 1109

22.1.3 Data-Oriented Thinking: The Performance Mental Model 1111

22.1.4 The Core Tension: Encapsulation vs Locality 1112

22.1.5 A Concrete Illustration in Modern C++ 1113

22.1.6 A Key DOD Technique: Structure of Arrays (SoA) vs Array of Structures (AoS) 1115

22.1.7 Identity and References: Pointers vs Handles 1116

22.1.8 How to Decide: A Practical Decision Framework 1117

22.1.9 The Hybrid Reality: OOD as Policy, DOD as Engine 1118

22.1.10 Engineering Checklist 1118

22.1.11 Summary 1119

22.2 Designing for Hardware 1119

22.2.1 The Premise: Hardware Is the Ground Truth 1119

22.2.2 The Hardware Constraints That Matter Most 1120

22.2.3 The Central Practice: Layout Data by Access Pattern 1123

22.2.4 Reducing Pointer Chasing: Indices and Handles 1124

22.2.5 Avoiding False Sharing: Aligning and Sharding 1125

22.2.6 Batching: Amortize Overhead and Improve Prefetch 1126

22.2.7 Cache-Friendly Algorithms: Sort, Group, and Stream 1127

22.2.8 I-Cache and Code Layout: Keep Hot Code Small 1127

22.2.9 Measurement: Hardware-Aware Design Must Be Evidence-Based . . . 1128

22.2.10 A Practical Hardware-Driven Design Workflow 1130

22.2.11 Engineering Checklist 1130

22.2.12 Summary 1131
22.3 ECS and Modern C++ 1131
22.3.1 Why ECS Exists: Data-Oriented Architecture at Scale 1131
22.3.2 The ECS Mental Model (In Hardware Terms) 1132
22.3.3 ECS as an Architectural Decomposition 1133
22.3.4 Modern C++ Enables Practical ECS Designs 1134
22.3.5 A Minimal ECS Core (Educational, Not a Full Engine) 1134
22.3.6 Archetypes vs Sparse Sets: Two Major ECS Storage Families 1139
22.3.7 ECS and Strong Typing: Safer Queries with Modern C++ 1140
22.3.8 Scheduling and Parallelism: Systems as Units of Work 1140
22.3.9 ECS and Resource Management in C++ 1141
22.3.10 Where ECS Fits in Long-Lived Systems 1142
22.3.11 Engineering Checklist 1143
22.3.12 Summary 1143
22.4 When Data Beats Objects 1144
22.4.1 The Practical Claim 1144
22.4.2 A Decision Framework: Identify the DOD Sweet Spot 1144
22.4.3 Why Objects Lose in These Cases 1146
22.4.4 A Concrete Demonstration: Same Problem, Two Shapes 1147
22.4.5 Canonical Situations Where Data Beats Objects 1149
22.4.6 A More Subtle Point: DOD Can Improve Maintainability Too 1151
22.4.7 The Hybrid Architecture Rule 1151
22.4.8 Practical Refactoring Steps: From Objects to Data (Safely) 1152
22.4.9 Engineering Checklist 1153
22.4.10 Summary 1153

VIII Large-Scale System Design 1154

23 Designing for Scale and Longevity 1155

23.1 Code That Lives 20+ Years . . . 1155
23.1.1 Longevity as an Explicit Design Goal . . . 1155
23.1.2 The Three Stability Layers . . . 1156
23.1.3 Designing Boundaries That Survive Decades . . . 1157
23.1.4 API Evolution Without Breaking the World . . . 1158
23.1.5 Data Longevity: Formats, Schema, and Migrations . . . 1161
23.1.6 Performance Longevity: Budgets, Not Optimizations . . . 1163
23.1.7 Correctness Longevity: Make Failure Predictable . . . 1164
23.1.8 Concurrency Longevity: Rules, Not Hope . . . 1166
23.1.9 Operational Longevity: Observability and Diagnose-Ability . . . 1168
23.1.10 Build and Tooling Longevity . . . 1169
23.1.11 Security Longevity: Assume the Threat Model Evolves . . . 1170
23.1.12 Human Longevity: Teams Change, Code Must Stay Legible . . . 1170
23.1.13 A Practical Longevity Checklist for Large C++ Systems . . . 1171
23.1.14 Closing Principle: Build for Change, Not for Today . . . 1173
23.2 Managing Technical Debt . . . 1173
23.2.1 What Technical Debt Really Means in Large C++ Systems . . . 1174
23.2.2 A Practical Taxonomy of Technical Debt . . . 1175
23.2.3 Debt That Is *Acceptable* vs. Debt That Is *Fatal* . . . 1176
23.2.4 Make Debt Visible: Measurement and Signals . . . 1177
23.2.5 Governance: Turning Debt into a Managed Portfolio . . . 1178
23.2.6 Prioritization: Choosing the Right Debt to Pay . . . 1179
23.2.7 Refactoring Strategies That Work at Scale in C++ . . . 1180
23.2.8 Testing Debt: The Highest ROI Category . . . 1185
23.2.9 Operational Debt: Paying for Faster Diagnosis . . . 1186

23.2.10 When Debt Repayment Becomes a Project 1187
23.2.11 Anti-Patterns in Technical Debt Management 1188
23.2.12 A Large-Scale Technical Debt Playbook (C++ Focused) 1189
23.2.13 Closing Principle: Debt Is Managed by Feedback Loops 1190
23.3 Design for Team Growth 1190
23.3.1 The Core Problem: Coordination Cost Dominates 1191
23.3.2 Organize Architecture Around Change and Ownership 1191
23.3.3 Design APIs for Many Callers and Many Authors 1193
23.3.4 Reduce Developer Friction: Build, Test, and Tooling 1196
23.3.5 Make Architecture Discoverable 1197
23.3.6 Prevent Architectural Drift 1198
23.3.7 Design for Safe Parallel Development 1199
23.3.8 Concurrency Rules That Scale Across Teams 1200
23.3.9 Operational Design for Many Teams 1202
23.3.10 Leadership Practices Encoded into the System 1203
23.3.11 A Checklist for Team-Scalable Design 1204
23.3.12 Closing Principle: Team Growth Is a First-Class Scalability Requirement 1205
23.4 Designing for Maintenance 1205
23.4.1 Maintenance as an Engineering Constraint 1206
23.4.2 The Maintainability Model: Reduce the Cost of Change 1206
23.4.3 Boundary Discipline: Maintenance Happens at Interfaces 1207
23.4.4 Dependency Management: The Most Common Maintenance Failure . . 1208
23.4.5 Consistency: The Maintenance Multiplier 1210
23.4.6 Testing as the Maintenance Safety Net 1212
23.4.7 Observability: Maintenance Requires Diagnose-Ability 1214
23.4.8 Refactoring Strategy: Maintainability Without Rewrites 1215
23.4.9 Maintenance-Friendly Modern C++ Practices 1216

23.4.10 Culture and Process Embedded in Design 1218
23.4.11 A Practical Checklist for Designing for Maintenance 1218
23.4.12 Closing Principle: Maintenance Is Designed, Not Hoped For 1219

24 Testing as Design Feedback 1220
24.1 Testability as a Design Property 1220
24.1.1 Why Testability Is a Design Property (Not an Afterthought) 1220
24.1.2 A Practical Definition of Testability 1221
24.1.3 The Main Enemies of Testability in C++ Systems 1222
24.1.4 Design Moves That Create Testability 1223
24.1.5 Control Time: Clocks as Dependencies 1226
24.1.6 Control Randomness: Deterministic Generators 1228
24.1.7 Design for Observability: Return Values, Events, and State Snapshots . 1228
24.1.8 Testability and Error Handling Design 1230
24.1.9 Testability Under Concurrency 1231
24.1.10 Testability and Architecture: Seams, Not Scaffolding 1233
24.1.11 Design Patterns That Increase Testability 1233
24.1.12 A Testability Checklist for Modern C++ Designs 1235
24.1.13 Closing Principle: Testability Predicts Longevity 1236
24.2 Designing Mockable Systems 1236
24.2.1 Mockability: What It Is and What It Is Not 1237
24.2.2 The Test Double Spectrum (Practical Taxonomy) 1237
24.2.3 Where to Put Seams: Mock at the Edges, Not the Core 1238
24.2.4 Design Technique 1: Functional Core, Imperative Shell 1239
24.2.5 Design Technique 2: Explicit Dependencies (Constructor Injection) . . 1240
24.2.6 Design Technique 3: Prefer Fakes for Stateful Boundaries 1241
24.2.7 Design Technique 4: Spies for Minimal Interaction Verification 1241
24.2.8 Design Technique 5: Control Time and Randomness 1243

24.2.9 Avoiding the Mocking Trap: Fragile Interaction Tests 1244
24.2.10 Mockable Interfaces: C++ Design Guidelines 1245
24.2.11 Mockability and Resource Management 1246
24.2.12 Contract Tests: A Scalable Alternative to Extensive Mocking 1247
24.2.13 A Mockability Checklist for Large C++ Systems 1247
24.2.14 Closing Principle: Mockability Is a Consequence of Good Boundaries . 1249
24.3 Property-Based Testing 1249
24.3.1 Why Property-Based Testing Matters in Large C++ Systems 1250
24.3.2 Core Concepts 1250
24.3.3 Design Feedback: What PBT Teaches You 1251
24.3.4 Property Shapes That Scale Well 1252
24.3.5 Generators: Getting Input Quality Right 1256
24.3.6 Shrinking and Counterexample Management 1257
24.3.7 Property-Based Testing and Mocking 1258
24.3.8 Property-Based Testing in C++: Integration Patterns 1258
24.3.9 PBT as Refactoring and Design Feedback 1260
24.3.10 Common Failure Modes and How to Avoid Them 1260
24.3.11 A Property-Based Testing Checklist 1261
24.3.12 Closing Principle: Properties Are Executable Invariants 1262
24.4 Using Tests to Validate Architecture 1262
24.4.1 Architecture as Constraints, Not Documentation 1263
24.4.2 Why Architectural Validation Must Be Automated 1264
24.4.3 What to Validate with Architectural Tests 1264
24.4.4 Architectural Testing Layers 1266
24.4.5 A Practical C++ Pattern: Architectural “Ports” Verified by Tests 1267
24.4.6 Contract Tests: Architecture Through Substitutability 1268
24.4.7 Architectural Tests for Dependency Rules (Practical Strategies) 1269

24.4.8 Using Tests to Validate Architectural Quality Attributes 1271
24.4.9 Making Architectural Tests Maintainable 1272
24.4.10 A Scalable Workflow: Architecture Validation as Continuous Feedback 1273
24.4.11 Closing Principle: Architectural Integrity Is a Testable Asset 1274

25 Evolution, Refactoring, and Survival 1275
25.1 Designing for Change 1275
25.1.1 The Economics of Change: Cost of Change as the Primary Metric . . . 1275
25.1.2 Principle 1: Make Change Local (Boundaries and Ownership) 1276
25.1.3 Principle 2: Make Contracts Explicit (So You Can Evolve Safely) . . . 1277
25.1.4 Principle 3: Make Change Safe (Feedback Loops) 1279
25.1.5 Principle 4: Prefer Incremental Evolution (Avoid Big-Bang Rewrites) . 1280
25.1.6 Principle 5: Evolve Data with Explicit Rules 1282
25.1.7 Principle 6: Reduce Coupling at Compile Time (C++ Specific) 1284
25.1.8 Refactoring as Survival: A Disciplined Workflow 1285
25.1.9 Designing for Team-Scale Change 1286
25.1.10 Common Anti-Patterns That Kill Evolvability 1286
25.1.11 A Practical Checklist: Designing for Change 1287
25.1.12 Closing Principle: Evolvability Is the Highest Form of Scalability . . . 1288
25.2 Safe Refactoring Strategies 1288
25.2.1 What Makes Refactoring Safe 1288
25.2.2 The Golden Rule: Never Refactor Without a Safety Net 1289
25.2.3 Strategy 1: Refactor in Small, Reversible Steps 1290
25.2.4 Strategy 2: Create Seams Before Changing Internals 1290
25.2.5 Strategy 3: Branch by Abstraction (Replace Internals Gradually) 1291
25.2.6 Strategy 4: Strangler Façade (Wrap and Replace) 1293
25.2.7 Strategy 5: Differential Testing (Reference vs. Optimized) 1293
25.2.8 Strategy 6: Functional Core, Imperative Shell (Move Side Effects Out) . 1294

25.2.9 Strategy 7: Control Nondeterminism Before Refactoring 1295
25.2.10 Strategy 8: Preserve ABI and Public API Stability 1296
25.2.11 Strategy 9: Refactor Data and Schemas Safely (Expand/Contract) . . . 1297
25.2.12 Strategy 10: Reduce Compile-Time Coupling Before Large Refactors . 1298
25.2.13 Refactoring Playbooks (Repeatable Procedures) 1299
25.2.14 Common Anti-Patterns (Unsafe Refactoring) 1300
25.2.15 Closing Principle: Safe Refactoring Is a System 1301
25.3 API Deprecation Policies 1301
25.3.1 Goals of a Deprecation Policy 1302
25.3.2 Change Taxonomy: What Kind of Change Is This? 1302
25.3.3 The Deprecation Lifecycle (A Practical Model) 1303
25.3.4 Compatibility Windows and Support Policies 1305
25.3.5 Communicating Deprecation: Make It Impossible to Miss 1305
25.3.6 Migration Patterns That Preserve Stability 1306
25.3.7 Deprecation Under ABI Constraints (C++ Library Reality) 1308
25.3.8 Enforcement: Make Deprecation Real 1310
25.3.9 Documentation Requirements (Without External Links Here) 1311
25.3.10 Special Cases and Accelerated Deprecation 1312
25.3.11 A Practical Deprecation Policy Template 1312
25.3.12 Closing Principle: Deprecation Is a Contract With Your Future 1313
25.4 Knowing When to Redesign 1313
25.4.1 Definitions: Refactor vs. Redesign vs. Rewrite 1314
25.4.2 The Core Question: Is Incremental Change Still Economically Viable? . 1314
25.4.3 Hard Signals That Suggest Redesign Is Needed 1315
25.4.4 Soft Signals (Warning Signs) That Often Precede Redesign 1317
25.4.5 Decision Framework: Redesign or Refactor? 1318
25.4.6 Redesign Patterns That Avoid Full Rewrites 1320

25.4.7 C++-Specific Redesign Triggers 1322
25.4.8 Avoiding the Rewrite Trap 1323
25.4.9 A Practical Redesign Readiness Checklist 1324
25.4.10 Closing Principle: Redesign Is a Controlled Evolution, Not a Reset . . 1324

IX Case Studies 1325

26 Designing a Core Library 1326
26.1 API Design Walkthrough 1326
26.1.1 Step 0: Define the Core Library's Mission and Non-Goals 1327
26.1.2 Step 1: Decide Stability Guarantees 1327
26.1.3 Step 2: Establish API Design Principles 1328
26.1.4 Step 3: Design Namespace and Header Structure 1329
26.1.5 Step 4: Design the Error and Result Model (Foundation of Contracts) . 1329
26.1.6 Step 5: Define Non-Owning Views and Safe Parameter Conventions . . 1332
26.1.7 Step 6: Design Extension Points (Events/Logging Without Coupling) . 1333
26.1.8 Step 7: Configuration as a Stable Read-Only Interface 1334
26.1.9 Step 8: Assemble a Case Study API: A Small Core "Client" 1335
26.1.10 Step 9: Provide a Testability Walkthrough (Design Feedback) 1336
26.1.11 Step 10: Deprecation and Evolution Built In 1338
26.1.12 Step 11: Performance and Allocation Considerations (Core Library Reality) 1338
26.1.13 Step 12: Final Core Library API Checklist 1339
26.1.14 Closing Principle: Core APIs Are Long-Term Infrastructure 1340
26.2 Ownership Decisions 1340
26.2.1 1. The Ownership Spectrum in C++ 1341
26.2.2 2. Ownership Rules for Public APIs (Core Library Policy) 1342
26.2.3 3. Borrowing: Designing Safe Non-Owning Interfaces 1345

26.2.4 4. Ownership in Return Values 1346
26.2.5 5. Lifetime, Concurrency, and Ownership 1348
26.2.6 6. Ownership and ABI: Designing Stable Public Types 1350
26.2.7 7. Ownership and Error Handling 1351
26.2.8 8. Ownership Policy for Extensions and Plug-ins 1352
26.2.9 9. A Repeatable Ownership Decision Procedure 1353
26.2.10 10. Ownership Checklist for a Core Library 1354
26.2.11 Closing Principle: Ownership Is Architecture 1354
26.3 Error and Contract Strategy 1355
26.3.1 1) Failure Taxonomy: Bug vs. Error vs. Fault 1355
26.3.2 2) Contracts: What the API Promises and Requires 1357
26.3.3 3) Strategy: Contracts Fail Fast; Errors Propagate 1358
26.3.4 4) A Unified Error Model for a Core Library 1358
26.3.5 5) Contract Enforcement: A Core `contract.hpp` 1362
26.3.6 6) Policy: Exceptions vs Result Types in a Core Library 1364
26.3.7 7) Composing Errors and Preserving Context 1365
26.3.8 8) Contracts for Object Lifetimes and State Machines 1367
26.3.9 9) Error Strategy and Performance Predictability 1368
26.3.10 10) A Core-Library Contract and Error Checklist 1368
26.3.11 Closing Principle: Contracts Define Correctness; Errors Enable Survival 1369

27 Designing a High-Performance Service 1370
27.1 Concurrency Model Selection 1370
27.1.1 1) First Principles: What You Must Decide Explicitly 1371
27.1.2 2) Model Landscape: Common Concurrency Models in Services 1373
27.1.3 3) A Selection Framework (Decision Procedure) 1377
27.1.4 4) A Practical Modern C++ Baseline: Hybrid Reactor + Worker Pool . 1379
27.1.5 5) Sharding as a Concurrency Model for Stateful Services 1383

27.1.6 6) Backpressure and Load Shedding: Non-Negotiable for Performance . 1384
27.1.7 7) Concurrency and Testability 1385
27.1.8 8) Common Mistakes in Concurrency Model Selection 1386
27.1.9 9) Selection Summary: Practical Recommendations 1386
27.1.10 Closing Principle: Concurrency Model Is Architecture 1387
27.2 Resource Management 1387
27.2.1 1) First-Class Resource Budgets 1388
27.2.2 2) RAII as the Foundation: Every Resource Has an Owner 1389
27.2.3 3) Bounded Queues and Backpressure 1392
27.2.4 4) Memory Management for High-Performance Services 1394
27.2.5 5) Connection and Descriptor Management 1396
27.2.6 6) Cancellation, Deadlines, and Time Budgets 1397
27.2.7 7) Thread and CPU Resource Management 1398
27.2.8 8) Resource Observability (If You Cannot Measure It, You Cannot Control It) 1399
27.2.9 9) Shutdown and Draining: Resources Must Be Reclaimed Predictably . 1399
27.2.10 10) Common Resource Management Failure Modes 1400
27.2.11 Resource Management Checklist for a High-Performance C++ Service . 1401
27.2.12 Closing Principle: Performance Is Resource Discipline 1402
27.3 Performance Tradeoffs 1402
27.3.1 1) Throughput vs. Tail Latency 1403
27.3.2 2) CPU vs. Memory 1404
27.3.3 3) Abstraction vs. Predictability 1405
27.3.4 4) Locking vs. Sharding vs. Message Passing 1406
27.3.5 5) Copying vs. Borrowing vs. Ownership Transfer 1407
27.3.6 6) Synchronous vs. Asynchronous Boundaries 1408
27.3.7 7) Determinism vs. Peak Speed 1409

27.3.8 8) Lock-Free vs. Lock-Based Data Structures 1410
27.3.9 9) Inlining and Header-Only vs. Build Times and Binary Size 1410
27.3.10 10) Observability vs. Overhead 1411
27.3.11 11) Security and Validation vs. Speed 1412
27.3.12 12) A Practical Decision Toolkit (How to Choose) 1413
27.3.13 Closing Principle: Performance Is a Set of Explicit Choices 1414

28 Designing a Long-Lived Framework 1415

28.1 Extension Points 1415
28.1.1 1) What Is an Extension Point? 1416
28.1.2 2) Classes of Extension Points 1417
28.1.3 3) Designing an Extension Point Contract 1420
28.1.4 4) Ownership and Lifetime Across Extension Boundaries 1421
28.1.5 5) Extension Point Patterns That Scale Over Years 1422
28.1.6 6) Binary Plugins: Designing a Safe C++ Plugin ABI 1424
28.1.7 7) Versioning and Evolution of Extension Points 1426
28.1.8 8) Security and Trust Boundaries 1427
28.1.9 9) Testability of Extension Points 1428
28.1.10 10) Extension Point Checklist (Framework-Grade) 1428
28.1.11 Closing Principle: Extension Points Are the Framework's Survival Mechanism 1429
28.2 Stability Guarantees 1429
28.2.1 1) The Three Stability Axes: Source, Binary, and Behavior 1430
28.2.2 2) A Framework-Grade Stability Policy (What You Promise) 1431
28.2.3 3) Versioning: Making Stability Machine-Checkable 1432
28.2.4 4) Deprecation as Part of Stability 1433
28.2.5 5) Techniques to Deliver ABI Stability in Modern C++ 1434
28.2.6 6) Behavioral Stability: Contracts You Must State 1437

28.2.7 7) Compatibility Testing: Enforcing Stability Automatically 1439
28.2.8 8) Migration Guarantees: How Users Upgrade Safely 1440
28.2.9 9) The Stability Contract Document (What You Publish) 1441
28.2.10 Closing Principle: Stability Is an Engineering System 1441
28.3 Versioning Strategy 1442
28.3.1 1) The Core Problem Versioning Must Solve 1442
28.3.2 2) Define What "Compatibility" Means in Your Framework 1443
28.3.3 3) Semantic Versioning as a Discipline (Not a Slogan) 1443
28.3.4 4) A Change Classification Matrix 1445
28.3.5 5) Versioning the Public Surface: API, ABI, and SPI 1446
28.3.6 6) Deprecation Windows as Part of Versioning 1447
28.3.7 7) Capability Detection Over Version Checks 1448
28.3.8 8) ABI Versioning for Binary Distribution (When Relevant) 1449
28.3.9 9) Release Lines: LTS, Feature Releases, and Support Windows 1450
28.3.10 10) Versioning Configuration and Defaults 1451
28.3.11 11) Automation: Making Versioning Enforceable 1452
28.3.12 12) A Practical Versioning Blueprint (Recommended) 1453
28.3.13 Closing Principle: Version Numbers Are a Contract You Must Enforce . 1454

X Design Wisdom 1455

29 Common Design Failures in C++ 1456
29.1 Overengineering 1456
29.1.1 1) The True Cost Model of Overengineering 1457
29.1.2 2) Common Forms of Overengineering in Modern C++ 1457
29.1.3 3) Why Overengineering Is Worse in C++ 1461
29.1.4 4) How Overengineering Manifests Over Time (Long-Lived Failure Pattern) 1461

29.1.5 5) Design Heuristics to Avoid Overengineering 1462
29.1.6 6) Concrete Refactoring: De-Overengineering Patterns 1463
29.1.7 7) A Practical Overengineering Checklist 1465
29.1.8 8) The Correct Alternative: Intentional Evolution 1465
29.1.9 Closing Principle: Complexity Must Pay Rent 1466
29.2 Leaky Abstractions 1466
29.2.1 1) What "Leak" Means in Engineering Terms 1467
29.2.2 2) Why Leaky Abstractions Are Common in C++ 1467
29.2.3 3) Common Leaky Abstractions in C++ Systems 1468
29.2.4 4) How Leaky Abstractions Create Long-Term Design Damage 1472
29.2.5 5) How to Detect Leaky Abstractions Early 1473
29.2.6 6) Designing Non-Leaky (Honest) Abstractions 1474
29.2.7 7) Repairing a Leaky Abstraction in a Long-Lived System 1476
29.2.8 Leaky Abstraction Checklist 1477
29.2.9 Closing Principle: Abstractions Must Be Either Tight or Honest 1477
29.3 Hidden Ownership 1478
29.3.1 1) What Counts as Hidden Ownership? 1479
29.3.2 2) Why Hidden Ownership Happens in C++ 1480
29.3.3 3) Common Hidden Ownership Anti-Patterns 1481
29.3.4 4) Hidden Ownership and Concurrency 1484
29.3.5 5) How to Make Ownership Visible (Framework-Grade Rules) 1486
29.3.6 6) Detecting Hidden Ownership in Existing Code 1488
29.3.7 7) Refactoring Strategy: From Hidden to Explicit Ownership 1488
29.3.8 Hidden Ownership Checklist 1489
29.3.9 Closing Principle: Ownership Must Be Visible to Be Maintainable . . . 1490
29.4 Premature Generalization 1490
29.4.1 1) What Premature Generalization Looks Like 1491

29.4.2 2) Why Premature Generalization Is Common in C++ 1491
29.4.3 3) The Real Cost Model: Degrees of Freedom 1492
29.4.4 4) Common Premature Generalization Anti-Patterns in Modern C++ . . 1493
29.4.5 5) How Premature Generalization Damages Long-Lived Systems 1496
29.4.6 6) How to Decide When Generalization Is Justified 1496
29.4.7 7) Prefer "Stable Seams" Over "General Solutions" 1497
29.4.8 8) Refactoring Out Premature Generalization 1498
29.4.9 Premature Generalization Checklist 1499
29.4.10 Closing Principle: Generalize Only Under Real Pressure 1499

30 Principles That Actually Survive Reality 1501
30.1 Simplicity Is Not Minimalism 1501
30.1.1 1) The Difference: Minimal Surface vs. Minimal Surprise 1502
30.1.2 2) Why Minimalism Fails in Real C++ Systems 1502
30.1.3 3) A More Accurate Definition of Simplicity 1505
30.1.4 4) Structural Simplicity: Adding the Right Kind of "More" 1506
30.1.5 5) When Minimalism *Is* Simplicity 1507
30.1.6 6) Practical Heuristics: How to Choose "Simple" Over "Minimal" . . . 1508
30.1.7 7) A C++ Case Study Pattern: "Small API" vs "Simple System" 1509
30.1.8 Closing Principle: Simplicity Is the Removal of Ambiguity 1510
30.2 Explicit Is Better Than Clever 1510
30.2.1 1) What "Explicit" Means in Real C++ Design 1511
30.2.2 2) Why Clever Code Fails Over Time 1511
30.2.3 3) Explicitness in the Language: Use the Features That Prevent Surprise 1512
30.2.4 4) Explicitness in Ownership and Lifetimes 1514
30.2.5 5) Explicitness in Error Handling and Contracts 1515
30.2.6 6) Explicitness in Concurrency 1517
30.2.7 7) Explicitness in Templates and Generic Code 1518

30.2.8 8) Explicitness in Architecture: Make the “Seams” Visible 1519
30.2.9 9) A Practical Checklist: Replace Cleverness With Explicitness 1519
30.2.10 Closing Principle: Explicitness Scales With Time and Teams 1520
30.3 Design for Humans First 1520
30.3.1 1) What “Design for Humans” Means in Practice 1521
30.3.2 2) Why Human-Centered Design Is Critical in Modern C++ 1521
30.3.3 3) Human-First Principles That Scale 1522
30.3.4 4) Human-First Error Handling: Make Failure Actionable 1525
30.3.5 5) Human-First Concurrency: Make Threading Understandable 1526
30.3.6 6) Human-First Performance: Make the Cost Model Visible 1527
30.3.7 7) Human-First Evolution: Make Change Safe 1528
30.3.8 8) Human-First Code Review and Team Growth 1529
30.3.9 9) A Practical Checklist 1529
30.3.10 Closing Principle: The System That Survives Is the One People Can Maintain 1530
30.4 When to Say “No” 1530
30.4.1 1) What “No” Means in Engineering Terms 1531
30.4.2 2) The Dominant Reasons to Say “No” 1532
30.4.3 3) “No” as a Design Tool: Where Features Should Live 1535
30.4.4 4) A Decision Framework: The “No” Checklist 1535
30.4.5 5) Patterns for Saying “No” Without Blocking Progress 1537
30.4.6 6) The Most Important Places to Say “No” in C++ 1538
30.4.7 7) Saying “No” Is Saying “Yes” to Longevity 1539
30.4.8 Closing Principle: Guard the Core, Empower the Edges 1539

Appendices **1541**

Appendix A: Design Patterns Reinterpreted in Modern C++ 1541
Appendix B: Idioms Every C++ Designer Must Master 1549

Appendix C: Checklist for Design Reviews 1563
Appendix D: Migration from Legacy C++ to Modern Design 1574
Appendix E: Glossary of Design Terms 1587

Epilogue **1602**

Software Design as an Ethical Responsibility **1602**

Why C++ Still Matters for Serious Systems **1613**

References **1622**

Fornt Matter

Foreword

Software does not fail because developers “forget syntax.” It fails because its *design* silently accumulates contradictions: unclear responsibilities, unstable boundaries, accidental coupling, and hidden assumptions about lifetimes, errors, and concurrency. These issues rarely announce themselves early. They emerge months or years later, when the system is under real pressure: a new feature must land quickly, a team doubles in size, a platform changes, performance constraints tighten, or a security incident forces a redesign under deadline.
This book is written for the long run.
Modern C++ is often described as a “difficult” language. That label is partially deserved, but it hides a more important truth: C++ is a language that makes design *consequences* visible. When you build systems in C++, the realities of ownership, resource lifetimes, binary boundaries, performance, and concurrency are never far away. You cannot easily ignore them. In that sense, Modern C++ is not merely a programming language; it is an engineering medium where architectural decisions become concrete, measurable, and enforceable.

Why another book on software design?

There are many excellent texts on design and architecture. Yet a gap remains for practitioners building long-lived, performance-critical systems: embedded platforms, trading systems, real-time services, rendering engines, storage engines, simulation stacks, and infrastructure

components that must remain correct and maintainable for a decade or more.
In such environments, design is not a set of slogans. Design is the discipline of:

- defining boundaries that remain stable under change,
- making ownership and lifetimes explicit,
- controlling dependencies so modules can evolve independently,
- treating error handling as part of the API contract,
- building abstractions that do not leak performance or correctness,
- and ensuring concurrency is a deliberate model, not an afterthought.

Modern C++ provides unique tools to express these goals precisely:

- deterministic resource management (RAII) as a default,
- value semantics and move semantics to control costs and aliasing,
- strong typing for making invalid states unrepresentable,
- generic programming with concepts for expressing constraints explicitly,
- compile-time computation and validation (constexpr/consteval),
- and well-defined mechanisms for controlling object lifetimes and ownership.

This book does not treat these features as “language tricks.” It treats them as *design instruments*.

What you should expect from this book

This book is not a catalog of patterns. Patterns are useful, but they are not the goal. The goal is to build a robust internal model for design decisions and their tradeoffs in systems that must live a long time.
You should expect:

- **Design principles grounded in engineering reality.** The discussion focuses on practical constraints: maintainability, scalability of teams, correctness, performance, and long-term evolution.
- **C++-specific guidance where it matters.** Ownership, lifetime, ABI boundaries, resource management, and concurrency are treated as first-class design concerns.
- **An emphasis on clarity and enforceability.** Modern C++ can express design intent in types, constraints, and interfaces, reducing the gap between "what we meant" and "what the code permits."
- **A bias toward stable boundaries and controlled dependencies.** Good design is largely the art of saying "no" to convenient shortcuts that turn into expensive liabilities later.

Modern C++ as a design language

A well-designed C++ system is not defined by how advanced its templates are or how clever its metaprogramming becomes. It is defined by how *readable and stable* its boundaries are, and by how safely it can evolve.
Consider a simple example of ownership clarity. An API that returns a raw pointer may force the caller to guess:

- Who owns the returned object?

- How long will it live?
- Is it nullable?
- Is it stable across threads?

Modern C++ encourages designs where these questions are answered in the type system and contract structure.

```
// Ownership is explicit: the caller receives unique ownership.
std::unique_ptr<Service> make_service(Config cfg);

// Non-owning observation is explicit: lifetime is not extended.
Service& service();          // must outlive the caller's use
Service* try_service() noexcept; // nullable, non-owning
```

This is not mere style. It is design: we are choosing which relationships are permitted and which are forbidden.
Similarly, value semantics are a design choice that reduces aliasing and hidden coupling:

```
// A value-type request object: easy to reason about, easy to test, easy to compose.
struct Request {
  std::string route;
  std::string method;
  std::vector<std::pair<std::string, std::string>> headers;
};
```

Value types tend to scale well in complex systems because they minimize shared mutable state. When identity is required, it must be introduced deliberately, with explicit ownership and concurrency strategy.

Design is the management of change

Long-lived systems are defined by the changes they survive:

- new requirements that contradict old assumptions,
- integrations with external systems,
- performance constraints that shift over time,
- evolving security expectations,
- platform and toolchain changes,
- and team turnover.

A design that survives change is usually not the one with the most features or the most flexibility; it is the one with the best *boundaries*. Boundaries are the places where the system says: “This is stable. Everything else may evolve behind it.”
In Modern C++, boundaries appear as:

- interfaces and contracts,
- module partitions and physical dependency rules,
- ownership and lifetime policies,
- ABI constraints where binary compatibility matters,
- and concurrency models with explicit thread-safety guarantees.

A mature design is one where the system can change *locally* without requiring global rewrites.

Correctness is a design property

Correctness does not emerge from unit tests alone. Tests detect failures; design prevents entire classes of failures from existing.
A design that encodes invariants and constraints reduces the space of possible bugs. Modern C++ helps you express invariants through strong types and explicit contracts.

```
// Strong types: reduce accidental mixing of units or identifiers.
struct UserId { std::uint64_t value; };
struct OrderId { std::uint64_t value; };

// Explicit: a function cannot accidentally accept the wrong id type.
Order load_order(UserId user, OrderId order);
```

Even when the type system cannot express a full invariant, good design makes the invariant visible and enforceable through narrow interfaces, clear responsibilities, and controlled mutation.

Performance is a design constraint, not an optimization phase

In long-lived systems, performance concerns are rarely solved by micro-optimizations alone. They are primarily solved by:

- choosing the right data ownership model,
- controlling allocations and memory layout,
- minimizing synchronization and shared mutable state,
- making hot paths simple and predictable,
- and ensuring abstractions do not impose invisible costs.

Modern C++ supports this with zero-cost abstractions, but only if the design is disciplined. Abstractions must be measured and justified. When they are not, they become a tax paid forever.

Concurrency is the design of safety under parallel execution

Concurrency failures are often design failures: unclear ownership across threads, hidden sharing, inconsistent synchronization, and APIs that silently permit misuse.
A serious system must define a concurrency model explicitly:

- Which objects are thread-confined?
- Which are immutable and shareable?
- Which are synchronized, and by what mechanism?
- Which operations are safe to call concurrently?

If your design does not answer these questions, the implementation will answer them for you—in unpredictable ways.

Who this book is for

This book is written for:

- C++ developers who want to design systems that remain clean under growth,
- engineers migrating from “working code” to sustainable architecture,
- architects responsible for performance, correctness, and long-term evolution,
- and teams building libraries, frameworks, services, or embedded systems in C++.

It is also useful for engineers coming from managed languages who want to understand why C++ systems place such emphasis on ownership, boundaries, and deterministic resource management.

How to read this book

You may read this book in two complementary ways:

- **Sequentially**, to build a coherent mental model from principles to architecture and large-scale design.
- **Selectively**, focusing on topics aligned with your current constraints (ownership, error handling, concurrency, API stability, performance-driven design).

In either approach, the most important practice is to continuously map each concept to your own codebase:

- Where are responsibilities unclear?
- Where do lifetimes depend on informal knowledge?
- Which modules cannot be changed without breaking unrelated parts?
- Which APIs hide failure modes?
- Where does concurrency rely on “hope” rather than explicit guarantees?

Design improves when you ask such questions relentlessly and answer them in the structure of the system, not only in comments.

A final word

Software design is not an aesthetic pursuit. It is the engineering of systems that other people can trust: trust to evolve them, trust to operate them, trust to secure them, trust to extend them, and trust to understand them under pressure.

Modern C++ is a demanding tool, but it rewards disciplined design with unusual power: the ability to build abstractions that are expressive, efficient, and precise; the ability to enforce constraints at compile time; and the ability to remain close to the realities of hardware and operating systems without sacrificing architectural clarity.
If you take one idea from this foreword, let it be this:

> Design is the set of decisions that makes future change cheaper, safer, and more predictable.

The chapters ahead are an invitation to treat those decisions as a first-class engineering responsibility.

`Ayman Alheraki`

Why Software Design Fails — and How C++ Forces Clarity

Software design fails far more often than software implementation. Most systems can be made to *work* in the short term. Far fewer can be made to *survive* under change: new requirements, new teams, new platforms, new performance constraints, and new security expectations. This preface explains why design failure is so common, why it is costly, and why Modern C++—despite its reputation—can be one of the most honest and effective environments for learning and practicing rigorous design.

Design failure is rarely a single mistake

Design failure is usually a slow accumulation of small omissions:

- responsibilities are not explicit,
- invariants are not enforced,
- dependencies grow without discipline,
- error handling is improvised at call sites,
- ownership and lifetimes are left to convention,

- concurrency is bolted on late,
- boundaries drift until the system becomes one large “ball of mud.”

None of these decisions looks catastrophic at the moment it is made. Each one is often justified by schedule pressure: *“We will clean it later.”* In long-lived systems, “later” usually becomes *never*, and the system steadily converts velocity into debt.
Design is not a document. Design is the shape of the codebase:

- what can be changed locally,
- what requires global coordination,
- what the type system forbids,
- what the interfaces allow,
- and what assumptions are forced into the open.

When design fails, it fails as an ecosystem. Developers become afraid to touch certain areas. Changes take longer, tests become harder to write, and defects become harder to reproduce. Eventually, the system “works” while the organization pays a continuous tax for every modification.

The recurring causes of design failure

Across industries, codebases tend to fail for a small set of repeating reasons. This book treats each reason as a solvable engineering problem.

1) Unclear boundaries

A system with unclear boundaries cannot evolve safely. When components do not have stable responsibilities, changes leak across the codebase. The symptom is familiar: a small feature request triggers edits in unrelated modules, followed by fragile patches and “temporary” workarounds.
Good boundaries are not primarily about folders. They are about *dependency direction* and *information hiding*:

- policies should not depend on mechanisms,
- high-level logic should not depend on low-level details,
- modules should expose narrow, stable interfaces,
- internal choices should remain replaceable.

Modern C++ offers strong mechanisms for enforcing boundaries physically and semantically:

- narrow headers and forward declarations,
- opaque types and PImpl where ABI stability matters,
- module interfaces where toolchains allow it,
- explicit ownership types that prevent accidental sharing.

2) Hidden ownership and ambiguous lifetimes

In many languages, garbage collection can hide lifetime problems until they become performance or latency failures. In C++, lifetime is always present. A pointer is not just a value; it implies a relationship that must be designed:

- Who owns the object?

- How long does it live?
- Can it be null?
- Is it shared across threads?
- Is the pointed-to object stable across reallocation or movement?

Design fails when these questions are not answered. Ambiguous ownership leads to leaks, double deletes, use-after-free bugs, and dangerous aliasing. Even when these failures do not occur immediately, they produce systems that cannot be reasoned about locally.
Modern C++ forces you to choose a model. The standard ownership vocabulary provides a design language:

```
// Unique ownership: one clear owner; transfer is explicit.
std::unique_ptr<Connection> open_connection(Config cfg);

// Shared ownership: use only when the lifetime truly must be shared.
std::shared_ptr<Cache> shared_cache();

// Non-owning reference: caller must obey lifetime constraints.
Cache& cache();
Cache* try_cache() noexcept; // nullable, non-owning
```

These types are not merely memory tools. They encode architectural intent:

- unique ownership tends to produce clear responsibility and predictable lifetimes,
- shared ownership tends to spread responsibility and must be justified,
- non-owning references require stable lifetime boundaries and discipline.

A design that does not make ownership explicit eventually becomes a system where no one knows what is safe to do.

3) “Everything is mutable” design

Uncontrolled mutability makes systems hard to test, hard to parallelize, and easy to break. When state can be modified from anywhere, every change becomes a global risk.
Modern C++ encourages a value-centric approach that reduces incidental sharing and makes data flow more explicit. Value types are easier to reason about, easier to compose, and safer to pass across boundaries.

```
// A value-type command: simple, testable, serializable, easy to log.
struct CreateUser {
  std::string name;
  std::string email;
};
```

Identity and shared mutation are sometimes unavoidable—but they must be introduced as deliberate design decisions:

- where mutation is allowed,
- who coordinates it,
- how invariants are preserved,
- and what concurrency strategy applies.

Design fails when identity and mutability spread as defaults.

4) Error handling treated as an implementation detail

If a system does not have a coherent error model, the real architecture is “whatever each call site does today.” The result is inconsistent behavior, lost context, partial failures, and APIs that are easy to misuse.
Modern C++ forces explicit choices:

- exceptions as a model for exceptional, non-local failure propagation,
- error codes for predictable failures and ABI-stable boundaries,
- optional/expected-style returns for locally handled outcomes.

Regardless of mechanism, the design goal is the same:

- errors should be *visible* in interfaces,
- failure should be *composable*,
- invariants should remain protected,
- and recovery paths should be explicit.

```
// A failure-aware API: callers cannot ignore the possibility of failure.
struct ParseError { std::string message; std::size_t position; };

std::variant<Ast, ParseError> parse(std::string_view input);
```

This style makes error semantics part of the type-level contract. Even when different mechanisms are chosen, the design must be explicit and consistent.

5) Premature generalization and abstraction inflation

A common failure mode is building abstractions before the constraints are understood. The code becomes a framework for hypothetical futures, not a system for today's requirements.
Modern C++ can amplify this risk because it offers powerful abstraction tools: templates, type erasure, metaprogramming, generic policies, compile-time computation. These tools are valuable, but only when guided by real constraints and measured costs.
Good design follows a discipline:

- begin with simple, concrete interfaces,

- extract abstractions only when duplication reveals stable structure,
- prefer narrow abstractions over universal frameworks,
- measure performance and complexity costs.

This book will repeatedly distinguish:

- abstraction that removes complexity *from callers*,
- versus abstraction that merely relocates complexity *into templates and indirection*.

6) Concurrency added late

Concurrency is not a library choice; it is a design choice. If concurrency is added after the architecture is set, the result is often:

- accidental sharing of mutable state,
- coarse locks and unpredictable latency,
- deadlocks, races, and subtle correctness failures,
- APIs that cannot express thread-safety guarantees.

Modern C++ does not let you pretend concurrency is free. The memory model, data race rules, and performance realities push the design to be explicit:

- which objects are thread-confined,
- which are immutable and shareable,
- which are synchronized and how,
- which operations are safe concurrently.

When done well, Modern C++ can produce concurrency designs that are clear and enforceable. When done poorly, C++ punishes ambiguity quickly.

How C++ forces clarity

C++ “forces clarity” not by being strict for its own sake, but because it exposes fundamental engineering truths:

Clarity about lifetime

Objects do not live forever. Resources must be acquired and released. RAII makes lifetime a structural property rather than a convention.

```
class File {
public:
  explicit File(const char* path);
  ~File();                    // release is deterministic
  File(File&&) noexcept;      // transfer of ownership is explicit
  File& operator=(File&&) noexcept;
  File(const File&) = delete; // no accidental sharing
  File& operator=(const File&) = delete;
};
```

This class design communicates policy:

- the resource is owned,
- ownership is unique,
- transfer is explicit,
- copies are forbidden.

Clarity about cost

In Modern C++, abstractions can be zero-cost, but not automatically. The language provides the *ability* to make costs explicit: copy vs move, allocation vs stack storage, virtual dispatch vs static dispatch. When the design hides costs, systems become unpredictable under load.

Clarity about boundaries

C++ makes dependency structure visible: headers include headers; build systems reflect coupling; ABI choices constrain interfaces. You cannot indefinitely hide a tangled architecture behind a dynamic runtime. The codebase becomes its own design proof.

Clarity about correctness rules

Undefined behavior is not a "bug type"; it is the absence of a defined contract between code and machine. It exists where design and implementation permit invalid states, data races, and broken invariants. A serious C++ design treats correctness as a property enforced by:

- types and constraints,
- narrow interfaces,
- invariants protected by constructors and encapsulation,
- disciplined ownership,
- deliberate concurrency models.

The purpose of this book

This book has one primary goal:

> To teach software design as an engineering discipline, using Modern C++ as a precise language for expressing and enforcing architectural intent.

It will not ask you to memorize rules. It will ask you to adopt a way of thinking:

- design is the management of complexity,
- design is the management of change,
- design is the enforcement of constraints,
- design is the continuous alignment between intent, code, and system behavior.

Modern C++ is particularly well-suited to this mission because it can encode intent at multiple levels:

- in types and value categories,
- in ownership and lifetime,
- in interfaces and error models,
- in compile-time constraints and validation,
- in physical boundaries and binary interfaces.

How this book is structured

The book progresses from fundamentals to long-lived architecture:

- It begins by defining design in terms of responsibilities, boundaries, and constraints.
- It then builds a Modern C++ design vocabulary: value vs identity, ownership, RAII, error models, and stable interfaces.

- It expands into compile-time design, concurrency models, and performance-driven architecture.
- It concludes with large-scale patterns, evolution strategies, and case studies that connect principles to real systems.

Each part is designed to answer a practical question:

- **What is the design decision here?**
- **What are the tradeoffs?**
- **How does Modern C++ help express and enforce the choice?**
- **How will this choice behave under change?**

A note on “most recent” practices

Modern C++ continues to evolve, but the core design truths remain stable:

- clarity beats cleverness,
- explicit ownership beats implicit conventions,
- stable boundaries beat global coupling,
- coherent error models beat ad-hoc handling,
- and concurrency must be designed, not patched.

Where the language offers newer tools—concepts, improved constexpr, modules, and emerging library utilities—this book adopts them when they reduce complexity or increase clarity. New features are treated as design instruments, not as trends.

The standard you should hold this book to

If you finish this book and your codebase still depends on undocumented lifetimes, hidden ownership, unstable boundaries, and ambiguous error semantics, then the book has failed. The measure of success is practical:

- can your team change modules without fear?
- can new developers learn the architecture quickly?
- can you reason locally about correctness and lifetimes?
- can you scale concurrency without turning the system into a locking puzzle?
- can you measure performance and know where the costs come from?

Those are design outcomes. Modern C++ is one of the best environments for achieving them—when used deliberately.
The chapters that follow provide a complete framework for doing so.

How to Read This Book

This book is designed with two complementary paths in mind: a sequential, immersive journey that builds a deep conceptual foundation, and a modular, topic-focused path for readers who seek targeted guidance on specific design challenges. Regardless of the path you choose, your engagement with the material should be active: design is not absorbed by passive reading, but by reflection, experimentation, and application.

Intended Audience

This book is written for software professionals who are serious about building robust, maintainable, and long-lived systems using Modern C++. It assumes familiarity with C++ fundamentals, including object-oriented programming, templates, and basic standard library usage. It is not a beginner's introduction to the syntax of C++, but rather a disciplined exploration of design principles as they manifest in real-world systems.
The book is appropriate for:

- Experienced C++ developers seeking to deepen their design skills.
- Architects and technical leads responsible for system-level decisions.
- Engineers transitioning from implementation to design-centric roles.
- Teams migrating legacy C++ codebases toward scalable architectures.

- Software professionals from other languages interested in Modern C++ design.

How the Book Is Structured

The book is divided into logical parts that reflect the natural progression of design thinking and practice:

- **Part I: Foundations of Software Design** introduces core principles of design independent of any specific language, and then grounds them in Modern C++.
- **Part II: Core Design Principles Revisited** revisits classical design concepts with Modern C++ lenses: responsibility, coupling, abstraction, and composition.
- **Part III: Object Model, Ownership, and Lifetime** focuses on C++ ownership semantics, value types, identity types, and resource management patterns.
- **Part IV: Interfaces, APIs, and Contracts** explores stable interface design, error handling, and API contracts that scale over time.
- **Part V: Compile-Time Design and Metaprogramming** covers compile-time features of Modern C++ that help express constraints and reduce runtime error surfaces.
- **Part VI: Concurrency and Parallel Design** examines explicit concurrency models and patterns that avoid common pitfalls in multi-threaded systems.
- **Part VII: Architectural Styles in C++** introduces higher-level architectural approaches applicable to large systems.
- **Part VIII: Large-Scale System Design** connects principles with practice in systems that must evolve over years or decades.

- **Part IX: Case Studies** provides real-world examples demonstrating how design principles are applied and validated.
- **Part X: Design Wisdom** synthesizes lessons learned and offers guidelines that endure beyond any specific project or language feature set.

Each chapter begins with clear motivations, problems it addresses, and the expected outcomes. Examples use Modern C++ idioms, emphasizing readability, correctness, and clarity of intent.

Reading Paths

Sequential Reading

A sequential reading is recommended for readers who are new to the discipline of software design or who want a comprehensive understanding of how design principles build upon one another. In this path, you will:

- Start with foundational concepts that shape how designers think about systems.
- Progress through increasing levels of abstraction and complexity.
- Reinforce earlier principles with concrete examples in later contexts.
- Build a mental model that integrates principles, patterns, and architectural styles.

Sequential reading fosters a cohesive understanding of the landscape of design, enabling insights that cut across individual chapters.

Topic-Focused Reading

Experienced practitioners or readers seeking guidance on specific problems can approach the book non-linearly. Each chapter is designed to stand on its own, presenting a complete exploration of its topic:

- If your immediate concern is API design, you may focus on Part IV.
- If you are tackling concurrency issues, start with Part VI.
- If compile-time constraints are central to your project, Part V offers concentrated insights.

This path allows you to dive directly into the challenges most relevant to your current work without waiting to cover foundational material first. Cross-references within chapters ensure that dependencies on other concepts are explained or recapped when necessary.

How to Approach Examples

Examples in the book serve two purposes: to illustrate design principles in context, and to demonstrate Modern C++ idioms that embody those principles. Examples are crafted to be realistic but concise enough for focused study.

- Code is presented with `minted` highlighting to emphasize syntax and relevant constructs.
- Each example is followed by an explanation of the design intent, alternatives, and tradeoffs.
- Readers are encouraged to compile and modify the examples, using them as laboratories for experimentation.

```
// Example: a simple, explicit resource manager using RAII.
class ResourceManager {
public:
```

```
  ResourceManager(ResourceHandle handle)
    : handle_{handle} {}
  ~ResourceManager() {
    if (handle_) {
      release(handle_);
    }
  }
  ResourceManager(ResourceManager&& other) noexcept
    : handle_{other.handle_} {
    other.handle_ = nullptr;
  }
  ResourceManager& operator=(ResourceManager&& other) noexcept {
    if (this != &other) {
      reset();
      handle_ = other.handle_;
      other.handle_ = nullptr;
    }
    return *this;
  }
  ResourceManager(const ResourceManager&) = delete;
  ResourceManager& operator=(const ResourceManager&) = delete;

private:
  void reset() {
    if (handle_) {
      release(handle_);
      handle_ = nullptr;
    }
  }

  ResourceHandle handle_;
};
```

In this example, the design goals are explicit:

- ownership is unique,
- lifetime management is deterministic,
- copying is prohibited to avoid ambiguous ownership,
- movement is explicit and safe.

Active Engagement

Reading this book is an active process. To internalize the material:

- Try to map principles to your existing codebases.
- Refactor small components using the design suggestions.
- Write new examples to test your understanding.
- Discuss design decisions with peers to validate and refine your thinking.
- Use the exercises at the end of chapters to solidify your mastery.

Exercises and Design Challenges

Each chapter includes exercises designed to reinforce the core concepts and to challenge you to apply them. These exercises are not trivial puzzles; they are scenarios derived from realistic design problems:

- Identify unclear boundaries in a given code snippet.
- Redesign an API to make error handling explicit.
- Refactor a class hierarchy to clarify ownership and lifetime.

- Evaluate concurrency models for safety and performance.
- Improve a module's interface for long-term stability.

Exercises include prompts, hints, and discussion points to guide your exploration.

Patterns and Anti-Patterns

Throughout the book, you will encounter design patterns reinterpreted through the lens of Modern C++. Rather than presenting patterns as prescriptive solutions, this book treats them as tools to be understood in context:

- When a pattern clarifies intent and reduces cognitive load, use it.
- When it obscures design intent or introduces unnecessary abstraction, reconsider it.
- Anti-patterns are presented not as curiosities, but as common signals of deeper design issues.

Each pattern discussion includes:

- Motivation and context,
- Modern C++ implementation,
- Limitations and alternatives,
- Impact on maintainability and scalability.

Reflection and Review

At the end of each major part, reflective summaries synthesize the key points and connect them with other areas of the book. These reflections help you:

- consolidate learning,
- recognize the relationships between concepts,
- prepare for practical application in your own systems.

A Living Discipline

Design is not static. As systems evolve and as Modern C++ itself evolves, your design thinking must remain adaptable. The principles in this book are enduring; the idioms and tools will continue to grow.
This book aims to equip you not just with solutions, but with a design mindset that remains valuable as languages, platforms, and environments change.

Conclusion

Whether you read this book from cover to cover or dip into it for targeted insights, your ultimate goal should be the same:
to write systems whose design remains comprehensible, evolvable, and resilient under change.
By engaging actively with the material, practicing the techniques, and reflecting on your own work, you will transform your understanding of design from a passive notion into an applied engineering discipline.

Target Audience

This book is written for engineers who build software that must remain correct, maintainable, and performant over long lifetimes. It is not a tutorial on C++ syntax; it is a design text that uses Modern C++ as a precise engineering language for expressing architectural intent.
The target audience includes professionals who must make design decisions that survive:

- evolving requirements and product scope,
- growing teams and organizational change,
- platform shifts and toolchain upgrades,
- tighter performance, latency, and memory constraints,
- security hardening and correctness guarantees,
- multi-threading and parallel execution.

Primary Audience

Professional C++ Developers

This book is primarily for C++ developers who already write production software and want to strengthen their ability to design systems that do not collapse under growth. Typical readers in

this category include engineers working on:

- performance-critical applications and services,
- systems and infrastructure components,
- embedded and real-time software,
- databases, storage engines, and caching systems,
- networking stacks, middleware, and messaging layers,
- simulation, CAD, robotics, and numerical systems,
- engines and toolchains (rendering, compilers, runtime libraries).

These engineers often face the same recurring design pressures: ownership, lifetime, concurrency, and boundary stability. The book provides a systematic framework to approach these pressures with modern design tools.

Technical Leads and Architects

If you are responsible for the architecture of a system, you are responsible for its future. This includes decisions about:

- component boundaries and dependency direction,
- API design and stability policies,
- error-handling models and failure semantics,
- performance budgets and measurement strategy,
- concurrency model and thread-safety guarantees,

- build structure, modularity, and long-term maintainability.

This book treats these responsibilities explicitly. It is written to support people who must defend architectural decisions with engineering reasoning, not preference or fashion.

Engineers Maintaining or Modernizing Legacy C++

Many organizations operate large C++ systems written over long periods. These codebases often contain:

- unclear ownership and informal lifetime assumptions,
- wide, unstable interfaces,
- accidental coupling across modules,
- ad-hoc error handling,
- tightly interwoven build dependencies,
- concurrency introduced without a coherent model.

This book is written to help you modernize design incrementally. It emphasizes strategies for improving architecture without large rewrites, and for introducing modern C++ principles while preserving operational stability.

Secondary Audience

Engineers from Managed Languages Seeking Systems-Level Design Literacy

Developers coming from managed environments (GC languages or highly dynamic runtime ecosystems) often encounter a new set of realities in C++:

- ownership must be explicit,
- lifetime must be engineered,
- concurrency cannot rely on implicit runtime guarantees,
- cost models (allocation, copying, synchronization) matter directly,
- interfaces must respect ABI and physical dependencies.

This book is useful for readers who already understand software design at a conceptual level but want to master how design becomes enforceable in a language where the machine model is close and explicit.

Advanced Students Preparing for Real-World Systems Work

If you are a senior student or early-career engineer aiming toward systems programming, this book provides a structured preparation path. It focuses on:

- design reasoning under constraints,
- building correct mental models of ownership and invariants,
- composing stable interfaces,
- understanding design tradeoffs in performance-critical environments.

This is not a beginner text, but a reader with strong motivation can treat it as a guide to become employable in demanding engineering environments.

What You Should Already Know

To get full value from this book, you should already be comfortable with:

- core C++ (classes, RAII, references, templates at a basic level),
- standard library fundamentals (containers, algorithms, strings),
- basic compilation concepts (headers, translation units, linking),
- basic debugging and testing workflow.

You do not need to be an expert in metaprogramming or concurrency to begin. Those areas are developed progressively with a design-first approach.

What You Will Learn

This book is designed to move the reader from implementation-level thinking to design-level competence. You will learn to:

- design stable boundaries that resist architectural erosion,
- express ownership and lifetime explicitly using modern C++ idioms,
- reduce coupling and enforce dependency direction,
- design failure-aware APIs with coherent error semantics,
- use the type system to enforce constraints and invariants,
- apply compile-time tools (concepts, constexpr) to validate design,
- select and implement concurrency models that remain safe under scale,

- design for performance as a structural property, not an afterthought,
- refactor and evolve large systems without destabilizing them.

Who This Book Is Not For

This book is not intended for:

- readers seeking an introductory C++ programming tutorial,
- readers who want only short-term “how-to” recipes without design reasoning,
- projects where performance, lifetime control, and architectural stability are irrelevant.

If your primary goal is quick prototyping without long-term maintenance responsibility, many environments and languages will feel more convenient. The purpose of this book is different: it is to help you design software that survives.

A Practical Self-Assessment

If you can answer “yes” to most of the following questions, you are in the right place:

- Do you maintain software that must live for years?
- Do you need designs that allow safe local changes?
- Do you care about predictable performance and memory usage?
- Do you work with concurrency, I/O, or real-time constraints?
- Do you want ownership and lifetimes to be explicit and verifiable?
- Do you want interfaces that remain stable under evolution?

If so, this book will meet you where your responsibilities are: in real engineering constraints, and in the discipline of long-lived software design.

```
// Design mindset checkpoint: when reading any API, ask:
struct Questions {
  // Who owns the returned object?
  // How long does it live?
  // Can failure be ignored?
  // What invariants are guaranteed?
  // Is it safe under concurrency?
};
```

The central theme of this book is that Modern C++ makes these questions unavoidable—and that answering them explicitly is what separates code that merely runs from systems that endure.

C++ Standards Covered (C++17 $\rightarrow$ C++26)

Modern C++ continues to evolve as a design language as well as a programming language. From C++17 through the emerging C++26 standard, the language has introduced features that not only enhance expressiveness and performance but also improve the way designers express invariants, constraints, modularity, and correctness. This chapter provides a clear and accurate overview of the standards that form the foundation for the design techniques in this book. Understanding which features are available and why they matter is essential for using Modern C++ as an explicit design language.

Why Standard Versions Matter

Each C++ standard formalizes a set of language features and standard library facilities. These features influence the way we structure code, enforce constraints, and architect systems. The standards covered in this book are:

- **C++17**
- **C++20**
- **C++23**
- **C++26 (Projected)**

Although not all compilers fully implement C++26 at the time of writing, many of its proposals have already influenced practice through experimental support in major toolchains.

C++17: A Foundation for Modern Design

C++17 solidified many features that are fundamental to clear design:

- **Structured Bindings**
  Structured bindings simplify decomposition of aggregates and tuples, making intent clearer in code that works with multiple values.

```
std::tuple<int, std::string> getUserInfo();
auto [id, name] = getUserInfo();
```

- **std::optional, std::variant, std::any**
  These types make uncertainty explicit. They allow designers to express "absence of value," "one of several types," or "type-erased value" clearly in interfaces.

```
std::optional<std::string> findConfig(std::string_view key);
std::variant<int, std::string> parseValue(std::string_view text);
```

- **constexpr Enhancements**
  C++17 extended `constexpr` to many standard library facilities, enabling more computation at compile time and reducing runtime surface area for errors.

- **Template Argument Deduction for Classes**
  This feature reduces verbosity and encourages the design of templates that are easier to use without typing long parameter lists.

```
std::pair p = std::make_pair(1, "text");
```

- **Improved Standard Library Algorithms**
  C++17 normalized the algorithm library, making common operations like `std::transform_reduce` and parallel algorithms easier to apply in design.

These capabilities allow designers to express concepts with reduced ceremony and clearer intent.

C++20: A Paradigm Shift in Expressiveness

C++20 introduced features that significantly enhance how we express design intent in code:

- **Concepts**
  Concepts provide a way to specify constraints on templates precisely. They improve error diagnostics and turn implicit requirements into explicit contracts.

```
template <std::integral T>
T add(T a, T b) {
    return a + b;
}
```

- **Ranges Library**
  Ranges change the way we think about collections and transformations, encouraging pipelines of operations that express design intent clearly and safely.

```
auto result = values | std::views::filter(isPositive) | std::views::transform(square);
```

- **Coroutines**
  Coroutines enable lazy computations, asynchronous flows, and resumable functions with minimal syntactic overhead, improving design for concurrency and streaming.

- **Modules**
  Modules begin a long-awaited replacement for the traditional header model, reducing unintended coupling and improving build-time scalability.

- **constexpr Enhancements**
  C++20 vastly expands `constexpr` support, making it possible to perform more complex computations at compile time, including dynamic containers in many cases.

These additions allow designers to treat compile-time structure as part of the architecture rather than an afterthought.

C++23: Refinement and Usability Enhancements

C++23 builds on C++20 with refinements that improve usability and expressive power:

- **Library Additions**
  Additions like `std::flat_map`, `std::flat_set`, and `std::expected` (where available) provide more precise tools for designing robust interfaces.

```
std::expected<int, std::string> computeValue(int x);
```

- **Standard Library Updates**
  Improvements to string and container interfaces, synchronization primitives, and time utilities make design with standard components more expressive and less error-prone.

- **Functional Additions**
  Enhancements to `std::span`, `std::simd` (where supported), and uniform initialization strengthen the ability to express design-level constraints on data structures.

C++23 furthers the goal of making the standard library a more complete toolbox for high-level design without sacrificing performance.

C++26: The Next Step

C++26 is the projected next major standard, with proposals that continue to elevate design expressiveness:

- **Reflection and Metaprogramming Improvements**
  Compile-time reflection and introspection features make it possible to reason about program structure at compile time, enabling higher-level design tooling and automatic interface generation.
- **Contracts and Static Reflection**
  Although not fully standardized in previous versions, ongoing proposals aim to add language-level contracts and stronger reflection mechanisms that enforce invariants and document intent statically.
- **Ranges and Views Refinement**
  Continued evolution of the Ranges library and associated utilities improves how designers define and compose operations on sequences of data.
- **Performance and Safety Enhancements**
  Proposals aimed at better memory safety diagnostics, improved lifetime analysis, and enhanced tooling integration are expected to influence design practices positively.

While C++26 features continue to be standardized at the time of writing, many compilers already offer them in experimental or preview modes, allowing early adopters to experiment with next-generation design features.

Standard Library Evolution

The standard library is as important as the core language in design. Between C++17 and C++26, the library has grown in areas that directly impact design:

- **Utilities for Error Handling** (`std::expected`) make error pathways explicit and composable.
- **Ranges and Views** transform the way data flows through systems.
- **Synchronization Primitives** (`std::latch`, `std::barrier`) support structured concurrency.
- **Networking and Executors** (via Library Fundamentals Extensions and TS) provide a foundation for scalable services.
- **Time and Chrono** improvements aid in designing systems with precise performance constraints.

These enhancements reduce the need for custom utilities and reinforce design consistency across codebases.

Compiler and Toolchain Support

To leverage the standards discussed in this book, modern compilers provide comprehensive support:

- GCC (GNU Compiler Collection)
- Clang/LLVM
- Microsoft Visual C++
- Embedded and specialty toolchains (where applicable)

Compiler support varies by version, particularly for the most recent features associated with C++26. Most compilers offer experimental flags to enable in-progress features. Readers are encouraged to consult their toolchain documentation for specific support details.

Design Implications of Standard Evolution

Each standard iteration affects design practice in important ways:

- **Expressiveness** increases the precision with which designers capture intent.
- **Safety** features reduce classes of errors at compile time.
- **Abstraction without cost** becomes more achievable with expanded `constexpr`, Concepts, and Ranges.
- **Modularity** improves with Modules and careful dependency management.
- **Error modeling** becomes more explicit and composable with tools like `std::expected`.

Design decisions in Modern C++ are not merely code choices; they are architectural choices shaped by the language's evolving capabilities.

Practical Guidance for Readers

As you read this book:

- Assume C++17 as the minimum baseline where structural clarity is prioritized.
- Use C++20 features to express contracts, constraints, and modular boundaries.
- Adopt C++23 enhancements where they simplify interfaces and reduce boilerplate.
- Explore C++26 features experimentally to future-proof designs.

When examples use newer features, alternative idioms are provided for environments where full support may not yet be available.

Conclusion

The standards covered in this book define the design vocabulary for Modern C++. They are not mere versions of a language; they are successive refinements that transform how systems are expressed, constrained, and evolved. Mastery of these standards enables you to use C++ not just as a tool for writing code, but as a language for engineering resilient and long-lived systems.

Notation, Terminology, and Conventions

This book uses a precise technical vocabulary. Software design fails frequently because teams use the same words to mean different things: "ownership" becomes confused with "lifetime," "interface" becomes confused with "API," "architecture" becomes confused with "folder structure," and "thread-safe" becomes a vague promise rather than a verifiable property. This chapter defines the notation, terminology, and conventions used throughout the book so that every design claim can be read unambiguously.

Typographic Conventions

- **Bold** highlights key terms at first introduction.
- *Italic* emphasizes a concept or provides a nuanced distinction.
- `Monospace` is used for identifiers, keywords, file names, command names, and short code fragments such as `std::unique_ptr` or `noexcept`.
- `C++ code` is presented using `minted`, with minimal but correct scaffolding.
- `Pseudo-code` is marked clearly and never presented as compilable C++.

Code Conventions

All C++ examples are written with the following goals:

- express design intent clearly,
- prioritize correctness and explicitness over cleverness,
- use standard library facilities when they communicate intent better than custom utilities,
- avoid unnecessary dependencies,
- favor “small, composable pieces” rather than monolithic examples.

Language Version in Examples

Examples generally assume a C++20 baseline unless a feature is explicitly introduced as belonging to C++17, C++23, or later. When an example uses a newer feature, the text states the assumption explicitly and, when useful, provides a conservative alternative.

Naming Conventions

Naming in examples follows common professional practice:

- Types use `PascalCase` (e.g., `OrderService`).
- Functions and variables use `snake_case` or `lowerCamelCase` depending on the idiom being illustrated. A single chapter remains consistent.
- Template parameters use `T`, `U`, or descriptive names such as `Value`, `Error`, `Allocator`.
- Boolean predicates use names that read as questions, such as `is_valid()`.

Examples Are Didactic, Not Frameworks

Most examples are intentionally small. They illustrate design decisions and tradeoffs rather than promote an architectural "one true structure." When realistic complexity is necessary (e.g., plugin boundaries, concurrency models), the complexity is introduced incrementally and justified.

Core Design Vocabulary

The following terms are used with strict meanings throughout the book.

Design

Design is the set of decisions that determine:

- structure (how the system is decomposed),
- responsibility (what each part owns and does),
- interaction (how parts communicate),
- constraints (what is allowed and forbidden),
- evolution (how change is absorbed over time).

Design is not a diagram; it is the shape of the codebase and the rules it enforces.

Architecture

Architecture is the high-level structure of a system: the major components, their boundaries, their dependency direction, and the governing rules that preserve these properties. Architecture is the design view that remains meaningful at scale (teams, years, deployments).

Component, Module, and Layer

These words are often used interchangeably in practice; this book distinguishes them:

- A **component** is a cohesive unit of behavior with a clear public interface. It may be built from multiple compilation units. It should be replaceable behind its interface.
- A **module** is a unit of *dependency management*. In classic C++, it is approximated with headers/libraries and build targets. With C++ modules, it can be an actual language-level boundary.
- A **layer** is a structural pattern where dependencies flow in one direction (e.g., UI depends on application services; application services depend on domain; domain depends on nothing).

Interface, API, and ABI

- An **interface** is a contract: the set of operations and behavioral guarantees exposed to a client.
- An **API** (Application Programming Interface) is the source-level interface: function signatures, types, semantics, error behavior, and usage rules.
- An **ABI** (Application Binary Interface) is the binary-level contract: calling conventions, name mangling, object layout expectations, exception propagation rules across boundaries, and linkage constraints.

When a chapter discusses “stability,” it will specify whether the concern is API stability, ABI stability, or both.

Policy and Mechanism

A crucial architectural distinction:

- **Policy** expresses *what* should happen (rules, decisions, business/domain logic).
- **Mechanism** expresses *how* it happens (I/O, storage, concurrency primitives, serialization, transport).

Good design ensures policy does not depend on mechanism. This is a primary means of achieving replaceability and testability.

Dependency Direction and Coupling

- A **dependency** exists when one unit must know about another to compile, link, or function.
- **Coupling** measures how hard it is to change one unit without affecting others.

This book treats dependency direction as a first-class design property. A large system stays healthy when dependencies flow in a controlled direction, and when unstable details do not infect stable policy code.

Cohesion

Cohesion measures whether a unit has a single, clear purpose. High cohesion tends to reduce coupling and improve maintainability. Low cohesion produces "grab bag" modules that become change hotspots.

Ownership and Lifetime

These terms are central to C++ design and are never used loosely:

- **Ownership** answers: who is responsible for releasing a resource?
- **Lifetime** answers: how long does an object or resource remain valid for use?

Ownership and lifetime often correlate, but they are not identical. This book explicitly marks whether a reference is owning or non-owning, and what lifetime guarantees apply.

```
// Ownership: the unique_ptr owns the object and releases it.
std::unique_ptr<Widget> owning_widget;

// Non-owning observation: does not extend lifetime.
Widget* observer = owning_widget.get();
```

Value Types and Identity Types

- A **value type** is defined by its state. Copying a value produces another value with the same meaning. Equality is typically state-based.
- An **identity type** is defined by its identity over time. Two objects may have equal state yet represent different entities. Aliasing and shared access are common concerns.

Design decisions in C++ often hinge on choosing value semantics versus identity semantics, because that choice affects aliasing, concurrency, caching, and API clarity.

Invariant

An **invariant** is a condition that must always hold for an object or module to be valid. Invariants are not comments; they are enforced properties expressed through constructors, encapsulation, types, and contracts.

Contract

A **contract** is the agreed behavior between a caller and a callee:

- preconditions (what must be true before calling),
- postconditions (what is guaranteed after success),
- invariants (what is always maintained),
- error behavior (how failure is reported and what remains valid).

When this book says "contract," it refers to semantics and guarantees, not merely language-level contract syntax.

Thread-Safety Terminology

Concurrency terms are used precisely:

- **Thread-confined**: an object may be used only from one thread.
- **Immutable**: state does not change after construction and is safely shareable.
- **Externally synchronized**: the caller must provide synchronization.
- **Internally synchronized**: the object manages its own synchronization.
- **Lock-free / wait-free**: used only with their strict technical meanings and only when the requirements are met.

Claims about thread safety are treated as design constraints that must be made explicit in APIs.

Error Handling Terminology

The book distinguishes the following categories:

- **Failure**: an operation cannot produce the requested result.
- **Error**: information describing a failure.
- **Exceptional condition**: a failure that is not part of normal control flow.
- **Recoverable failure**: the caller can proceed with an alternate path.
- **Unrecoverable failure**: continuing would violate invariants or correctness expectations.

Mechanisms such as exceptions, error codes, `std::optional`, and expected-like results are presented as design tools. The key is not the mechanism, but the coherence and composability of the error model.

Complexity and "Cost" Terms

When the book says "cost," it may refer to:

- runtime cost (latency, throughput, CPU cycles),
- memory cost (allocations, footprint, cache behavior),
- concurrency cost (contention, synchronization overhead),
- build cost (compile time, link time, dependency propagation),
- cognitive cost (understandability, maintainability, change risk).

Modern C++ enables zero-cost abstractions in many cases, but the book treats cost as a measurable property that must be respected, not assumed.

Compilation and Build Terminology

Because physical structure influences architecture in C++, the book uses:

- **translation unit**: the compiled result of a source file after preprocessing.
- **ODR (One Definition Rule)**: the rule governing definitions across translation units.
- **linkage**: how symbols are connected across compiled units.
- **build target**: a library or executable unit defined by the build system.

When the book discusses "dependency management," it includes both conceptual dependencies (design-level) and physical dependencies (build-level).

Conventions for Requirements and Guarantees

To keep design claims unambiguous, this book uses consistent language:

- **must** indicates a required property for correctness.
- **should** indicates a strong recommendation with known tradeoffs.
- **may** indicates an allowed choice depending on context.
- **never** indicates a practice that is unsafe or architecturally damaging in the given context.

When a recommendation is context-sensitive, the text makes the context explicit: performance constraints, ABI stability, real-time requirements, safety-critical environments, or organizational scale.

Conventions for Example Contracts

Many examples annotate API intent using comments that reflect contracts:

```
// Precondition: input must be valid UTF-8.
// Postcondition: returns a normalized string on success.
// Error: returns an error object; no partial state escapes.
auto normalize_text(std::string_view input)
  -> std::variant<std::string, NormalizeError>;
```

Such comments are not decoration; they represent design requirements that must be enforced by structure and tests. The book repeatedly shows how to convert informal assumptions into enforceable constraints.

Final Note

A technical book about design must enforce the same discipline it teaches. This chapter exists so that the reader can interpret every later statement with precision. When the book uses terms like *ownership*, *boundary*, *contract*, or *thread-safe*, it is not offering a mood or an opinion. It is describing an engineering property that can be designed, implemented, measured, and validated.

Part I

Foundations of Software Design

What Software Design Really Is

1.1 Design vs Coding vs Architecture

Software engineering teams frequently use the words *design*, *coding*, and *architecture* as if they were interchangeable. They are not. Confusing them leads to misaligned expectations, unstable systems, and "progress" that looks productive (many commits) while the system becomes harder to change. This section establishes a rigorous distinction among the three, then shows how Modern C++ makes those distinctions enforceable in code.

1.1.1 Three activities, three different levels of responsibility

Coding (construction). **Coding** is the act of translating a chosen design into executable software: writing functions, implementing algorithms, wiring components, and validating behavior through debugging, unit tests, integration tests, and verification activities. Coding answers: *"How do we implement the solution that has already been chosen?"*

Design. **Software design** defines the internal structure of a system or component: its decomposition into units, the responsibilities of each unit, and the interfaces and interactions that allow those units to collaborate. Design answers: *"What parts do we build, what does each part do, and how do they fit together?"* Design produces artifacts such as component interfaces, invariants, error models, concurrency rules, and data-flow strategies.

Architecture. **Software architecture** is the set of system structures needed to reason about the system at scale: the major elements, their relationships, and the externally visible properties that govern how the system is built, deployed, operated, and evolved. Architecture answers: *"What are the big pieces, how are they organized, and what rules constrain the entire system over time?"*

1.1.2 A scale model: what changes when you move between levels

A useful mental model is to treat the three levels as different "zoom" positions:

- **Architecture** is the system-level blueprint and governance: boundaries, dependency direction, deployment topology, and the rules that protect evolvability.
- **Design** is the component-level blueprint: responsibilities, interfaces, invariants, error semantics, lifetime/ownership, and concurrency contracts.
- **Coding** is the implementation-level realization: algorithms, data structures, control flow, resource acquisition/release, and tests that validate correctness.

The confusion begins when teams treat architecture as folder structure, design as class diagrams drawn once, and coding as the only work that matters. In reality:

- architecture is not a diagram; it is the set of constraints that prevent the system from degrading,
- design is not decoration; it is the contract surface and responsibility map that makes local change safe,
- coding is not merely typing; it is a disciplined process of translating intent into behavior without violating constraints.

1.1.3 What each level produces

Architecture produces:

- **System structures:** major subsystems and their relations (e.g., UI, application services, domain, infrastructure).
- **Dependency rules:** what may depend on what (and what must never depend on what).
- **Cross-cutting decisions:** security boundaries, data ownership domains, observability strategy, and scalability model.
- **Evolution constraints:** compatibility strategy (API/ABI), versioning, and migration boundaries.

Design produces:

- **Component decomposition:** classes, modules, libraries, and services with clear responsibilities.
- **Interfaces and contracts:** preconditions, postconditions, invariants, error behavior, and concurrency guarantees.
- **Data and ownership model:** value vs identity choices, lifetime boundaries, and resource management strategy.
- **Interaction model:** synchronous vs asynchronous collaboration, callback/event/pipeline choices, and failure propagation.

Coding produces:

- **Executable implementations:** algorithms, concrete classes, I/O, serialization, and integration code.

- **Verification artifacts:** unit tests, integration tests, property tests, benchmarks, fuzzing harnesses.
- **Operational artifacts:** logs, metrics hooks, tracing context propagation, configuration loaders.

1.1.4 Why design and architecture fail in real projects

Design and architecture most often fail for predictable reasons:

- **No explicit constraints:** boundaries exist only in people's heads; the codebase does not enforce them.
- **Hidden ownership and lifetime:** APIs return raw pointers and references without specifying who owns what and for how long.
- **Inconsistent error model:** failures are handled ad-hoc, leading to partial states and fragile recovery paths.
- **Uncontrolled dependencies:** convenience includes and shortcuts create cycles; modules become inseparable.
- **Architecture treated as "one-time":** a diagram is produced early and then ignored while the code evolves without rules.

Modern C++ does not automatically prevent these failures. What it does provide is a language in which design constraints can be made explicit and mechanically checkable.

1.1.5 How Modern C++ forces clarity

C++ forces clarity because it makes key design questions unavoidable:

- **Lifetime:** objects do not live by default; lifetimes must be engineered.

- **Ownership:** resource responsibility must be explicit or it becomes unsafe.
- **Cost:** copies, allocations, and synchronization have real, measurable impact.
- **Boundaries:** physical dependencies (headers, linkage, ABI) expose coupling early.
- **Concurrency:** data races are not "bugs"; they are violations of the language's execution model.

This encourages a discipline where design intent is expressed through types and interfaces rather than informal convention.

1.1.6 A concrete mapping: one requirement, three levels

Suppose the system must provide "user profile lookup" with caching and safe concurrent access.

Architecture-level decisions might include:

- caching is an infrastructure concern, not a domain rule,
- application services may depend on domain + interfaces, but not on concrete database drivers,
- the cache is shared across requests and must be concurrency-safe by contract,
- observability is mandatory at service boundaries (request id, latency, failures).

Design-level decisions might include:

- define an interface for profile storage and a separate interface for cache policy,
- choose value types for immutable profile snapshots returned to callers,
- define explicit error semantics: "not found" vs "I/O failure" vs "parse failure",

- specify thread-safety: whether the cache is internally synchronized or externally synchronized.

Coding-level decisions might include:

- implement LRU or TinyLFU policy,
- implement parsing, serialization, and retry behavior,
- write unit tests for cache behavior and integration tests for the storage backend,
- add metrics around cache hit rate and storage latency.

1.1.7 Design expressed as types and contracts (Modern C++ examples)

Example 1: ownership and lifetime as design. A design that returns an owning pointer expresses transfer of responsibility explicitly:

```
struct Profile { /* value object */ };

class IProfileStore {
public:
  virtual ~IProfileStore() = default;

  // Contract: returns a value snapshot; no aliasing with internal storage.
  virtual std::optional<Profile>
  try_load_profile(std::string_view user_id) = 0;
};
```

This design choice has architectural consequences:

- Returning `Profile` by value strongly suggests *value semantics* and avoids shared mutable state.
- Using `std::optional` makes absence explicit in the API contract.

Example 2: dependency direction as design. Introduce an interface for storage so policy code does not depend on mechanism:

```
class ProfileService {
public:
  explicit ProfileService(IProfileStore& store) : store_{store} {}

  std::optional<Profile> find(std::string_view user_id) {
    return store_.try_load_profile(user_id);
  }

private:
  IProfileStore& store_; // non-owning; lifetime managed by composition root
};
```

Here, the design makes the dependency direction visible:

- high-level service depends on an interface, not on a database driver,
- ownership is not ambiguous: `ProfileService` observes `store_` and does not own it.

Example 3: concurrency guarantees as part of design. A cache component must declare its thread-safety model. Two common designs:

```
class CacheExternallySynchronized {
public:
  // Contract: caller must hold the required lock for all calls.
  std::optional<Profile> get(std::string_view key);
  void put(std::string_view key, Profile value);
};

class CacheInternallySynchronized {
public:
```

```
  // Contract: thread-safe for concurrent calls.
  std::optional<Profile> get(std::string_view key);
  void put(std::string_view key, Profile value);

private:
  std::mutex m_;
};
```

This is a design decision, not an implementation detail. A system cannot scale safely unless the concurrency contract is explicit at the API boundary.

1.1.8 Common misunderstandings and corrections

- **"Architecture is about technologies."**
  Technology selection matters, but architecture is primarily about structures, constraints, and relationships that govern evolution.
- **"Design is class diagrams."**
  Class diagrams may document some aspects, but design is fundamentally about responsibilities, invariants, interfaces, failure semantics, ownership, and interactions.
- **"Coding will reveal the right design."**
  Implementation provides feedback, but without explicit constraints the codebase will drift toward accidental complexity.
- **"C++ makes design harder."**
  C++ makes *implicit* design harder. It rewards explicit design by making ownership, lifetime, cost, and boundaries enforceable.

1.1.9 Practical checklist: what to decide before you code

Before writing significant code, a competent design effort should answer:

- What are the **stable boundaries** and what is allowed to change behind them?
- What is the **dependency direction** among major modules?
- Which types are **values** and which are **identities**?
- For every resource, who **owns** it and what is its **lifetime**?
- What is the system's **error model** and how do failures compose?
- What is the **concurrency model** and what are thread-safety guarantees?
- What are the **performance budgets** (latency, memory, throughput) that shape the design?

If these are answered only implicitly, the system will still acquire answers—but by accident, not by design. In Modern C++, accidents are expensive.

1.1.10 Summary

- **Coding** realizes decisions as executable behavior.
- **Design** defines component structure, responsibilities, interfaces, contracts, and collaboration.
- **Architecture** defines system structures and global constraints that preserve evolvability and govern long-term change.

The thesis of this book is that Modern C++ is a uniquely effective environment for design discipline because it can encode design intent in types, lifetimes, interfaces, and dependency structure. When you embrace that explicitness, you produce systems that remain understandable and resilient under pressure.

1.2 Design as Constraint Management

At its core, software design is not about writing code. It is about making explicit decisions that shape how a system behaves, evolves, and withstands change over time. One of the most precise ways to understand this process is to view design as the art and science of *constraint management*. This section explains what constraints are, why they exist in software systems, how they influence design decisions, and how Modern C++ provides tools to express, enforce, and validate them.

1.2.1 What Are Constraints in Software Systems?

A **constraint** is any rule that limits the space of valid solutions. In physical engineering, a bridge must support load without failing; the laws of physics constrain design. In software, constraints arise from multiple sources:

- **Functional requirements**: what the system must do.
- **Correctness expectations**: invariants that must always hold.
- **Performance budgets**: limits on latency, memory usage, throughput.
- **Safety and security requirements**: predictable behavior under attack or fault.
- **Organizational boundaries**: how teams divide ownership and responsibility.
- **Platform and ABI rules**: how code is linked, deployed, and evolved over binary boundaries.

Design is the process of making these constraints explicit and manageable rather than hidden, ad-hoc, or deferred.

1.2.2 Why Constraint Management Is Central to Design

When a system is small, constraints are few and often informal. As systems grow, constraints accumulate, interact, and sometimes conflict. Without explicit design:

- constraints remain implicit and undocumented,
- violations surface unpredictably,
- tradeoffs are revisited inconsistently,
- modifications require deep understanding of unspoken assumptions.

A design that *manages* constraints creates boundaries that contain complexity, codifies invariants so they are verifiable, and structures code to make violations visible at compile time rather than at runtime.

1.2.3 Types of Constraints in Software Design

Behavioral Constraints. These are rules about what the system must do and must not do:

- Precondition: a condition that must be true before an operation begins.
- Postcondition: a condition that must hold when an operation completes.
- Invariant: a condition that must always hold for an object or subsystem.

```
// Example: invariants expressed through constructors and types.
class Account {
public:
  Account(double balance) : balance_{balance} {
    if (balance_ < 0.0) {
      throw std::invalid_argument("Balance must be non-negative");
    }
```

```
  }

  double get_balance() const noexcept {
    return balance_;
  }

private:
  double balance_;
};
```

Ownership and Lifetime Constraints. Who owns a resource and how long it lives profoundly influence correctness and performance.

```
// Unique ownership constraint:
std::unique_ptr<Widget> make_widget();

// Shared ownership constraint:
std::shared_ptr<Cache> global_cache();
```

Concurrency Constraints. Concurrency increases the constraint complexity:

- Which objects are thread-confined?
- Which are shareable and how are they synchronized?
- What operations are safe to invoke concurrently?

```
class ThreadSafeCounter {
public:
  void increment() {
    std::lock_guard<std::mutex> lg{m_};
    ++count_;
  }
```

```
  long value() const {
    std::lock_guard<std::mutex> lg{m_};
    return count_;
  }

private:
  mutable std::mutex m_;
  long count_{0};
};
```

Performance Constraints. Performance constraints govern how data is organized, how often allocations occur, and what operations are allowed on hot paths.

```
std::vector<int> prepare_buffer(std::size_t n) {
  std::vector<int> buf;
  buf.reserve(n); // constraint: avoid reallocations
  // fill buffer
  return buf;
}
```

Dependency and Modularity Constraints. These constraints limit which modules may depend on each other, promoting modular design and independent evolution.

```
// Forward declaration reduces physical dependency.
class Database;

class Service {
public:
  explicit Service(Database& db) : db_{db} {}
private:
  Database& db_;
};
```

1.2.4 Constraints as Design Tools, Not Obstacles

A constraint is not a mere restriction. It is a design *tool*:

- It prevents invalid states.
- It makes assumptions explicit and machine-checkable.
- It reduces the space of behaviors the system must account for.
- It enables local reasoning: modules can be understood in isolation.

When constraints are implicit, they become sources of errors; when explicit, they are sources of clarity.

1.2.5 Constraint Visibility Through Types

Types in C++ serve as a primary mechanism for making design constraints visible and enforceable:

- A type encodes a set of valid values.
- A function signature restricts the set of permissible calls.
- A template constraint (concept) restricts the domain of valid instantiations.

```
// A concept expressing a constraint.
template <typename T>
concept Addable = requires(T a, T b) {
  { a + b } -> std::same_as<T>;
};

template <Addable T>
T add(T a, T b) {
  return a + b;
}
```

1.2.6 Constraint Enforcement at Compile Time vs Runtime

One of the reasons Modern C++ is effective for design is the ability to enforce constraints at compile time:

- **Compile-time enforcement** moves detection of violations earlier in the development cycle, reducing debugging costs.
- **Runtime enforcement** remains important for dynamic conditions, but should be localized and documented.

```
// Example: constexpr enables compile-time checks.
constexpr int factorial(int n) {
  return (n <= 1) ? 1 : (n * factorial(n - 1));
}

static_assert(factorial(5) == 120);
```

1.2.7 Constraint Relaxation and Evolution

Systems evolve. Design must accommodate change without letting constraints become brittle. Constraint management includes:

- **Relaxation**: intentionally broadening a constraint when requirements change.
- **Refinement**: tightening a constraint to improve correctness or performance.
- **Versioning**: accommodating multiple constraint sets for evolving interfaces.

A well-designed system has mechanisms for evolving constraints without destabilizing dependent modules.

1.2.8 Tradeoffs and Constraint Conflicts

Not all constraints are compatible. Design often involves navigating tradeoffs:

- Strong isolation vs performance (copying vs shared state).
- Concurrency safety vs latency (locking vs lock-free).
- Generality vs simplicity (generic abstractions vs concrete types).

Good design does not avoid tradeoffs; it makes them explicit and justifiable.

1.2.9 Constraint Management Examples

Error handling constraint example. A function that must communicate success and failure explicitly may return a type that encodes both:

```
std::expected<int, std::string> parse_integer(std::string_view text) {
  try {
    size_t pos;
    int value = std::stoi(std::string{text}, &pos);
    if (pos != text.size()) {
      return std::unexpected{"Trailing characters"};
    }
    return value;
  } catch (const std::exception& e) {
    return std::unexpected{e.what()};
  }
}
```

This design makes the constraint explicit: callers must handle both success and error.

Concurrency constraint example. A thread-safe queue enforces constraints on concurrent access:

```
template <typename T>
class ThreadSafeQueue {
public:
  void push(T value) {
    std::lock_guard<std::mutex> lg{m_};
    q_.push(std::move(value));
  }

  std::optional<T> try_pop() {
    std::lock_guard<std::mutex> lg{m_};
    if (q_.empty()) {
      return std::nullopt;
    }
    T front = std::move(q_.front());
    q_.pop();
    return front;
  }

private:
  std::queue<T> q_;
  mutable std::mutex m_;
};
```

The design constraint here is a guarantee of safe concurrent calls for `push` and `try_pop`.

1.2.10 Summary

Software design framed as constraint management:

- treats constraints as first-class design artifacts,

- uses types and interfaces to make constraints visible and enforceable,
- separates compile-time and runtime enforcement where appropriate,
- embraces tradeoffs as integral to decisions,
- and ensures that constraints evolve without destabilizing the system.

Viewing design as constraint management converts "soft" intentions into "hard" boundaries that machines can check and humans can reason about. This perspective is central to building systems that remain comprehensible, correct, and evolvable over their lifetime.

1.3 Design as Decision Making

Software design is often described as "structure," "architecture," or "planning." Those descriptions are incomplete. A more precise and operational definition is this: **software design is decision making under constraints**. The codebase you inherit is not merely a collection of implementations; it is the crystallized record of thousands of decisions—some deliberate, many accidental. This section explains what counts as a design decision, why decision quality determines system longevity, how to make decisions explicit and reviewable, and how Modern C++ helps encode decisions so they remain enforceable years later.

1.3.1 The core premise: everything that matters is a decision

A design decision is any choice that:

- reduces the space of possible implementations,
- creates constraints for future changes,
- affects correctness, performance, security, or evolvability,

- changes how teams collaborate on the codebase.

A design decision is not necessarily "big." Choosing to return a raw pointer vs `std::unique_ptr` is a design decision. Choosing exceptions vs explicit error returns is a design decision. Choosing value semantics vs identity semantics is a design decision. The majority of long-term software pain comes from decisions that were never treated as decisions.

1.3.2 Deliberate decisions vs accidental decisions

Deliberate decisions. Deliberate decisions are made consciously, based on requirements and tradeoffs, documented in code through types and interfaces, and validated by tests and measurements. They tend to:

- have clear intent,
- expose constraints to the compiler,
- remain stable under change,
- be reviewable by others.

Accidental decisions. Accidental decisions occur when a team avoids choosing explicitly. The system still acquires a behavior, but by default, habit, framework pressure, or convenience. Accidental decisions tend to:

- hide constraints in informal convention,
- create undefined ownership and lifetime assumptions,
- produce inconsistent error handling,
- increase coupling and reduce replaceability.

A primary goal of this book is to help you convert accidental decisions into deliberate ones.

1.3.3 What makes a design decision “good”?

A good design decision is not the “most modern” or the “most elegant.” It is the decision that best satisfies the system’s constraints and future evolution. Good decisions usually share these properties:

- **They make invalid states hard to represent.**
- **They localize change.** A change in one area does not require rewriting unrelated areas.
- **They reduce hidden coupling.**
- **They are testable and observable.**
- **They have a clear cost model.** Performance and memory behavior are predictable.
- **They are explicit at the API boundary.** Callers cannot accidentally misuse the system.

Modern C++ excels when decisions are expressed through types, interfaces, and lifetimes that tools can check.

1.3.4 The decision surface: where design actually happens

Design decisions appear at multiple levels. The most important ones can be grouped into a practical taxonomy.

1) Boundary decisions

- Where does this component begin and end?
- What knowledge is allowed inside the boundary, and what is forbidden?
- What is stable, and what is replaceable?

In C++, boundary decisions have physical consequences: headers, libraries, modules, and ABI constraints. A boundary that is not enforced becomes fiction.

2) Dependency decisions

- Which direction do dependencies flow?
- Does high-level policy depend on low-level mechanisms?
- Can we compile and test this component in isolation?

3) Data model decisions

- Which entities are value types and which are identity types?
- Is mutation required? If so, where is it permitted?
- Is sharing allowed, and under what rules?

4) Ownership and lifetime decisions

- Who owns each resource?
- When is it released?
- Is lifetime tied to scope, to a component, or to a shared graph?

5) Error and failure decisions

- What constitutes a failure?
- How is it represented in interfaces?
- Is recovery local or propagated?
- What state remains valid after failure?

6) Concurrency decisions

- What is the concurrency model: message passing, shared state with locks, actors, or task-based execution?
- Which objects are thread-confined, immutable, externally synchronized, or internally synchronized?
- What is the policy for cancellation, timeouts, and backpressure?

7) Performance and resource-budget decisions

- Where are allocations allowed?
- What is the latency budget for hot paths?
- What is the memory and cache behavior of core data structures?

A team that cannot name these decisions is not designing; it is coding and hoping.

1.3.5 Encoding decisions in Modern C++

In many environments, decisions are recorded in documents. In C++, the best decisions are recorded in code itself, because code is executable and checkable. Modern C++ provides a vocabulary for embedding decisions into interfaces.

Ownership decision: raw pointer vs `unique_ptr` vs reference

```
// Decision: caller receives ownership; lifetime is transferred.
std::unique_ptr<Connection> open_connection(Config cfg);

// Decision: optional observation; no ownership; nullable.
Connection* try_connection() noexcept;
```

```
// Decision: non-null observation; no ownership; caller must ensure lifetime.
Connection& connection();
```

Each signature encodes a different decision about responsibility and lifetime. Good design makes those decisions explicit and consistent across the system.

Value vs identity decision

```
// Value type: state defines meaning; easy to copy/move; no shared mutation implied.
struct Money {
  std::int64_t cents{};
};

// Identity type: represents a long-lived entity; often shared, mutable, and needs lifetime
↪  rules.
class Account {
public:
  explicit Account(std::uint64_t id) : id_{id} {}
  std::uint64_t id() const noexcept { return id_; }

private:
  std::uint64_t id_;
};
```

A system that defaults everything to identity introduces aliasing and concurrency problems. A system that defaults everything to value avoids many of those problems and introduces identity only where it is justified.

Constraint decision: template requirements made explicit with concepts

```
template <typename T>
concept Serializable = requires(const T& v) {
```

```
  { v.serialize() } -> std::same_as<std::string>;
};

template <Serializable T>
std::string to_wire(const T& v) {
  return v.serialize();
}
```

This is a design decision: the function refuses types that do not satisfy the constraint. The compiler becomes an enforcement tool rather than a passive compiler of accidental assumptions.

Failure decision: explicit result types

```
struct IoError {
  int code{};
  std::string message;
};

// Decision: the caller must handle success or error explicitly.
std::variant<std::string, IoError> read_text_file(std::string_view path);
```

Making failure explicit at the interface boundary is a design choice that reduces ambiguity, improves testing, and prevents silent ignoring of error conditions.

1.3.6 The economics of decisions: why early clarity is cheaper

Design decisions have a cost curve:

- When a system is small, changing decisions is cheap.
- As usage spreads, the decision becomes embedded in many call sites.
- As teams grow, the decision becomes a coordination and compatibility problem.

- At scale, changing the decision becomes an ecosystem migration.

This is why interface decisions are disproportionately important. An unstable interface is not merely an inconvenience; it is a multiplier on future change cost.

1.3.7 Decision records: keeping intent alive

A mature engineering organization treats important design decisions as assets:

- **Make them explicit in interfaces and types.**
- **Name the decision in code review.**
- **Protect it with tests and invariants.**
- **Measure its performance implications.**
- **Document it briefly where it cannot be inferred from code.**

In this book, the strongest form of documentation is *enforced intent*:

- a type that cannot represent invalid states,
- an API that makes ownership unambiguous,
- a constraint checked by the compiler,
- a boundary supported by the build system and module structure.

1.3.8 A practical design-decision checklist

Before committing to a design, ask:

- **What decision is being made?** Name it explicitly.

- **What constraint does it satisfy?** (Correctness, performance, security, evolution, team scale.)
- **What tradeoff does it introduce?** (Complexity, runtime cost, API friction, flexibility.)
- **Can misuse be prevented mechanically?** (Types, concepts, RAII, encapsulation.)
- **How will this decision evolve?** (Versioning, migration path, compatibility plan.)
- **How will we validate it?** (Tests, benchmarks, static analysis, runtime assertions.)

If you cannot answer these questions, the "design" is not a design yet.

1.3.9 Summary

Design as decision making means:

- identifying decisions rather than letting defaults choose for you,
- making tradeoffs explicit and justifiable,
- encoding decisions into types, interfaces, and boundaries,
- protecting decisions with tests, invariants, and measurements,
- evolving decisions deliberately rather than rewriting in panic.

Modern C++ is a powerful environment for this discipline because it provides a rich vocabulary for expressing intent precisely and a compilation model that rewards explicit boundaries and contracts. The result is not merely code that runs, but systems whose design remains clear, enforceable, and resilient under change.

1.4 Why Design Matters More Than Language

A recurring misconception in software engineering is that language selection is the primary determinant of system quality. Languages matter, but they rarely dominate the outcomes that organizations care about: correctness under change, maintainability across teams, security posture, operational stability, and predictable performance. In practice, **software design**—the set of structural decisions that govern responsibilities, boundaries, dependencies, invariants, and evolution—dominates the long-term success or failure of a system.
This section explains why design matters more than language, what language can and cannot do for you, and how Modern C++ can amplify good design while severely exposing bad design.

1.4.1 Language is a tool; design is the system

A programming language provides:

- syntax and semantics to express computations,
- runtime and compilation models,
- standard libraries and ecosystem conventions,
- safety rails (strong typing, memory safety models, exceptions, etc.).

But a language does *not* automatically provide:

- stable boundaries,
- coherent dependency direction,
- clear ownership and lifetime policy,
- consistent error semantics,

- explicit concurrency contracts,
- local reasoning at scale.

Those properties are **design outcomes**. They must be chosen, encoded, and enforced. A weak design implemented in a "safer" language remains weak: it becomes hard to change, hard to test, difficult to scale across teams, and expensive to operate. Conversely, a strong design in a low-level language can remain reliable and evolvable for decades.

1.4.2 What actually breaks long-lived systems

In long-lived systems, failure rarely comes from a missing language feature. The common causes are structural:

- **Boundary erosion:** modules leak details; everything depends on everything.
- **Responsibility ambiguity:** the same concept is implemented in multiple places with divergent rules.
- **Hidden ownership and lifetime:** resources are shared implicitly; lifetimes are assumed, not guaranteed.
- **Ad-hoc error handling:** inconsistent failure conventions; partial states escape; recovery is unpredictable.
- **Unmanaged concurrency:** thread-safety promises are informal; sharing rules are unclear; data races appear.
- **Unmeasured performance:** costs are accidental; hot paths are cluttered; allocations and contention explode.

These are design problems. A language may help you *express* solutions, but it cannot decide the structure for you.

1.4.3 Why language debates are attractive but misleading

Language debates are attractive because language choice is visible, binary, and emotionally charged. Design is harder:

- design is multi-dimensional (correctness, performance, evolution, team structure),
- design is contextual (constraints differ across domains),
- design requires tradeoffs and discipline,
- design success is measured over years, not weeks.

As a result, teams sometimes treat language selection as a substitute for design thinking: "If we pick the right language, the architecture will be fine." That is a category error. Good design requires explicit decisions regardless of language.

1.4.4 What language *can* do: amplify good design or amplify bad design

Languages influence design outcomes by shaping the cost of certain choices:

- Some languages make certain errors harder (e.g., memory safety models).
- Some languages make certain designs easier to express (e.g., algebraic data types, traits, generics).
- Some languages impose runtime constraints that affect architecture (GC behavior, dynamic dispatch costs, ABI rules).

However, language does not replace design. It modifies the terrain on which design decisions are made.
A useful principle is:

> A language can raise the floor, but it cannot guarantee the ceiling.

You can raise the minimum level of safety or expressiveness, but you cannot guarantee high-quality boundaries, coherent contracts, or evolvable architecture without deliberate design.

1.4.5 Design dominates because change dominates

Most software cost is not the first implementation. It is the *years of change*:

- new features under deadline,
- new integrations,
- refactoring due to scaling requirements,
- security hardening,
- platform and toolchain migrations,
- team turnover and onboarding.

Design is the system's ability to absorb change *locally*. Language choice influences productivity and defect rates, but design determines whether change remains localized or becomes a cross-cutting disaster.

1.4.6 Design clarity is created at interfaces, not inside implementations

In long-lived systems, **interfaces** matter more than implementations because:

- interfaces spread widely (many call sites),
- interfaces encode assumptions (ownership, lifetimes, errors),
- interfaces define dependency direction,
- interfaces define compatibility constraints (API/ABI stability).

Modern C++ enables a particularly explicit interface vocabulary. For example, a return type is not merely “what data comes back”; it encodes design intent:

```
// Ownership transfer: caller becomes responsible for lifetime.
std::unique_ptr<Service> make_service(Config cfg);

// Non-owning, non-null: lifetime must be guaranteed elsewhere.
Service& service();

// Non-owning, nullable: caller must handle absence.
Service* try_service() noexcept;

// Value-return: caller receives a snapshot, no aliasing implied.
Profile load_profile(UserId id);
```

This is design. It constrains usage and makes the contract visible. A language that permits vague interfaces without friction may hide these decisions; C++ tends to force them into the open.

1.4.7 The myth of “language as architecture”

Teams sometimes equate architecture with language-level constructs:

- “We use classes, therefore we have OOP design.”
- “We use templates, therefore we have generic design.”
- “We use a framework, therefore we have architecture.”

These are misconceptions. Architecture is not the presence of a mechanism; it is the discipline of using mechanisms to enforce boundaries and manage dependencies.
A monolith can be well-architected. A microservice system can be poorly architected. Likewise, a project using advanced language features can be badly designed if responsibilities and constraints are unclear.

1.4.8 Modern C++ as a design stress test

C++ has sharp edges. That is precisely why it is valuable for learning and practicing real design discipline. C++ tends to reveal design problems earlier because:

- ownership and lifetime are explicit concerns,
- undefined behavior punishes incorrect assumptions,
- performance costs are visible and measurable,
- physical dependencies expose coupling (headers, linking, ABI).

This does not mean "C++ is better than other languages." It means that C++ is a medium where design intent must often be made explicit to avoid serious failure modes. When design is strong, C++ can produce systems with:

- predictable performance,
- tight control of resources,
- clear boundaries and contracts,
- long-term stability.

1.4.9 A design-first comparison: the same idea, different languages

Consider a simple requirement: parse input into a domain object, report failure explicitly, and prevent partial state from escaping.
In Modern C++, the design can be expressed with explicit result types:

```
struct ParseError {
  std::string message;
  std::size_t position{};
```

```
};

struct User {
  std::string name;
  std::string email;
};

// Design: parsing either produces a valid User or a descriptive error.
std::variant<User, ParseError> parse_user(std::string_view input);
```

The important point is not the syntax. It is the contract:

- success and failure are explicit and composable,
- invalid state is not returned as a partially constructed object,
- the interface forces the caller to handle the possibility of failure.

Any language can express a version of this. The difference is whether the language encourages explicitness and whether the codebase enforces the contract systematically. Design is the discipline that makes the contract consistent across the system.

1.4.10 When language matters more (and why it still does not replace design)

There are situations where language choice has outsized impact:

- safety-critical domains may require certain safety guarantees or certification constraints,
- certain runtime models (GC, JIT, sandboxing) affect latency and predictability,
- ecosystem availability can dominate time-to-market,
- ABI constraints matter for plugin systems and binary distribution.

Even there, design remains the determinant of system coherence:

- A GC language does not guarantee good lifetime boundaries; it changes the failure modes.
- A memory-safe language does not guarantee stable dependency direction; it changes the bug classes.
- A rich ecosystem does not guarantee clarity; it can increase architectural sprawl without boundaries.

Language shifts constraints; design manages them.

1.4.11 Practical guidance: a design-first mindset for C++ systems

In Modern C++, treat the following as design decisions, not implementation details:

- **Value vs identity:** prefer value semantics unless identity is necessary.
- **Ownership and lifetime:** encode policies in types (`unique_ptr`, references, explicit lifetimes).
- **Error model:** choose a coherent strategy and reflect it in APIs.
- **Concurrency model:** define thread-safety guarantees explicitly at boundaries.
- **Dependency direction:** enforce it in build targets and module structure.
- **Invariants:** ensure invalid states are unrepresentable or immediately rejected.

1.4.12 Summary

- Language choice influences expression, tooling, ecosystems, and some classes of errors.
- Design determines boundaries, responsibilities, constraints, and the cost of change.
- Most real system failures arise from structural decay, not missing language features.

- Modern C++ is powerful because it can encode design intent precisely and because it exposes ambiguity quickly.

In long-lived systems, design is the primary engineering asset. Languages are tools for expressing that asset. The more explicit the language is about lifetimes, costs, and boundaries, the more it rewards disciplined design—and the less it tolerates design by accident.

1.5 Why C++ Exposes Bad Design Faster Than Other Languages

C++ is not "unforgiving" because it is old or because its syntax is complex. It is unforgiving because it is close to the machine model and because it provides enormous freedom. That freedom turns design mistakes into concrete failures: crashes, data races, undefined behavior, ABI breaks, performance collapse, build-time explosions, and maintenance paralysis. In many other languages, poor design can remain hidden behind dynamic runtime behavior, garbage collection, reflection, or permissive dependency structures. In C++, the same design debt surfaces earlier because the language and toolchain make critical properties explicit.
This section explains *why* bad design becomes visible quickly in C++, and how Modern C++ provides disciplined tools to turn this pressure into clarity.

1.5.1 C++ makes key engineering realities unavoidable

Bad design often means "we avoided deciding." C++ forces decisions in areas where other environments allow ambiguity to persist:

- **Ownership and lifetime** cannot be ignored without risk.
- **Cost models** (copies, allocations, indirection) are visible and measurable.

- **Concurrency correctness** is governed by a strict memory model with severe consequences for data races.
- **Binary and build boundaries** expose dependency and coupling problems early.

In other words, C++ converts vague design choices into concrete obligations.

1.5.2 1) Ownership and lifetime: ambiguity becomes failure

In many ecosystems, memory management is handled by the runtime. That can hide lifetime mistakes for a long time, but it does not eliminate them; it shifts them into different failure modes (latency spikes, pressure on the collector, retained memory graphs, implicit sharing). In C++, lifetime mistakes surface as:

- use-after-free,
- double deletion,
- leaks that accumulate predictably,
- dangling references and invalid iterators,
- subtle aliasing bugs.

These failures are not “C++ problems”; they are **design problems**:

- Who owns the object?
- When is it destroyed?
- Can it be shared?
- Can it outlive the scope that created it?

Modern C++ provides a vocabulary to make these decisions explicit.

```cpp
// Good design: ownership is explicit.
std::unique_ptr<Session> make_session(Config cfg);

// Sharing is explicit and thus reviewable.
std::shared_ptr<Cache> shared_cache();

// Non-owning access is explicit: lifetime must be guaranteed elsewhere.
Session& session();
Session* try_session() noexcept;
```

When a design refuses to answer these questions, C++ will answer them through failures.

1.5.3 2) Undefined behavior is a design stress test

In C++, certain incorrect assumptions do not merely cause errors; they can trigger **undefined behavior** (UB), where the program has no defined semantics. UB is not a "runtime exception"; it is the absence of a valid contract between your program and the language rules.
Bad design patterns that often lead to UB include:

- violating object lifetime rules (dangling references/pointers),
- breaking aliasing assumptions,
- data races (which are UB by the language model),
- violating preconditions of the standard library.

In other languages, similar mistakes might cause predictable exceptions. In C++, they can corrupt program state silently. That harshness forces serious teams to design for:

- explicit invariants,
- narrow interfaces,

- clear ownership,
- disciplined concurrency models.

The result is that design debt becomes visible sooner because the language does not hide the consequences.

1.5.4 3) The C++ memory model exposes concurrency mistakes early

Concurrency is a design problem, not a library choice. C++ makes this explicit because:

- data races are not “bugs”; they are *undefined behavior* under the standard,
- atomic operations and synchronization primitives have precise semantics,
- thread-safety cannot be assumed; it must be designed and documented.

A poorly designed system that allows accidental sharing will eventually exhibit:

- non-reproducible failures,
- rare corruption under load,
- performance collapse due to contention,
- deadlocks due to unclear lock ordering.

Modern C++ encourages explicit concurrency contracts at API boundaries:

```
// Thread-confined: must only be used on one thread.
class Connection {
public:
  void send(std::string_view msg);
};
```

```cpp
// Internally synchronized: safe for concurrent calls.
class ThreadSafeLog {
public:
  void write(std::string_view line);

private:
  std::mutex m_;
};
```

If a codebase lacks such explicit contracts, C++ makes the resulting failures difficult enough that teams are forced to confront the design.

1.5.5 4) Cost transparency punishes architectural ambiguity

In performance-sensitive systems, design mistakes often appear as:

- excessive allocations,
- accidental copies,
- cache-unfriendly data layouts,
- unpredictable latency spikes,
- unnecessary dynamic dispatch or indirection.

C++ makes these costs observable:

- value categories distinguish copy vs move,
- object layout and allocation strategies are under your control,
- abstractions can be zero-cost, but only if designed carefully.

A design that is careless about cost might “work” in other environments but become unusable in C++ contexts where latency and memory budgets are strict. C++ thus exposes architectural cost mistakes quickly.

```
// Design choice: avoid repeated reallocations by reserving capacity.
std::vector<int> build_data(std::size_t n) {
  std::vector<int> v;
  v.reserve(n);
  for (std::size_t i = 0; i < n; ++i) v.push_back(static_cast<int>(i));
  return v;
}
```

This is not micro-optimization; it is respecting a cost constraint that should be part of the design.

1.5.6 5) Physical dependencies reveal coupling

C++ has a compilation model where include relationships, templates, and link boundaries are tangible. Poor modular design becomes visible as:

- compile times exploding,
- unnecessary rebuild cascades,
- brittle build ordering,
- circular dependencies that block refactoring,
- wide headers that force global knowledge.

This is one of the fastest ways C++ exposes bad design: **the build system becomes a diagnostic tool.** When every small change triggers huge recompilations, the architecture is often too coupled.
Modern C++ offers mechanisms to reduce this:

- forward declarations and pImpl to minimize header exposure where appropriate,
- careful header hygiene and separation of interface from implementation,
- modules (where supported) to formalize boundaries and reduce accidental coupling.

```
// Forward declaration: reduces dependency surface of the header.
class Database;

class UserService {
public:
  explicit UserService(Database& db) : db_{db} {}
  void create_user(std::string_view name);

private:
  Database& db_; // non-owning; lifetime managed elsewhere
};
```

Again, this is design: deciding what knowledge is allowed to cross a boundary.

1.5.7 6) ABI and binary boundaries punish unstable interfaces

Many C++ systems require:

- shared libraries distributed separately,
- plugins loaded at runtime,
- long-lived SDKs used by external clients,
- components compiled with different toolchains or versions.

In these contexts, bad design is exposed through ABI breakage:

- changing class layouts breaks clients,

- exceptions across boundaries can be unsafe,
- inline functions and templates can leak implementation details,
- allocation/deallocation across module boundaries can be mismatched.

A design that ignores these realities often “works on my machine” but fails in production deployments. Therefore, C++ systems that live long must treat ABI stability as part of architecture, not as a late concern.

1.5.8 7) Tooling and sanitizers make consequences measurable

Modern C++ development commonly uses powerful analysis tools (static analyzers, sanitizers, fuzzing, and hardened builds). These tools amplify the language’s tendency to expose design flaws by making them detectable early:

- lifetime errors are caught under sanitizers,
- data races become visible under dynamic race detection,
- contract violations appear through assertions and defensive checks,
- undefined behavior can be surfaced under UB checks.

The existence of these tools is not an excuse for weak design. It is a reason to design in ways that the tools can validate.

1.5.9 C++ is not “hard”; vague design is hard

Many developers experience C++ as difficult because C++ makes vague design choices painful:

- unclear ownership becomes memory hazards,
- unclear concurrency becomes data races,

- unclear boundaries become build-time and ABI failures,
- unclear performance constraints become unpredictable latency.

When design is explicit, Modern C++ becomes significantly more manageable because the code communicates intent and constraints clearly.

1.5.10 Design discipline: turning pressure into clarity

To use C++ as an advantage rather than a penalty, adopt a discipline that encodes decisions:

- **Make ownership explicit** in interfaces; avoid ambiguous raw owning pointers.
- **Prefer value semantics** for data transfer; introduce identity deliberately.
- **Define error models** consistently; make failure visible at boundaries.
- **Declare concurrency contracts** in types and APIs; document thread-safety precisely.
- **Control dependencies** physically (headers/modules/build targets) and logically (dependency direction).
- **Protect invariants** through constructors, encapsulation, and narrow interfaces.

1.5.11 Summary

C++ exposes bad design faster because it makes crucial properties explicit:

- lifetimes and ownership cannot be safely ignored,
- undefined behavior punishes incorrect assumptions,
- the memory model demands explicit concurrency discipline,

- costs are transparent and must be managed,
- physical dependencies reveal coupling quickly,
- ABI boundaries punish unstable interfaces.

This is not a weakness of C++. It is an engineering reality. For long-lived systems, early exposure of design flaws is a feature: it forces teams to build with explicit contracts, stable boundaries, and disciplined constraints. That is the foundation on which resilient software is built.

Complexity: The Real Enemy

2.1 Accidental vs Essential Complexity

Complexity is the dominant cost driver in long-lived software. It increases defect rates, slows delivery, raises onboarding time, and turns simple changes into risky operations. The most important design skill is not adding features or patterns; it is *controlling complexity*. A foundational distinction is between **essential complexity** (inherent to the problem) and **accidental complexity** (introduced by our solution).

2.1.1 Definitions

Essential complexity. **Essential complexity** is the irreducible complexity of the domain itself. It comes from:

- the rules of the business or problem space,
- the number of real-world states and transitions the system must represent,
- correctness and safety requirements,
- concurrency, distribution, and failure conditions that are inherent to the environment.

If you remove essential complexity, you no longer solve the problem.

Accidental complexity. **Accidental complexity** is complexity added by our choices:

- overly clever abstractions or premature generalization,
- unnecessary indirection layers,
- scattered responsibilities and unclear ownership,
- inconsistent error handling and ad-hoc conventions,
- build and dependency tangles,
- avoidable runtime overhead and framework-driven boilerplate.

If you remove accidental complexity, you still solve the problem—often better.

2.1.2 Why this distinction matters in design

Design decisions should be evaluated by a single question:

> Does this decision reduce essential complexity, or does it merely add accidental complexity?

Good design does not pretend that essential complexity can be eliminated. Instead, it:

- makes essential complexity *explicit* and *localized*,
- prevents accidental complexity from spreading across the system,
- turns implicit assumptions into explicit constraints.

In long-lived systems, the most common failure mode is not that essential complexity grows (that is expected), but that accidental complexity grows faster than the team's ability to understand and control it.

2.1.3 Where essential complexity comes from

Essential complexity is typically created by the nature of reality:

- **State and invariants:** accounts cannot go negative; orders have lifecycle rules; permissions must be enforced.
- **Time and concurrency:** simultaneous updates, race conditions, partial failures, timeouts.
- **Distribution:** retries, idempotency, eventual consistency, network partitions.
- **Security:** authentication, authorization, auditing, and threat models.
- **Regulatory constraints:** logging requirements, data retention, privacy rules.

Design cannot remove these. It can only represent them in a way that is understandable and enforceable.

2.1.4 Where accidental complexity comes from

Accidental complexity usually comes from human factors:

- **Ambiguity:** unclear ownership, unclear lifetimes, unclear thread-safety contracts.
- **Convenience coupling:** “just include that header,” “just call the database here,” “just share the pointer.”
- **Generalization without evidence:** frameworks for requirements that do not exist yet.
- **Abstraction inflation:** adding interfaces and layers that do not reduce cognitive load.
- **Inconsistent conventions:** different error models, naming schemes, logging styles, and dependency patterns in different modules.

- **Tooling and build misuse:** global headers, heavy template exposure, monolithic targets that force rebuild cascades.

A key property of accidental complexity is that it tends to be invisible in the short term and devastating in the long term.

2.1.5 A practical diagnostic: the "complexity budget"

Every team has a limited complexity budget: a finite amount of system behavior it can understand, maintain, and evolve safely. Essential complexity consumes that budget by necessity. Accidental complexity consumes it wastefully.
A system becomes fragile when:

- essential complexity rises (as products evolve),
- accidental complexity rises (as shortcuts accumulate),
- and the team's capacity to reason about the code does not rise at the same rate.

Design is the discipline of keeping accidental complexity low enough that the inevitable essential complexity remains manageable.

2.1.6 Modern C++ and the two kinds of complexity

Modern C++ can reduce accidental complexity when used to encode constraints and make intent explicit:

- **RAII** makes resource lifetime deterministic and local.
- **Value semantics** reduce aliasing and hidden coupling.
- **Strong types** prevent invalid states and accidental misuse.

- **Concepts** make template requirements explicit and diagnostics clearer.
- **Standard vocabulary types** (`std::optional`, `std::variant`, `std::expected` where available) make uncertainty and failure explicit in APIs.

However, C++ can also *increase* accidental complexity when misused:

- excessive template metaprogramming for aesthetics,
- deep inheritance trees that hide behavior,
- unclear ownership through raw pointers,
- overuse of shared ownership without a clear lifetime model,
- header sprawl that collapses modular boundaries.

Therefore, Modern C++ is best understood as a powerful constraint language: it rewards deliberate design and punishes ambiguity.

2.1.7 Example 1: Essential complexity represented cleanly

Assume an essential domain rule: "An `Order` must follow a strict lifecycle and cannot skip states." That complexity is essential. The design goal is to represent it explicitly and prevent invalid transitions.

```
enum class OrderState { Created, Paid, Shipped, Completed, Cancelled };

class Order {
public:
  explicit Order(std::uint64_t id) : id_{id} {}

  OrderState state() const noexcept { return state_; }
```

```
  bool pay() {
    if (state_ != OrderState::Created) return false;
    state_ = OrderState::Paid;
    return true;
  }

  bool ship() {
    if (state_ != OrderState::Paid) return false;
    state_ = OrderState::Shipped;
    return true;
  }

  bool complete() {
    if (state_ != OrderState::Shipped) return false;
    state_ = OrderState::Completed;
    return true;
  }

  bool cancel() {
    if (state_ == OrderState::Completed) return false;
    state_ = OrderState::Cancelled;
    return true;
  }

private:
  std::uint64_t id_{};
  OrderState state_{OrderState::Created};
};
```

This design does not eliminate essential complexity; it *contains* it:

- the rule lives in one place,
- invalid transitions are rejected deterministically,

- the rest of the system does not need to replicate the rules.

2.1.8 Example 2: Accidental complexity caused by unclear ownership

Consider a service that returns a raw pointer. The interface hides the most important question: who owns the object?

```
// Accidental complexity: ownership is ambiguous.
Widget* get_widget(std::string_view id);
```

Callers now must guess:

- Should the pointer be deleted?
- Is it cached and shared?
- How long does it remain valid?
- Can it be used concurrently?

A design that makes ownership explicit reduces accidental complexity:

```
// Explicit decisions: ownership and lifetime are visible in the API.
std::unique_ptr<Widget> make_widget(std::string_view id); // transfer ownership
Widget& widget(std::string_view id);                      // non-owning, non-null
Widget* try_widget(std::string_view id) noexcept;         // non-owning, nullable
```

Here, the complexity does not disappear; it becomes *explicit* and *checkable*. This is how design converts accidental complexity into manageable constraints.

2.1.9 Example 3: Accidental complexity from “abstraction inflation”

A common anti-pattern is to create multiple indirection layers without a clear constraint that requires them.

```
// Accidental complexity: indirection without a concrete need.
struct IDataProvider { virtual ~IDataProvider() = default; virtual Data get() = 0; };
struct IDataProviderFactory { virtual ~IDataProviderFactory() = default;
  virtual std::unique_ptr<IDataProvider> create() = 0; };

struct DataProviderManager {
  explicit DataProviderManager(IDataProviderFactory& f) : f_{f} {}
  Data fetch() { return f_.create()->get(); } // transient objects and hidden cost
private:
  IDataProviderFactory& f_;
};
```

This may be justified in a plugin/ABI world. Without that constraint, it often adds:

- extra allocations,
- extra virtual dispatch,
- extra concepts for readers to hold in their head,
- little reduction in essential complexity.

A simpler design that matches actual constraints is usually superior:

```
// If no runtime replacement is required, prefer direct composition.
class DataProvider {
public:
  Data get();
};
```

```
class Service {
public:
  explicit Service(DataProvider& p) : p_{p} {}
  Data fetch() { return p_.get(); }
private:
  DataProvider& p_;
};
```

The rule is not “avoid interfaces.” The rule is: **introduce indirection only when it buys you a constraint you truly need** (replaceability, testing seams, ABI boundaries, plugin architecture, etc.).

2.1.10 Techniques for reducing accidental complexity

1) Make constraints explicit at boundaries

- ownership and lifetime in types,
- failure and absence in return types,
- thread-safety in API contracts.

2) Prefer value semantics for data transfer

Value-oriented designs reduce aliasing and hidden coupling, making systems easier to test and parallelize.

3) Reduce dependency surface

Minimize what crosses module boundaries:

- narrow headers,
- forward declarations where appropriate,

- avoid leaking implementation details through templates when not necessary.

4) Delay generalization until structure is stable

Generalize only when duplication reveals a stable abstraction, not when imagination predicts future needs.

5) Establish a coherent error model

A unified approach to failure avoids duplicated conventions and inconsistent recovery.

2.1.11 A decision framework: “Is this complexity essential?”

Before introducing a new mechanism, ask:

- What essential constraint does this mechanism satisfy?
- What accidental complexity does it introduce (concept count, indirection, build coupling, runtime cost)?
- Can the constraint be satisfied with a simpler structure?
- Does it make the system easier to reason about locally?
- Can misuse be prevented mechanically (types, RAII, constraints)?

If the mechanism does not reduce essential complexity or enforce a needed constraint, it is likely accidental complexity.

2.1.12 Summary

- **Essential complexity** is inherent to the problem; it must be represented, not denied.

- **Accidental complexity** is created by our choices; it must be minimized aggressively.
- Strong design makes essential complexity explicit and localized while preventing accidental complexity from spreading.
- Modern C++ is a powerful tool for reducing accidental complexity by encoding constraints in types, lifetimes, and interfaces—but it can also amplify accidental complexity when used for unnecessary abstraction or cleverness.

The central design task is therefore clear: *pay the unavoidable cost of essential complexity, and refuse to pay the compound interest of accidental complexity.*

2.2 Scaling Code vs Scaling Teams

Software complexity grows not just with the number of lines of code, but with the number of people who must understand, modify, and extend that code. Scaling a codebase for performance or features is one challenge; scaling a team that collaborates effectively on that code is another. These two dimensions interact in profound ways: a design that supports one does not necessarily support the other. In long-lived systems, team scale often becomes the dominant driver of architectural decisions, because design choices that worked for small teams can become bottlenecks, communication hazards, or risk multipliers as the team grows. This section analyzes the differences, the traps, and how Modern C++ design practices can support both.

2.2.1 What it means to scale code

Scaling code refers to:

- increasing the size of the codebase without exponential growth in complexity,
- preserving clarity and correctness as features are added,

- maintaining low cognitive load for developers working on individual components,
- enabling composition of modules without fragile dependencies.

A codebase scales when:

- its internal structure mirrors domain structure cleanly,
- interfaces are stable and explicit,
- invariants and constraints are visible and enforced,
- accidental complexity is minimized,
- duplication is limited and abstracted where justified.

2.2.2 What it means to scale teams

Scaling teams refers to:

- increasing the number of engineers collaborating on the codebase,
- splitting work into independent or loosely coordinated streams,
- maintaining productivity without exponential communication overhead,
- preserving correctness and consistency despite parallel efforts,
- onboarding new members efficiently.

Team scale creates pressures that small teams rarely experience:

- the need for clear boundaries to reduce merge conflicts,
- the need for explicit contracts to reduce coordination overhead,
- the need for documented invariants to reduce knowledge silos,
- the need for modular compilation to reduce build contention.

2.2.3 Why scaling teams is harder than scaling code

When a small team grows, the biggest friction arises from social and cognitive dependencies, not from raw lines of code. Classic software engineering research (e.g., Brook's Law) observes that the number of communication paths grows rapidly with team size. To work effectively, a system must exhibit:

- **Local reasoning**: developers can make changes in isolation with confidence.
- **Stable contracts**: APIs do not change capriciously, reducing the need for cross-team coordination.
- **Clear ownership**: each team or subteam owns defined areas of the codebase.
- **Modular boundaries**: teams can work on different modules without inducing coupling conflicts.

A design that scales code but not teams will result in:

- frequent merge conflicts,
- fragile cross-module dependencies,
- informal conventions that do not scale,
- slow onboarding of new engineers,
- decision bottlenecks at central authority figures.

2.2.4 Interfaces and boundaries: the nexus of code and team scaling

The most important artifact for both scaling code and teams is the **interface**. A well-designed interface:

- reduces the cognitive load on the consumer of a component,
- clarifies the responsibilities and guarantees of a module,
- enables parallel development without constant negotiation,
- allows independent evolution of implementation.

In Modern C++, interfaces are encoded in:

- abstract base classes with narrow responsibilities,
- value semantics for simple data exchange,
- strong types that prevent misuse,
- concepts that constrain template usage meaningfully,
- header boundaries or module boundaries to enforce dependency rules.

For example, consider a service interface that must be safely serializable and thread-safe:

```
// Interface with explicit constraints: serializable and thread-safe.
class IService {
public:
  virtual ~IService() = default;
  virtual std::string serialize() const = 0;
  virtual void execute() = 0;
};
```

This interface allows multiple teams to implement independently, provided they honor the contract.

2.2.5 Dependency direction and team autonomy

One of the core principles for scaling teams is the *direction of dependencies*. If high-level policy depends on low-level details, teams working on high-level features must coordinate with those implementing low-level modules. This creates bottlenecks.
A better design in Modern C++ uses **inversion of control** and **dependency injection**, where high-level components depend on narrow interfaces, and the concrete implementations are provided externally. For example:

```
class Renderer {
public:
  virtual void draw() const = 0;
};

class Application {
public:
  explicit Application(Renderer& r) : renderer_{r} {}
  void run() {
    // high-level logic independent of concrete renderer
    renderer_.draw();
  }
private:
  Renderer& renderer_;
};
```

Teams working on `Application` and concrete `Renderer` implementations can progress independently, coordinating only on the shared interface.

2.2.6 Concurrency, teams, and merge contention

As teams scale, so does parallel work. Concurrency is not just a runtime concern; it is a design concern that interacts with team activity:

- Tests should not interfere with each other due to shared global state.
- Subsystems under development by different teams should avoid shared mutable state when possible.
- Thread-safety contracts must be clear in public interfaces to avoid implicit coordination requirements.

Modern C++ encourages explicit concurrency contracts:

```
// Thread-safe queue with explicit synchronization
template <typename T>
class ThreadSafeQueue {
public:
  void push(T value) {
    std::lock_guard<std::mutex> guard{mutex_};
    queue_.push(std::move(value));
  }

  std::optional<T> try_pop() {
    std::lock_guard<std::mutex> guard{mutex_};
    if (queue_.empty()) return std::nullopt;
    T front = std::move(queue_.front());
    queue_.pop();
    return front;
  }
private:
  std::queue<T> queue_;
  mutable std::mutex mutex_;
};
```

This explicitness reduces accidental coupling through global state, an important factor when many developers interact with shared subsystems.

2.2.7 Build and modularity considerations

Build performance and modular separation are technical concerns with social impact. When every small change triggers large recompilation, or when code boundaries are blurred, teams suffer:

- slower feedback loops,
- increased merge conflicts,
- reduced willingness to refactor,
- fear of breaking dependent modules.

Modern C++ gives tools to manage this:

- **C++ Modules** (where supported) to reduce header inclusion footprints,
- **Forward declarations** and opaque types to minimize dependencies,
- **Narrow interfaces** and clear separation of declarations and definitions to accelerate builds.

A well-partitioned build structure allows teams to work on different modules without frequent rebuilds of the entire codebase.

2.2.8 Onboarding and knowledge transfer

As teams grow, new members must become productive quickly. Design that scales teams:

- uses consistent naming conventions,
- encapsulates invariants in types and interfaces,
- minimizes accidental complexity,

- documents design decisions alongside code,
- uses types to enforce constraints rather than comments or external artifacts alone.

For example, a strong type for user identifiers prevents confusion:

```
// Strong type for user identity reduces accidental misuse.
struct UserId {
  std::uint64_t value;
};
```

This simple construct improves readability, reduces class-of-bug mistakes, and signals intent: user identity is not just an integer.

2.2.9 Governance, ownership, and team boundaries

In large organizations, code ownership aligns with team boundaries. Design should respect those boundaries:

- Components that are owned by different teams should communicate through stable, narrow interfaces.
- Teams should avoid shared mutable state unless governed by clear concurrency contracts.
- Libraries should be versioned and have compatibility guarantees where external use is expected.

Modern C++ design can support these needs with:

- **ABI-safe interfaces** for public-facing modules,
- **Encapsulation** to reduce the need for cross-team coordination when internals evolve,
- **Explicit lifetime and ownership models** to avoid accidental contract violation.

2.2.10 Summary

- Scaling code and scaling teams are related but distinct challenges. A system that scales in complexity does not automatically support a growing team.
- Interfaces and modular boundaries are the linchpins of both code and team scaling.
- Modern C++ supports precise expression of contracts, ownership, concurrency, and dependency direction, which are essential for team autonomy and parallel development.
- Clear design, explicit constraints, and disciplined use of language features reduce accidental complexity and enable teams to scale effectively.
- Organizational and build structure choices interact with design to affect how teams coordinate, merge, and evolve the system over time.

2.3 Cognitive Load in Large Systems

Large systems rarely fail because individual functions are difficult. They fail because the *mental workload required to understand and change the system* exceeds what humans can sustain reliably. That workload is called **cognitive load**: the amount of working memory and attention required to perform a task such as reading code, reasoning about behavior, debugging, reviewing a change, or extending an API.
In long-lived software, cognitive load becomes the dominant constraint. Performance issues can be profiled. Memory issues can be measured. But cognitive overload silently produces the worst outcomes: shallow understanding, risky changes, fear-driven development, over-reliance on a few experts, and architectural decay.

2.3.1 What cognitive load means in software engineering

Cognitive load in a codebase is the mental effort required to answer questions such as:

- What does this component do, and what does it *not* do?
- What are the inputs, outputs, invariants, and failure modes?
- Who owns this object, and how long does it live?
- What can be called concurrently, and what must be synchronized?
- What changes are safe locally, and what changes are system-wide?

A large system is maintainable when an engineer can answer these questions *locally* from the code and its interfaces, without requiring tribal knowledge or heroic debugging.

2.3.2 Why cognitive load explodes as systems grow

Cognitive load grows faster than code size because it is driven by **interactions**:

- more modules means more boundaries,
- more boundaries means more contracts,
- more contracts means more assumptions,
- more assumptions means more ways to violate invariants.

Two systems can have the same number of lines of code but radically different cognitive load depending on:

- coupling and dependency direction,
- clarity of ownership and lifetime,
- uniformity of error handling,
- stability of public interfaces,

- predictability of concurrency rules,
- consistency of naming and conventions.

2.3.3 Three kinds of load in code comprehension

It is useful to separate cognitive load into three engineering-relevant categories:

Intrinsic load (problem complexity). The unavoidable complexity of the domain. For example, a trading engine, a medical device, or a distributed storage system has real essential rules. Intrinsic load cannot be eliminated; it must be represented clearly.

Extraneous load (accidental complexity). Complexity introduced by the solution: unclear ownership, inconsistent conventions, scattered responsibilities, unnecessary indirection, and opaque abstractions. This is the primary target of design improvement.

Germane load (learning and building mental models). The effort invested in forming correct mental models. Good design increases germane load in a healthy way by making the system *learnable*: concepts have names, boundaries are stable, and invariants are enforced.
A mature design reduces extraneous load so that intrinsic and germane load remain manageable.

2.3.4 The cognitive units of a system: concepts, not classes

Humans do not scale by reading more code; they scale by reasoning in **chunks**:

- domain concepts (Order, Account, Session),
- architectural roles (Service, Repository, Adapter),
- policies (Retry policy, Cache policy),

- invariants (non-negative balance, valid UTF-8, monotonic timestamp),
- boundaries (public API vs internal details).

A design succeeds when these chunks are:

- named consistently,
- represented by types and interfaces,
- enforced by invariants,
- isolated behind boundaries.

A design fails when a single concept is scattered across many files with no authoritative home.

2.3.5 High cognitive load smells in large systems

The following are reliable indicators of excessive cognitive load:

- **Ambiguous ownership:** raw owning pointers, unclear lifetime rules, hidden sharing.
- **Wide interfaces:** “god” services exposing many unrelated operations.
- **Implicit contracts:** preconditions and thread-safety rules exist only in comments or tribal knowledge.
- **Inconsistent error model:** sometimes exceptions, sometimes error codes, sometimes null, sometimes logging-and-continuing.
- **Hidden global state:** behavior depends on mutable singletons and implicit configuration.
- **Uncontrolled dependencies:** everything includes everything; small changes rebuild the world.

- **Temporal coupling:** methods must be called in a specific order without being expressed in the type system.

These smells often appear together because they share a root cause: design decisions were not encoded into enforceable structure.

2.3.6 How Modern C++ can reduce cognitive load

Modern C++ provides mechanisms that make design intent *explicit* and therefore easier to understand:

1) Express ownership and lifetime in types

```
// Ownership is explicit: transfer ownership to the caller.
std::unique_ptr<Session> create_session(Config cfg);

// Non-owning reference: requires external lifetime management.
Session& session();

// Nullable observation: absence is explicit.
Session* try_session() noexcept;
```

These signatures reduce cognitive load because a reader does not need to guess the lifetime policy.

2) Prefer value semantics for data exchange

Value semantics reduce aliasing, which reduces the number of hidden interactions an engineer must consider:

```
struct Profile {
  std::string name;
```

```
  std::string email;
};

// Value-return: caller receives a snapshot. No shared mutable state implied.
Profile load_profile(UserId id);
```

3) Use vocabulary types for uncertainty and alternatives

```
std::optional<Profile> try_load_profile(UserId id);

// Alternatives are explicit and exhaustively handled.
using ParseResult = std::variant<Profile, ParseError>;
ParseResult parse_profile(std::string_view text);
```

The cognitive burden shifts from guessing to reading explicit types.

4) Enforce template requirements with concepts

Large systems often suffer from "generic code that accepts anything until it explodes in errors." Concepts reduce that load by stating requirements up front:

```
template <typename T>
concept HashableKey = requires(T k) {
  { std::hash<T>{}(k) } -> std::convertible_to<std::size_t>;
  { k == k } -> std::convertible_to<bool>;
};

template <HashableKey Key, typename Value>
class Cache {
  // ...
};
```

This is not only correctness. It is a readability feature: the design constraint is visible.

2.3.7 Local reasoning: the primary cognitive scalability goal

A system scales cognitively when most changes can be made with **local reasoning**:

- Understand one component and its interface.
- Make a change behind that interface.
- Validate locally (tests) without needing global knowledge.

Local reasoning is destroyed by:

- shared mutable state,
- scattered invariants,
- side effects hidden behind innocuous names,
- broad dependencies.

Local reasoning is enabled by:

- clear boundaries,
- stable interfaces,
- explicit ownership and error semantics,
- deterministic resource management (RAII),
- controlled dependency direction.

2.3.8 Reducing cognitive load in APIs: narrow, explicit, and stable

A high-load API is typically one that is wide and implicit:

```
// High cognitive load: wide interface, unclear contracts.
class SystemManager {
public:
  void init();
  void start();
  void stop();
  void reset();
  void updateAll();
  void doWork(int mode);
  void setConfig(const char* text);
  int lastError();
  // ...
};
```

A lower-load design narrows responsibilities and makes contracts explicit:

```
struct StartError { std::string message; };

class IRunner {
public:
  virtual ~IRunner() = default;
  virtual std::optional<StartError> start() = 0;
  virtual void request_stop() noexcept = 0;
};

class IConfiguration {
public:
  virtual ~IConfiguration() = default;
  virtual std::string_view get(std::string_view key) const noexcept = 0;
};
```

Even if the implementation is complex, the cognitive load is contained because responsibilities are separated and contracts are visible.

2.3.9 Cognitive load and concurrency in large systems

Concurrency multiplies cognitive load because it adds time and interleavings. A good design reduces that load by being explicit about the concurrency model.
Two safe and cognitively efficient approaches are:

Immutability and value passing. Prefer immutable snapshots shared across threads.

```
struct Snapshot {
  std::vector<int> values;
};

// If Snapshot is treated as immutable after construction,
// it can be shared safely without locks.
std::shared_ptr<const Snapshot> current_snapshot();
```

Encapsulated synchronization. If mutation is needed, encapsulate locks inside a component and document the thread-safety contract.

```
class ThreadSafeCounter {
public:
  void increment() {
    std::lock_guard<std::mutex> g{m_};
    ++v_;
  }

  std::int64_t value() const {
    std::lock_guard<std::mutex> g{m_};
    return v_;
```

```
  }

private:
  mutable std::mutex m_;
  std::int64_t v_{0};
};
```

The goal is not “no locks.” The goal is **no ambiguous sharing rules**.

2.3.10 Build structure as cognitive structure

In C++, physical structure affects cognitive load:

- heavy headers and broad includes increase both compile time and mental coupling,
- template leakage exposes internal details to the entire system,
- monolithic targets blur boundaries and ownership.

A system with clean build boundaries tends to have cleaner mental boundaries:

- fewer allowed dependencies,
- smaller public surfaces,
- fewer “global” design assumptions.

When build changes require touching many unrelated targets, it is often a symptom of architectural coupling and a predictor of cognitive overload.

2.3.11 Practical design techniques to lower cognitive load

The following techniques consistently reduce cognitive load in large systems:

- **Information hiding:** hide unstable decisions behind stable interfaces.
- **Strong types:** make illegal states unrepresentable (IDs, units, validated strings).
- **RAII everywhere:** make resource lifetimes deterministic and local.
- **One error model per layer:** consistent failure semantics.
- **Narrow interfaces:** one responsibility per abstraction.
- **Dependency discipline:** direction is enforced by the build system and module structure.
- **Predictable naming:** consistency beats creativity.
- **Minimize hidden state:** avoid global mutable singletons.

2.3.12 A cognitive-load checklist for design reviews

Use these questions in design reviews and code reviews:

- Can a new engineer understand the API contract from the signature alone?
- Is ownership explicit at all boundaries?
- Are failure modes explicit and consistent?
- Are invariants enforced by construction and encapsulation?
- Is thread-safety stated and enforced (or clearly delegated)?
- Can this component be tested in isolation without heavy scaffolding?
- Does changing internals require changing many unrelated call sites?
- Are dependencies minimal and directional?

If the answer is often “no,” the system is accumulating cognitive debt.

2.3.13 Summary

Cognitive load is the real scalability limit in large systems:

- Essential complexity cannot be removed, only represented cleanly.
- Accidental complexity is optional and must be fought continuously.
- Modern C++ reduces cognitive load when it is used to encode constraints in types, lifetimes, and interfaces.
- The primary design goal for large systems is **local reasoning**: enabling safe change without global understanding.

In a long-lived codebase, the best design is the one that allows competent engineers to make correct changes with bounded mental effort, even years after the original authors are gone.

2.4 Entropy and Software Decay

Complexity does not merely appear; it accumulates. Even competent teams discover that a once-clean system gradually becomes harder to change, slower to build, and more fragile under load. This phenomenon is often described informally as **entropy**: the natural drift from an orderly, comprehensible structure toward disorder and unpredictability. In software, entropy is not a physical law, but it behaves like one because of human, organizational, and economic forces. Without deliberate countermeasures, software tends to decay.
This section explains what software entropy means, why decay is the default outcome, what forms decay takes in real systems, and how Modern C++ design techniques can slow or reverse it.

2.4.1 What “entropy” means in software

In software engineering, **entropy** refers to the growth of disorder in a system’s structure and behavior:

- boundaries become porous,
- dependencies become tangled,
- invariants become implicit and inconsistent,
- local changes have global consequences,
- performance becomes unpredictable,
- confidence in change declines.

Entropy is a *systems property*. It is not "bad programmers" or "old code." It is what happens when:

- requirements evolve continuously,
- many people change the code over time,
- deadlines reward short-term solutions,
- correctness constraints are not enforced mechanically.

2.4.2 Why decay is the default outcome

Software decays because incentives often favor immediate delivery over structural integrity:

- shipping a feature is visible; preventing future complexity is invisible,
- shortcuts provide immediate benefit; the cost arrives later and is diffuse,
- new engineers optimize locally; only architects see system-level damage,
- as teams grow, coordination costs encourage "quick hacks" across boundaries.

When design constraints exist only as documentation or tribal knowledge, entropy accelerates because the codebase cannot defend itself. In long-lived systems, the absence of enforcement is equivalent to the absence of architecture.

2.4.3 The main mechanisms of software decay

Software decay typically manifests through a small number of recurring mechanisms.

1) Boundary erosion

Boundaries are rules about what knowledge is allowed where. Boundary erosion occurs when:

- internal details leak into public headers,
- modules start calling each other directly for convenience,
- “temporary” shortcuts become permanent dependencies,
- test code relies on internal implementation details.

In C++, boundary erosion is often visible through:

- expanding header include graphs,
- growing public APIs that expose internals,
- increasingly frequent rebuild cascades.

2) Coupling growth and dependency tangles

As dependencies spread, change becomes expensive:

- a small modification triggers many rebuilds,
- a simple refactor requires touching unrelated modules,

- two components cannot evolve independently.

A common pattern is the emergence of a few *central* modules that everything depends on. These become:

- change bottlenecks,
- merge-conflict hotspots,
- sources of global regressions.

3) Contract drift

A contract is the behavior a caller relies on: invariants, error semantics, performance expectations, thread-safety guarantees. Contract drift occurs when:

- the implementation changes but the contract is undocumented,
- different call sites assume different rules,
- error handling becomes inconsistent across layers,
- thread-safety is assumed rather than declared.

Over time, a system becomes "correct by accident"—and then stops being correct.

4) Uncontrolled shared state

Shared mutable state is a major entropy accelerator:

- hidden global state makes behavior context-dependent,
- concurrency introduces interleavings that are hard to reason about,
- unit testing becomes difficult because tests interfere with each other,
- debugging becomes probabilistic under load.

5) Local optimizations that create global incoherence

Engineers often optimize locally under pressure:

- duplicating logic rather than establishing a coherent policy,
- bypassing a layer to "fix it quickly,"
- introducing a special-case branch that becomes a permanent feature.

Each local decision may be rational. Their aggregate produces architectural disorder.

6) Toolchain and build friction

Build friction is both a symptom and a cause of decay:

- slow builds discourage refactoring,
- fear of breaking the build encourages minimal changes,
- templates in public headers amplify rebuild costs,
- missing modular boundaries promote "include everything" habits.

2.4.4 How to recognize entropy: practical signals

You can often detect entropy before it becomes catastrophic by monitoring signals that correlate with structural decay:

- **Change amplification:** small changes require editing many files.
- **Rising defect density:** bug fixes cause regressions in unrelated areas.
- **Slow onboarding:** new developers need long mentoring to be productive.

- **Fear-driven development:** teams avoid refactoring; everything is “too risky.”
- **Build-time growth:** compile and link times climb without proportional feature growth.
- **Interface instability:** frequent breaking API changes and ripple effects.
- **Inconsistent conventions:** multiple competing patterns for errors, logging, and ownership.

These are not cultural problems; they are design and governance problems that can be corrected.

2.4.5 Modern C++ as an entropy control tool

Modern C++ provides strong mechanisms to fight entropy, but only when used to encode design constraints.

1) Encode lifetime and ownership explicitly

Ambiguous lifetime accelerates decay because it forces developers to rely on tribal knowledge.

```
// Ownership transfer is explicit.
std::unique_ptr<Resource> acquire_resource();

// Non-owning, non-null access is explicit.
Resource& resource();

// Non-owning, nullable access is explicit.
Resource* try_resource() noexcept;
```

These signatures reduce the need for informal rules.

2) Prefer value semantics to reduce aliasing

Aliasing is a cognitive and correctness tax. Value semantics reduce hidden sharing:

```
struct Snapshot {
  std::vector<int> data;
};

// Copying Snapshot is a deliberate act; shared mutable state is avoided by design.
Snapshot read_snapshot();
```

3) Use vocabulary types for explicit uncertainty and failure

```
std::optional<Item> try_find_item(Id id);

// Alternative outcomes are explicit and must be handled.
using LoadResult = std::variant<Item, LoadError>;
LoadResult load_item(Id id);
```

This prevents contract drift where callers interpret failure differently.

4) Make constraints machine-checkable with concepts

```
template <typename T>
concept NonEmptyString = requires(const T& s) {
  { s.empty() } -> std::convertible_to<bool>;
  { s.size() }  -> std::convertible_to<std::size_t>;
};

template <NonEmptyString S>
void set_name(const S& name) {
  // design assumes "string-like"; constraint is visible
}
```

When constraints are explicit, they do not silently drift as code evolves.

5) Reduce dependency sprawl through boundary discipline

In C++, a powerful way to fight entropy is to reduce the knowledge crossing headers:

- keep headers narrow,
- forward-declare where appropriate,
- avoid leaking heavy template internals across boundaries,
- isolate unstable details behind opaque types when needed.

```
// Header-level boundary hygiene.
class Database; // forward declaration

class UserRepository {
public:
  explicit UserRepository(Database& db) : db_{db} {}
  void store(User user);

private:
  Database& db_; // non-owning; lifetime managed by composition root
};
```

2.4.6 Entropy is fought with *continuous design work*

The most dangerous myth is that architecture is a one-time activity. In reality:

- every feature changes the system,
- every change tests the integrity of boundaries,
- every shortcut becomes a precedent.

Therefore, entropy control requires:

- explicit design constraints (ownership, dependency direction, error models),
- enforcement mechanisms (types, build rules, code review checklists),
- continuous refactoring of accidental complexity,
- tests that protect invariants and contracts,
- measurement of build, runtime, and operational costs.

2.4.7 A practical anti-decay strategy for long-lived C++ systems

1) Establish invariants and protect them

Define invariants in code and validate them at boundaries:

- constructors reject invalid states,
- internal representation remains private,
- public functions preserve invariants.

2) Freeze interfaces, evolve implementations

Treat public interfaces as a strategic asset:

- keep them small,
- stabilize semantics and error behavior,
- evolve internals behind the boundary.

3) Restrict dependency direction

Make it easy to do the right thing and hard to do the wrong thing:

- separate “policy” from “mechanism”,
- prevent infrastructure from contaminating domain logic,
- enforce layering via build targets and include rules.

4) Avoid shared mutable global state

Prefer:

- value passing,
- immutability,
- explicit ownership graphs,
- message passing where appropriate.

5) Make refactoring normal

Entropy cannot be stopped only by rules; it must be reduced by regular cleanup:

- remove dead abstractions,
- merge duplicate policies,
- shrink headers and dependency surfaces,
- simplify error handling pathways,
- strengthen invariants where violations have been observed.

2.4.8 Summary

Software decay is the default outcome of long-lived development:

- entropy grows through boundary erosion, coupling, contract drift, shared state, and build sprawl,
- teams experience decay as rising cognitive load, fear of change, and instability,
- Modern C++ can fight entropy by making design constraints explicit and enforceable,
- the decisive factor is continuous design discipline, not one-time architecture work.

The goal is not to freeze the system. The goal is to keep the system *coherent* while it evolves. In long-lived systems, coherence is the most valuable form of stability.

2.5 Why "It Works" Is Not a Design Argument

In engineering discussions, one phrase ends design thinking faster than almost anything else: *"It works."* The statement is often factually true in the narrowest sense: the code compiles, tests pass, a demo succeeds, and the feature ships. But as an argument about design quality, "it works" is almost meaningless. It speaks only about *today's observed behavior under limited conditions*. Design is about *tomorrow*: change, scale, failures, constraints, and long-term cost.
This section explains why "it works" is insufficient, what design must prove instead, and how Modern C++ encourages a discipline where correctness, performance, and evolvability are demonstrated rather than assumed.

2.5.1 Working is a baseline, not a justification

Every correct system must work; therefore, "it works" cannot differentiate good design from bad design. Many designs "work":

- designs that are dangerously coupled,
- designs that leak resources slowly,
- designs that fail under concurrency,
- designs that have undefined behavior that does not manifest in tests,
- designs that collapse when requirements evolve,
- designs that break when moved to a different compiler or platform.

A design argument must therefore address properties that are not captured by immediate functionality.

2.5.2 Design is about properties beyond functionality

A robust design can be evaluated by engineering properties that remain meaningful under change:

- **Correctness under stress:** behavior under invalid input, partial failures, and boundary conditions.
- **Stability under evolution:** ability to add features without rewiring unrelated code.
- **Local reasoning:** ability to change a component without global knowledge.
- **Explicit contracts:** preconditions, invariants, error behavior, thread-safety guarantees.
- **Performance predictability:** stable latency and memory usage under production load.
- **Operational clarity:** observability, debuggability, and failure diagnosis.
- **Security posture:** resistance to misuse, abuse, and unexpected inputs.

Design quality is measured by these properties over time. “It works” does not mention any of them.

2.5.3 Why “it works” persists: local success hides global cost

The phrase persists because local success is visible:

- a feature demo is immediate,
- functional tests can pass,
- performance can look acceptable at small scale,
- undefined behavior can remain silent,
- coupling can remain hidden while the system is small.

But the true cost appears later:

- when the next feature touches the same code,
- when the team grows and knowledge becomes distributed,
- when concurrency increases under load,
- when deployment conditions differ from test environments,
- when the system must be maintained by people who did not write it.

In other words, “it works” often means “we have not paid the bill yet.”

2.5.4 C++ makes the gap between “works” and “is designed” painfully real

Modern C++ exposes the gap between working behavior and sound design because:

- ownership and lifetime must be correct, not merely plausible,
- data races are undefined behavior, not “rare bugs”,

- ABI boundaries punish unstable interfaces,
- cost models are visible: allocations, copies, contention,
- the compilation model exposes coupling through build time and dependencies.

A design that "works" in a demo can still be a design failure in C++:

- a dangling pointer may not crash under tests,
- a data race may not manifest until production load,
- a performance regression may appear only with realistic data volume.

Therefore, Modern C++ engineering requires proof-oriented design arguments.

2.5.5 What a design argument should sound like

A design argument must specify the constraint being satisfied and the mechanism that enforces it. Instead of:

> "It works."

A real design argument sounds like:

- **Ownership is explicit:** "the API returns `std::unique_ptr` to transfer responsibility; no raw owning pointers."
- **Invariants are protected:** "invalid states are rejected at construction; representation remains private."
- **Failure is explicit:** "this operation returns an explicit error alternative; callers cannot ignore it by accident."

- **Concurrency model is defined:** “components are either thread-confined or internally synchronized, and that is documented in the interface.”
- **Dependencies are directional:** “policy depends on interfaces, not on concrete mechanisms; build targets enforce this.”
- **Costs are controlled:** “hot paths allocate zero memory; allocations are constrained to setup phases.”

These statements can be tested, reviewed, and validated.

2.5.6 Example 1: “It works” but the ownership contract is undefined

```
// "Works" today, but the design is ambiguous.
// Who owns the returned pointer? Who deletes it? Is it cached?
Widget* get_widget(std::string_view id);
```

This interface increases complexity because every caller must guess the ownership model. The design fix is to encode the decision:

```
// Ownership transfer: caller is responsible.
std::unique_ptr<Widget> make_widget(std::string_view id);

// Non-owning, non-null: lifetime guaranteed elsewhere.
Widget& widget(std::string_view id);

// Non-owning, nullable: absence is explicit.
Widget* try_widget(std::string_view id) noexcept;
```

Now the system does not rely on implicit knowledge. A reviewer can evaluate the design.

2.5.7 Example 2: “It works” but concurrency is undefined

```
// "Works" in tests, fails under load.
// Nothing states whether concurrent calls are allowed.
class Cache {
public:
  void put(std::string key, std::string value);
  std::string get(std::string_view key);
};
```

A design that scales must make thread-safety a contract. Two valid designs exist; each implies different constraints:

```
// Design A: externally synchronized (caller provides locking).
class CacheExternallySynchronized {
public:
  std::optional<std::string> get(std::string_view key) noexcept;
  void put(std::string key, std::string value);
};

// Design B: internally synchronized (thread-safe for concurrent calls).
class CacheInternallySynchronized {
public:
  std::optional<std::string> get(std::string_view key) const;
  void put(std::string key, std::string value);

private:
  mutable std::mutex m_;
};
```

“It works” does not choose a concurrency model. Design must.

2.5.8 Example 3: “It works” but error semantics are incoherent

```
// Ambiguous: returns empty string on failure OR on legitimate empty content.
std::string read_file(std::string_view path);
```

The fix is to make failure explicit in the type:

```
struct IoError {
  int code{};
  std::string message;
};

// Explicit: either content or error.
std::variant<std::string, IoError> read_file(std::string_view path);
```

Now a caller cannot accidentally treat failure as valid data. The design contract is obvious.

2.5.9 Design requires evidence: tests, measurement, and enforcement

A design argument is credible only when it is supported by evidence:

- **Unit and integration tests** validate functional correctness and invariants.
- **Property tests and fuzzing** explore edge cases and invalid inputs.
- **Static analysis and sanitizers** detect lifetime errors, undefined behavior, and races.
- **Benchmarks** validate performance constraints and regression budgets.
- **Build boundaries** enforce modularity and dependency direction.

In mature engineering cultures, design claims are not believed because they sound confident. They are believed because the system enforces them mechanically.

2.5.10 A checklist: replacing “it works” with design-level questions

When someone says “it works,” respond with design questions:

- What are the ownership and lifetime guarantees?
- What is the error model and what remains valid after failure?
- What are the invariants, and how are they enforced?
- What is the concurrency contract? What is safe to call concurrently?
- What are the dependency rules? What does this component depend on?
- What are the performance budgets? Where are allocations allowed?
- How will this evolve without breaking clients?
- What evidence validates these claims (tests, analysis, measurements)?

If these questions are unanswered, the design is not done—even if the code runs.

2.5.11 Summary

- “It works” is a statement about observed behavior, not design quality.
- Design quality is about long-term properties: evolvability, local reasoning, explicit contracts, performance predictability, and correctness under stress.
- C++ exposes bad design quickly because lifetime, concurrency, ABI, and cost constraints are real and unforgiving.
- A valid design argument names constraints, encodes them in interfaces and types, and supports them with evidence.

In long-lived systems, the correct engineering response to “it works” is:

> “What does it guarantee, under what constraints, and how do we know?”

The Philosophy of Modern C++ Design

3.1 Zero-Cost Abstractions

A defining idea behind Modern C++ design is the principle of **zero-cost abstractions**. The phrase is often repeated and frequently misunderstood. It does *not* mean "abstractions are free" in an absolute sense, and it does not mean C++ magically removes all overhead. It means something more precise and more demanding:

> **You should be able to write high-level, expressive abstractions in C++ such that, when you use them correctly, you do not pay for what you do not use, and you do not pay extra compared to a reasonable low-level implementation of the same semantics.**

This section clarifies the concept, distinguishes it from naive performance claims, and shows how Modern C++ enables abstractions that are both *expressive* and *predictable in cost*, which is essential for long-lived systems.

3.1.1 What "zero-cost" means (and what it does not mean)

The positive definition. A C++ abstraction is **zero-cost** when the compiler can:

- eliminate the abstraction boundary (inline, devirtualize, constant-fold),

- reduce runtime indirection and allocations to what is semantically necessary,
- specialize code to the actual types and values used,
- generate machine code comparable to hand-written low-level code.

The two core promises. In practice, the principle is captured by two promises:

- **Pay only for what you use:** unused features do not impose overhead.
- **No hidden tax for expressiveness:** using a well-designed abstraction does not add overhead beyond the inherent cost of the semantics you requested.

What it does *not* mean.

- It does not guarantee that every abstraction is free; some semantics require real work.
- It does not remove algorithmic complexity; an $O(n^2)$ algorithm remains $O(n^2)$.
- It does not eliminate I/O, synchronization, cache misses, or memory bandwidth limits.
- It does not excuse poor design: a wasteful architecture remains wasteful even in C++.

Zero-cost is therefore a *design discipline* supported by the language and compiler model.

3.1.2 Why zero-cost abstractions matter for long-lived systems

Long-lived systems must continuously balance two pressures:

- **Rising complexity:** more features, more teams, more environments.
- **Hard constraints:** latency budgets, memory limits, throughput targets, power constraints, real-time requirements.

Abstractions are necessary to control complexity, but abstractions can also hide costs. In many ecosystems, abstraction layers often impose unavoidable runtime overhead (late binding, dynamic dispatch, runtime reflection, GC pressure). In Modern C++, well-chosen abstractions can remain:

- **readable** (clear intent),
- **maintainable** (local reasoning),
- **measurable** (predictable costs),
- **optimizable** (compilers can remove overhead).

This is a major reason C++ remains dominant in domains where both *software scale* and *performance scale* must coexist.

3.1.3 The enabling mechanisms in Modern C++

Zero-cost abstractions are enabled by a set of language and compilation properties:

1) Static polymorphism (templates) and specialization

Templates allow the compiler to generate code specialized to concrete types, enabling inlining and elimination of indirection:

```
template <typename T>
T clamp(T v, T lo, T hi) {
  if (v < lo) return lo;
  if (v > hi) return hi;
  return v;
}

// The compiler generates a specialized version for each used T.
// No virtual dispatch is required; calls are eligible for inlining.
```

2) Inlining and optimization across boundaries

Modern C++ design often prefers small, composable functions. When the compiler can see the definition, it can inline and optimize away layers:

```
inline int twice(int x) noexcept { return x + x; }

int f(int a) {
  return twice(a) + twice(a);
}
// At optimization, this is typically reduced to arithmetic with no call overhead.
```

3) RAII: deterministic resource management without runtime frameworks

RAII is a design pattern that becomes a language-strength in C++:

- it expresses ownership explicitly,
- it makes cleanup automatic and local,
- it does not require GC or runtime reference counting unless you choose it.

```
class File {
public:
  explicit File(const char* path);
  ~File();                    // closes handle deterministically
  File(const File&) = delete;
  File& operator=(const File&) = delete;
  File(File&&) noexcept;
  File& operator=(File&&) noexcept;
};
```

The abstraction is high-level (“a File”), but the cost is exactly the inherent cost of opening/closing a file.

4) Value categories and move semantics

Move semantics reduce accidental copies and allow efficient composition of APIs:

```
std::string make_message() {
  std::string s = "hello";
  s += " world";
  return s; // moved or elided; avoids deep copies in optimized builds
}
```

This supports designs that are both clean and efficient: return-by-value becomes an idiomatic, low-cost abstraction.

5) `constexpr` and compile-time computation

When logic can be evaluated at compile time, runtime cost disappears:

```
constexpr int fib(int n) {
  return (n <= 1) ? n : fib(n - 1) + fib(n - 2);
}

constexpr int v = fib(10); // computed at compile time
```

6) Concepts: constraints that improve correctness without runtime checks

Concepts constrain templates at compile time, reducing misuse and improving diagnostics without runtime overhead:

```
#include <concepts>

template <typename T>
concept Addable = requires(T a, T b) {
  { a + b } -> std::same_as<T>;
};
```

```
template <Addable T>
T add(T a, T b) { return a + b; }
```

3.1.4 A cost model vocabulary: where overhead hides

To design zero-cost abstractions, you must recognize where costs enter the system. Typical cost sources include:

- **Dynamic dispatch:** virtual calls and interface-based designs.
- **Heap allocation:** hidden allocations in containers, type erasure, or "convenience" APIs.
- **Indirection and pointer chasing:** cache misses and branch misprediction.
- **Synchronization:** locks, atomics, contention and wakeups.
- **Conversions and copies:** unintended deep copies, temporary objects.
- **Exception paths:** not the cost of *having* exceptions, but the cost of using them for control flow or mixing models inconsistently.

A zero-cost abstraction does not eliminate these costs universally; it prevents *unintentional* costs and ensures that intentional costs are explicit and justified.

3.1.5 Examples of zero-cost design in practice

Example 1: `std::string_view` as a non-owning view

Non-owning views allow APIs to accept text without forcing allocation/copying:

```
#include <string_view>
```

```
bool starts_with(std::string_view s, std::string_view prefix) noexcept {
  return s.size() >= prefix.size() &&
         s.substr(0, prefix.size()) == prefix;
}
```

This design can be efficient because it:

- avoids allocation,
- avoids copying,
- makes lifetime rules explicit (a view does not extend lifetime).

Example 2: `std::span` as a view over contiguous memory

```
#include <span>

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

This supports a clean interface that can accept arrays, vectors, and other contiguous buffers without copying.

Example 3: Static polymorphism instead of virtual dispatch

```
template <typename Strategy>
int compute(int x, Strategy s) {
  return s(x);
}

struct Square {
```

```
  int operator()(int x) const noexcept { return x * x; }
};
```

When `Strategy` is known, the call can be inlined, producing code equivalent to calling a plain function.

3.1.6 When abstractions are *not* zero-cost

C++ allows zero-cost abstractions, but it also allows abstractions with real overhead. Recognizing them is part of professional design.

Type erasure and indirect calls

Type erasure (e.g., `std::function`) buys flexibility at a cost:

- possible heap allocation,
- indirect call (cannot always inline),
- larger object size and potential cache effects.

```
#include <functional>

std::function<int(int)> f = [](int x) { return x + 1; };
// Useful for callbacks and plugin boundaries, but not free.
```

The correct design stance is not “avoid it,” but:

> Use type erasure when the flexibility is essential, and keep it away from hot paths unless the cost is acceptable.

Shared ownership and reference counting

`std::shared_ptr` provides shared lifetime management, but reference counting has costs:

- atomic increments/decrements in many implementations,
- contention under heavy sharing,
- complex ownership graphs that increase cognitive load.

```
#include <memory>

std::shared_ptr<Node> a = std::make_shared<Node>();
std::shared_ptr<Node> b = a; // refcount operations (real work)
```

Again, the abstraction is correct when shared ownership is essential; it is accidental complexity when used to avoid designing lifetimes.

Over-abstraction and “patterns as decoration”

Some abstractions add cognitive cost and runtime cost without reducing essential complexity:

- deep inheritance hierarchies,
- factories of factories without a real constraint (ABI/plugins/testing seam),
- unnecessary indirection layers that do not improve replaceability or clarity.

In C++, such patterns are doubly harmful: they obscure intent and often introduce measurable overhead.

3.1.7 Zero-cost as a design method, not a slogan

To apply the principle effectively, treat it as a decision framework:

1) State the semantic intent

- What does the abstraction represent?
- What invariants does it enforce?
- What failure modes exist?
- What concurrency rules apply?

2) State the cost intent

- Are allocations allowed here?
- Is dynamic dispatch allowed?
- Is locking allowed?
- Is copying allowed?
- What is the performance budget?

3) Encode intent into interfaces

- ownership via `unique_ptr`/references/views,
- failure via `optional`/`variant`/expected-like results,
- constraints via concepts and types,
- cost hints via value semantics and move-only types.

4) Validate with measurement

Zero-cost is ultimately empirical:

- benchmarks for hot paths,
- profiling for allocation and contention,
- inspection and analysis when appropriate.

3.1.8 A pragmatic guideline set for Modern C++ design

The following guidelines capture the practical essence of zero-cost abstraction design in long-lived systems:

- Prefer **value types** for data transfer; introduce identity and sharing deliberately.
- Prefer **static polymorphism** in hot paths; use virtual dispatch and type erasure at boundaries where flexibility is essential.
- Make **allocations explicit**; avoid hidden heap usage in core loops.
- Use **views** (`string_view`, `span`) to avoid copying, but document lifetime expectations.
- Use **RAII** to make resource management deterministic and local.
- Use **concepts** to express requirements without runtime checks.
- Keep interfaces **narrow** so compilers can optimize and humans can reason locally.

3.1.9 Summary

Zero-cost abstractions are a foundational philosophy of Modern C++ design:

- They enable expressive code without forcing unnecessary runtime overhead.
- They depend on the compiler's ability to specialize, inline, and optimize away abstraction layers.
- They require explicit design decisions about ownership, lifetime, constraints, and cost models.
- They do not eliminate inherent costs; they eliminate accidental costs and hidden taxes.

When practiced correctly, zero-cost abstraction design is not about writing "faster code." It is about building systems that remain *clean, scalable, and predictable* as they grow in size, team count, and lifetime.

3.2 Pay-for-What-You-Use Principle

Modern C++ design is guided by a cost discipline that is both technical and architectural: the **pay-for-what-you-use** principle. This principle is often mentioned alongside "zero-cost abstractions," but it deserves its own treatment because it is not merely about micro-optimizations. It is about *cost transparency and cost containment* at system scale. The principle can be stated precisely:

> **A C++ program should not incur runtime or memory costs for features, flexibility, or generality that it does not use. If you choose additional semantics (dynamic dispatch, shared ownership, synchronization, heap allocation), you should pay explicitly for those semantics—and be able to justify that cost.**

This section explains the design meaning of pay-for-what-you-use, the types of costs it governs, the mechanisms C++ provides to realize it, and the failure modes that occur when teams violate the principle.

3.2.1 Why the principle exists: cost is a design constraint

In long-lived systems, costs are not abstract:

- embedded and real-time systems have strict latency and memory limits,
- servers have throughput and tail-latency budgets,
- desktop and mobile systems have power and responsiveness constraints,
- safety-critical systems require predictable behavior under worst-case conditions.

A design that hides costs becomes unstable over time:

- hot paths acquire accidental allocations,
- abstractions accumulate indirection,
- shared ownership spreads without a lifetime model,
- concurrency becomes contention-heavy,
- unpredictable latency makes SLOs impossible to meet.

Pay-for-what-you-use is therefore a rule of *architectural hygiene*: the system should not pay hidden taxes for convenience or unused generality.

3.2.2 The three main cost dimensions

In practice, the principle governs three main cost dimensions.

Runtime cost. Time spent at runtime:

- CPU cycles (extra branches, virtual calls, indirection),
- synchronization and contention costs,
- allocator overhead,
- cache misses and memory stalls.

Memory cost. Footprint and allocation behavior:

- object size inflation due to vtables or type erasure,
- heap fragmentation and allocator metadata,
- reference-counting control blocks,
- redundant buffering and copying.

Build and complexity cost. Costs that affect the engineering system:

- compile time and rebuild cascades,
- binary size growth,
- link-time overhead,
- cognitive load from over-generalized abstractions.

A design that respects pay-for-what-you-use aims to keep these costs *proportional* to actual requirements.

3.2.3 How Modern C++ enables pay-for-what-you-use

1) Static polymorphism vs dynamic polymorphism

Dynamic polymorphism (virtual functions) provides runtime substitution but costs:

- an indirection at call sites,
- reduced inlining opportunities,
- object size increase (vptr) and vtable structures,
- ABI coupling across boundaries.

Static polymorphism (templates) provides substitution at compile time:

- no virtual dispatch,
- calls are eligible for inlining,
- the compiler specializes code to the used types.

```
// Static polymorphism: pay only when you instantiate with a concrete type.
template <typename Strategy>
int compute(int x, Strategy s) {
  return s(x); // eligible for inlining
}

struct Square {
  int operator()(int v) const noexcept { return v * v; }
};
```

This does not mean “never use virtual functions.” It means:

> Use dynamic polymorphism at boundaries where runtime substitution is essential; use static polymorphism where performance and optimization matter and the set of types is known at compile time.

2) Value semantics, move semantics, and copy elision

Returning values is often efficient in Modern C++ because moves and copy elision avoid unnecessary copying:

```
std::vector<int> build(std::size_t n) {
  std::vector<int> v;
  v.reserve(n);
  for (std::size_t i = 0; i < n; ++i) v.push_back(static_cast<int>(i));
  return v; // moved or elided; avoids deep copy
}
```

Design impact:

- interfaces can be clean and composable without forcing heap allocation,
- cost remains proportional to the actual data moved.

3) RAII and deterministic resource management

RAII ensures that resources are acquired and released with predictable scope-based cost, without runtime frameworks:

```
class LockGuard {
public:
  explicit LockGuard(std::mutex& m) : m_{m} { m_.lock(); }
  ~LockGuard() { m_.unlock(); }
  LockGuard(const LockGuard&) = delete;
  LockGuard& operator=(const LockGuard&) = delete;
private:
  std::mutex& m_;
};
```

The abstraction is expressive, but its cost is exactly the cost of locking and unlocking—nothing more, nothing hidden.

4) Views instead of copies: `std::span` and `std::string_view`

Views allow functions to accept data without copying:

- you pay for iteration and access,
- you do not pay for ownership transfer, allocation, or deep copy.

```
#include <span>
#include <string_view>

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

bool has_prefix(std::string_view s, std::string_view p) noexcept {
  return s.size() >= p.size() && s.substr(0, p.size()) == p;
}
```

The design obligation is to manage lifetime explicitly: a view does not extend lifetime.

5) Compile-time computation: `constexpr` and `consteval`

When a value is known at compile time, the runtime cost can be eliminated:

```
consteval std::uint32_t fnv1a(std::string_view s) {
  std::uint32_t h = 2166136261u;
  for (unsigned char c : s) {
    h ^= c;
    h *= 16777619u;
  }
  return h;
```

```
}

constexpr auto id = fnv1a("UserCreated");
```

The system pays at compile time for computing the constant, not at runtime for recomputing it.

6) Concepts: constraints without runtime checks

Concepts make template requirements explicit and reject invalid uses at compile time rather than adding runtime checks:

```
#include <concepts>

template <typename T>
concept Incrementable = requires(T x) {
  { ++x } -> std::same_as<T&>;
};

template <Incrementable T>
void step(T& x) { ++x; }
```

This improves correctness and reduces the need for defensive runtime logic in generic code.

3.2.4 Common violations: hidden costs disguised as convenience

Violations of pay-for-what-you-use often appear as “simple” convenience choices that impose systemic tax.

Hidden heap allocation via type erasure

```
#include <functional>

// Useful, but may allocate and usually prevents inlining.
std::function<int(int)> f = [](int x) { return x + 1; };
```

Design guidance:

- Use type erasure where the flexibility is essential (callbacks, plugin systems).
- Avoid it in hot loops unless measured and justified.

Shared ownership as a substitute for lifetime design

```
#include <memory>

auto p = std::make_shared<Node>();
auto q = p; // reference count operations; may be atomic
```

Shared ownership is powerful when the system truly has shared lifetime. It becomes accidental complexity when used to avoid making ownership decisions.

Unnecessary dynamic dispatch

Designs built around deep virtual hierarchies often pay ongoing runtime cost and reduce optimization opportunities. Virtual dispatch is justified when:

- runtime substitution is required,
- ABI boundaries require a stable interface,
- plugin architectures need late binding.

Otherwise, static polymorphism or direct composition often yields simpler, faster, more analyzable code.

Allocation-heavy interfaces in core paths

A common high-level mistake is to design APIs that force allocation even when the caller could reuse buffers or provide storage:

```
// High hidden cost: always allocates and copies.
std::string format_message(int code);
```

Alternative designs can make costs explicit:

```
#include <span>
#include <string_view>

// Caller provides storage: allocation becomes optional.
std::size_t format_message_into(int code, std::span<char> out);

// Or return a view when formatting is not required.
std::string_view message_text(int code) noexcept;
```

3.2.5 Pay-for-what-you-use at the architecture level

The principle is not limited to low-level code. It scales to architecture decisions:

Feature containment

If a system has optional features (telemetry, plugins, advanced parsing), a pay-for-what-you-use architecture:

- compiles them out when unused (build-time selection),
- isolates their dependencies,
- prevents them from inflating core runtime cost.

Dependency containment

A design that depends on large frameworks globally pays:

- build time,

- binary size,
- cognitive load,
- and often runtime initialization overhead.

Pay-for-what-you-use architecture keeps the core minimal and pushes heavy dependencies to the edges.

3.2.6 A practical discipline: making cost part of design review

To operationalize pay-for-what-you-use, treat cost as part of the design contract:

- Where are allocations allowed?
- Is dynamic dispatch allowed here?
- Are exceptions part of the error contract or reserved for exceptional paths?
- Is locking allowed on this path?
- Does this API force copies, or can it accept views/spans?
- What is the expected complexity and data volume?

In Modern C++ teams, these questions are not optimization trivia. They are architecture-level constraints that protect the system from cost creep.

3.2.7 Summary

The pay-for-what-you-use principle is a cost discipline that defines Modern C++ design philosophy:

- You should not incur cost for unused features or unnecessary generality.

- C++ enables this through static polymorphism, inlining, RAII, value semantics, views, and compile-time computation.
- Violations often appear as convenient abstractions that hide allocation, indirection, synchronization, or build-time coupling.
- At system scale, respecting the principle requires cost-aware interfaces, feature containment, and dependency discipline.

In long-lived systems, this principle is not about writing clever code. It is about preventing cost and complexity from accumulating silently until the system becomes unmanageable.

3.3 Determinism, Explicitness, and Control

Modern C++ design is built on a distinctive engineering philosophy: **systems should behave predictably, communicate intent explicitly, and keep control of costs and lifetimes in the hands of the designer**. These three pillars—**determinism**, **explicitness**, and **control**—form a coherent worldview. They are not "style preferences"; they are engineering strategies for building long-lived systems where correctness, performance, reliability, and evolvability matter under real constraints.
This section defines each pillar precisely, explains why they reinforce each other, and shows how Modern C++ expresses them in types, interfaces, and runtime behavior.

3.3.1 Determinism: predictable behavior under defined contracts

What determinism means in software systems. Determinism is the property that a system's behavior is predictable and repeatable given the same inputs and environment. In practice, determinism is not absolute; it is a spectrum. What matters is that the system has:

- **defined semantics** for normal operation,

- **defined failure modes** for invalid inputs and exceptional situations,
- **bounded variability** in timing and resource usage where required.

Why determinism is a design goal. Long-lived systems are engineered, tested, deployed, and maintained by humans. Determinism reduces:

- heisenbugs (bugs that disappear under observation),
- non-reproducible production failures,
- debugging time and operational uncertainty,
- safety risks in real-time and critical systems.

Determinism in Modern C++: the deterministic baseline. C++ provides a deterministic baseline in key areas:

- object lifetime tied to scope (RAII),
- deterministic destruction order within scope rules,
- explicit control of allocation and deallocation,
- predictable value semantics and move semantics.

```
class ScopeTimer {
public:
  explicit ScopeTimer(std::string_view name)
    : name_{name}, start_{std::chrono::steady_clock::now()} {}

  ~ScopeTimer() {
    auto end = std::chrono::steady_clock::now();
    auto us = std::chrono::duration_cast<std::chrono::microseconds>(end - start_).count();
```

```
    std::cout << name_ << ": " << us << "us\n";
  }

private:
  std::string_view name_;
  std::chrono::steady_clock::time_point start_;
};

void f() {
  ScopeTimer t{"f"};
  // ... work ...
} // deterministic end-of-scope action
```

The critical point is not timing; it is the *deterministic execution of cleanup and finalization*.

3.3.2 Explicitness: design intent must be visible and checkable

Explicitness as a complexity-control strategy. In large systems, implicit assumptions become the primary source of bugs and decay. Explicitness means:

- contracts are visible in interfaces,
- invariants are enforced by construction,
- ownership and lifetime are encoded in types,
- failure and absence are represented explicitly,
- concurrency rules are declared, not assumed.

Explicitness reduces cognitive load because engineers can infer behavior from code structure rather than tribal knowledge.

Ownership and lifetime explicitness

```
// Ownership transfer is explicit.
std::unique_ptr<Connection> open_connection(Config cfg);

// Non-owning, non-null access is explicit.
Connection& connection();

// Non-owning, nullable access is explicit.
Connection* try_connection() noexcept;
```

A design that returns raw pointers everywhere forces readers to guess. A design that uses ownership vocabulary types makes the contract self-documenting and mechanically enforceable.

Failure explicitness

```
struct ParseError {
  std::string message;
  std::size_t position{};
};

using ParseResult = std::variant<Value, ParseError>;

ParseResult parse_value(std::string_view text);
```

This makes a design decision explicit: parsing can fail, and callers must handle it.

Constraint explicitness (concepts)

```
#include <concepts>

template <typename T>
concept HashableKey = requires(T k) {
  { std::hash<T>{}(k) } -> std::convertible_to<std::size_t>;
```

```
  { k == k } -> std::convertible_to<bool>;
};

template <HashableKey Key, typename Value>
class Cache { /* ... */ };
```

The constraint becomes part of the type contract, not a comment.

3.3.3 Control: intentional tradeoffs and cost visibility

Control is not "manual management." A common misunderstanding is that "control" means writing low-level code everywhere. In Modern C++, control means:

- the ability to choose the right abstraction level,
- the ability to choose cost models intentionally,
- the ability to localize complex mechanisms behind stable boundaries,
- the ability to achieve predictable performance and memory behavior.

Control is architectural: it is the ability to *enforce* your decisions and prevent accidental costs from spreading.

Control over allocations and data movement

```
std::vector<int> build(std::size_t n) {
  std::vector<int> v;
  v.reserve(n); // cost decision: avoid reallocations
  for (std::size_t i = 0; i < n; ++i) v.push_back(static_cast<int>(i));
  return v; // moved or elided; predictable ownership transfer
}
```

Control means the designer can decide where allocations occur and avoid accidental allocations in hot paths.

Control over indirection and dispatch

```
// Static polymorphism: no virtual dispatch in the hot path.
template <typename Strategy>
int compute(int x, Strategy s) {
  return s(x);
}

struct FastStrategy {
  int operator()(int x) const noexcept { return x * 2; }
};
```

Dynamic dispatch is still valuable at system boundaries, but control means you decide where it is used and where it is not.

Control over resource lifetimes

RAII is the flagship mechanism for control: resources are tied to scope and composition, not to garbage collection heuristics or external frameworks.

```
class Buffer {
public:
  explicit Buffer(std::size_t n) : data_(n) {}

  std::span<std::byte> bytes() noexcept {
    return std::span<std::byte>{data_.data(), data_.size()};
  }

private:
  std::vector<std::byte> data_;
};
```

Here, lifetime is deterministic, costs are explicit, and usage does not require global runtime policies.

3.3.4 How the three pillars reinforce each other

These pillars are not independent; they amplify each other:

- **Explicitness enables determinism:** when ownership, invariants, and failure are explicit, behavior is predictable.
- **Determinism enables control:** predictable lifetimes and costs allow the system to meet performance and reliability budgets.
- **Control enables explicitness:** when the language allows precise expression of cost and lifetime, interfaces can state real contracts, not vague wishes.

This is why Modern C++ is well-suited for systems where correctness and efficiency must coexist.

3.3.5 What happens when these pillars are violated

Violations of determinism, explicitness, and control generate recognizable pathologies:

When determinism is lost

- non-reproducible failures under load,
- unpredictable tail latency,
- debugging that depends on timing and observation,
- subtle state corruption due to undefined behavior.

When explicitness is lost

- ownership becomes tribal knowledge,
- invariants drift across modules,
- failure conventions become inconsistent,
- concurrency rules are assumed and violated.

When control is lost

- hidden allocations appear in hot paths,
- abstraction layers impose accidental overhead,
- performance regressions are hard to diagnose,
- dependency tangles make refactoring unsafe.

These are not merely code issues; they are design failures that compound over time.

3.3.6 Practical design techniques that embody the philosophy

The following techniques operationalize the three pillars in Modern C++ systems:

- **Prefer value semantics** for data transfer; introduce sharing deliberately.
- **Use RAII pervasively** to make lifetimes deterministic.
- **Encode ownership in APIs** using references, `unique_ptr`, views, and handles.
- **Make failure explicit** using `optional`, `variant`, or expected-like results where appropriate.

- **Declare concurrency contracts** (thread-confined vs synchronized vs immutable sharing).
- **Control dependency direction** using narrow headers, build targets, and stable interfaces.
- **Measure costs** and enforce budgets (allocations, latency, contention) as design constraints.

3.3.7 Summary

Determinism, explicitness, and control are the philosophical foundation of Modern C++ design:

- **Determinism** gives predictable behavior and reproducibility.
- **Explicitness** makes contracts visible, checkable, and resistant to drift.
- **Control** makes cost models intentional and prevents hidden taxes.

In long-lived systems, these pillars transform "code that runs" into software that remains comprehensible, stable, performant, and safe under continuous change.

3.4 RAII as a Design Philosophy

RAII (Resource Acquisition Is Initialization) is often taught as a C++ idiom for avoiding memory leaks. That is correct but far too narrow. In Modern C++, RAII is a **design philosophy** that shapes the architecture of long-lived systems: it turns resource management from a scattered runtime concern into a *deterministic, local, and enforceable contract*. RAII is not primarily about memory; it is about **ownership, lifetime, invariants, and failure safety** for *all* resources. This section treats RAII as a first-class design principle, explains its system-scale benefits, and shows how it drives explicitness and reliability in Modern C++ architectures.

3.4.1 What counts as a “resource”

In RAII, a resource is anything that must be acquired and later released, and where failure to release causes harm. Common categories include:

- **Memory:** heap allocations, buffers, arenas, memory-mapped regions.
- **OS handles:** files, sockets, pipes, timers, processes, threads.
- **Locks and synchronization:** mutexes, read-write locks, semaphores, condition variables.
- **Transactions and logical resources:** database transactions, undo scopes, temporary invariants.
- **External device resources:** GPU contexts, camera handles, driver resources.
- **Protocol state:** acquired leases, reservations, subscriptions, registrations.

A design that manages these resources manually with ad-hoc conventions will eventually decay. RAII creates a uniform mechanism: **tie acquisition to object construction and release to object destruction**.

3.4.2 RAII: the deterministic contract

RAII makes a strong promise:

> If an object exists, it owns a valid resource (or represents a well-defined empty state). When the object is destroyed, the resource is released exactly once.

This promise turns resource management into a local property:

- the scope defines lifetime,

- destruction is deterministic,
- cleanup occurs even when control flow is interrupted.

In long-lived systems, local reasoning is the highest form of scalability: engineers can understand resource ownership without tracing every possible exit path.

3.4.3 Why RAII is a design philosophy, not a cleanup trick

RAII is philosophical because it changes how you design APIs and components:

- **Ownership becomes explicit:** a type represents responsibility.
- **Error handling becomes consistent:** failure to acquire prevents construction.
- **Invariants become stable:** constructed objects satisfy invariants by design.
- **Interfaces become safer:** fewer "must call cleanup" rules exist.
- **Exception safety becomes systematic:** cleanup is automatic.

A system designed around RAII is not one where "we remember to free resources," but one where *forgetting is structurally difficult*.

3.4.4 RAII and exception safety: correctness under non-linear control flow

Regardless of whether your error model is exceptions or explicit returns, non-linear control flow exists:

- early returns,
- multiple exit points,
- errors returned from deeper calls,

- exceptions (if used),
- cancellation and timeouts in concurrent systems.

Manual cleanup across these paths is fragile. RAII solves this by binding cleanup to destruction:

```
class File {
public:
  explicit File(const char* path); // acquires OS handle
  ~File();                         // releases OS handle

  File(const File&) = delete;
  File& operator=(const File&) = delete;

  File(File&&) noexcept;
  File& operator=(File&&) noexcept;

  void write(std::string_view s);

private:
  int fd_{-1};
};

void log_message(const char* path, std::string_view msg) {
  File f{path};       // if acquisition fails, construction fails
  f.write(msg);       // any early return or exception still triggers destructor
}                      // deterministic release
```

This is not an implementation detail. It is a design decision that makes the system robust against control-flow complexity.

3.4.5 RAII and invariants: “valid by construction”

A central theme of Modern C++ design is that objects should maintain invariants:

- representation is private,
- constructors establish a valid state,
- operations preserve validity,
- invalid states are rejected early.

RAII supports this because acquisition and initialization happen together.

```
class LockedCounter {
public:
  explicit LockedCounter(std::mutex& m, std::int64_t& v)
    : lock_{m}, value_{v} {} // invariant: lock is held while object exists

  void increment() noexcept { ++value_; }

private:
  std::lock_guard<std::mutex> lock_;
  std::int64_t& value_;
};
```

Here, the invariant is structural: as long as a `LockedCounter` exists, the mutex is held. The lock cannot be forgotten.

3.4.6 RAII and ownership models: move-only design

Modern C++ makes RAII powerful through move semantics and clear ownership types:

- **unique ownership** (move-only, single responsible owner),
- **shared ownership** (reference-counted, used when shared lifetime is essential),
- **non-owning views** (references, pointers, `span`, `string_view`).

RAII is most effective when ownership is explicit. Unique ownership is the default for resources that must be released exactly once:

```
class Socket {
public:
  Socket();           // open
  ~Socket();          // close
  Socket(const Socket&) = delete;
  Socket& operator=(const Socket&) = delete;
  Socket(Socket&&) noexcept;
  Socket& operator=(Socket&&) noexcept;

  void send(std::span<const std::byte> bytes);

private:
  int fd_{-1};
};
```

This design communicates:

- copying is forbidden because it would duplicate ownership,
- moving transfers ownership, making lifetime explicit.

3.4.7 RAII at architecture scale: the “composition root”

Large systems must create and connect subsystems:

- logging,
- configuration,
- thread pools,

- database connections,
- caches,
- network endpoints.

A mature RAII architecture centralizes ownership at a **composition root**:

- the top-level object graph owns resources,
- subsystems receive references or non-owning handles,
- lifetimes become hierarchical and predictable.

```
class App {
public:
  App()
    : config_{load_config()},
      log_{config_},
      db_{config_},
      service_{db_, log_} {}

  void run();

private:
  Config config_;
  Logger log_;
  Database db_;
  Service service_; // holds non-owning references to db_ and log_ internally
};
```

This approach prevents scattered global singletons and reduces cognitive load: resource lifetime is visible in one place.

3.4.8 RAII and concurrency: locking, joining, and cancellation safety

Concurrency introduces resources that must be released deterministically:

- locks must be released,
- threads must be joined or detached correctly,
- tasks must be canceled or completed cleanly,
- waiting must avoid deadlocks and leaks.

RAII is the dominant strategy for making concurrency safe:

```
class JoiningThread {
public:
  explicit JoiningThread(std::thread t) noexcept : t_{std::move(t)} {}
  ~JoiningThread() {
    if (t_.joinable()) t_.join(); // deterministic cleanup
  }

  JoiningThread(const JoiningThread&) = delete;
  JoiningThread& operator=(const JoiningThread&) = delete;
  JoiningThread(JoiningThread&&) noexcept = default;
  JoiningThread& operator=(JoiningThread&&) noexcept = default;

private:
  std::thread t_;
};
```

Even if an exception occurs or an early return happens, the thread is joined, avoiding a common class of production failures.

3.4.9 Common RAII mistakes that create accidental complexity

RAII is powerful, but misuse creates new problems. Typical mistakes include:

1) Owning raw pointers

Owning raw pointers hides ownership and invites leaks and double deletion. Prefer:

- direct members for mandatory ownership,
- `std::unique_ptr` for optional/dynamic ownership,
- `std::shared_ptr` only when shared lifetime is essential.

2) Destructors that throw

Destructors should not emit exceptions. A destructor should be a safe finalization path.

3) "Two-phase initialization"

Designs that require `init()` after construction are fragile because they create partially valid states. Prefer constructors (or factory functions) that produce fully valid objects.

4) Hidden global ownership

Global singletons hide lifetime, complicate tests, and increase coupling. Prefer explicit ownership graphs at the composition root.

3.4.10 RAII as a policy: making invalid states unrepresentable

The deepest value of RAII is that it prevents a class of invalid states:

- "resource acquired but not released",

- “resource released twice”,
- “resource used after release”,
- “lock held without being released.”

Designing with RAII means representing these states as impossible or highly constrained, rather than relying on discipline.

3.4.11 A practical RAII checklist for design reviews

When reviewing APIs and architecture, use these questions:

- Does this type *own* a resource? If yes, is ownership explicit and unique by default?
- Is the resource acquired during construction or in a factory that returns a fully valid object?
- Is cleanup guaranteed on every path (including early return and exceptions)?
- Are partial states prevented (no mandatory `init()` step)?
- Are non-owning references clearly labeled and lifetime-safe?
- Are concurrency resources (locks/threads) managed by RAII wrappers?

If the answers are unclear, the design is inviting entropy.

3.4.12 Summary

RAII is not merely a memory-management trick; it is a Modern C++ design philosophy:

- It creates deterministic lifetime and cleanup, which enables reliable local reasoning.
- It makes ownership explicit and enforces “release exactly once” semantics.

- It improves exception safety and robustness under complex control flow.
- It supports architecture-scale clarity by centralizing ownership and preventing hidden global lifetimes.
- It reduces accidental complexity by turning cleanup into structure rather than convention.

A long-lived C++ system that fully embraces RAII is not one that "remembers to clean up." It is one whose design makes forgetting unlikely, misuse difficult, and resource lifetimes transparent.

3.5 Compile-Time as a Design Tool

Modern C++ is unusual among mainstream languages in how much **design work** it can move from runtime into compile time. This is not merely an optimization technique. Used deliberately, compile-time facilities become a **design tool**: they express constraints, enforce contracts, eliminate invalid states, and make whole categories of errors impossible to ship.
In long-lived systems, the goal is not to maximize compile-time cleverness. The goal is to use compile-time mechanisms to make interfaces *unambiguous*, *hard to misuse*, and *cheap to run*. When applied with discipline, compile-time design reduces operational risk and cognitive load while improving performance predictability.

3.5.1 What "compile-time as a design tool" means

Treat compile time as a place where you can:

- **encode invariants** (legal states, valid ranges, required capabilities),
- **constrain generic code** (requirements are explicit, diagnostics are actionable),
- **select policies** without runtime branching (strategy chosen statically),
- **precompute constants** (tables, hashes, parsers for fixed inputs),

- **prevent misuse** by construction (types and constraints guide the user),
- **shift failures left** (errors appear at build time instead of production).

This is a design philosophy: correctness and intent are expressed in types and compile-time rules, not hidden in documentation or runtime checks.

3.5.2 The compile-time toolbox in Modern C++

Modern C++ provides several layers of compile-time capability:

- **Types and templates:** static polymorphism, policy-based design, specialization.
- **Type traits and metafunctions:** reasoning about types (`std::is_*`, `std::remove_*`, `std::conditional_t`).
- **Compile-time functions:** `constexpr` and `consteval`.
- **Initialization guarantees:** `constinit` (static initialization safety).
- **Constraints:** concepts and `requires` (explicit requirements for templates).
- **Assertions:** `static_assert` (compile-time contract enforcement).
- **Constant parameters:** non-type template parameters (NTTP) for sizes, policies, tags.

Each feature is powerful; together they form a design language for building constrained, reliable systems.

3.5.3 1) Compile-time constraints: make requirements explicit

A recurring source of complexity in large systems is generic code that accepts "anything" and fails later with unreadable errors or undefined behavior. Concepts and constraints convert that into explicit design contracts.

```cpp
#include <concepts>
#include <cstddef>
#include <functional>

template <typename T>
concept HashableKey =
  requires(const T& k) {
    { std::hash<T>{}(k) } -> std::convertible_to<std::size_t>;
    { k == k } -> std::convertible_to<bool>;
  };

template <HashableKey Key, typename Value>
class Cache {
public:
  void put(Key key, Value value);
  // ...
};
```

Design impact:

- the interface states what it needs,
- invalid uses are rejected at compile time,
- error messages become part of the developer experience,
- runtime checks are avoided because the contract is structural.

3.5.4 2) Compile-time invariants: invalid states become unrepresentable

Design quality improves when constraints are enforced mechanically. For many invariants, compile time can enforce them completely.

Example: fixed-size buffers as a type-level invariant

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

template <std::size_t N>
class FixedBuffer {
public:
  constexpr std::size_t size() const noexcept { return N; }
  std::span<std::byte, N> bytes() noexcept { return std::span<std::byte, N>{data_}; }
  std::span<const std::byte, N> bytes() const noexcept { return std::span<const std::byte,
  ↪  N>{data_}; }

private:
  std::array<std::byte, N> data_{};
};
```

Here, buffer size is not a runtime variable; it is part of the type. Many mistakes become impossible:

- passing a buffer of the wrong size,
- forgetting required capacity checks in APIs that demand a specific size,
- mixing protocols that require different fixed payload lengths.

3.5.5 3) Compile-time computation: move work out of runtime

Compile-time computation is often presented as an optimization, but it is also design:

- it makes costs predictable,
- it removes runtime failure modes for fixed inputs,
- it allows prevalidated constants and tables.

Example: compile-time hashing for stable identifiers

```
#include <cstdint>
#include <string_view>

consteval std::uint32_t fnv1a_32(std::string_view s) {
  std::uint32_t h = 2166136261u;
  for (unsigned char c : s) {
    h ^= c;
    h *= 16777619u;
  }
  return h;
}

constexpr std::uint32_t EventId_UserCreated = fnv1a_32("UserCreated");
constexpr std::uint32_t EventId_UserDeleted = fnv1a_32("UserDeleted");
```

Design impact:

- identifiers are stable and cheap at runtime,
- no runtime parsing or string hashing is required in hot paths,
- the mapping from semantic names to numeric IDs remains explicit.

3.5.6 4) Policy selection without runtime branching

Large systems frequently need configurable behavior: logging strategies, allocation strategies, retry policies, synchronization approaches. Compile-time policy selection yields designs that remain flexible without introducing runtime overhead everywhere.

Example: policy-based retry strategy

```
#include <chrono>
```

```
#include <thread>

struct NoRetry {
  template <typename F>
  auto run(F&& f) {
    return f();
  }
};

struct FixedRetry {
  int attempts{3};
  std::chrono::milliseconds delay{50};

  template <typename F>
  auto run(F&& f) {
    for (int i = 0; i < attempts; ++i) {
      if (auto r = f(); r) return r;
      std::this_thread::sleep_for(delay);
    }
    return f(); // last attempt
  }
};

template <typename RetryPolicy>
class Client {
public:
  explicit Client(RetryPolicy p) : policy_{p} {}

  template <typename Operation>
  auto call(Operation&& op) {
    return policy_.run(std::forward<Operation>(op));
  }
```

```
private:
  RetryPolicy policy_;
};
```

Design impact:

- flexibility is achieved through composition,
- the policy is explicit in the type and construction,
- callers can choose behavior without global flags,
- hot-path overhead is proportional to the chosen semantics.

3.5.7 5) `static_assert`: design contracts that cannot be ignored

`static_assert` is a design tool when used to enforce architectural assumptions and invariants that must never silently drift.

```
#include <type_traits>
#include <cstdint>

struct PacketHeader {
  std::uint16_t type;
  std::uint16_t flags;
  std::uint32_t length;
};

static_assert(std::is_trivially_copyable_v<PacketHeader>,
              "PacketHeader must be trivially copyable for safe binary IO.");

static_assert(sizeof(PacketHeader) == 8,
              "PacketHeader size is part of the protocol contract.");
```

This is not “cleverness”; it is engineering discipline:

- protocol assumptions become executable constraints,
- accidental padding or layout changes are detected immediately,
- reviewers see the contract at the point of definition.

3.5.8 6) `constinit`: preventing static initialization hazards

Global objects with dynamic initialization can introduce order-of-initialization problems across translation units. `constinit` is a design tool for enforcing safe initialization properties.

```
#include <array>

constinit std::array<int, 4> Ports = { 80, 443, 8080, 8443 };
// If this cannot be constant-initialized, compilation fails.
```

Design impact:

- initialization becomes deterministic and safe,
- the architecture avoids hidden startup hazards,
- failures are compile-time, not production-time.

3.5.9 7) Compile-time types as domain language

Strong types often reduce both bugs and cognitive load by making the domain explicit. When combined with compile-time constraints, they become a robust design language.

Example: strong IDs prevent category errors

```
#include <cstdint>

struct UserId { std::uint64_t value; };
struct OrderId { std::uint64_t value; };

void load_user(UserId);
void load_order(OrderId);

// load_user(OrderId{1}); // does not compile: category error prevented
```

This is compile-time enforcement of domain correctness. It prevents an entire class of production defects.

3.5.10 Costs and tradeoffs: compile-time is not free

Using compile-time mechanisms has real costs that must be treated as design constraints:

- **Compile-time cost:** heavy templates and metaprogramming increase build time.
- **Binary size:** template instantiations can increase code size.
- **Debuggability:** excessive type-level complexity can reduce readability.
- **Tooling friction:** diagnostics and IDE performance may degrade with extreme genericity.

Therefore, “compile-time as a design tool” must be governed by a maturity rule:

> Use compile-time mechanisms to encode *stable* constraints and invariants. Avoid pushing unstable product variability into templates when it harms clarity or build performance.

3.5.11 A pragmatic design checklist

Use the following checklist to apply compile-time design responsibly:

- Can a misuse be prevented by the type system rather than a runtime check?
- Is the constraint stable and central to the architecture (protocol size, thread-safety policy, ownership)?
- Will the compile-time mechanism reduce cognitive load for typical developers?
- Does the compile-time choice improve error messages and usability of the API?
- Is the added build-time and complexity cost justified by long-term risk reduction?

If the answer to these is “yes,” compile time is often the best place for the rule.

3.5.12 Summary

Compile-time capabilities in Modern C++ are not only about speed:

- They are a **design tool** for encoding invariants, constraints, and policies.
- They shift failures left, making entire bug classes impossible to ship.
- They enable predictable cost models by eliminating runtime indirection and recomputation when semantics are known.
- They must be used with discipline to avoid build-time explosion and needless complexity.

In long-lived systems, compile-time design is a form of architectural governance: it keeps the system honest, explicit, and mechanically enforceable as it evolves.

Part II

Core Design Principles Revisited (Through C++)

Responsibility, Cohesion, and Boundaries

4.1 Single Responsibility Reinterpreted

4.1.1 Why the Classic SRP Needs a Modern Restatement

The **Single Responsibility Principle (SRP)** is frequently repeated, frequently misunderstood, and often misapplied—especially in large C++ systems where performance, ABI, concurrency, and build times impose real architectural constraints.
A modern restatement that works well for long-lived C++ systems is:

> **A software unit should have one axis of change:** it should be responsible for *one coherent policy* and should change only when that policy changes.

This framing moves SRP away from superficial interpretations like "a class should do one thing" and toward an engineering interpretation:

- **Responsibility is not a method count.** A class can be small and still violate SRP if it mixes unrelated policies.
- **Responsibility is not a layer name.** "Service" or "Manager" is not a responsibility; it is often a warning sign.
- **Responsibility is an axis of change.** SRP is about stabilizing code under evolving requirements.

4.1.2 Responsibility as a Boundary, Not a Size Constraint

In real codebases, responsibilities are discovered by observing **change pressure**:

- Requirements change (business rules, validation policies, pricing rules).
- Platforms change (OS APIs, filesystems, time sources, locale behavior).
- Infrastructure changes (database engine, messaging protocol, serialization format).
- Runtime constraints change (latency targets, memory ceilings, thread model).
- Security rules change (authorization policy, auditing requirements).

SRP asks: **Can we isolate each category of change behind a boundary so changes do not cascade?**
In C++, boundaries must consider not only correctness but also:

- **Binary compatibility and ABI stability** (public headers, inline functions, templates).
- **Build-time coupling** (include graph, macro and header pollution).
- **Ownership and lifetime** (RAII, exceptions, error propagation).
- **Concurrency semantics** (thread-safety guarantees, data race avoidance).

4.1.3 A Practical Definition of "One Responsibility" in C++

A unit (class, module, component, library) has **one responsibility** if:

1. It owns **one policy** (one set of rules/decisions).
2. It has **one reason to change** (one category of requirement change).
3. It offers **one stable contract** to clients.

4. It hides details whose change should not affect clients (including build and ABI impact).

Conversely, SRP is violated when a unit mixes:

- **Policy and mechanism** (business rules + threading + serialization).
- **Domain and infrastructure** (entities + SQL + logging + HTTP).
- **Decision and orchestration** (choosing rules + sequencing workflows + retry logic).
- **Stable and volatile dependencies** (core types including rapidly changing headers).

4.1.4 Cohesion: The Technical Signal Behind SRP

SRP and cohesion are tightly connected. High cohesion means:

- The members of a unit collaborate to support the same concept.
- Data fields and invariants exist to enforce the same rules.
- Public methods form a focused vocabulary around one concept.

Low cohesion typically manifests as:

- Methods that operate on disjoint subsets of fields.
- Frequent conditional branching by “mode” or “type”.
- “Utility” methods that do not rely on class invariants.
- Changes that touch unrelated areas of the unit.

In C++, cohesion has a special build-time dimension:

- **Header cohesion:** a header should expose one concept, not become a dependency dump.
- **Include minimality:** dependencies are part of responsibility boundaries.

4.1.5 The SRP Trap: "One Class, One Job" Is Too Naive

A realistic C++ component may handle multiple *operations* while still respecting SRP, if all operations are facets of the same policy.
Example: a `Money` type might provide arithmetic, formatting rules, and rounding rules. SRP is not about restricting operations, but about preventing *unrelated* axes of change. If formatting rules vary independently from arithmetic correctness policies, they likely belong behind separate boundaries.
SRP questions to ask:

- "If formatting rules change, will arithmetic code change?"
- "If storage backend changes, will domain rules change?"
- "If logging policy changes, will business logic change?"

4.1.6 SRP Reinterpreted Through Modern C++ Concepts

1) RAII Separates Resource Responsibility from Business Responsibility

C++ makes lifetime and ownership explicit. SRP encourages isolating resource management:

- A class responsible for a domain concept should not be responsible for OS handles.
- Resource wrappers should enforce ownership invariants and nothing else.

```
#include <cstdio>
#include <stdexcept>

class FileHandle {
public:
  explicit FileHandle(const char* path, const char* mode)
```

```
    : f_(std::fopen(path, mode)) {
    if (!f_) throw std::runtime_error("fopen failed");
  }

  ~FileHandle() noexcept {
    if (f_) std::fclose(f_);
  }

  FileHandle(const FileHandle&) = delete;
  FileHandle& operator=(const FileHandle&) = delete;

  std::FILE* get() const noexcept { return f_; }

private:
  std::FILE* f_;
};

// Domain logic does not own OS semantics; it consumes a service:
class ReportWriter {
public:
  void write(std::FILE* out) {
    // domain formatting policy lives here (and can evolve)
    std::fputs("report\n", out);
  }
};
```

The boundary is clear:

- `FileHandle` changes when OS/file-handle policy changes.
- `ReportWriter` changes when reporting policy changes.

2) Modules and Header Hygiene Make Responsibility Boundaries Real

Modern C++ increasingly treats **dependency control** as architectural work:

- Public headers define the contract and should be stable.
- Volatile dependencies should be hidden in implementation files.
- Heavy templates in public headers expand the responsibility surface area (build-time coupling).

A practical SRP rule for C++:

> **If a dependency frequently changes, it should not be required by your stable public types.**

3) Concepts Improve SRP by Making Contracts Explicit

With concepts, generic code can express a narrow responsibility:

- A template can be responsible for one algorithmic policy.
- The concept boundary prevents accidental coupling to unintended types.

```
#include <concepts>
#include <string_view>

template <class T>
concept LogSink = requires(T s, std::string_view msg) {
  { s.write(msg) } -> std::same_as<void>;
};

template <LogSink Sink>
```

```
class Logger {
public:
  explicit Logger(Sink& sink) : sink_(sink) {}

  void info(std::string_view msg) {
    // Responsibility: logging policy (formatting / levels / routing)
    sink_.write(msg);
  }

private:
  Sink& sink_;
};
```

The logger here is not responsible for:

- File handles,
- network transports,
- rotation,
- compression,
- authentication.

Those belong to other axes of change.

4) Coroutines and Async Workflows Multiply Reasons to Change

When a unit mixes domain rules with asynchronous orchestration, SRP often breaks:

- domain policy changes,
- retry/backoff changes,

- cancellation policy changes,
- scheduling/executor changes.

A stable pattern:

- Keep domain logic in synchronous, testable units.
- Keep async orchestration in separate units that compose domain operations.

4.1.7 How SRP Fails in the Real World: Common Anti-Patterns in C++

The "God Object" Service

Symptoms:

- A single class includes database, HTTP, JSON, logging, caching headers.
- Most methods accept broad context objects and branch heavily.
- Changes in one subsystem force rebuilds and retests everywhere.

Consequences:

- Build times explode due to include coupling.
- Unit tests become integration tests.
- Concurrency bugs appear because shared state is everywhere.

The "Entity That Knows Too Much"

A domain entity that also performs persistence or transport concerns (SQL/JSON/HTTP) violates SRP. In C++, it also contaminates the entity's header with volatile dependencies.

The "Utility Header" That Grows Forever

A utility header often becomes a dumping ground. It violates SRP at the file level:

- unrelated responsibilities share one compilation boundary,
- unrelated changes trigger recompilation of unrelated code.

4.1.8 SRP at Multiple Scales: Class, Module, Component, Library

SRP is not only a class rule. In long-lived systems, it must be applied across scales:

Class Level

- Maintain a focused invariant.
- Keep private data cohesive (methods operate on the same invariant).
- Avoid mixing policy with I/O and threading.

File and Header Level

- One header should expose one concept and minimal dependencies.
- Implementation details belong in `.cpp` (or in a private module partition).
- Prefer forward declarations when feasible; include only what is required.

Module and Library Level

- A library should represent one area of policy: e.g., *serialization*, *networking*, *storage access*, *domain rules*.
- Do not blend stable APIs with volatile third-party dependencies in the same public surface.

- Consider PIMPL or façade layers for ABI stability when your audience includes third parties.

4.1.9 Engineering SRP: Boundary Techniques That Work in Modern C++

Technique 1: Split Policy From Mechanism

- **Policy:** decisions and rules.
- **Mechanism:** how the rule is executed (threads, sockets, files, syscalls).

If policy changes, mechanism should not. If mechanism changes, policy should not.

Technique 2: Use Interfaces for Volatile Dependencies

A stable core should depend on abstractions, not volatile headers.

```
#include <string_view>

struct AuditSink {
  virtual ~AuditSink() = default;
  virtual void record(std::string_view event) = 0;
};

class AccountPolicy {
public:
  explicit AccountPolicy(AuditSink& audit) : audit_(audit) {}

  void deposit(int amount) {
    // Responsibility: domain rule (validation, invariants)
    if (amount <= 0) return;
```

```
    balance_ += amount;
    audit_.record("deposit");
  }

private:
  int balance_{0};
  AuditSink& audit_;
};
```

Now:

- domain policy changes do not require changing audit mechanism,
- audit mechanism changes do not require recompiling domain rules (if boundary is stable).

Technique 3: Treat "Error Policy" as a Responsibility

In C++, error handling is not decoration. It is a policy:

- exceptions vs error codes,
- recoverable vs fatal errors,
- logging and telemetry,
- retry and compensation strategies.

Mixing error policy into domain logic often creates multiple axes of change. A robust design isolates:

- domain invariants and validations,
- error mapping and transport-level translation (HTTP codes, gRPC statuses),
- logging/telemetry policy.

Technique 4: Dependency Direction Is Part of SRP

A common SRP violation is when low-level modules depend on high-level policies due to convenience. A practical rule:

> **The module that changes least should not depend on modules that change more.**

This aligns SRP with boundary stability:

- core domain should not include JSON, SQL drivers, web frameworks,
- adapters should translate between stable core contracts and volatile infrastructure.

4.1.10 A Decision Checklist: Does This Unit Violate SRP?

Use the following checklist during design reviews:

1. **Change triggers:** Can you name more than one independent reason this unit changes?
2. **Dependency footprint:** Does its public header include volatile dependencies?
3. **Build impact:** Does changing it trigger recompilation across unrelated areas?
4. **Testing signal:** Do its unit tests require database/network/filesystem?
5. **Concurrency signal:** Does it combine data invariants with threading primitives?
6. **Vocabulary:** Do its method names belong to multiple unrelated domains?

If two or more items are true, SRP is likely violated.

4.1.11 SRP Does Not Mean "No Collaboration"

SRP does *not* imply that units should be isolated islands. It implies:

- Collaboration happens through **stable contracts**.
- The system's behavior emerges from composition of focused responsibilities.
- Boundaries prevent ripple effects.

In C++ specifically, SRP is also about:

- minimizing compile-time and ABI blast radius,
- isolating ownership and lifetime policies,
- separating concurrency and scheduling from domain invariants,
- keeping the stable core free of volatile infrastructure.

4.1.12 Summary: SRP as "Axis of Change" for Long-Lived C++ Systems

A modern C++ interpretation of SRP is not a slogan about class size. It is an architectural discipline:

- Identify axes of change.
- Create boundaries that isolate those changes.
- Use C++ tools (RAII, concepts, modules, header hygiene, PIMPL, interfaces) to enforce boundaries.
- Measure success by reduced ripple effects: fewer rebuilds, fewer retests, fewer regressions.

When SRP is applied this way, "single responsibility" becomes a practical strategy for designing software that remains coherent, maintainable, and evolvable over years of change.

4.2 High Cohesion in Practice

4.2.1 Cohesion as the Engineering Signal Behind Good Design

Cohesion describes how strongly the elements inside a unit (class, module, library, component) belong together. In long-lived systems, cohesion is not a "style preference"; it is a measurable driver of:

- maintainability under continuous change,
- testability and fault isolation,
- compile-time scalability and dependency control (especially in C++),
- correctness under concurrency (clear invariants and ownership),
- disciplined public APIs with stable contracts.

A practical restatement:

> **A cohesive unit implements one coherent concept, enforced by one set of invariants, expressed through a small vocabulary, and protected by a small dependency surface.**

High cohesion does *not* mean "small." It means **internally consistent**. A large unit can be cohesive if every part exists to serve the same concept and invariants. A tiny unit can be incohesive if it mixes unrelated policies.

4.2.2 Cohesion vs. Coupling: The Non-Negotiable Pair

Cohesion and coupling are not independent:

- **High cohesion** makes it easier to keep **coupling low** because the unit exposes a focused interface.
- **Low cohesion** often forces **high coupling** because unrelated clients depend on unrelated parts of the same unit.

In C++, the cost of coupling is amplified by:

- heavy include graphs and template instantiation,
- macro and header pollution,
- build-system scale effects (incremental rebuild times),
- ABI stability concerns for distributed artifacts.

Therefore, for Modern C++ architecture, cohesion is both a design principle *and* a build-system survival strategy.

4.2.3 Forms of Cohesion You Can Recognize in Code Reviews

When reviewing code, you rarely need formal theory. You need recognizable patterns:

Functional Cohesion (Best)

All operations contribute to a single, well-defined purpose. The unit is centered on one invariant or one algorithmic responsibility.

Communicational Cohesion (Often Good)

Operations act on the same data (same state/invariant), even if they perform different actions.

Sequential Cohesion (Sometimes Acceptable)

Output of one operation becomes the input of the next, forming a pipeline. Good when it represents a real workflow unit.

Temporal Cohesion (Warning Sign)

Operations are grouped because they happen "around the same time" (startup/shutdown), not because they share a concept. Frequently leads to "init managers" that grow endlessly.

Logical/Conditional Cohesion (Strong Warning)

Operations are grouped because they are "similar" and selected by flags or enums (`if(mode==...)`). This is a common source of rigidity and bug farms.

Coincidental Cohesion (Worst)

A dumping ground: unrelated helpers, unrelated policies, unrelated dependencies.

4.2.4 The Core Mechanism of High Cohesion in C++: Invariants

In C++, cohesion becomes concrete when a type owns a clear invariant:

- data members exist to maintain that invariant,
- methods exist to preserve or evolve it,
- private helpers exist only to support it,
- external dependencies are avoided unless they are intrinsic to the concept.

A cohesive class typically has:

- a small number of data members that are all "about the same thing,"

- methods that touch most members (not disjoint subsets),
- few boolean flags controlling unrelated modes,
- a focused, predictable API vocabulary.

Example: Cohesion Through a Single Invariant (Range Validity)

```cpp
#include <stdexcept>
#include <cstdint>

class ByteRange {
public:
  ByteRange(std::uint64_t begin, std::uint64_t end)
    : begin_(begin), end_(end) {
    if (begin_ > end_) throw std::invalid_argument("invalid range");
  }

  std::uint64_t begin() const noexcept { return begin_; }
  std::uint64_t end()   const noexcept { return end_;   }
  std::uint64_t size()  const noexcept { return end_ - begin_; }

  bool contains(std::uint64_t x) const noexcept {
    return x >= begin_ && x < end_;
  }

  ByteRange intersect(ByteRange other) const {
    const auto b = (begin_ > other.begin_) ? begin_ : other.begin_;
    const auto e = (end_   < other.end_)   ? end_   : other.end_;
    if (b > e) return ByteRange{0, 0};
    return ByteRange{b, e};
```

```
  }

private:
  std::uint64_t begin_;
  std::uint64_t end_;  // invariant: begin_ <= end_
};
```

Why this is cohesive:

- One concept: a half-open range.
- One invariant: `begin_ <= end_`.
- All methods serve that invariant and concept.
- No I/O, no logging, no unrelated policies.

4.2.5 API Vocabulary: Cohesion at the Interface Level

High cohesion is visible in names. A cohesive API forms a small, consistent vocabulary. Incohesion often reveals itself as a mixed vocabulary:

- domain language mixed with infrastructure language (`price()`, `connect()`, `serializeJson()`, `mutexLock()`),
- mixed abstraction levels (high-level use cases next to low-level byte manipulations),
- "grab bag" methods that are not conceptually related to the type's invariant.

A pragmatic review technique:

> **If you can't describe the class in one sentence without using "and," cohesion is suspect.**

4.2.6 Cohesion at File and Header Level: Dependency Surface as a Design Boundary

In Modern C++, cohesion must be enforced at the **header boundary**:

- a header should expose one concept,
- includes should be minimal and concept-necessary,
- unstable dependencies must not leak into stable public headers.

A unit can be logically cohesive yet practically incohesive if its header pulls half the world. This is why **dependency cohesion** matters:

- Do clients include this header because they need one concept, or because it's the only place to get it?
- Does the header include unrelated headers "just in case"?
- Do changes in unrelated dependencies force rebuilds of unrelated components?

A strong rule for long-lived systems:

> **The public header is the unit's contract. Every include is part of the contract.**

4.2.7 High Cohesion in Components: "Stable Core, Volatile Edge"

At system scale, cohesion means each component owns one coherent policy:

- **Core domain component:** invariants and domain rules.
- **Storage component:** persistence mechanism and mapping.
- **Transport component:** HTTP/gRPC protocol and translation.

- **Observability component:** logging, metrics, tracing policy.

The typical failure mode is a component that mixes:

- domain rules + SQL + JSON + logging + retries + threading.

That is not one component; it is multiple components welded together, guaranteeing painful change cascades.

4.2.8 Cohesion Under Concurrency: Thread-Safety Is a Responsibility Boundary

Concurrency often destroys cohesion when:

- locking policy is scattered across methods,
- invariants are not clearly protected by one lock strategy,
- both data rules and scheduling rules are mixed in the same unit.

A cohesive approach:

- One unit owns the **data invariant**.
- Another unit (or a wrapper) owns the **synchronization policy**.

Example: Separate Invariant From Synchronization Policy

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

class NameInvariant {
```

```
public:
  void set(std::string v) {
    if (v.empty()) throw std::invalid_argument("empty name");
    value_ = std::move(v);
  }

  const std::string& get() const noexcept { return value_; }

private:
  std::string value_{"default"};
};

class ThreadSafeName {
public:
  void set(std::string v) {
    std::lock_guard<std::mutex> lk(m_);
    inv_.set(std::move(v));
  }

  std::string get_copy() const {
    std::lock_guard<std::mutex> lk(m_);
    return inv_.get();
  }

private:
  mutable std::mutex m_;
  NameInvariant inv_;
};
```

Here cohesion is preserved:

- `NameInvariant` changes when naming rules change.
- `ThreadSafeName` changes when synchronization policy changes.

4.2.9 Measuring Cohesion: Practical Signals and Lightweight Metrics

You do not need heavy tooling to detect cohesion problems, but it can help. Common pragmatic signals:

- **Disjoint fields:** different methods touch different subsets of members.
- **Flag-driven behavior:** many methods branch on mode flags or type enums.
- **Include explosion:** the header includes many unrelated libraries.
- **Mixed tests:** unit tests require database/network/filesystem.
- **Churn hotspots:** file changes frequently for unrelated reasons.
- **Large "friend" networks:** many friends indicate broken encapsulation boundaries.

A lightweight quantitative hint (not a law):

- **LCOM-style thinking:** if methods share few fields, cohesion is likely low.

Use such numbers as smoke detectors, not as correctness proofs.

4.2.10 Refactoring Toward High Cohesion: Patterns That Work

1) Extract Concept (Split by Invariant)

When a subset of methods and fields form their own invariant, extract a new type that owns it.

2) Extract Policy (Split by Axis of Change)

When a unit changes for two independent reasons (formatting rules and storage rules), split policies.

3) Introduce Adapter (Keep Volatile Dependencies at the Edge)

Keep third-party and protocol headers out of core headers. Put translation in adapters.

4) Replace Flags With Types

Flag-driven cohesion failures often become clean designs by replacing mode flags with separate types.

Example: Replace Mode Flag With Separate Types

```
#include <string_view>

struct PlainFormatter {
  std::string format(std::string_view s) const {
    return std::string{s};
  }
};

struct JsonFormatter {
  std::string format(std::string_view s) const {
    // minimal illustrative escaping; real JSON escaping would be broader
    std::string out;
    out.reserve(s.size() + 2);
    out.push_back('"');
    for (char c : s) {
```

```
      if (c == '"' || c == '\\') out.push_back('\\');
      out.push_back(c);
    }
    out.push_back('"');
    return out;
  }
};
```

Instead of a single incohesive class:

- `Formatter(mode)` with branching and mixed responsibilities,

you get cohesive types, each with one formatting policy.

4.2.11 High Cohesion in Modern C++ Builds: Keep Public Surfaces Small

C++ projects often fail the cohesion test because of build architecture:

- a “core” header includes networking and database headers,
- templates and inline functions expose private dependencies,
- macros leak across components and change meaning.

Practical cohesion rules for build-scale systems:

- Public headers: minimal includes, stable types, stable vocabulary.
- Implementation files: include the heavy and volatile dependencies.
- Prefer passing abstract interfaces or small value types across boundaries.
- Avoid “global configuration headers” that become universal coupling points.

4.2.12 Review Checklist: "Is This Unit Cohesive?"

Use this checklist during design reviews:

1. **One sentence:** Can you describe the unit without using "and"?
2. **Invariant:** Can you name the invariant it owns and enforces?
3. **Vocabulary:** Do method names belong to one domain language?
4. **Field sharing:** Do most methods work with the same core state?
5. **Dependencies:** Does the public surface avoid volatile includes?
6. **Testing:** Can you test it without external infrastructure?
7. **Concurrency:** Are data rules separated from scheduling/locking policy?

If multiple answers are "no," cohesion is likely low.

4.2.13 Summary: Cohesion as a Boundary Discipline

High cohesion is the practice of designing units that:

- express one coherent concept,
- enforce one set of invariants,
- offer a small, consistent interface vocabulary,
- minimize dependency surface (especially in public headers),
- keep volatile mechanisms at the edges,
- preserve testability and change isolation.

In long-lived Modern C++ systems, cohesion is not merely "clean code"—it is the foundation that keeps architecture stable as requirements, platforms, and tooling evolve.

4.3 Responsibility vs Ownership

4.3.1 Why This Distinction Matters in Modern C++

In software design discussions, the word *responsibility* is often used loosely, and the word *ownership* is sometimes treated as synonymous with it. In Modern C++, this confusion is costly because C++ makes ownership explicit and enforceable through the type system, RAII, and move semantics.
A robust, C++-accurate separation is:

- **Responsibility** answers: *"Who decides? Who enforces the rule? Who owns the policy?"*
- **Ownership** answers: *"Who holds the resource? Who destroys it? Who controls lifetime?"*

These are orthogonal axes. A unit can be responsible for a policy while not owning any resources. A unit can own resources while not being responsible for the domain policy that uses them.

4.3.2 Definitions That Scale Beyond Single Classes

Responsibility (Policy Ownership)

Responsibility is a **semantic** property:

- It is about **rules**, **invariants**, and **behavioral guarantees**.
- It is stable when aligned with a clear axis of change (SRP).
- It defines **why** something must behave in a certain way.

Responsibility is ideally localized:

- When a rule changes, one primary unit changes.
- Other units remain unaffected because they depend on contracts, not internals.

Ownership (Lifetime Control)

Ownership is a **mechanical** property:

- It is about **lifetime**, **resource release**, and **exclusive vs shared control**.
- In C++, it is expressed through RAII and types: values, references, `unique_ptr`, `shared_ptr`, file/socket wrappers, etc.
- It defines **how long** something exists and **who cleans it up**.

4.3.3 A Design Rule: Never Mix Policy and Lifetime Accidentally

A frequent architecture failure in C++ occurs when:

- a policy class begins to allocate and own infrastructure objects,
- a resource manager begins to embed business rules,
- an orchestration unit becomes the lifetime owner of everything.

A practical rule for long-lived systems:

> **Let policy depend on stable abstractions. Let ownership live at the boundary where lifetimes are naturally managed.**

4.3.4 Responsibility Without Ownership: "Rules Over Objects"

A unit may be responsible for correctness but not own the resources it operates on.

Example: Domain Policy Does Not Own Persistence or Transport

```cpp
#include <string>
#include <string_view>
#include <stdexcept>

struct User {
  std::string name;
  int age{};
};

class UserPolicy {
public:
  // Responsibility: validation rules (policy)
  void validate(const User& u) const {
    if (u.name.empty()) throw std::invalid_argument("name required");
    if (u.age < 0 || u.age > 130) throw std::invalid_argument("invalid age");
  }
};
```

`UserPolicy` is responsible for rules, but owns nothing. Ownership (where `User` is stored, who keeps it alive, how it is persisted) is separate.
This separation is beneficial because:

- policy changes do not force changes in storage,
- policy is trivially testable (pure, deterministic),
- policy can be reused across components.

4.3.5 Ownership Without Responsibility: “RAII as a Mechanical Contract”

A unit may own a resource and guarantee cleanup, while being intentionally ignorant of domain rules.

Example: RAII Socket Wrapper Owns Lifetime, Not Business Meaning

```
#include <utility>
#include <stdexcept>

class SocketHandle {
public:
  explicit SocketHandle(int fd) : fd_(fd) {
    if (fd_ < 0) throw std::runtime_error("invalid socket");
  }

  ~SocketHandle() noexcept {
    // close(fd_) would occur here (omitted)
  }

  SocketHandle(const SocketHandle&) = delete;
  SocketHandle& operator=(const SocketHandle&) = delete;

  SocketHandle(SocketHandle&& other) noexcept : fd_(std::exchange(other.fd_, -1)) {}
  SocketHandle& operator=(SocketHandle&& other) noexcept {
    if (this != &other) {
      // close(fd_) would occur here
      fd_ = std::exchange(other.fd_, -1);
    }
    return *this;
```

```
  }

  int get() const noexcept { return fd_; }

private:
  int fd_{-1};
};
```

This class owns the lifetime. It is not responsible for:

- authentication policy,
- application protocol,
- retry behavior,
- rate limits,
- business semantics of the messages.

4.3.6 Where Confusion Happens: The "Owner" Is Not Always the "Responsible"

The most common confusion patterns:

Confusion 1: "The object that holds a pointer is responsible for everything"

Holding a pointer is about lifetime and access, not necessarily policy. A pointer member often means:

- dependency injection,
- composition,

- an adapter boundary,
- or simply ownership.

It does not automatically imply responsibility for the pointed-to object's domain meaning.

Confusion 2: "The orchestrator owns everything, therefore it must decide everything"

Composition roots (application startup, dependency wiring, main controller) often *own* lifetimes. But they should remain **policy-thin**, otherwise they become a "god component" with many axes of change.

Confusion 3: "shared_ptr means shared responsibility"

`shared_ptr` only means shared *lifetime*. It says nothing about responsibility. In fact, using `shared_ptr` excessively often obscures responsibility boundaries.

4.3.7 Responsibility Boundaries: Who Owns the Invariant?

A rigorous way to assign responsibility is:

- Identify **the invariant**.
- Assign responsibility to the unit that can **enforce it consistently**.
- Ensure other units cannot violate it accidentally.

In C++, invariants are defended through:

- encapsulation (private state),
- constructors establishing valid states,
- methods preserving valid states,

- types encoding constraints (strong types),
- RAII ensuring cleanup paths are correct even under exceptions.

4.3.8 Ownership Models and Their Design Meaning

C++ provides multiple ownership models. Each has architectural implications.

Value Ownership (Local, Deterministic)

Values are the default:

- best locality,
- simplest reasoning,
- avoids lifetime complexity,
- strongest cohesion.

Exclusive Ownership (`std::unique_ptr`)

Signals:

- one clear owner,
- deterministic destruction,
- movable but not copyable,
- ideal for plugin-like boundaries and polymorphism without shared lifetime.

Shared Ownership (`std::shared_ptr`)

Signals:

- lifetime shared among multiple parties,
- destruction is non-deterministic relative to any single owner,
- risk of cycles and hidden retention.

Shared ownership is often a last resort in long-lived systems; it can obscure responsibility and complicate testing.

Non-owning References and Views

References, raw pointers (as non-owning), and views (`std::span`, `string_view`) encode:

- access without lifetime control,
- explicit dependence on external owner,
- improved architectural clarity when documented.

4.3.9 The Key Architectural Move: "Ownership at the Edge, Responsibility at the Core"

A stable pattern for long-lived systems:

- **Core:** responsibility-heavy, ownership-light.
- **Edges:** ownership-heavy, responsibility-light.

Core logic should:

- focus on invariants and rules,
- avoid direct dependency on volatile infrastructure,
- be easily testable and deterministic.

Edges should:

- manage resources (threads, sockets, DB connections, files),
- adapt protocols and libraries,
- translate errors and data representations.

4.3.10 Case Study: Service Composition Root vs Domain Service

Consider a typical application boundary:

- A **composition root** creates objects and wires dependencies.
- A **domain service** applies rules.

The composition root may own lifetimes of many objects, but should not embed rules. The domain service should own the policy, but may not own long-lived resources.

Example: Composition Root Owns Lifetimes, Domain Service Owns Rules

```
#include <memory>
#include <string_view>

struct AuditSink {
  virtual ~AuditSink() = default;
  virtual void record(std::string_view e) = 0;
};
```

```
class AccountRules {
public:
  explicit AccountRules(AuditSink& audit) : audit_(audit) {}

  void deposit(int& balance, int amount) {
    if (amount <= 0) return; // policy choice
    balance += amount;
    audit_.record("deposit");
  }

private:
  AuditSink& audit_; // non-owning: responsibility uses it, does not own it
};

class App {
public:
  App(std::unique_ptr<AuditSink> audit)
    : audit_(std::move(audit)), rules_(*audit_) {}

  void run() {
    int bal = 0;
    rules_.deposit(bal, 10);
  }

private:
  std::unique_ptr<AuditSink> audit_; // ownership at the edge
  AccountRules rules_;               // responsibility in the core
};
```

This makes intent explicit:

- `App` owns lifetime of the sink (infrastructure choice).
- `AccountRules` owns the deposit policy.

4.3.11 Design Smells: When Responsibility and Ownership Are Misaligned

Smell 1: "Policy class owns threads, sockets, and caches"

A policy class that starts threads or manages sockets typically violates SRP:

- domain rule changes and runtime infrastructure changes are independent,
- mixing them creates multiple axes of change.

Smell 2: "Resource wrapper knows business semantics"

RAII wrappers should be dumb about business rules; they should enforce mechanical safety.

Smell 3: "Everything is shared_ptr"

If every dependency becomes shared ownership, it is often compensating for missing architecture:

- unclear lifetimes,
- missing composition root,
- unclear boundaries,
- fear of dangling references due to poorly defined ownership.

Smell 4: "The destructor does business work"

If destruction triggers business operations (network calls, database writes), the class likely mixes ownership mechanics with domain policy, making behavior unpredictable and hard to test.

4.3.12 Practical Checklist for Reviews

Use this checklist in code reviews:

1. **Can you name the invariant/policy?** If not, responsibility is unclear.
2. **Can you name the owner?** If not, lifetime is unclear.
3. **Are they intentionally separate?** If not, coupling is accidental.
4. **Do public headers leak heavy dependencies?** If yes, responsibility boundary is too wide.
5. **Is shared ownership used only when required?** If not, architecture is likely compensating for unclear lifetimes.
6. **Is the core testable without infrastructure?** If not, responsibility and ownership are probably mixed.

4.3.13 Summary

In Modern C++ architecture:

- **Responsibility** is about policy, decisions, invariants, and axes of change.
- **Ownership** is about lifetime, destruction, resource release, and control.
- Keeping them distinct is a primary technique for achieving SRP, high cohesion, low coupling, and testability.

- A durable pattern is: **responsibility in the stable core, ownership at the system edges**.

When responsibility and ownership are aligned intentionally (not accidentally), systems become easier to evolve, safer under concurrency, cheaper to build, and significantly more robust over long lifetimes.

4.4 Designing with Lifetimes

4.4.1 Lifetimes as a First-Class Design Dimension

In Modern C++, **lifetime design** is not a low-level implementation detail; it is a core architectural decision. Because C++ allows direct control over allocation, destruction, aliasing, and concurrency, lifetime mistakes frequently become:

- correctness failures (use-after-free, double delete, dangling references),
- security vulnerabilities (memory corruption, data exposure),
- concurrency failures (data races, invalid sharing, lifetime races),
- performance regressions (unnecessary heap churn, reference counting overhead),
- architectural brittleness (unclear ownership leading to global shared pointers).

A mature design treats lifetimes as explicit contracts:

> **Every reference implies a lifetime assumption. Good design makes those assumptions explicit, enforceable, and verifiable.**

4.4.2 The Three Questions That Drive Lifetime-Oriented Design

Any non-trivial C++ system should answer these questions systematically:

1. **Who owns it?** (exclusive, shared, or no ownership)
2. **How long must it live?** (scope, request, session, process)
3. **Who may access it concurrently?** (single thread, confined, shared with synchronization)

If a design cannot answer these clearly, it will eventually compensate with:

- pervasive `shared_ptr`,
- large object graphs with unclear roots,
- global singletons and hidden lifetimes,
- defensive copying that destroys performance.

4.4.3 Lifetime Vocabulary: Values, Owners, Borrowers, and Views

A practical lifetime model for Modern C++ design uses four roles:

- **Value (self-contained):** lifetime is bound to scope or container.
- **Owner:** controls destruction (RAII); may transfer ownership via move.
- **Borrower:** holds a non-owning reference; assumes another owner outlives it.
- **View:** a lightweight borrower that refers to a contiguous range or string data without owning (`std::span`, `std::string_view`).

This vocabulary is design-critical because it guides which types should appear at boundaries.

4.4.4 The Default Rule: Prefer Value Semantics

Value semantics is the most robust lifetime strategy:

- lifetimes are obvious and deterministic,
- ownership is unambiguous,
- exception safety is natural with RAII,
- testability improves (no shared global graphs),
- performance is often superior due to locality and fewer allocations.

Modern C++ (move semantics, NRVO, small-buffer optimizations) makes value semantics viable for many cases that historically required pointers.

Example: Value-Based Composition Root

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

struct Config {
  std::string db_host;
  int db_port{};
};

class App {
public:
  explicit App(Config cfg) : cfg_(std::move(cfg)) {}

private:
```

```
  Config cfg_; // owned by value, deterministic lifetime
};
```

Here the lifetime is trivial: `App` owns `Config` by value.

4.4.5 RAII as the Lifetime Backbone

In C++, **RAII** is the foundational lifetime discipline:

- resource acquisition is tied to initialization,
- release is tied to destruction,
- correctness holds under exceptions and early returns.

RAII is not only for memory. It applies to:

- file descriptors and handles,
- sockets and connections,
- locks and synchronization primitives,
- transactions and rollback guards,
- temporary privilege changes and security tokens.

Example: Transaction Guard

```
#include <utility>

struct DbTx {
  void commit() noexcept;
  void rollback() noexcept;
```

```
};

class TxGuard {
public:
  explicit TxGuard(DbTx& tx) : tx_(tx) {}

  void commit() noexcept {
    committed_ = true;
    tx_.commit();
  }

  ~TxGuard() noexcept {
    if (!committed_) tx_.rollback();
  }

private:
  DbTx& tx_;
  bool committed_{false};
};
```

This design makes the lifetime contract explicit:

- a transaction must be committed or rolled back,
- rollback is guaranteed even on exceptions.

4.4.6 Owning Pointers: Expressing Exclusive and Shared Lifetime

When values are not feasible (polymorphism, large graphs, stable addresses), ownership must be explicit.

Exclusive Ownership: `std::unique_ptr`

`unique_ptr` communicates:

- one owner,
- deterministic destruction,
- move-only transfer of lifetime.

Shared Ownership: `std::shared_ptr`

`shared_ptr` communicates:

- lifetime shared among multiple parties,
- destruction tied to reference count reaching zero,
- potential for cycles (requires `weak_ptr`),
- non-trivial overhead and more complex reasoning.

A durable guideline for long-lived systems:

> **Use shared ownership only when multiple independent parts of the system truly co-own the same lifetime.**

4.4.7 Borrowing and Non-Owning Access: References, Raw Pointers, and Views

Non-owning access is essential for performance and clear architecture, but must be explicit:

- a reference signals non-null and borrowed access,
- a raw pointer can signal optional borrowed access (nullable),
- views signal borrowed ranges and must never outlive the underlying storage.

Example: Borrowed Access with Explicit Nullability

```
#include <string_view>

struct Logger {
  void info(std::string_view) noexcept;
};

class Service {
public:
  explicit Service(Logger* log) : log_(log) {} // nullable, borrowed

  void run() noexcept {
    if (log_) log_->info("running");
  }

private:
  Logger* log_; // non-owning; caller must ensure lifetime
};
```

The lifetime obligation is on the caller; the class does not own the logger.

4.4.8 Lifetime-Aware Interfaces: Make Ownership Transfer Obvious

API boundaries should make lifetime semantics unmistakable.

Preferred Patterns

- Accept by value when you intend to take ownership (`T`).
- Accept by `std::unique_ptr<T>` when ownership transfer is required.

- Accept by reference (`T&`) or pointer (`T*`) for borrowing.
- Return values by value when possible; modern C++ optimizes this well.
- Return `unique_ptr` to transfer ownership to the caller.

Example: Ownership-Explicit Factory and Consumer

```
#include <memory>
#include <string>

struct Widget {
  explicit Widget(std::string name) : name(std::move(name)) {}
  std::string name;
};

std::unique_ptr<Widget> make_widget(std::string name) {
  return std::make_unique<Widget>(std::move(name)); // ownership returned
}

class WidgetUser {
public:
  void set_widget(std::unique_ptr<Widget> w) { // ownership taken
    w_ = std::move(w);
  }

  Widget* widget() noexcept { return w_.get(); } // borrowed access

private:
  std::unique_ptr<Widget> w_;
};
```

This interface forces correct lifetime reasoning.

4.4.9 Lifetime at Architecture Scale: Scopes and Roots

In long-lived systems, lifetimes naturally align to architectural scopes. A useful classification:

- **Scope lifetime:** local computations; stack objects and temporaries.
- **Object lifetime:** owned members within an aggregate; destroyed with the owner.
- **Request lifetime:** per user request, per job, per command.
- **Session lifetime:** authenticated user session, connection session.
- **Process lifetime:** composition root objects, services, caches.

A key design rule:

> **Make lifetime roots explicit. Hidden roots create hidden leaks and hidden coupling.**

Composition Root as the Lifetime Owner

A composition root (often `main`) typically owns process-lifetime objects and wires dependencies. Policy should not live there; lifetime management can.

```
#include <memory>

struct AuditSink { virtual ~AuditSink() = default; };
struct FileAuditSink : AuditSink {};

struct Rules {
  explicit Rules(AuditSink& a) : audit(a) {}
```

```
  AuditSink& audit; // borrowed; rules are responsibility-heavy, ownership-light
};

int main() {
  auto audit = std::make_unique<FileAuditSink>(); // owned at the root
  Rules rules{*audit};                            // borrows dependency
  (void)rules;
}
```

4.4.10 Lifetimes and Concurrency: Avoid Lifetime Races

Concurrency introduces a unique category of bug: **lifetime races**. Even if an object is not used-after-free in a single-thread model, concurrent access can free it while another thread still holds a reference.
A disciplined approach is to combine:

- clear ownership roots,
- confinement (thread-local ownership where possible),
- synchronized shared access when required,
- explicit handoff semantics (move-only messages, queues).

Message Passing with Move-Only Ownership (Handoff)

```
#include <memory>
#include <queue>
#include <mutex>

struct Job {
```

```
  virtual ~Job() = default;
  virtual void run() = 0;
};

class JobQueue {
public:
  void push(std::unique_ptr<Job> j) {
    std::lock_guard<std::mutex> lk(m_);
    q_.push(std::move(j)); // ownership handoff into the queue
  }

  std::unique_ptr<Job> pop() {
    std::lock_guard<std::mutex> lk(m_);
    if (q_.empty()) return {};
    auto j = std::move(q_.front());
    q_.pop();
    return j; // ownership handoff out of the queue
  }

private:
  std::mutex m_;
  std::queue<std::unique_ptr<Job>> q_;
};
```

This pattern is lifetime-safe:

- no shared ownership by default,
- explicit transfer across boundaries,
- minimal aliasing.

4.4.11 Aliasing as a Lifetime Risk: Design to Reduce It

Even with correct ownership, **aliasing** creates risk:

- multiple references to the same mutable object complicate invariants,
- shared mutation amplifies concurrency hazards,
- observers may outlive the observed unless carefully structured.

A design that reduces aliasing typically:

- uses value semantics for immutable or stable data,
- uses unique ownership for mutable state,
- uses views for read-only access within known bounds,
- avoids exposing raw internal pointers from owners.

4.4.12 Lifetime-Oriented Error Safety: The Strong Exception Guarantee as a Design Tool

Lifetime design and error handling are inseparable in C++:

- RAII ensures cleanup under exceptions.
- Resource acquisition should complete fully or fail without side effects.
- Mutations should preserve invariants even if an exception occurs mid-operation.

A practical guideline:

> **Construct fully valid objects, then publish them. Avoid partially initialized objects escaping.**

4.4.13 Design Smells That Indicate Lifetime Problems

Smell 1: "Everything is shared_ptr"

Often signals unclear ownership roots and fear of dangling references.

Smell 2: "Raw owning pointers"

Owning raw pointers typically indicates missing RAII and unclear destruction responsibility.

Smell 3: "Factories returning raw pointers"

This hides ownership transfer and makes leaks likely.

Smell 4: "Destructors with side effects"

Business logic in destructors makes behavior unpredictable and fragile.

Smell 5: "Long-lived references to short-lived storage"

Common with `string_view` and `span` when views escape their intended scope.

4.4.14 A Practical Design Checklist: Designing with Lifetimes

Use this checklist during reviews:

1. **Ownership is explicit:** values, `unique_ptr`, `shared_ptr` only where justified.
2. **Borrowing is explicit:** references for non-null, pointers for optional, views for ranges.
3. **Lifetimes align with scopes:** request/session/process lifetimes are clear and rooted.
4. **No hidden global roots:** minimize singletons; make long-lived owners visible.

5. **Aliasing is minimized:** avoid exposing mutable shared state widely.
6. **Concurrency is respected:** no lifetime races; prefer handoff via move-only messages.
7. **Exception safety is engineered:** RAII everywhere, invariants preserved on failure.

4.4.15 Summary

Designing with lifetimes is a central discipline in Modern C++ architecture:

- Lifetimes define the true boundaries of safety and responsibility.
- RAII and value semantics are the default tools for correctness and clarity.
- Ownership must be explicit; borrowing must be explicit; views must remain bounded.
- Concurrency requires designing against lifetime races through confinement and ownership handoff.
- Clear lifetime roots and stable interfaces produce systems that remain maintainable for years.

When lifetimes are designed intentionally, the system becomes easier to reason about, safer under change, more testable, and dramatically more resilient in production.

Coupling, Dependencies, and Direction

5.1 Physical vs Logical Dependencies

5.1.1 Two Kinds of Dependency Every Long-Lived C++ System Must Control

In large systems, the word *dependency* hides two distinct phenomena:

- **Logical dependencies** are *semantic*: one policy, concept, or abstraction relies on another concept to make sense.
- **Physical dependencies** are *mechanical*: one translation unit, header, module, or binary artifact must be present to compile, link, load, or run another.

A long-lived architecture succeeds when it aligns these:

> **The physical dependency graph should approximate the logical dependency graph.**

Failure to align them creates a familiar crisis:

- logically unrelated changes trigger wide rebuilds,
- a "stable" subsystem becomes hard to modify due to build coupling,
- dependency direction becomes unclear or inverted,
- modularity claims collapse under include graphs and linkage realities.

5.1.2 Logical Dependencies: The “Needs” Relationship

A logical dependency exists when one part of the design relies on another to define correct behavior.
Examples:

- A domain rule logically depends on a concept such as `Money` or `AccountId`.
- A sorting algorithm logically depends on the concept “`LessThanComparable`” rather than a specific container.
- An authorization policy logically depends on the concept “current principal” and “permission model”.

Logical dependencies should be expressed through:

- stable abstractions and contracts,
- small vocabulary types (strong types, value objects),
- interfaces and concepts,
- explicit preconditions and invariants.

In a mature architecture:

- logical dependencies are intentional and documented,
- dependency direction is decided to control change ripple effects.

5.1.3 Physical Dependencies: The “Must Include/Link/Load” Relationship

A physical dependency exists when the compiler, linker, or runtime requires another artifact.
In C++, physical dependencies appear in multiple layers:

- **Preprocessor/include dependencies:** headers required for declarations and definitions.
- **Template and inline dependencies:** code must be visible at compile time.
- **Link dependencies:** object files and libraries required to resolve symbols.
- **Runtime dependencies:** dynamic libraries, plugins, and resource files required at load or run.
- **Toolchain dependencies:** compiler options, ABI mode, standard library configuration.

Physical dependency costs in long-lived systems:

- long incremental build times,
- fragile build graphs,
- accidental ODR violations,
- ABI breakage across libraries,
- difficulty extracting modules, components, or microservices.

5.1.4 The Key Insight: Logical Decoupling Can Be Defeated by Physical Coupling

You can have an elegant logical design and still fail in practice if physical dependencies are unmanaged. Two classic examples in C++:

Example 1: "Logically clean" domain type with physically heavy headers

A domain type that includes JSON, SQL, networking headers in its public header becomes physically coupled to volatile dependencies. This causes rebuild storms and makes the domain layer unstable.

Example 2: Templates that leak internal implementation

Templates are logically powerful but physically expensive:

- every include becomes a compilation dependency,
- every client instantiates code, increasing compile time,
- internal changes can force widespread recompilation.

5.1.5 Dependency Direction: Logical Direction vs Physical Direction

In a healthy system, both logical and physical dependency direction should follow the architecture:

- high-level policies should not depend on low-level details,
- stable core should not depend on volatile frameworks,
- infrastructure should depend on domain contracts, not the other way around.

But C++ makes it easy to invert physical direction accidentally:

- including a framework header in a domain header,
- embedding a third-party type as a member in a core type,
- exposing templates that require heavy implementation headers everywhere.

5.1.6 A Mental Model: "Concept Graph" vs "Build Graph"

You can think of two graphs:

- **Concept graph (logical):** nodes are concepts, policies, abstractions; edges represent semantic need.

- **Build graph (physical):** nodes are headers, translation units, libraries; edges represent include/link requirements.

Good architecture keeps the build graph thin, layered, and stable, while the concept graph remains expressive.

5.1.7 Common C++ Mechanisms That Create Physical Dependencies

1) Header Includes

The simplest and most expensive mechanism. Every include is a physical edge that:

- expands the compilation boundary,
- imports macros and configuration,
- increases rebuild scope.

2) Inline Functions

Inlining in headers makes definitions visible everywhere and turns implementation into a physical dependency.

3) Templates

Templates are the ultimate physical dependency amplifier. They are necessary, but they must be managed carefully:

- keep template interfaces narrow,
- keep template bodies small,
- push heavy work into non-template helpers when possible.

4) Macros and Configuration Headers

Macros are invisible physical dependencies:

- they change meaning across translation units,
- they create subtle ODR and ABI risks,
- they defeat locality and reasoning.

5) Link-Time Dependencies

Even if headers are clean, linking can impose coupling:

- static libraries pull in transitive dependencies,
- symbol visibility and link order matter in some environments,
- ABI compatibility becomes a long-term constraint.

5.1.8 Engineering Strategies to Align Physical With Logical Dependencies

Strategy 1: Keep Public Headers Minimal and Stable

A practical rule:

> **Your public headers define your physical architecture. Keep them small, stable, and concept-focused.**

Techniques:

- forward declare when possible,
- avoid including third-party headers in stable interfaces,
- prefer references/pointers or abstract interfaces to heavy member types.

Strategy 2: Use PIMPL for ABI and Dependency Control

PIMPL makes implementation a private physical dependency while preserving a stable logical contract.

```cpp
#include <memory>
#include <string>

class Parser {
public:
  Parser();
  ~Parser();
  Parser(Parser&&) noexcept;
  Parser& operator=(Parser&&) noexcept;

  Parser(const Parser&) = delete;
  Parser& operator=(const Parser&) = delete;

  bool parse(std::string_view input);

private:
  struct Impl;
  std::unique_ptr<Impl> p_; // hides heavy headers and volatile dependencies
};
```

With this:

- logical dependency: clients depend on the parsing contract,
- physical dependency: clients do not depend on parser internals.

Strategy 3: Separate Interface and Implementation Libraries

Split artifacts:

- **interface target:** headers, small stable types, contracts,
- **implementation target:** heavy code, third-party dependencies.

This reduces physical blast radius while preserving logical structure.

Strategy 4: "Stable Core, Volatile Edge" Dependency Rule

Make the stable core depend only on:

- standard library,
- stable internal contracts,
- small value types.

Push volatile dependencies to the edge:

- frameworks,
- database drivers,
- serialization libraries,
- OS- and vendor-specific APIs.

Strategy 5: Prefer Type Erasure or Virtual Interfaces at Boundaries

To prevent template-heavy dependencies from leaking to clients, use boundary techniques.

```cpp
#include <memory>
#include <string_view>

struct Sink {
  virtual ~Sink() = default;
  virtual void write(std::string_view msg) = 0;
};

class Logger {
public:
  explicit Logger(std::shared_ptr<Sink> s) : sink_(std::move(s)) {}

  void info(std::string_view msg) {
    sink_->write(msg);
  }

private:
  std::shared_ptr<Sink> sink_;
};
```

This keeps:

- logical dependency: logger depends on “sink contract”,
- physical dependency: logger clients do not need sink implementation headers.

5.1.9 Recognizing Misalignment: Practical Smells

Smell 1: “One-line change triggers a massive rebuild”

This indicates physical dependency coupling, often due to:

- heavy common headers,
- inline/template expansion in public headers,
- transitive include chains.

Smell 2: "Domain layer includes framework headers"

Logical layering is violated physically.

Smell 3: "Every change in a private detail is an ABI break"

Public exposure of implementation types creates a physical contract that is too wide.

Smell 4: "Macros determine behavior across modules"

Hidden physical coupling defeats logical boundaries.

5.1.10 Design Checklist: Physical vs Logical Dependencies

Use the following checklist in reviews:

1. **Logical clarity:** Can you name why the dependency exists in terms of policy/contract?
2. **Physical necessity:** Does the dependent code truly need this header, or only a forward declaration?
3. **Public surface:** Are heavy/volatile headers avoided in public interfaces?
4. **Template containment:** Are templates constrained to where they provide real value?
5. **Artifact layering:** Do stable libraries avoid depending on volatile libraries?
6. **Build hygiene:** Would changing a private detail avoid forcing unrelated rebuilds?

5.1.11 Summary

Logical dependencies describe how concepts and policies relate. **Physical dependencies** describe compilation, linking, and runtime constraints.
Long-lived Modern C++ systems require both to be engineered deliberately:

- Align the build graph with the concept graph.
- Keep public headers minimal and stable.
- Push volatile dependencies to the edges.
- Use PIMPL, interface/implementation splits, and boundary abstractions to shrink physical coupling.
- Treat templates and inlining as powerful tools with physical costs that must be paid consciously.

When physical and logical dependencies are aligned, architecture becomes enforceable, rebuilds become predictable, and change becomes affordable over years of system evolution.

5.2 Dependency Direction and Stability

5.2.1 Why Dependency Direction Is an Architectural Control Lever

In long-lived systems, most engineering cost is not in writing code the first time, but in **changing** code safely over years. Dependency direction determines how far changes propagate. When dependency direction is engineered correctly:

- stable code stays stable,
- volatile code absorbs change,

- core policies are protected from framework churn,
- builds scale (fewer rebuild storms),
- testing becomes cheaper (stable units test in isolation).

In Modern C++ specifically, dependency mistakes are amplified by:

- include graphs and template instantiation,
- ABI and binary compatibility constraints,
- strong coupling through headers and inline code,
- concurrency and lifetime semantics crossing boundaries.

A practical restatement:

> **Dependency direction is the primary mechanism by which you decide where change is allowed to live.**

5.2.2 Stability: What It Really Means in Engineering Terms

A unit is **stable** if it changes rarely and is difficult or expensive to change. Stability can be driven by:

- wide usage (many dependents),
- hard compatibility constraints (ABI, API contracts),
- high assurance requirements (security, safety, regulatory),
- deep foundational role (types, invariants, domain vocabulary).

A unit is **volatile** if it changes frequently due to:

- integration with external systems,
- evolving protocols and formats,
- UI, UX, and product iteration,
- performance tuning and operational feedback,
- vendor and framework upgrades.

Good architecture aligns dependencies so that:

> **volatile units depend on stable units, not the other way around.**

5.2.3 The Core Law: Dependencies Should Point Toward More Stable Code

The key design law is simple:

- If A depends on B, then changes in B can affect A.
- Therefore, A should depend on things that are **more stable** than A.

When the direction is inverted (stable depends on volatile), you get:

- ripple effects: a framework upgrade forces core redesign,
- build storms: changes in edge headers recompile core libraries,
- brittle releases: operational changes require core changes,
- testing collapse: stable units become untestable without infrastructure.

5.2.4 Two Kinds of Direction in C++: Logical and Physical

In C++ you must manage direction in two graphs:

- **Logical direction:** concepts, policies, and abstractions depend on other concepts.
- **Physical direction:** headers, templates, and link dependencies determine what rebuilds and what must ship together.

A stable architecture requires both to align.

5.2.5 Stable Core vs Volatile Edge: A Practical Layering Model

A pragmatic model for modern systems:

- **Core domain (most stable):** vocabulary types, invariants, business rules.
- **Application services (stable):** use cases and orchestration policy.
- **Adapters (volatile):** database, HTTP, serialization, message brokers.
- **Frameworks/IO (most volatile):** OS APIs, third-party libraries, network stacks.

The direction rule:

> **Dependencies must point inward: edges depend on core contracts, not vice versa.**

In C++ terms, that implies:

- core headers must not include database or web framework headers,
- core must not embed framework-owned types in public interfaces,
- adapters translate between core types and external representations.

5.2.6 The Dependency Inversion Move: Make Stable Policy Independent of Volatile Details

The essential technique is to introduce a stable abstraction and make volatile details implement it.

Example: Stable Policy Depends on a Stable Interface

```
#include <string_view>

struct Clock {
  virtual ~Clock() = default;
  virtual long long now_unix_seconds() const noexcept = 0;
};

class SessionPolicy {
public:
  explicit SessionPolicy(const Clock& clock) : clock_(clock) {}

  bool expired(long long issued_at, long long ttl_seconds) const noexcept {
    // Responsibility: expiration policy (stable)
    const auto now = clock_.now_unix_seconds();
    return (now - issued_at) >= ttl_seconds;
  }

private:
  const Clock& clock_; // borrowed; stable policy does not own volatile mechanism
};
```

Now the volatile part (system time source, platform API) can change without changing the policy.

Volatile Adapter Implements the Interface

```
#include <chrono>

struct SystemClock final : Clock {
  long long now_unix_seconds() const noexcept override {
    using namespace std::chrono;
    return duration_cast<seconds>(system_clock::now().time_since_epoch()).count();
  }
};
```

5.2.7 Stability and Build-Time Architecture: Headers Are Dependencies

In C++, build-time architecture is real architecture. A stable core is not stable if:

- its public headers include volatile headers,
- it contains templates that require volatile definitions to be visible,
- it exposes inline functions that depend on volatile details.

Therefore, stability engineering includes:

- minimal public headers,
- forward declarations where feasible,
- PIMPL when ABI and dependency hiding matter,
- interface/implementation library splits.

5.2.8 Stability as "Change Cost": Fan-In and Fan-Out Intuition

Even without formal metrics, engineers can reason about stability:

- **High fan-in** (many depend on it) ⇒ expensive to change ⇒ must be stable.
- **High fan-out** (it depends on many) ⇒ likely volatile and fragile.

A common failure is a unit with both:

- high fan-in (widely used),
- and high fan-out (depends on many external libs).

That unit becomes a systemic risk: it is both hard to change and constantly forced to change.

5.2.9 Stability Patterns That Work in Modern C++

Pattern 1: Policy Classes Are Ownership-Light

Stable policies should:

- avoid owning resources (threads, sockets, connections),
- accept collaborators by reference/pointer,
- remain deterministic and testable.

Pattern 2: Composition Root Owns Lifetimes

The composition root (startup wiring) is where ownership concentrates. It is allowed to depend on volatile details because it is, by nature, volatile.

Pattern 3: Adapters Translate at the Boundary

Adapters:

- depend on frameworks,
- know protocols and formats,
- translate to/from core vocabulary types,
- isolate churn and upgrades.

Pattern 4: Facade to Contain Third-Party Spread

When a third-party library must be used widely, wrap it in a local facade:

- reduces physical dependency spread,
- centralizes upgrade work,
- protects the stable core from vendor details.

5.2.10 Anti-Patterns: How Direction and Stability Fail

Anti-Pattern 1: Core Depends on Framework

Symptoms:

- core types embed framework types as members,
- core headers include framework headers,
- domain logic cannot compile without the framework.

Consequence:

- framework upgrades become architecture changes,
- testing requires infrastructure,
- core becomes volatile.

Anti-Pattern 2: “Shared_ptr Everywhere” as a Stability Patch

This often appears when lifetimes are unclear across layers. It increases coupling and hides direction issues:

- lifetimes become global,
- destruction becomes unpredictable,
- responsibility and ownership blur across boundaries.

Anti-Pattern 3: “Common” Library as a Dependency Dump

- everything depends on it (high fan-in),
- it depends on everything (high fan-out),
- it becomes impossible to evolve safely.

5.2.11 A Concrete Rule Set for C++ Teams

The following rules enforce dependency direction and stability:

1. **No framework headers in core headers.**
2. **No third-party types in core public interfaces.**
3. **Policies depend on abstractions, not mechanisms.**

4. **Ownership is concentrated at edges (composition root).**
5. **Adapters translate between external formats and core types.**
6. **Templates in core interfaces must be small and stable.**
7. **Prefer forward declarations and stable value types in public headers.**

5.2.12 Worked Example: Inverting Direction to Stabilize the Core

Suppose we have a policy that needs persistence.

Stable abstraction

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

struct UserRecord {
  std::string name;
  int age{};
};

struct UserRepo {
  virtual ~UserRepo() = default;
  virtual std::optional<UserRecord> find_user(std::string_view name) = 0;
  virtual void save_user(const UserRecord& u) = 0;
};

class UserRules {
public:
  explicit UserRules(UserRepo& repo) : repo_(repo) {}
```

```
  void birthday(std::string_view name) {
    auto u = repo_.find_user(name);
    if (!u) return;
    ++u->age;              // stable policy
    repo_.save_user(*u);   // stable contract
  }

private:
  UserRepo& repo_;
};
```

Volatile adapter (e.g., SQL implementation) lives outside

The SQL library and schema details do not appear in `UserRules`. They are confined to an adapter that implements `UserRepo`. This preserves:

- stable core rules,
- volatile infrastructure changes localized to adapters,
- physical dependency containment.

5.2.13 Review Checklist: Direction and Stability

Use this checklist during architecture reviews:

1. **Direction:** Do dependencies point from volatile to stable?
2. **Core purity:** Can the core compile without external frameworks?
3. **Public surfaces:** Are core headers minimal and stable?

4. **Fan-in/fan-out:** Are highly depended-on units low in dependencies?
5. **Adapters:** Are protocols, formats, and drivers isolated?
6. **Ownership:** Are lifetimes owned at the edges, not in policies?
7. **Testability:** Can core policy be tested without infrastructure?

5.2.14 Summary

Dependency direction and stability are inseparable in long-lived Modern C++ systems:

- direction determines where change pressure lands,
- stable units must depend only on more stable units,
- the core must be protected from volatile frameworks through inversion and adapters,
- physical dependency control (headers, templates, ABI surfaces) is essential for real stability,
- clear ownership roots and clean boundaries prevent lifetime and concurrency coupling from undermining direction.

When dependency direction is engineered as a first-class constraint, architecture becomes enforceable, change becomes affordable, and the system remains evolvable over years of continuous requirements and platform evolution.

5.3 Header Dependencies vs Module Dependencies

5.3.1 Why This Comparison Matters in Long-Lived C++ Systems

Modern C++ systems live or die by dependency control. For decades, C++ has relied on **headers** and the preprocessor to share declarations and definitions. This model scales functionally, but it

scales poorly in large codebases because it creates:

- large transitive include graphs,
- fragile macro-driven configuration,
- long incremental build times,
- accidental ODR violations through multiple definitions,
- architectural erosion where physical coupling defeats logical layering.

C++ **modules** were introduced to provide a language-level mechanism for dependency management:

- declarations are *imported*, not textually included,
- macros do not automatically leak across boundaries,
- the compiler can build and cache module interfaces,
- dependency graphs become clearer and less accidental.

This section compares **header dependencies** and **module dependencies** in engineering terms and provides practical guidance for designing stable dependency direction in a mixed world where many libraries still ship as headers.

5.3.2 Header Dependencies: Textual Inclusion and Its Consequences

What a Header Dependency Really Is

A header dependency means:

- the preprocessor textually inserts the header content into the translation unit,

- all transitive includes become part of the compilation unit,
- macros and configuration may change meaning based on include order,
- inline and template definitions become visible, expanding compile-time coupling.

Engineering Consequences of Header Dependencies

- **Transitive growth:** Including one header often includes dozens or hundreds.
- **Build instability:** Small changes in a common header can trigger widespread rebuilds.
- **Macro leakage:** Macros can unintentionally affect unrelated code.
- **ODR sensitivity:** Multiple definitions can occur due to inconsistent macros/configuration.
- **Weak boundaries:** The include graph often differs from the intended architecture.

Header Hygiene as Dependency Engineering

Header-based systems must follow strict hygiene:

- include only what you use,
- minimize includes in public headers,
- prefer forward declarations when valid,
- avoid macros in public headers,
- keep inline/template bodies small in stable interfaces,
- isolate third-party includes in implementation files.

5.3.3 Module Dependencies: Import-Based Semantics and Stronger Boundaries

What a Module Dependency Really Is

A module dependency means:

- a translation unit **imports** a compiled module interface,
- names become visible without textual inclusion,
- the compiler can cache and reuse compiled interface artifacts,
- macros and textual order effects are largely contained.

Engineering Consequences of Module Dependencies

- **Reduced transitive surface:** importing does not recursively paste text.
- **Faster builds (when supported):** compiled interfaces can reduce repeated parsing.
- **Cleaner boundaries:** export lists define what is public.
- **Less macro interference:** module interfaces are typically insulated from macro pollution.
- **Better architecture enforcement:** import graphs can more closely mirror design graphs.

5.3.4 Conceptual Mapping: Include Graph vs Import Graph

Include Graph (Headers)

- edges are created by `#include`,
- edges are often accidental and transitive,

- order matters,
- textual substitution expands compilation units.

Import Graph (Modules)

- edges are created by `import`,
- edges are more explicit and intentional,
- exported names form a stable contract,
- compiled interface units reduce repeated parsing.

5.3.5 Code Illustration: Headers vs Modules (Same Intent, Different Physics)

Header-based interface

```
// math_api.hpp
#pragma once
#include <cstdint>

struct Vec2 {
  float x{};
  float y{};
};

float dot(Vec2 a, Vec2 b) noexcept;
```

```
// math_api.cpp
#include "math_api.hpp"
```

```
float dot(Vec2 a, Vec2 b) noexcept {
  return a.x * b.x + a.y * b.y;
}
```

Physical dependency:

- every includer re-parses `math_api.hpp` and its transitive includes,
- macros from includers may influence included headers if they are not robust.

Module-based interface

```
// math_api.ixx (module interface unit)
export module math_api;
export struct Vec2 {
  float x{};
  float y{};
};
export float dot(Vec2 a, Vec2 b) noexcept;
```

```
// math_api.cpp (module implementation unit)
module math_api;

float dot(Vec2 a, Vec2 b) noexcept {
  return a.x * b.x + a.y * b.y;
}
```

Consumer:

```
import math_api;

float f() {
```

```
  return dot(Vec2{1,2}, Vec2{3,4});
}
```

Physical dependency:

- consumer imports a compiled interface,
- the compiler does not need to textually include the interface,
- exported names define the public surface precisely.

5.3.6 Where Modules Improve Coupling and Direction

1) Public Surface Becomes Explicit

Headers often expose more than intended:

- helper types,
- private macros,
- transitive dependencies.

Modules encourage a deliberate contract:

- export only what clients need,
- keep internal helpers non-exported.

2) Macro Containment Improves Architectural Predictability

In header ecosystems, macro collisions and configuration switches can leak across the project. Modules reduce this risk by isolating interface compilation from consumer macro state.

3) Reduced Accidental Coupling Through Transitive Includes

Headers frequently force unrelated dependencies into a client because:

- a public header includes another public header for convenience,
- templates require definitions,
- legacy patterns rely on “umbrella headers.”

Modules reduce the need for umbrella headers and can make dependencies more intentional.

4) Better Build Scalability (When Tooling Is Ready)

A core engineering motivation for modules is improved build throughput:

- repeated parsing of the same headers is reduced,
- compiled interfaces can be cached across translation units,
- dependency edges can be processed more efficiently.

5.3.7 Where Headers Remain Dominant (Practical Reality)

Despite the advantages of modules, real systems still rely heavily on headers because:

- most third-party libraries are header-based,
- many ecosystems are template-heavy (generic libraries),
- toolchains and build systems may have uneven module support across platforms,
- legacy codebases require long migration periods.

Therefore, long-lived systems often adopt a **hybrid architecture**:

- stable internal components gradually become modules,
- third-party dependencies remain headers but are isolated behind adapters,
- public API surfaces are carefully constrained.

5.3.8 Design Guidance: How to Engineer Dependency Direction in a Hybrid World

Guideline 1: Treat Public Headers Like Public Modules

Even if you are not using modules, design headers as if they were module interfaces:

- minimal includes,
- minimal exported surface,
- stable vocabulary types only,
- no volatile dependencies.

Guideline 2: Hide Volatile Dependencies Behind Non-Template Boundaries

If a volatile third-party library is header-heavy:

- confine it in `.cpp` files,
- wrap it behind a stable interface,
- expose only stable types across the boundary.

Guideline 3: Keep Template Interfaces Narrow

Templates create unavoidable physical dependencies. Use them where they provide true value, but:

- avoid templates in high-fan-in (highly depended-on) headers,
- push heavy work into non-template helpers,
- prefer type-erasure or virtual interfaces at architectural boundaries.

Guideline 4: Use Modules to Stabilize the Core First

If you adopt modules, prioritize:

- core vocabulary and domain contracts,
- stable utilities with high fan-in,
- frequently included interfaces.

This yields the most benefit because:

- the same interfaces are imported repeatedly,
- build caching has a large effect,
- physical coupling is reduced in the highest-impact region.

5.3.9 Migration Strategy: From Headers to Modules Without Breaking the System

A realistic migration approach:

1. **Stabilize headers first:** clean interfaces, reduce includes, remove macro leakage.

2. **Identify high fan-in headers:** these produce the most rebuild cost.
3. **Create module interfaces for stable APIs:** export only the needed surface.
4. **Keep adapters at edges:** isolate third-party headers behind translation layers.
5. **Avoid exposing module-internal types accidentally:** export deliberately.

5.3.10 Smells and Failure Modes

Smell 1: "Umbrella header" becomes the architecture

If all clients include a single mega-header, physical coupling defeats logical direction.

Smell 2: "Public templates everywhere"

Public templates force definitions into consumer compilation units. This can dominate build time and widen physical dependency graphs.

Smell 3: "Macro configuration controls behavior"

If behavior depends on preprocessor state at include sites, architecture becomes fragile and hard to reason about.

Smell 4: "Module boundaries mirror directory boundaries, not design boundaries"

Modules should represent architectural contracts, not filesystem convenience.

5.3.11 Review Checklist: Header vs Module Dependencies

Use this checklist during architecture reviews:

1. **Is the public surface explicit and minimal?**

2. **Are volatile dependencies excluded from stable interfaces?**
3. **Do dependency edges reflect design layering?**
4. **Are templates confined to where they provide essential value?**
5. **Are macros prevented from leaking across contracts?**
6. **Will a small change trigger broad rebuilds? If yes, why?**

5.3.12 Summary

Header dependencies are textual and transitive; they expand compilation units, leak macros, and can erase architectural boundaries if not controlled. **Module dependencies** are import-based and contract-driven; they strengthen boundaries, reduce accidental coupling, and can improve build scalability when toolchains support compiled interfaces effectively.
For long-lived Modern C++ systems, the practical stance is:

- engineer physical dependency direction as seriously as logical direction,
- design headers as disciplined contracts even before adopting modules,
- adopt modules first in stable, high fan-in parts of the system,
- isolate third-party header ecosystems behind adapters to protect stability.

5.4 Designing for Replacement

5.4.1 Replacement as the Real Definition of "Maintainability"

In long-lived systems, the most expensive changes are rarely feature additions; they are **replacements**:

- replacing a database engine,
- replacing a serialization format,
- replacing a network stack or protocol,
- replacing a UI framework,
- replacing an authentication provider,
- replacing a logging/telemetry pipeline,
- replacing an entire subsystem that has become unmaintainable.

A system is genuinely maintainable when it can survive these replacements without a rewrite. Therefore, a practical definition:

> **Design for replacement means making future substitutions cheaper than future repairs.**

In Modern C++, replacement design is inseparable from dependency direction:

- stable policies must not depend on volatile details,
- physical dependencies (headers, templates, ABI surfaces) must not leak replacement candidates into stable interfaces,
- lifetimes and ownership must remain local to replaceable subsystems.

5.4.2 The Replacement Contract: What Must Remain Stable

To design for replacement, you must identify what remains stable under change. In most systems, the stable parts are:

- **domain vocabulary** (core types and invariants),
- **use-case contracts** (what the system must accomplish),
- **semantic guarantees** (correctness rules, security properties, ordering, idempotency),
- **observability expectations** (what must be logged/recorded, not how).

The replaceable parts are typically:

- **mechanisms** (drivers, frameworks, protocols),
- **vendors** (third-party libraries, cloud services),
- **formats** (JSON, Protobuf, custom binary),
- **infrastructure** (SQL engines, message brokers, caches).

Designing for replacement means:

> **Keep stable meaning in the core. Keep replaceable mechanism at the edges.**

5.4.3 Two Preconditions for Replaceability

1) A Stable Boundary

A replaceable subsystem must be behind a boundary that is:

- narrow (few operations),
- semantically rich (expresses intent),
- independent of vendor-specific types,
- stable across years.

2) Dependency Inversion

The stable core must depend on an abstraction that represents the needed capability, while the replaceable subsystem implements it.

5.4.4 A Practical Architecture Pattern: Ports and Adapters (Through C++)

A widely effective replacement-oriented structure is:

- **Port:** a stable interface owned by the core (what the core needs).
- **Adapter:** a replaceable implementation owned by the edge (how it is done).

In Modern C++ this is reinforced by:

- clean header boundaries,
- type erasure or interfaces to avoid template leakage,
- RAII wrappers to contain resource lifetimes inside adapters,
- translation layers that prevent third-party types from reaching the core.

5.4.5 Example: Designing a Storage Subsystem for Replacement

Stable core vocabulary

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

struct UserId {
  std::string value;
```

```
};

struct User {
  UserId id;
  std::string name;
  int age{};
};
```

Port owned by the core

```
#include <vector>

struct UserRepo {
  virtual ~UserRepo() = default;

  virtual std::optional<User> find(UserId id) = 0;
  virtual void save(const User& u) = 0;

  // contract is about intent, not about SQL/NoSQL details
  virtual std::vector<User> find_by_name(std::string_view prefix) = 0;
};
```

Core policy depends only on the port

```
#include <stdexcept>

class UserRules {
public:
  explicit UserRules(UserRepo& repo) : repo_(repo) {}
```

```
  void rename(UserId id, std::string new_name) {
    if (new_name.empty()) throw std::invalid_argument("empty name");
    auto u = repo_.find(id);
    if (!u) return;

    u->name = std::move(new_name); // stable policy
    repo_.save(*u);                // stable boundary
  }

private:
  UserRepo& repo_;
};
```

At this point, replacement is cheap:

- replace SQL with another SQL engine,
- replace SQL with NoSQL,
- replace local file storage with remote service,
- without changing `UserRules`.

5.4.6 Preventing Replacement Failure: Do Not Leak Vendor Types

Replaceability fails when vendor types cross the boundary:

- core types contain framework types as members,
- public interfaces accept/return third-party containers,
- exceptions and error types are vendor-specific across the boundary,

- templates force clients to include vendor headers.

A strict rule for long-lived systems:

> **Replaceable subsystems must not appear in core public headers.**

5.4.7 Replacement at the Physical Level: Headers, ABI, and Build Graph

Even if the logical boundary is clean, replacement can be expensive if physical coupling leaks:

- core headers include adapter headers “for convenience,”
- templates in the core require adapter definitions,
- ABI depends on third-party layout types.

Practical containment tools:

- **PIMPL:** hide implementation and third-party headers.
- **Interface/implementation split:** keep stable interfaces separate from volatile code.
- **Forward declarations:** reduce transitive include dependencies.
- **Local facades:** wrap third-party libraries behind a narrow internal API.

5.4.8 Designing for Replacement of Cross-Cutting Concerns

Some subsystems are not “components” but cross-cutting concerns:

- logging and auditing,
- metrics and tracing,
- configuration sources,

- security and authentication,
- time source,
- randomness source.

These must be replaceable because operational requirements change.

Example: Replaceable Observability Sink

```
#include <string_view>

struct AuditSink {
  virtual ~AuditSink() = default;
  virtual void record(std::string_view event) = 0;
};

class MoneyTransferPolicy {
public:
  explicit MoneyTransferPolicy(AuditSink& audit) : audit_(audit) {}

  void transfer(int& from, int& to, int amount) {
    if (amount <= 0) return;
    if (from < amount) return;
    from -= amount;
    to   += amount;
    audit_.record("transfer");
  }

private:
  AuditSink& audit_; // stable core depends on abstraction, not on log framework
```

```
};
```

Replaceability is achieved because:

- the core knows only `AuditSink`,
- adapters can send events to files, syslog, structured logs, or tracing pipelines.

5.4.9 Replacement Requires Testability: Contract Tests Over Implementation Tests

To replace a component safely, you need tests that validate the boundary contract, not internal mechanics.
A replacement-friendly test strategy includes:

- **core unit tests:** verify domain policy independent of adapters (using fakes/mocks),
- **adapter integration tests:** verify adapter correctness against real infrastructure,
- **contract tests:** the adapter must satisfy the port contract (behavioral guarantees).

Minimal fake for core testing

```
#include <unordered_map>

class FakeUserRepo : public UserRepo {
public:
  std::optional<User> find(UserId id) override {
    auto it = db_.find(id.value);
    if (it == db_.end()) return std::nullopt;
    return it->second;
  }
```

```
  void save(const User& u) override {
    db_[u.id.value] = u;
  }

  std::vector<User> find_by_name(std::string_view prefix) override {
    std::vector<User> out;
    for (auto& [_, u] : db_) {
      if (u.name.rfind(prefix, 0) == 0) out.push_back(u);
    }
    return out;
  }

private:
  std::unordered_map<std::string, User> db_;
};
```

The core can be tested without any real database, which makes replacement safer and faster.

5.4.10 Designing for Replacement Under Concurrency and Lifetimes

Replacement often fails due to lifetime coupling:

- shared ownership leaks across layers,
- long-lived references tie core objects to adapter lifetimes,
- thread-safety assumptions differ between implementations.

Replacement-friendly lifetime rules:

- keep ownership at composition roots and adapters,

- pass dependencies as references/pointers (borrowed) into stable core policies,
- avoid sharing mutable state across boundaries; prefer request-scoped objects,
- encode thread-safety expectations in the port contract (or document them explicitly).

5.4.11 Replacement Smells: Signals That a Subsystem Cannot Be Swapped

Smell 1: The core includes vendor headers

A clear physical coupling issue; replacement will be painful.

Smell 2: Domain types embed vendor types

This turns vendor details into part of the stable vocabulary.

Smell 3: Boundary is too wide

Too many methods, too many dependencies, too many semantic promises. Replacement becomes a rewrite.

Smell 4: Boundary is too low-level

If the interface exposes low-level operations (`executeSQL`, `sendPacket`), the core is coupled to the mechanism, not the intent.

Smell 5: No contract tests

Without tests that validate the boundary behavior, replacements become risky and slow.

5.4.12 A Practical Checklist for "Replaceability by Design"

Use this checklist for any subsystem you expect to replace:

1. **Stable port:** is there a narrow interface expressing intent?
2. **No vendor leakage:** do public headers avoid third-party types?
3. **Direction:** does the core depend on the port, while adapters depend on the core vocabulary?
4. **Physical isolation:** are third-party headers confined to adapter implementation units?
5. **Lifetime containment:** does the adapter own its resources internally?
6. **Thread-safety contract:** are concurrency expectations explicit at the boundary?
7. **Contract tests:** can an implementation be swapped and validated quickly?

5.4.13 Summary

Designing for replacement is an architectural commitment:

- define what must remain stable (meaning, contracts, invariants),
- isolate what will change (mechanisms, vendors, formats),
- invert dependencies so stable policy depends on stable abstractions,
- prevent physical coupling through disciplined headers and boundary patterns,
- validate replaceability through contract tests and clear lifetime ownership.

In Modern C++, replacement design is the most practical proof that coupling is controlled and that dependency direction is correct. If a subsystem can be replaced without rewriting the core, the architecture is working.

Abstraction Without Damage

6.1 Abstraction vs Indirection

6.1.1 The Central Confusion: "Abstract" Is Not "Indirect"

In design discussions, *abstraction* and *indirection* are frequently treated as synonyms. In Modern C++, confusing them produces architectures that are:

- slower (unnecessary dynamic dispatch and allocations),
- harder to test (interfaces everywhere, behavior nowhere),
- harder to read (layers of pass-through code),
- harder to optimize (lost inlining and blurred data ownership),
- harder to evolve (abstractions that do not match domain change axes).

A rigorous distinction:

- **Abstraction** is about **meaning and contract**: capturing essential properties while hiding irrelevant detail.
- **Indirection** is about **mechanism of access**: calling or reaching something through an extra level (pointer, virtual call, handle, wrapper, dispatcher).

You can have abstraction without indirection. You can have indirection without abstraction. Good design uses both intentionally and sparingly.

6.1.2 Abstraction: Compressing Meaning, Not Adding Layers

What Abstraction Does

Abstraction creates:

- a **vocabulary** the system can speak,
- a **contract** clients can rely on,
- a **boundary** that reduces cognitive load,
- a **stability surface** that survives change.

Abstraction answers:

- "What is this concept?"
- "What must always be true?"
- "What operations define it?"
- "What details are irrelevant to clients?"

Abstraction Can Be Purely Static in C++

In C++, abstraction often appears as:

- strong types and value objects,
- constrained templates and concepts,

- algorithms expressed generically over a concept,
- RAII wrappers that encode safety invariants.

None of these require runtime indirection.

Example: Abstraction Without Indirection (Strong Type)

```
#include <string>
#include <stdexcept>
#include <utility>

class UserId {
public:
  explicit UserId(std::string v) : v_(std::move(v)) {
    if (v_.empty()) throw std::invalid_argument("UserId empty");
  }

  const std::string& value() const noexcept { return v_; }

private:
  std::string v_; // invariant: non-empty
};
```

This is abstraction:

- it encodes meaning (`UserId` is not `std::string`),
- it enforces invariants,
- it reduces incorrect mixing of unrelated strings,
- it adds no runtime indirection beyond owning the data.

6.1.3 Indirection: A Costly Power Tool

What Indirection Does

Indirection adds an extra hop between caller and callee or between user and data:

- pointer/reference indirection,
- virtual dispatch,
- function pointer callbacks,
- type erasure (runtime dispatch through a wrapper),
- message queues and actor mailboxes,
- plugin boundaries and dynamic loading.

Indirection answers:

- "How do we switch implementations at runtime?"
- "How do we decouple lifetime and ownership?"
- "How do we isolate build or ABI dependencies?"
- "How do we cross threads or processes safely?"

Indirection *always* has costs:

- performance cost (branch mispredict, loss of inlining, cache misses),
- complexity cost (lifetime, ownership, aliasing),
- debugging cost (stack traces, indirection layers),
- reasoning cost (who owns what? who guarantees what?).

Example: Indirection Without Abstraction (Pointless Layer)

```
struct Service {
  int do_work(int x) { return x * 2; }
};

struct ServiceProxy {
  Service* s{};
  int do_work(int x) { return s->do_work(x); } // indirection but no new contract
};
```

This adds indirection but provides no meaningful abstraction:

- no new invariant,
- no new semantic contract,
- no boundary that isolates change,
- only an extra hop.

6.1.4 The Rule: Use Abstraction to Remove Detail, Use Indirection to Enable Change

A design remains healthy when:

- abstraction reduces cognitive load and stabilizes meaning,
- indirection is introduced only to solve a concrete problem: runtime variation, plugin boundary, test seam, or dependency inversion.

A pragmatic decision guide:

> **If you cannot name the specific variability or boundary the indirection enables, do not add it.**

6.1.5 Abstraction Mechanisms in Modern C++ (Often Without Indirection)

1) Value Objects and Strong Types

Use them to encode invariants, units, identities, and domain vocabulary. Benefits:

- fewer invalid states,
- fewer parameter mix-ups,
- more readable and stable APIs.

2) Concepts and Constrained Templates

Concepts create abstraction by expressing requirements precisely. They often allow:

- generic code with clear contracts,
- compile-time enforcement,
- zero runtime dispatch.

```
#include <concepts>
#include <string_view>

template <class T>
concept Sink = requires(T s, std::string_view msg) {
  { s.write(msg) } -> std::same_as<void>;
};

template <Sink S>
void log_info(S& sink, std::string_view msg) {
  sink.write(msg); // abstraction through concept, no virtual dispatch
}
```

3) RAII as Safety Abstraction

RAII expresses the contract “resource is acquired and released safely” with minimal indirection. It encodes a lifetime invariant and reduces error surface.

6.1.6 Indirection Mechanisms in Modern C++ (Choose Intentionally)

1) Virtual Interfaces (Runtime Polymorphism)

Use when:

- runtime selection is required,
- ABI stability matters,
- compilation dependencies must be minimized,
- plugin architectures require stable binary boundaries.

Cost:

- virtual dispatch,
- heap allocations often appear,
- more complex lifetime considerations.

2) Type Erasure (Runtime Polymorphism Without Inheritance)

Use when:

- you want runtime polymorphism but avoid exposing templates,
- you want a stable, uniform boundary type.

Cost:

- indirect calls,
- potential allocations (depending on small-buffer optimization),
- more complex debugging.

3) Dependency Injection via References (Low-Cost Indirection)

Passing collaborators by reference/pointer can be the lightest indirection that still enables replacement and testing.

```
#include <string_view>

struct Clock {
  virtual ~Clock() = default;
  virtual long long now() const noexcept = 0;
};

class Policy {
public:
  explicit Policy(const Clock& c) : c_(c) {}

  bool expired(long long t, long long ttl) const noexcept {
    return (c_.now() - t) >= ttl; // stable policy
  }

private:
  const Clock& c_; // indirection enables replacement, but policy remains cohesive
};
```

6.1.7 The “Abstraction Damage” Failure Modes

Abstraction can damage a design when it becomes:

1) Premature

Abstraction built before variability is understood often:

- locks in the wrong interface,
- forces unnatural shapes onto the domain,
- increases code volume and friction.

2) Leaky

Clients still need to know internal details, defeating the purpose. In C++, leaky abstractions often come from:

- exposing third-party types in interfaces,
- requiring clients to manage lifetimes and synchronization,
- passing raw buffers and magic flags instead of expressing intent.

3) Indirection-Heavy

Too many layers and runtime dispatch points cause:

- performance unpredictability,
- brittle debugging,
- unclear ownership and responsibility.

6.1.8 A Practical Decision Matrix (Through C++)

Use these questions to decide the right tool:

- **Do we need runtime variability?**
  - No ⇒ prefer value abstractions, concepts, templates.
  - Yes ⇒ consider virtual interfaces or type erasure.
- **Is this a stable boundary with many dependents?**
  - Yes ⇒ minimize physical dependencies; virtual/type-erased boundary may be justified.
  - No ⇒ keep it simple; avoid heavy indirection.
- **Is performance critical and predictable latency required?**
  - Yes ⇒ avoid unnecessary indirection; keep hot paths static.
  - No ⇒ prioritize clarity and replaceability.
- **Is build scalability a constraint?**
  - Yes ⇒ reduce header/template spread; isolate implementation details.

6.1.9 Combining Both Safely: Abstraction With Controlled Indirection

The best long-lived designs combine:

- **abstraction in the core:** stable domain vocabulary and invariants,
- **indirection at the edges:** replaceable adapters for I/O, frameworks, protocols.

A durable pattern:

Keep the core abstract but direct. Keep the edges indirect but thin.

6.1.10 Review Checklist: Abstraction vs Indirection

Use this checklist in reviews:

1. **Abstraction test:** Does this type reduce detail and encode a real contract?
2. **Invariants:** Does it enforce at least one meaningful invariant?
3. **Indirection test:** What concrete variability or boundary does the indirection enable?
4. **Cost awareness:** Is this in a hot path where virtual calls/allocations matter?
5. **Ownership clarity:** Are lifetimes explicit across the boundary?
6. **Physical coupling:** Does the abstraction leak heavy dependencies into public headers?
7. **Replaceability:** Can we swap the implementation without rewriting core policy?

6.1.11 Summary

- **Abstraction** captures meaning, invariants, and contracts; it can be purely static in C++ and often costs nothing at runtime.
- **Indirection** is a mechanism for runtime variability, dependency inversion, and boundary isolation; it has real costs in performance and complexity.
- Good Modern C++ design uses abstraction to stabilize the core and uses indirection only where it buys replaceability, test seams, ABI stability, or build scalability.
- The goal of "abstraction without damage" is achieved when abstractions reduce complexity while indirection remains intentional, minimal, and justified.

6.2 Interfaces, Concepts, and Type Erasure

6.2.1 Three Ways to Express "A Contract" in Modern C++

Long-lived C++ systems succeed when contracts are explicit and stable. Modern C++ offers three primary mechanisms to express a contract:

- **Interfaces (runtime polymorphism)**: contracts expressed through virtual functions and dynamic dispatch.
- **Concepts (compile-time polymorphism)**: contracts expressed as constraints on types and operations, verified by the compiler.
- **Type erasure (runtime polymorphism without inheritance)**: contracts expressed through a uniform wrapper type that forwards calls indirectly.

These are not interchangeable. Each represents a trade-off across:

- performance predictability and inlining,
- build times and physical dependencies,
- ABI stability and binary boundaries,
- testability and replaceability,
- error messages and user experience,
- flexibility vs simplicity.

This section provides an engineering-level comparison and shows how to use each mechanism without damaging the design.

6.2.2 Interfaces: Runtime Contracts With Strong Boundary Properties

What an Interface Buys You

A classic interface in C++ is a base class with virtual functions. Its engineering benefits are boundary-related:

- **Runtime substitutability**: the implementation can be chosen at runtime.
- **Stable object code boundaries**: interface users only need the interface header, not the implementation.
- **Reduced template spread**: avoids exposing template bodies in public headers.
- **Natural plugin and ABI patterns**: common in shared-library and plugin designs.

Costs of Interfaces

- virtual dispatch (indirect call),
- loss of inlining across the boundary,
- object lifetime and ownership are often heap-based,
- more runtime failure modes if invariants are not enforced,
- interface evolution is constrained (adding virtual functions breaks ABI unless carefully managed).

Minimal interface example: stable contract, replaceable implementations

```
#include <string_view>

struct Sink {
```

```
  virtual ~Sink() = default;
  virtual void write(std::string_view msg) = 0;
};

class Logger {
public:
  explicit Logger(Sink& s) : sink_(s) {}

  void info(std::string_view msg) {
    // Responsibility: logging policy (format/level decisions)
    sink_.write(msg);
  }

private:
  Sink& sink_; // borrowed: Logger does not own the sink
};
```

This design makes replacement cheap:

- one `Sink` can write to a file,
- another can write to syslog,
- another can emit structured events,
- without changing `Logger`.

6.2.3 Concepts: Compile-Time Contracts With Zero Runtime Dispatch

What Concepts Buy You

Concepts express contracts as requirements on a type. Their engineering advantages:

- **No runtime indirection**: calls are direct and inlineable.
- **Strong static validation**: incorrect types fail at compile time.
- **Better generic API clarity**: the contract is explicit and checkable.
- **Local performance**: enables optimization and constant propagation.

Costs of Concepts

- increased compile-time surface (templates instantiate in many translation units),
- build-time coupling (headers must carry definitions),
- possible code bloat if many instantiations occur,
- error messages are improved versus unconstrained templates but still can be heavy in large generic stacks.

Concept example: a sink contract without inheritance

```
#include <concepts>
#include <string_view>

template <class T>
concept SinkLike = requires(T t, std::string_view msg) {
  { t.write(msg) } -> std::same_as<void>;
};

template <SinkLike SinkT>
class LoggerT {
public:
  explicit LoggerT(SinkT& s) : sink_(s) {}
```

```
  void info(std::string_view msg) {
    sink_.write(msg); // direct call, can inline
  }

private:
  SinkT& sink_;
};
```

This is strong abstraction with no indirection, but it creates physical dependencies:

- `LoggerT` is a template: its definition lives in headers.
- Every translation unit that uses it instantiates it.

6.2.4 Type Erasure: Runtime Polymorphism Without Exposing Templates or Inheritance

What Type Erasure Buys You

Type erasure creates a single runtime type that can hold any object that satisfies an implied contract. It sits between interfaces and concepts:

- **Runtime substitutability** like interfaces,
- **No inheritance required** for implementations,
- **Stable API surface** (one concrete wrapper type),
- **Reduced template spread** at module boundaries,
- **Encapsulation of implementation types** (no need to expose their headers to consumers).

Costs of Type Erasure

- indirect calls through function pointers or vtable-like structures,
- potential heap allocation (unless small-buffer optimization is used),
- complexity in correct copy/move semantics,
- debugging indirection and less transparent call stacks.

A minimal, explicit type-erased `AnySink`

The following example illustrates the structure of type erasure clearly. It emphasizes correctness (move-only) and avoids pretending that type erasure is "free."

```
#include <string_view>
#include <memory>
#include <utility>
#include <type_traits>

class AnySink {
public:
  AnySink() = default;

  template <class T>
  AnySink(T obj) : self_(std::make_unique<Model<T>>(std::move(obj))) {}

  AnySink(AnySink&&) noexcept = default;
  AnySink& operator=(AnySink&&) noexcept = default;

  AnySink(const AnySink&) = delete;
  AnySink& operator=(const AnySink&) = delete;
```

```
  void write(std::string_view msg) {
    self_->write(msg);
  }

private:
  struct Concept {
    virtual ~Concept() = default;
    virtual void write(std::string_view msg) = 0;
  };

  template <class T>
  struct Model final : Concept {
    explicit Model(T v) : value(std::move(v)) {}
    void write(std::string_view msg) override { value.write(msg); }
    T value;
  };

  std::unique_ptr<Concept> self_;
};
```

Use it like this:

```
#include <string_view>
#include <cstdio>

struct StdoutSink {
  void write(std::string_view msg) {
    std::fwrite(msg.data(), 1, msg.size(), stdout);
    std::fwrite("\n", 1, 1, stdout);
```

```
  }
};

class Logger {
public:
  explicit Logger(AnySink s) : sink_(std::move(s)) {}

  void info(std::string_view msg) {
    sink_.write(msg);
  }

private:
  AnySink sink_;
};
```

This yields:

- a stable `Logger` interface with one concrete field type,
- runtime substitutability without requiring inheritance in sinks,
- controlled indirection and a clear ownership model.

6.2.5 How to Choose: The Engineering Trade-Offs

Decision axis 1: Runtime vs Compile-Time Variation

- **Need runtime selection** (plugins, runtime config, dependency injection): prefer **interfaces** or **type erasure**.
- **Need compile-time selection** (performance-critical generic algorithms): prefer **concepts**.

Decision axis 2: Stable ABI Boundaries

- For shared libraries and stable binary APIs, templates are painful.
- Interfaces and type erasure create stable boundary types.
- PIMPL and adapter layers complement these boundaries.

Decision axis 3: Build Scalability and Physical Dependencies

- Concepts/templates increase physical coupling: header definitions must be visible.
- Interfaces/type erasure reduce physical coupling: consumers need only the contract.

Decision axis 4: Performance and Hot Paths

- Concepts enable inlining and optimization.
- Interfaces/type erasure impose indirect calls and can add allocations.
- A common pattern is: concepts inside hot loops, indirection at the edges.

Decision axis 5: Ease of Extension by Third Parties

- Interfaces require inheritance and vtable discipline.
- Concepts require satisfying syntactic/semantic requirements (no inheritance).
- Type erasure allows third parties to provide any type that matches the required operations.

6.2.6 Avoiding Design Damage: Common Failure Modes

Failure 1: "Interfaces Everywhere" Without Real Variability

Adding virtual interfaces without a proven variability axis:

- increases code size,
- adds indirection cost,
- reduces clarity,
- complicates lifetimes.

Failure 2: "Concepts Everywhere" in Public, High Fan-In Boundaries

Overusing templates in highly depended-on boundaries:

- increases build times,
- increases ODR risk with macros/configuration,
- spreads implementation details through headers.

Failure 3: Type Erasure With Unclear Ownership and Copy Semantics

Type erasure wrappers must have an intentional ownership model:

- move-only wrappers are simpler and often sufficient,
- copyable wrappers require cloning semantics and careful exception safety.

6.2.7 A Robust Pattern: "Concepts Internally, Type Erasure at the Boundary"

A common long-lived-system pattern:

- Use **concepts** and templates for internal algorithms where performance matters.
- Use **type erasure** or **interfaces** at architectural boundaries to:

  - reduce template spread,
  - avoid exposing third-party headers,
  - preserve ABI stability,
  - keep build graphs stable.

6.2.8 Review Checklist: Interfaces, Concepts, Type Erasure

Use this checklist during design reviews:

1. **Variability axis:** what is expected to vary, and when (compile time vs runtime)?
2. **Boundary stability:** is this a public/high fan-in API that must remain stable?
3. **Build cost:** will templates increase rebuild scope unacceptably?
4. **Performance:** is this in a hot path where indirect calls matter?
5. **Ownership:** are lifetimes explicit (borrowed vs owned) across the abstraction?
6. **Leak control:** does the abstraction avoid leaking third-party types and headers?

6.2.9 Summary

Modern C++ provides three contract mechanisms, each with a distinct role:

- **Interfaces** provide runtime substitution and strong boundary properties, but introduce virtual dispatch and constrain ABI evolution.
- **Concepts** provide compile-time contracts with zero runtime cost and excellent optimization potential, but increase physical dependencies and compile-time load.

- **Type erasure** provides runtime substitution with a uniform stable type and no inheritance requirement, but introduces indirection and potential allocation costs.

A long-lived design avoids damage by choosing the mechanism that matches the real variability axis:

- **concepts** for internal high-performance generic code,
- **interfaces/type erasure** for stable boundaries, plugin points, and dependency isolation.

6.3 Virtual Dispatch vs Static Polymorphism

6.3.1 Why This Choice Decides Whether Abstraction Helps or Hurts

In Modern C++ design, the most common source of "abstraction damage" is choosing the wrong polymorphism mechanism. Two families dominate:

- **Virtual dispatch (dynamic polymorphism)**: runtime selection via virtual functions and vtables.
- **Static polymorphism**: compile-time selection via templates, CRTP, and concepts.

Both are valid. Both are powerful. Both can produce disasters when used outside their natural domain.
A long-lived system does not ask: *"Which one is better?"* It asks:

> **Where must variation be decided at runtime, and where can it be decided at compile time without harming build scale and maintainability?**

6.3.2 Definitions: Two Ways to Substitute Behavior

Virtual Dispatch (Runtime Polymorphism)

Virtual dispatch means:

- the call target is chosen at runtime,
- the caller talks to a base-class interface,
- the callee is reached through an indirect call (vtable).

Static Polymorphism (Compile-Time Polymorphism)

Static polymorphism means:

- the call target is chosen at compile time,
- substitution happens through templates and overload resolution,
- the compiler can inline and optimize aggressively.

6.3.3 The Real Trade-Off: Runtime Flexibility vs Physical Dependency Cost

The deeper engineering trade-off is not only runtime speed; it is **system-level cost**:

- Virtual dispatch increases runtime indirection but can decrease build coupling.
- Static polymorphism eliminates runtime indirection but often increases compilation coupling.

In long-lived systems, build time and dependency stability are strategic resources.

6.3.4 Virtual Dispatch: When It Is the Correct Tool

Virtual Dispatch Strengths

Virtual dispatch is ideal when you need:

- **runtime selection** (configuration, plugins, dependency injection),
- **stable boundaries** between libraries (especially across shared libraries),
- **reduced template spread** in high fan-in APIs,
- **replaceable subsystems** where implementations evolve independently from the core,
- **test seams** (injecting fakes/mocks with minimal friction).

Virtual Dispatch Costs and Risks

- indirect call cost (branch prediction, pipeline disruption),
- reduced inlining opportunities,
- potential heap allocation patterns (common with polymorphic object graphs),
- tighter constraints on interface evolution (ABI and vtable layout),
- lifetime and ownership complexity if object graphs are not rooted clearly.

Example: Virtual interface as a stable boundary

```
#include <string_view>

struct Compressor {
  virtual ~Compressor() = default;
```

```
  virtual void compress(std::string_view in) = 0;
};

class Pipeline {
public:
  explicit Pipeline(Compressor& c) : c_(c) {}

  void run(std::string_view payload) {
    // stable policy: pipeline decisions
    c_.compress(payload); // runtime substitution
  }

private:
  Compressor& c_; // borrowed; ownership can live in a composition root
};
```

This design allows:

- switching compression algorithm at runtime,
- plugin loading,
- isolating third-party library churn behind implementations.

6.3.5 Static Polymorphism: When It Is the Correct Tool

Static Polymorphism Strengths

Static polymorphism is ideal when you need:

- **zero-cost abstractions** (no virtual call overhead),
- **inlining and optimization** in hot paths,

- **value-based designs** and small composable algorithms,
- **compile-time enforcement** of contracts via concepts,
- **fine-grained generic reuse** across types and containers.

Static Polymorphism Costs and Risks

- increased compile times and rebuild scope,
- larger binary size due to many instantiations,
- public template APIs widen physical dependency surfaces,
- error messages and debugging can be harder in deep generic stacks,
- interface stability can be harder when templates expose details.

Example: Static polymorphism with concepts

```
#include <concepts>
#include <string_view>

template <class T>
concept CompressorLike = requires(T t, std::string_view in) {
  { t.compress(in) } -> std::same_as<void>;
};

template <CompressorLike C>
class PipelineT {
public:
  explicit PipelineT(C& c) : c_(c) {}
```

```
  void run(std::string_view payload) {
    c_.compress(payload); // direct call, can inline
  }

private:
  C& c_;
};
```

This is excellent for performance-critical pipelines but creates template instantiations across translation units.

6.3.6 CRTP: Static Polymorphism Without Virtuals (But With Discipline)

CRTP (Curiously Recurring Template Pattern) is a static polymorphism pattern often used to share implementation while allowing derived classes to customize behavior.

CRTP example

```
#include <string_view>

template <class Derived>
class CompressorBase {
public:
  void compress(std::string_view in) {
    // common pre/post work
    static_cast<Derived&>(*this).compress_impl(in);
  }
};

class FastCompressor : public CompressorBase<FastCompressor> {
```

```
public:
  void compress_impl(std::string_view in) {
    (void)in;
    // fast algorithm
  }
};
```

CRTP is powerful but must be constrained:

- it exposes implementation in headers,
- it increases compile-time coupling,
- it can become unreadable if overused.

6.3.7 Performance Reality: The Cost Is Often Not the Call Itself

A mature perspective:

- The cost of a virtual call can be negligible if the operation dominates (I/O, allocations, parsing).
- The cost can be significant in tight loops where the called operation is tiny and frequent.
- Cache locality and memory layout often matter more than dispatch mechanism.

Therefore, the correct approach is architectural:

- use static polymorphism in hot inner loops and algorithms,
- use virtual boundaries at system edges and replaceable subsystems.

6.3.8 Build and Boundary Reality: C++ Physical Dependencies Decide Scaling

Static polymorphism typically requires:

- templates in headers,
- more transitive includes,
- more rebuild propagation.

Virtual dispatch boundaries typically allow:

- smaller headers (only interface),
- implementation hidden in `.cpp`,
- stable ABI boundaries,
- reduced rebuild surface.

For long-lived systems, a high fan-in API should be extremely cautious with templates. A practical rule:

> **If many components depend on it, prefer a stable runtime boundary unless static polymorphism is essential for performance.**

6.3.9 Interface Evolution: Static vs Virtual Stability

Virtual Interface Evolution Constraints

Adding or changing virtual functions can break binary compatibility. Therefore, virtual interfaces require:

- careful versioning strategy,
- stable minimal surface,
- preference for non-virtual extension points (free functions, adapters, composition).

Template Interface Evolution Constraints

Template APIs can be source-compatible but still painful:

- small signature changes can force rebuilds across the entire codebase,
- template constraints changes can break compilation for some types,
- error messages may shift and become more complex.

6.3.10 A Strong Pattern: "Static Inside, Virtual Outside"

A proven architecture in performance-sensitive systems:

- **Inside core algorithms:** static polymorphism, concepts, templates.
- **At component boundaries:** virtual interfaces or type erasure.

This preserves:

- performance in hot code,
- replaceability at edges,
- build scalability and stable dependency direction.

6.3.11 Worked Example: Virtual Boundary With Static Inner Loop

Stable runtime boundary

```
#include <string_view>

struct HashAlgo {
  virtual ~HashAlgo() = default;
  virtual unsigned long long hash(std::string_view s) const noexcept = 0;
};
```

Static inner implementation options

```
#include <string_view>

struct Fnv1a64 {
  unsigned long long operator()(std::string_view s) const noexcept {
    unsigned long long h = 14695981039346656037ULL;
    for (unsigned char c : s) {
      h ^= c;
      h *= 1099511628211ULL;
    }
    return h;
  }
};
```

Adapter that implements the virtual contract

```
#include <utility>

class HashAlgoFnv final : public HashAlgo {
```

```
public:
  unsigned long long hash(std::string_view s) const noexcept override {
    return fnv_(s); // static inner loop remains inline-friendly
  }
private:
  Fnv1a64 fnv_{};
};
```

The system gets:

- runtime replaceability of hash algorithms,
- fast implementation inside each algorithm,
- stable boundary that does not force template spread across users.

6.3.12 Design Checklist: Choosing Between Virtual and Static

Use the following checklist:

1. **Runtime variability needed?**
   - Yes ⇒ virtual dispatch or type erasure.
   - No ⇒ static polymorphism.
2. **Hot path?**
   - Yes ⇒ static polymorphism for inner loops; avoid virtual calls in tight loops.
   - No ⇒ virtual boundaries are acceptable and often cleaner.
3. **High fan-in API?**
   - Yes ⇒ avoid template-heavy public surfaces; prefer virtual/type-erased boundary.

4. **ABI stability required?**
    - Yes ⇒ virtual boundaries (or type erasure with stable wrapper) plus careful versioning.
5. **Build scalability critical?**
    - Yes ⇒ minimize templates in widely included headers.

6.3.13 Summary

- **Virtual dispatch** enables runtime substitution, plugin boundaries, and reduced template spread, but introduces indirect calls and tighter ABI/interface evolution constraints.
- **Static polymorphism** enables zero-cost abstraction and aggressive optimization, but increases compile-time coupling, rebuild scope, and potential code bloat.
- Long-lived Modern C++ systems typically succeed with a hybrid: **static polymorphism inside, virtual/type-erased boundaries outside**.
- The correct choice is driven by variability timing (runtime vs compile time), hot-path requirements, build scaling, and stability of boundaries.

6.4 When Abstraction Becomes a Liability

6.4.1 The Purpose of Abstraction and the Point Where It Breaks

Abstraction exists to:

- reduce cognitive load by hiding irrelevant detail,
- stabilize meaning and contracts across change,

- isolate volatility behind boundaries,
- enable reuse without copying logic.

However, abstraction becomes a **liability** when it no longer reduces complexity, but *relocates* it into:

- unclear interfaces,
- accidental indirection layers,
- leaky boundaries,
- build and linkage coupling,
- runtime unpredictability,
- debugging and operational opacity.

The defining test:

> **An abstraction is justified only if it makes change cheaper and reasoning simpler for the next decade of system evolution.**

6.4.2 Liability Category 1: Premature Abstraction (Wrong Contract Too Early)

What Happens

Premature abstraction occurs when an interface is designed before the system has learned what truly varies. This usually produces:

- the wrong axis of variation captured in the API,

- missing essential operations that appear later,
- over-generalized interfaces that force awkward implementations,
- frozen mistakes because many dependents form quickly.

C++-Specific Damage

In C++, premature abstraction is amplified by:

- header exposure: early template/interface decisions spread widely,
- ABI stability: once shipped, virtual interfaces are hard to change,
- ownership semantics: early choices lock lifetime models (shared vs unique),
- build graphs: early header design determines rebuild scope for years.

Practical prevention

- Start with concrete implementations.
- Extract abstraction only after at least two competing implementations exist or are inevitable.
- Prefer narrow abstractions that express intent, not mechanism.

6.4.3 Liability Category 2: Over-Generalization (The "Lowest Common Denominator" Interface)

What Happens

An interface designed to serve many different consumers often becomes:

- too broad (many methods, many responsibilities),
- too weak (insufficient semantics to be useful),
- too permissive (allows invalid states and misuse),
- shaped by accidental needs rather than core invariants.

The symptom

- many flags,
- "options" structs with dozens of fields,
- `void*` or untyped key-value dictionaries,
- generic `execute()` that hides real semantics.

Design correction

- Split by axis of change (responsibility).
- Replace flags with types (separate cohesive policies).
- Use strong types and value objects to express invariants.

6.4.4 Liability Category 3: Leaky Abstractions (Clients Still Need the Details)

What Happens

A leaky abstraction is one where:

- clients must learn internal details to use it correctly,
- hidden constraints exist but are not encoded in types/contracts,

- performance behavior depends on undocumented implementation properties.

C++-Specific leakage vectors

- exposing third-party types in public interfaces,
- requiring callers to manage lifetimes across the boundary,
- requiring callers to coordinate thread-safety externally,
- exposing raw buffers and expecting hidden alignment/ownership rules.

Example: Leaky lifetime abstraction with `string_view`

```
#include <string_view>

class TokenStore {
public:
  // Liability: returns a view whose lifetime depends on hidden internal storage
  ↪  rules.
  std::string_view last_token() const noexcept { return last_; }

  void set_token(std::string_view t) {
    last_ = t; // BUG: storing a view to caller-owned memory
  }

private:
  std::string_view last_{};
};
```

The abstraction claims “I store tokens” but actually stores *borrowed views*. Clients must now understand lifetimes and storage rules, which defeats the abstraction.

Fix: own what you claim to own

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

class TokenStore {
public:
  std::string_view last_token() const noexcept { return last_; }

  void set_token(std::string_view t) {
    last_.assign(t.data(), t.size()); // store an owned copy
  }

private:
  std::string last_{};
};
```

6.4.5 Liability Category 4: Indirection Without Meaning (Layering as a Habit)

What Happens

Many systems accumulate layers that add no new contract:

- proxies that forward calls,
- managers that route to other managers,
- wrappers around wrappers with no new invariants,
- “service” classes that are just pass-through.

This is a common failure in C++ enterprise systems where “clean architecture” becomes a checklist rather than a reasoning discipline.

The test

> **If removing the layer does not remove an invariant, a boundary, or a variability axis, the layer is likely pure liability.**

6.4.6 Liability Category 5: Abstraction That Breaks Local Reasoning (Hidden Global Effects)

What Happens

Abstractions become liabilities when they hide global side effects:

- implicit thread creation,
- hidden global caches,
- hidden blocking I/O,
- implicit logging, retries, and backoff policies,
- implicit memory ownership across unknown roots.

In C++, these hidden effects create:

- unpredictable latency,
- deadlocks and contention,
- shutdown ordering failures,
- lifetime races.

A strong boundary rule

> **Do not hide thread-safety policy, blocking behavior, or ownership transfer behind innocent-looking APIs.**

6.4.7 Liability Category 6: Abstraction That Destroys Performance Predictability

What Happens

Abstraction becomes a performance liability when it introduces:

- virtual dispatch in hot loops,
- heap allocation on every call path,
- reference counting on performance-critical paths,
- type erasure where static polymorphism would suffice,
- excessive copying due to unclear value vs reference semantics.

C++-specific performance failure modes

- lost inlining across virtual boundaries,
- cache misses due to pointer chasing,
- branch mispredictions from indirect calls,
- code bloat from uncontrolled templates.

The discipline:

- static polymorphism for inner loops,
- runtime indirection at replaceable edges,
- keep allocation strategies explicit and localized.

6.4.8 Liability Category 7: Abstraction That Explodes Build Times and Physical Coupling

What Happens

In C++, the physical dependency graph is architecture. Abstractions become liabilities when they:

- live as template-heavy public headers,
- include large transitive header sets,
- expose implementation details inline,
- drag volatile third-party headers into stable interfaces.

This creates:

- rebuild storms,
- slow CI,
- fragile include-order dependencies,
- ODR risks from macro-driven configuration,
- difficulty forming reusable libraries or modules.

Mitigations

- minimize public headers; forward declare aggressively where correct,
- push heavy includes to `.cpp`,
- use PIMPL for stable APIs with volatile internals,
- use runtime boundaries or type erasure to avoid template spread in high fan-in APIs,
- adopt modules for stable interfaces where toolchains allow.

6.4.9 Liability Category 8: Abstraction That Freezes the Wrong Responsibility

What Happens

If an abstraction groups responsibilities incorrectly:

- it becomes a “common” dependency dumping ground,
- it forces unrelated clients to depend on unrelated policies,
- it creates a high fan-in/high fan-out hub,
- it becomes both hard to change and constantly forced to change.

A high-risk indicator:

> **A library that everybody uses and that depends on everything is a structural liability.**

6.4.10 Modern C++ Correctives: How to Build Abstraction Without Liability

Corrective 1: Encode Meaning in Types

Use:

- strong types for IDs and units,
- value objects to own invariants,
- constrained templates and concepts for algorithmic contracts.

Corrective 2: Make Lifetimes and Ownership Explicit

Use:

- values by default,
- `unique_ptr` for exclusive ownership transfer,
- `shared_ptr` only for true shared lifetime,
- views (`span`, `string_view`) only within proven lifetime boundaries.

Corrective 3: Put Indirection Where It Buys Replaceability

- Interfaces/type erasure at subsystem boundaries (adapters, plugins).
- Static polymorphism inside algorithms and hot paths.
- Avoid layers that add no invariants or boundary semantics.

Corrective 4: Design Contracts Around Intent

Replace mechanism-shaped APIs with intent-shaped APIs:

- not `executeSQL(query)`, but `save(User)` or `find(UserId)`,
- not `sendPacket(bytes)`, but `publish(Event)` or `request(Action)`.

Corrective 5: Validate Abstractions With Replacement Tests

An abstraction is proven when:

- two implementations can be swapped without rewriting core policy,
- contract tests validate behavior,
- performance and observability remain within predictable bounds.

6.4.11 A Practical Checklist: "Is This Abstraction a Liability?"

Use this checklist during reviews:

1. **Contract clarity:** can you state the invariant/guarantee in one sentence?
2. **Variability axis:** what specifically is expected to vary and when?
3. **Leak test:** do clients need internal details (lifetimes, threading, formats) to use it safely?
4. **Layer test:** does this layer add a real invariant or boundary?
5. **Performance test:** is indirection/allocations introduced on hot paths?
6. **Build test:** does it drag heavy dependencies into stable headers?
7. **Evolution test:** does it freeze the wrong responsibility (high fan-in and high fan-out)?
8. **Replacement test:** can you swap implementations without redesigning the core?

6.4.12 Summary

Abstraction becomes a liability when it stops reducing complexity and starts creating it:

- premature or over-generalized contracts,
- leaky boundaries that expose lifetimes, threading, or vendor details,
- indirection layers without semantic value,
- hidden global effects that break local reasoning,
- performance unpredictability from unnecessary dispatch/allocations,
- build-time explosions from template/header spread,
- frozen responsibilities that form unstable dependency hubs.

“Abstraction without damage” in Modern C++ is achieved by:

- encoding meaning in types and invariants,
- keeping ownership explicit and local,
- placing indirection only where replaceability is required,
- shaping contracts around intent,
- enforcing physical dependency discipline at header and module boundaries.

Composition Over Inheritance (Correctly Applied)

7.1 Inheritance: Legitimate Uses Only

7.1.1 The Position of Modern C++: Inheritance Is Not Evil, but It Is Expensive

Modern C++ does not prohibit inheritance; it *prices* it. Inheritance is a language mechanism with strong expressive power, but it introduces costs that are architectural, not merely syntactic:

- **Strong coupling**: base-class changes propagate to all derived classes.
- **Fragile contracts**: semantic promises must hold under substitution; violations are subtle and common.
- **Lifetime complexity**: polymorphic deletion, ownership, and virtual destructors become mandatory concerns.
- **Dispatch indirection**: virtual calls reduce inlining and introduce runtime unpredictability.
- **Testing and reasoning burden**: behavior is distributed across class hierarchies and override chains.

- **Binary/API stability constraints**: virtual interfaces constrain evolution, especially across library boundaries.

Therefore, the correct rule is not "avoid inheritance." It is:

> **Use inheritance only when it expresses a stable substitutability contract and when that contract is the simplest correct tool for the job.**

7.1.2 Two Kinds of Inheritance in C++: Implementation vs Interface

To avoid confusion, distinguish these:

- **Implementation inheritance**: deriving to reuse code and data layout (base provides implementation).
- **Interface inheritance**: deriving to model a contract and enable runtime substitution (base provides virtual API, little or no data).

Modern, long-lived C++ designs accept **interface inheritance** under strict conditions. They strongly discourage **implementation inheritance** except in narrow, controlled scenarios.

7.1.3 Legitimate Use #1: Pure Interface for Runtime Polymorphism at Boundaries

When it is legitimate

This is the primary legitimate use in production systems:

- you need runtime variation (plugins, configuration, dependency injection),
- you need to decouple stable policy from volatile implementations,
- you want to reduce physical dependency spread (avoid templates in public APIs),
- you want stable boundaries between components and libraries.

Rules for a good interface base class

- Keep it **small** and **cohesive**.
- Make destruction safe: **virtual destructor**.
- Avoid data members (prefer stateless interface).
- Prefer non-throwing destructors.
- Avoid forcing clients into complex ownership models.

Example: Legitimate interface inheritance

```
#include <string_view>

struct Storage {
  virtual ~Storage() = default;
  virtual void put(std::string_view key, std::string_view value) = 0;
  virtual bool get(std::string_view key, std::string& out) = 0;
};

class CachePolicy {
public:
  explicit CachePolicy(Storage& s) : s_(s) {}

  void store(std::string_view k, std::string_view v) {
    // stable policy uses stable contract
    s_.put(k, v);
  }

private:
```

```
  Storage& s_; // borrowed: policy does not own storage
};
```

Here inheritance expresses a **substitutability contract**: any `Storage` implementation can be substituted at runtime.

7.1.4 Legitimate Use #2: Non-Virtual Interface (NVI) to Enforce Invariants

The Non-Virtual Interface pattern uses inheritance to:

- keep a stable public operation,
- enforce invariants in the base class,
- delegate variability to protected virtual hooks.

Legitimate when:

- there is a true algorithm skeleton that must remain correct,
- derived classes only customize steps,
- the base class must enforce preconditions/postconditions.

```
#include <string_view>

class Parser {
public:
  virtual ~Parser() = default;

  bool parse(std::string_view input) {
    // invariant enforcement and stable algorithm skeleton
    if (input.empty()) return false;
```

```
    before_parse(input);
    const bool ok = parse_impl(input);
    after_parse(ok);
    return ok;
  }

protected:
  virtual void before_parse(std::string_view) {}
  virtual bool parse_impl(std::string_view) = 0;
  virtual void after_parse(bool) {}

private:
  // public API is non-virtual; derived classes cannot bypass invariants
};
```

NVI is legitimate because it constrains derived behavior under a stable algorithmic contract.

7.1.5 Legitimate Use #3: Traits/Mixins via CRTP (Static Polymorphism Inheritance)

CRTP uses inheritance as a compile-time technique. It can be legitimate when it:

- provides small, orthogonal mixins,
- avoids deep hierarchies,
- keeps behavior local and explicit,
- does not expose fragile override chains.

Example: CRTP mixin for comparable types

```
template <class Derived>
struct EqualityComparable {
  friend bool operator!=(const Derived& a, const Derived& b) {
    return !(static_cast<const Derived&>(a) == static_cast<const Derived&>(b));
  }
};

struct UserId : EqualityComparable<UserId> {
  int v{};
  friend bool operator==(const UserId& a, const UserId& b) { return a.v == b.v; }
};
```

This is inheritance used for **compile-time composition**. It remains legitimate because it is shallow and does not create runtime substitutability risks.

7.1.6 Legitimate Use #4: Framework-Required Inheritance (Controlled Adapters Only)

Some libraries require inheritance to integrate (UI frameworks, plugin interfaces, callbacks). This can be legitimate if:

- inheritance is confined to the edge adapter layer,
- domain/core logic does not depend on framework base classes,
- the adapter translates between framework types and core types.

A strict boundary rule:

> **When inheritance is required by a framework, isolate it at the edge. Never let it shape the core vocabulary or policies.**

7.1.7 Legitimate Use #5: Polymorphic Value Semantics (Rare, Carefully Designed)

Sometimes you need value-like behavior with runtime polymorphism. This is rare and must be explicit (clone or type erasure patterns). It is legitimate when:

- you truly need copyable polymorphic objects,
- you provide a safe clone mechanism,
- you keep ownership clear.

```
#include <memory>

struct Shape {
  virtual ~Shape() = default;
  virtual double area() const noexcept = 0;
  virtual std::unique_ptr<Shape> clone() const = 0;
};

class ShapeValue {
public:
  explicit ShapeValue(std::unique_ptr<Shape> s) : p_(std::move(s)) {}

  ShapeValue(const ShapeValue& other) : p_(other.p_->clone()) {}
  ShapeValue& operator=(const ShapeValue& other) {
    if (this != &other) p_ = other.p_->clone();
    return *this;
  }

  ShapeValue(ShapeValue&&) noexcept = default;
```

```
  ShapeValue& operator=(ShapeValue&&) noexcept = default;

  double area() const noexcept { return p_->area(); }

private:
  std::unique_ptr<Shape> p_;
};
```

This preserves a clear rule: `ShapeValue` owns the polymorphic object and defines copy semantics.

7.1.8 Illegitimate (or High-Risk) Uses: What to Avoid

Risk 1: Implementation Inheritance to Reuse Code

Common anti-pattern:

- base class holds data members and implementation,
- derived classes override some methods,
- coupling becomes extreme,
- base class becomes a “god class.”

Why it fails:

- base class changes break derived classes,
- invariants are fragile and often violated,
- multiple responsibilities become entangled.

Prefer:

- composition of helper objects,
- free functions and algorithms,
- private members,
- policy-based templates when appropriate.

Risk 2: Deep Hierarchies

Deep hierarchies create:

- non-local reasoning,
- override chains that are hard to track,
- brittle specialization that breaks substitution.

A practical rule:

> **If the behavior you need is best described as combining parts, do not build a hierarchy. Compose.**

Risk 3: Multiple Inheritance of Implementation

Multiple inheritance can be legitimate for pure interfaces, but multiple inheritance of implementation is high risk:

- diamond problems,
- ambiguous bases,
- fragile layout and initialization,
- difficult maintenance.

Use multiple inheritance only for:

- combining **pure interfaces**,
- small CRTP mixins,
- carefully controlled traits.

7.1.9 Composition Over Inheritance: The Real Meaning

"Composition over inheritance" does not mean "never inherit." It means:

- reuse behavior by composing objects and algorithms,
- inherit only to express a real substitutability contract or constrained customization hook,
- keep inheritance shallow and local,
- isolate framework inheritance at the edges.

7.1.10 Practical Rules for Teams (C++-Specific)

Use these rules as a policy:

1. **Default to composition.** Inheritance requires justification.
2. **Interface inheritance only by default:** base class has no data, small virtual API.
3. **Always use a virtual destructor** for polymorphic bases.
4. **Avoid public data members in base classes.**
5. **Prefer NVI** when the base must enforce invariants.
6. **Keep hierarchies shallow.** Avoid deep inheritance trees.

7. **Multiple inheritance only for pure interfaces or mixins.**

8. **Do not leak framework bases into the core.** Confine them to adapters.

7.1.11 Review Checklist: "Is Inheritance Legitimate Here?"

1. **Is there a real substitutability contract?** (not just "shares code")

2. **Is runtime variation required?** If not, prefer static techniques or composition.

3. **Can the base class remain small and stable?**

4. **Are invariants enforced?** (consider NVI)

5. **Are ownership and lifetimes explicit?**

6. **Is the hierarchy shallow and understandable?**

7. **Is inheritance isolated to the edge when driven by frameworks?**

7.1.12 Summary

Inheritance is legitimate in Modern C++ primarily for:

- stable runtime interfaces at boundaries,
- controlled customization with invariant enforcement (NVI),
- small CRTP mixins and traits (static composition),
- framework-required adapters confined to the edges,
- rare polymorphic value semantics with explicit cloning.

Inheritance becomes a liability when it is used for code reuse, deep hierarchies, or mixing responsibilities. A long-lived system treats inheritance as a specialized tool: used intentionally, kept shallow, and placed where it buys replaceability and stability without corrupting the core.

7.2 Composition Patterns in Modern C++

7.2.1 Composition as the Default Engineering Strategy

Composition means building behavior by **assembling parts** rather than extending a base class. In Modern C++, composition is the default because it:

- keeps coupling local (dependencies are explicit members),
- preserves value semantics and deterministic lifetimes (RAII),
- avoids fragile base-class contracts and deep hierarchies,
- supports replacement (swap a collaborator without rewriting the system),
- improves testability (inject fakes at boundaries),
- scales better in build systems (fewer transitive inheritance dependencies).

A practical rule:

> **If the relationship is "has-a" or "uses-a", compose. Only inherit for a true substitutability contract.**

This section presents modern composition patterns that are widely applicable in long-lived C++ systems, with emphasis on responsibility boundaries, lifetimes, and dependency direction.

7.2.2 Pattern 1: Constructor Injection With Reference Semantics (Lightweight Composition)

Intent

Compose a policy object from stable collaborators without owning them. This is ideal for:

- high-level policies,
- application services,
- testable units,
- avoiding shared ownership graphs.

Example

```
#include <string_view>

struct AuditSink {
  virtual ~AuditSink() = default;
  virtual void record(std::string_view event) = 0;
};

struct Clock {
  virtual ~Clock() = default;
  virtual long long now() const noexcept = 0;
};

class SessionPolicy {
public:
  SessionPolicy(Clock& clock, AuditSink& audit)
```

```
    : clock_(clock), audit_(audit) {}

  bool expired(long long issued_at, long long ttl) {
    const auto age = clock_.now() - issued_at;
    const bool ex = age >= ttl;
    if (ex) audit_.record("session expired");
    return ex;
  }

private:
  Clock& clock_;      // borrowed dependency
  AuditSink& audit_;  // borrowed dependency
};
```

Notes

- Composition keeps lifetimes clear: the composition root owns long-lived objects.
- The policy remains ownership-light and deterministic.

7.2.3 Pattern 2: Owning Composition With Value Members (RAII Aggregation)

Intent

Compose a subsystem from owned value members when:

- the composed objects are small or movable,
- lifetimes should match the parent object,
- you want deterministic teardown order.

Example

```
#include <string>
#include <utility>

class ConnectionPool {
public:
  explicit ConnectionPool(std::string dsn) : dsn_(std::move(dsn)) {}
  void warmup();
private:
  std::string dsn_;
};

class AppServices {
public:
  explicit AppServices(std::string dsn)
    : pool_(std::move(dsn)) {}

  void start() {
    pool_.warmup();
  }

private:
  ConnectionPool pool_; // owned by value; deterministic lifetime
};
```

Notes

- Prefer value members when ownership is straightforward.
- This typically yields better locality and fewer allocations.

7.2.4 Pattern 3: Owning Composition With `std::unique_ptr` (Stable Addresses / Polymorphism)

Intent

Use exclusive ownership when:

- the object is polymorphic,
- it is large or non-movable,
- it requires a stable address,
- you want to hide implementation (PIMPL-like shape).

Example

```
#include <memory>
#include <string_view>

struct Sink {
  virtual ~Sink() = default;
  virtual void write(std::string_view msg) = 0;
};

class Logger {
public:
  explicit Logger(std::unique_ptr<Sink> sink)
    : sink_(std::move(sink)) {}

  void info(std::string_view msg) { sink_->write(msg); }
```

```
private:
  std::unique_ptr<Sink> sink_; // exclusive ownership, replaceable implementation
};
```

Notes

- Ownership is explicit and local.
- Avoid `shared_ptr` unless there is true shared lifetime.

7.2.5 Pattern 4: Policy-Based Design (Static Composition With Templates)

Intent

Compose behavior at compile time, especially in hot paths, by injecting policies as template parameters. Benefits:

- zero virtual dispatch,
- inlining and optimization,
- compact, explicit variability.

Example: Strategy via template policy

```
#include <string_view>

struct NoAudit {
  void record(std::string_view) noexcept {}
};

struct StdAudit {
```

```
  void record(std::string_view) noexcept;
};

template <class AuditPolicy>
class TransferService {
public:
  explicit TransferService(AuditPolicy audit = {}) : audit_(std::move(audit)) {}

  void transfer(int& from, int& to, int amount) {
    if (amount <= 0 || from < amount) return;
    from -= amount;
    to   += amount;
    audit_.record("transfer");
  }

private:
  AuditPolicy audit_; // composed policy object
};
```

Notes

- Best for internal code where template spread is acceptable.
- Avoid template-heavy public APIs with many dependents.

7.2.6 Pattern 5: Decorator via Composition (Add Behavior Without Inheritance)

Intent

Wrap an object to add cross-cutting behavior (logging, caching, retries) without modifying the original type.

Example: Logging decorator for a repository

```
#include <string_view>
#include <optional>

struct User { int id{}; };

struct UserRepo {
  virtual ~UserRepo() = default;
  virtual std::optional<User> find(int id) = 0;
};

struct AuditSink {
  virtual ~AuditSink() = default;
  virtual void record(std::string_view event) = 0;
};

class LoggingUserRepo final : public UserRepo {
public:
  LoggingUserRepo(UserRepo& inner, AuditSink& audit)
    : inner_(inner), audit_(audit) {}
```

```
  std::optional<User> find(int id) override {
    audit_.record("repo.find");
    return inner_.find(id);
  }

private:
  UserRepo& inner_;
  AuditSink& audit_;
};
```

Notes

- The *composition* is the point: the wrapper holds an inner object.
- Inheritance here is only to satisfy the runtime interface contract; behavior is composed.

7.2.7 Pattern 6: Type Erasure as a Composition Boundary (Runtime Flexibility Without Inheritance)

Intent

Use a type-erased wrapper when:

- you want runtime substitution,
- you do not want implementations to inherit,
- you want a single stable member type,
- you want to avoid template spread across modules.

Example: minimal type-erased callable

```
#include <functional>
#include <string_view>

class Handler {
public:
  explicit Handler(std::function<void(std::string_view)> fn)
    : fn_(std::move(fn)) {}

  void operator()(std::string_view msg) const { fn_(msg); }

private:
  std::function<void(std::string_view)> fn_;
};
```

Notes

- Type erasure introduces indirection and potential allocation, but it is often justified at boundaries.
- Keep the erased contract small and cohesive.

7.2.8 Pattern 7: Composition Root (Wiring Dependencies in One Place)

Intent

Centralize object construction and ownership so the rest of the system is:

- ownership-light,
- testable,

- replaceable,
- not polluted by framework types.

Example

```
#include <memory>

struct FileSink : Sink { void write(std::string_view) override; };

int main() {
  auto sink = std::make_unique<FileSink>();      // process lifetime owned here
  Logger log{std::move(sink)};                   // transfers ownership
  log.info("started");
}
```

Notes

- Most lifetime problems disappear when ownership is concentrated at the root.
- The root is allowed to depend on volatile frameworks; the core should not.

7.2.9 Pattern 8: Component + Facade (Contain Third-Party Dependency Spread)

Intent

Wrap a third-party library behind a small facade to:

- reduce physical dependency spread,
- centralize upgrades,

- prevent vendor types from leaking into core interfaces.

Example: facade shape

```
#include <string_view>

class JsonFacade {
public:
  // Stable, intent-level API; implementation can change
  bool is_valid(std::string_view text) const;
  std::string normalize(std::string_view text) const;
};
```

Notes

- The facade is a composition boundary: it owns and hides the third-party dependency.
- Clients depend on your contract, not the vendor's surface.

7.2.10 Composition Under Modern Constraints: Lifetimes, Concurrency, and Build Graphs

Composition patterns in Modern C++ must respect three constraints:

1) Lifetimes and Ownership

- Prefer values for clear ownership.
- Use `unique_ptr` for exclusive ownership transfer and polymorphism.
- Use references/pointers to borrow dependencies.
- Use `shared_ptr` only for true shared lifetime, not as a convenience to avoid thinking.

2) Concurrency and Isolation

- Prefer message passing and move-only handoff between threads.
- Avoid exposing shared mutable state through composed objects unless synchronization is part of the contract.

3) Build Scalability

- Template-based composition increases header coupling.
- Runtime composition (interfaces/type erasure) can reduce template spread at boundaries.
- Keep public headers minimal; push heavy includes into `.cpp`.

7.2.11 Composition Smells and Corrective Actions

Smell 1: "Everything is shared_ptr"

Likely indicates unclear ownership and global lifetimes. Corrective:

- move ownership to the composition root,
- switch to references and `unique_ptr` where appropriate.

Smell 2: "Decorator chains become a maze"

If behavior requires too many wrappers, the boundary contract is likely wrong. Corrective:

- redesign the contract around intent,
- collapse redundant layers,
- keep cross-cutting concerns in a coherent middleware pipeline.

Smell 3: "Template policies in public APIs with high fan-in"

Corrective:

- move template composition behind internal boundaries,
- expose stable runtime boundary types.

7.2.12 Review Checklist: Composition Patterns

1. **Is the relationship "has-a/uses-a" rather than "is-a"?**
2. **Are ownership and lifetimes explicit and localized?**
3. **Are replaceable mechanisms isolated behind boundaries?**
4. **Are hot paths free of unnecessary indirection/allocations?**
5. **Are public headers kept stable and minimal?**
6. **Does composition improve testability (inject fakes easily)?**

7.2.13 Summary

Composition patterns are the primary mechanism for building long-lived Modern C++ systems:

- Use reference-based injection for ownership-light policies.
- Use value members for deterministic RAII aggregation.
- Use `unique_ptr` composition for polymorphism and stable addresses.
- Use policy-based templates for zero-cost variability in internal hot code.
- Use decorators to add behavior without changing core types.

- Use type erasure to create stable runtime boundaries without inheritance requirements.
- Centralize ownership in a composition root to make the rest of the system clean and replaceable.
- Use facades to contain third-party dependency spread and preserve stability.

When composition is applied correctly, it produces systems that are easier to test, cheaper to evolve, and more resilient to replacement over years of change.

7.3 Policy-Based Design

7.3.1 What Policy-Based Design Means in Modern C++

Policy-based design is **compile-time composition**: you build a type by combining *policies* (small classes or callables) that each encode one decision.
Instead of inheritance hierarchies, policy-based design uses:

- templates (including default template parameters),
- concepts/constraints to specify required behavior,
- value members (policies stored inside the composed type),
- static dispatch and inlining.

The goal is not "generic programming for its own sake." The goal is architectural:

- isolate variability,
- prevent coupling to concrete mechanisms,
- allow high-performance substitution without runtime dispatch,

- keep responsibilities cohesive and explicit.

A concise description:

> **A policy is a small, replaceable unit of behavior. Policy-based design composes a system from such units at compile time.**

7.3.2 Why Policy-Based Design Belongs in "Composition Over Inheritance"

Inheritance tends to bind:

- structure,
- behavior,
- and extension points

into a single coupling mechanism.
Policy-based design separates them:

- class composition is explicit in the template parameter list,
- behavior is a value member or static function,
- extension points are expressed as requirements (concepts) rather than override chains.

This is composition in its strongest form:

- no base class contract fragility,
- no vtables,
- no virtual destructors,
- no runtime indirection unless you choose to add it.

7.3.3 Core Benefits (When Used Correctly)

1) Zero-cost substitutability

The compiler chooses the implementation at compile time:

- enables inlining,
- enables constant propagation,
- removes virtual call overhead.

2) Cohesion by construction

Each policy should represent one responsibility:

- allocation strategy,
- logging/auditing strategy,
- error handling strategy,
- concurrency strategy,
- caching strategy,
- retry/backoff strategy.

3) Explicit variability

The variability axis is visible in the type definition:

- readers see what changes and what does not,
- maintainers can add a policy without refactoring a hierarchy.

4) Testability without mocks

Policies can be swapped with test doubles at compile time.

7.3.4 Primary Costs (The Design Must Pay Them Honestly)

Policy-based design is not free:

- **compile-time cost**: templates increase build times and rebuild scope,
- **binary size**: many instantiations can bloat the binary,
- **API stability**: template interfaces in widely used headers are hard to evolve quietly,
- **diagnostics**: errors can become complex without good constraints.

Therefore, policy-based design is best applied:

- inside components and libraries,
- in hot paths where performance matters,
- behind stable runtime boundaries when the API has high fan-in.

7.3.5 The Shape of a Policy: A Small, Focused Behavioral Unit

A policy is typically one of:

- a callable object with `operator()`,
- a small class with a named method (`log()`, `allocate()`, `on_error()`),
- a type providing static functions,
- a traits type providing constants and types.

7.3.6 Example 1: Logging Policy (No Runtime Dispatch)

Policies

```
#include <string_view>
#include <cstdio>

struct NoLog {
  void log(std::string_view) const noexcept {}
};

struct StdoutLog {
  void log(std::string_view msg) const noexcept {
    std::fwrite(msg.data(), 1, msg.size(), stdout);
    std::fwrite("\n", 1, 1, stdout);
  }
};
```

Composed type

```
#include <string_view>
#include <utility>

template <class LogPolicy = NoLog>
class Worker {
public:
  explicit Worker(LogPolicy log = {}) : log_(std::move(log)) {}

  void run(std::string_view job) {
    log_.log("start");
    (void)job; // work
```

```
    log_.log("done");
  }

private:
  LogPolicy log_;
};
```

This design allows:

- `Worker<>` in production hot paths with no logging cost,
- `Worker<StdoutLog>` in debugging builds,
- without any virtual base classes or runtime checks.

7.3.7 Example 2: Error Handling Policy (Return vs Throw vs Terminate)

A critical engineering axis is error strategy. Policy-based design lets you encode it explicitly.

Policies

```
#include <stdexcept>
#include <string_view>

struct ThrowOnError {
  [[noreturn]] void fail(std::string_view msg) const {
    throw std::runtime_error(std::string(msg));
  }
};

struct TerminateOnError {
  [[noreturn]] void fail(std::string_view) const noexcept {
```

```
    std::terminate();
  }
};
```

Composed type

```
#include <string_view>
#include <utility>

template <class ErrorPolicy>
class Parser {
public:
  explicit Parser(ErrorPolicy ep = {}) : ep_(std::move(ep)) {}

  int parse_int(std::string_view s) {
    if (s.empty()) ep_.fail("empty input");
    int v = 0;
    for (char c : s) {
      if (c < '0' || c > '9') ep_.fail("non-digit");
      v = v * 10 + (c - '0');
    }
    return v;
  }

private:
  ErrorPolicy ep_;
};
```

Now the system chooses a semantic contract at compile time:

- throwing parser,

- terminating parser (safety-critical, fail-fast),
- and you can add more policies (error codes, expected types) later.

7.3.8 Using Concepts to Make Policies Explicit and Safe

Unconstrained templates accept anything and fail late with poor diagnostics. Modern C++ should constrain policies.

Policy concept

```
#include <concepts>
#include <string_view>

template <class P>
concept LogPolicy = requires(const P p, std::string_view msg) {
  { p.log(msg) } noexcept -> std::same_as<void>;
};
```

Constrained composition

```
#include <utility>

template <LogPolicy P>
class SafeWorker {
public:
  explicit SafeWorker(P p = {}) : log_(std::move(p)) {}
  void run(std::string_view job) {
    log_.log("run");
    (void)job;
  }
```

```
private:
  P log_;
};
```

This turns policy-based design from “template magic” into a readable contract.

7.3.9 Policy Interactions: Avoiding the “Combinatorial Explosion”

A common fear is that policies create too many combinations. The solution is architectural discipline:

- Keep policies **orthogonal** (each one axis of variability).
- Avoid policies that leak assumptions about other policies.
- Prefer **small number of sanctioned configurations** (typedefs / using-aliases).
- Use a stable runtime boundary if the configuration must vary at runtime.

Provide named configurations

```
using FastWorker = Worker<NoLog>;
using DebugWorker = Worker<StdoutLog>;
```

7.3.10 Where Policy-Based Design Works Best

1) Low-level libraries and utilities

- allocators and memory strategies,
- containers and buffering,
- parsers and formatters,
- numeric kernels.

2) Performance-critical internal components

- networking pipelines,
- packet processing,
- game loops,
- trading systems,
- embedded real-time components.

3) Safety and correctness enforcement

- error strategy policies,
- overflow handling policies,
- bounds checking policies,
- logging/auditing policies.

7.3.11 Where Policy-Based Design Becomes Harmful

1) High fan-in public APIs

If hundreds of targets include a template-heavy public header:

- build time can dominate,
- small changes cause rebuild storms,
- versioning becomes difficult.

2) Runtime variation requirements

If behavior must change based on configuration, user input, or plugins:

- compile-time policy selection cannot help,
- you need interfaces or type erasure boundaries.

3) Policies that encode too much domain logic

Policies are best for *mechanisms and strategies*, not for core domain rules that require clarity over genericity.

7.3.12 Integration Pattern: Policy-Based Internals Behind a Stable Boundary

A strong long-lived-system pattern is:

- keep internal performance code policy-based,
- expose a stable runtime boundary (interface or facade) to the rest of the system.

This preserves:

- optimization and configurability,
- build scalability,
- API stability,
- architectural layering.

7.3.13 Review Checklist: Policy-Based Design

Use this checklist to judge a policy-based design:

1. **Variability axis:** can you name exactly what the policy varies?
2. **Cohesion:** is each policy one responsibility?
3. **Orthogonality:** do policies avoid hidden coupling between each other?
4. **Constraints:** are policies constrained with concepts (or clear requirements)?
5. **Fan-in:** will this template surface be included by many dependents?
6. **Runtime need:** does variation need to occur at runtime instead?
7. **Sanctioned configs:** do you provide a small set of named combinations?

7.3.14 Summary

Policy-based design is compile-time composition that replaces inheritance hierarchies with explicit, orthogonal strategies. It is most valuable when:

- performance matters,
- variability is known at compile time,
- contracts can be expressed with concepts,
- the design benefits from explicit, cohesive responsibility separation.

It becomes harmful when template-heavy public APIs increase physical coupling and build cost, or when the variability must be chosen at runtime. The long-lived-system approach is hybrid:

- policy-based design internally for efficiency and clarity,
- stable runtime boundaries externally for replaceability and scalable builds.

7.4 Mixins, CRTP, and Compile-Time Composition

7.4.1 Why Compile-Time Composition Exists

Composition is not only runtime object assembly; Modern C++ also supports **compile-time composition**: building types by combining behavior through templates rather than virtual hierarchies.
The motivation is engineering:

- preserve **zero-cost abstraction**: no vtables, no virtual calls,
- enable aggressive inlining and optimization in hot paths,
- express small, reusable behavior units as **orthogonal building blocks**,
- avoid runtime ownership graphs and fragile base-class substitution contracts.

However, compile-time composition is a power tool:

- it can improve clarity and performance,
- or it can explode build times, complexity, and readability.

This section explains three closely related techniques:

- **Mixins** (behavioral building blocks),
- **CRTP** (Curiously Recurring Template Pattern),
- **compile-time composition** (assembling types through templates and constraints).

7.4.2 Definitions

Mixin

A mixin is a template base class that *adds* behavior to a derived type. It is not intended to model "is-a" in the domain sense; it is intended to model:

> **"has this capability."**

CRTP

CRTP is a pattern where a derived class passes itself as a template argument to a base class:

- base class calls derived behavior via `static_cast<Derived&>(*this)`,
- enabling static polymorphism and shared implementation.

Compile-Time Composition

A family of techniques where types are assembled from:

- mixins,
- policy classes,
- concepts and constrained templates,
- and value members,

with variability resolved at compile time.

7.4.3 Pattern 1: Basic Mixin for Capability Injection

Intent

Add a small, orthogonal capability to many types without repeating code.

Example: Comparable mixin (safe and small)

```
template <class Derived>
struct EqualityComparable {
  friend bool operator!=(const Derived& a, const Derived& b) {
    return !(a == b);
  }
};

struct UserId : EqualityComparable<UserId> {
  int v{};
  friend bool operator==(const UserId& a, const UserId& b) { return a.v == b.v; }
};
```

Engineering properties

- no runtime dispatch,
- no ownership complexity,
- behavior is local and visible,
- best used for small operators or utilities.

7.4.4 Pattern 2: CRTP as Static Polymorphism (Algorithm Skeleton Without Virtuals)

Intent

Define an algorithm skeleton once, but allow customization points in derived types, while enforcing invariants in the base.

Example: NVI-like structure with CRTP

```
#include <string_view>

template <class Derived>
class ParserBase {
public:
  bool parse(std::string_view input) {
    if (input.empty()) return false;           // invariant
    before(input);                             // shared step
    const bool ok = derived().parse_impl(input); // customization point
    after(ok);                                 // shared step
    return ok;
  }

private:
  Derived& derived() { return static_cast<Derived&>(*this); }

  void before(std::string_view) {}
  void after(bool) {}
};

class IntParser : public ParserBase<IntParser> {
public:
  bool parse_impl(std::string_view s) {
    for (char c : s) if (c < '0' || c > '9') return false;
    return true;
  }
};
```

Why this is legitimate

- invariants and the skeleton stay centralized,
- customization is explicit (`parse_impl`),
- no virtual dispatch cost,
- no ABI/vtable constraints.

7.4.5 Pattern 3: Stackable Mixins (Multiple Capabilities by Composition)

Intent

Compose multiple orthogonal behaviors by stacking mixins as base classes.

Example: Stackable capability mixins

```
#include <string_view>
#include <cstdio>

template <class Derived>
struct Named {
  std::string_view name() const noexcept {
    return static_cast<const Derived&>(*this).name_impl();
  }
};

template <class Derived>
struct Printable {
  void print() const {
    auto sv = static_cast<const Derived&>(*this).to_string_impl();
```

```
    std::fwrite(sv.data(), 1, sv.size(), stdout);
    std::fwrite("\n", 1, 1, stdout);
  }
};

class Component : public Named<Component>, public Printable<Component> {
public:
  std::string_view name_impl() const noexcept { return "Component"; }
  std::string_view to_string_impl() const noexcept { return "Component{}"; }
};
```

Engineering caution

Stackable mixins can become unreadable if:

- too many mixins are stacked,
- mixins are not orthogonal,
- mixins interact implicitly (hidden coupling).

7.4.6 Concept-Constrained CRTP and Mixins (Modern C++ Discipline)

Unconstrained templates accept anything and fail late. Modern C++ should constrain expectations.

Example: constrain a CRTP requirement

```
#include <concepts>
#include <string_view>

template <class D>
```

```
concept HasParseImpl = requires(D d, std::string_view s) {
  { d.parse_impl(s) } -> std::same_as<bool>;
};

template <class Derived>
requires HasParseImpl<Derived>
class SafeParserBase {
public:
  bool parse(std::string_view input) {
    if (input.empty()) return false;
    return static_cast<Derived&>(*this).parse_impl(input);
  }
};
```

This improves:

- diagnostics,
- readability of the contract,
- safety of generic composition.

7.4.7 Compile-Time Composition vs Runtime Composition: The Correct Boundary

A long-lived system often uses a hybrid:

- **Compile-time composition** inside performance-critical algorithms and low-level utilities.
- **Runtime composition** (interfaces/type erasure) at subsystem boundaries for replacement, build scalability, and ABI stability.

A pragmatic boundary rule:

> **Use CRTP/mixins where the behavior is small, local, and performance-sensitive. Use runtime boundaries where replacement and stability matter more than inlining.**

7.4.8 Failure Modes: When Mixins and CRTP Become a Liability

Failure 1: "Mixin soup" (too many capabilities)

Symptoms:

- inheritance lists become long and unreadable,
- behavior is scattered across many templates,
- surprising name lookup and overload resolution issues occur.

Correctives:

- reduce mixin count,
- prefer value-member composition for large behaviors,
- group cohesive capabilities into a single policy object.

Failure 2: Implicit coupling between mixins

If mixin A assumes mixin B is present, the composition is no longer orthogonal. Correctives:

- encode dependencies explicitly via requirements/concepts,
- refactor into a single cohesive component.

Failure 3: ODR/build explosion

CRTP and mixins live in headers. In high fan-in code, this can:

- slow builds,
- increase rebuild scope,
- increase binary size from many instantiations.

Correctives:

- confine templates to internal libraries,
- keep public APIs non-template where feasible,
- expose stable runtime boundaries for widely used components.

Failure 4: Debugging complexity

Static composition can:

- create long template backtraces,
- hide call paths behind inlining,
- reduce clarity in stack traces.

Correctives:

- keep patterns small and explicit,
- prefer clear names for customization points (`*_impl`),
- constrain templates to reduce diagnostic noise.

7.4.9 Practical Rules for Safe Use in Production

1. **Keep mixins small and orthogonal.**
2. **Keep CRTP bases lightweight and behavior-focused.**
3. **Avoid deep CRTP chains.** One or two levels is typically enough.
4. **Constrain requirements with concepts** to improve correctness and diagnostics.
5. **Prefer value composition for large behaviors** and for ownership-heavy logic.
6. **Avoid exposing template-heavy designs as widely used public APIs** unless necessary.

7.4.10 Review Checklist: Mixins and CRTP

1. **Does this represent a capability, not a domain "is-a" relationship?**
2. **Is the behavior small enough to justify header-only composition?**
3. **Are responsibilities orthogonal, with no hidden coupling between mixins?**
4. **Are customization points explicit and constrained?**
5. **Will this increase build times or instantiation count unacceptably?**
6. **Would runtime composition be better at this boundary?**

7.4.11 Summary

Mixins and CRTP are compile-time composition tools that allow Modern C++ to:

- reuse behavior without virtual hierarchies,
- enforce algorithm skeletons and invariants without runtime dispatch,

- assemble types from orthogonal capabilities.

They are legitimate and powerful when kept small, constrained, and local. They become liabilities when overused, when mixins interact implicitly, or when template-heavy public APIs damage build scalability and readability. The long-lived-system strategy is hybrid:

- use CRTP/mixins for internal performance-sensitive composition,
- use runtime boundaries at subsystem edges for replaceability and stability.

Part III

Object Model, Ownership, and Lifetime Design

Value Types vs Identity Types

8.1 What Makes an Object a “Value”

8.1.1 The Engineering Definition of a Value Object

In long-lived systems, “value” is not a philosophical label; it is an operational design decision. An object is a **value** when it is defined primarily by its **state** (its contained data), not by a persistent identity.
A practical engineering definition:

> **A value object is one where copying produces an equivalent object, equality compares state, and the object can be freely moved, stored, and replaced without changing what it *is*.**

This implies that a value:

- is **substitutable** by another instance with the same state,
- does not require stable addresses to preserve meaning,
- does not represent a unique, long-lived “entity” in the domain,
- is naturally compatible with RAII and deterministic lifetime management.

8.1.2 Values vs Identities: The Boundary of Meaning

Values and identities differ in what makes the object *the same* across time.

Value sameness (state-based)

- two objects are the "same" if their observable states are equal,
- copying is semantic duplication, not "creating a different thing."

Identity sameness (continuity-based)

- the object is the "same" if it is the same entity across time,
- copying does not create another "same entity"; it creates a new entity or is forbidden.

This chapter section focuses on the value side: what properties make a design safely behave like a value.

8.1.3 The Canonical Properties of Value Types in Modern C++

Property 1: Regularity (Value Semantics as a Contract)

A strong value type behaves like an *ordinary* type:

- it can be default-constructed (when it makes sense),
- copied and moved,
- assigned,
- compared for equality,
- and destroyed safely.

In practice, value types aim to be **regular**: they support predictable behavior under copying, assignment, and comparison. This regularity enables:

- use in containers,
- safe passing by value,
- local reasoning about lifetime,
- trivial refactoring when algorithms change.

Property 2: Copying Preserves Meaning

Copying a value should produce an independent object that represents the same value. Therefore:

- copying must not create hidden aliasing,
- copying must not share mutable internal state unless it is semantically immutable.

Property 3: Equality is State-Based

Equality should mean:

- if two values compare equal, they are interchangeable in most computations.

This implies:

- `operator==` compares the essential state,
- equality is consistent and stable over the lifetime of the compared objects,
- equality does not depend on address, allocation, or incidental representation.

Property 4: No External Ownership Obligations

A value does not impose lifetime obligations on the caller. Values:

- own their resources (if any) via RAII,
- do not require the caller to manage hidden lifetimes,
- do not leak raw owning pointers.

Property 5: Replacement-Friendly (Substitutable and Swappable)

Values should support:

- swapping,
- moving between containers,
- reassigning,
- returning by value.

Replacement-friendly design minimizes coupling to specific locations or allocation sites.

8.1.4 A Value is Not "Small"; a Value is "Well-Behaved"

A common misconception: values must be small POD structs. In Modern C++, values can own resources and still remain values:

- `std::string`, `std::vector`, `std::filesystem::path`,
- any RAII type that copies/moves with clear semantics.

The key is semantic correctness:

- copying and moving must preserve meaning,
- resource ownership must be explicit and safe,
- invariants must be enforced.

8.1.5 Example: A Strong Value Object With an Invariant

A value object is often used to encode domain invariants as types.

```
#include <string>
#include <string_view>
#include <stdexcept>
#include <utility>

class Money {
public:
  // invariant: amount is in minor units (e.g., cents) and non-negative
  explicit Money(long long minor_units) : cents_(minor_units) {
    if (cents_ < 0) throw std::invalid_argument("negative money");
  }

  long long cents() const noexcept { return cents_; }

  friend bool operator==(Money a, Money b) noexcept { return a.cents_ == b.cents_; }
  friend bool operator!=(Money a, Money b) noexcept { return !(a == b); }

  friend Money operator+(Money a, Money b) { return Money(a.cents_ + b.cents_); }
  friend Money operator-(Money a, Money b) {
    if (a.cents_ < b.cents_) throw std::invalid_argument("underflow");
    return Money(a.cents_ - b.cents_);
```

```
  }

private:
  long long cents_{};
};
```

This is a value because:

- copying produces the same monetary value,
- equality is state-based,
- invariants are enforced,
- it has no identity beyond its value.

8.1.6 Non-Examples: When an Object Is Not a Value

Non-example 1: Equality based on address

```
struct Widget {
  int x{};
  friend bool operator==(const Widget& a, const Widget& b) noexcept {
    return &a == &b; // identity comparison, not value comparison
  }
};
```

This is not a value comparison. It encodes identity semantics.

Non-example 2: Copying shares mutable state (hidden aliasing)

```
#include <memory>
```

```
class Bag {
public:
  Bag() : p_(std::make_shared<int>(0)) {}

  void set(int v) { *p_ = v; }
  int  get() const { return *p_; }

private:
  std::shared_ptr<int> p_; // copying Bag shares mutable state
};
```

If Bag is intended to be a value, this is a violation: copying creates aliasing; changes in one instance affect the other. That is an identity-like behavior (shared identity of the internal resource), not value semantics.

8.1.7 Value Types and Resource Ownership: The RAII Requirement

A value type that owns resources must obey RAII:

- resource acquisition happens in construction,
- release happens in destruction,
- copying and moving preserve valid states and clear ownership rules.

In a well-designed value:

- moving transfers ownership, leaving the moved-from object valid,
- copying duplicates the state (deep copy or equivalent semantics),
- no “manual free” is required by clients.

8.1.8 The Rule of Zero and the Rule of Five: Value-Semantics Guidance

Value types should aim for the **Rule of Zero**:

- prefer composing from members that already manage their own resources,
- let the compiler generate destructor/copy/move operations safely.

If a type directly owns a raw resource (file descriptor, handle, memory), it may need the **Rule of Five**:

- destructor,
- copy constructor,
- copy assignment,
- move constructor,
- move assignment,

to define correct ownership behavior.
The key point for design:

> **If you cannot define correct copy semantics, the type is not a value and should not pretend to be one.**

8.1.9 Value Objects in Domain Modeling: "Entity vs Value" Mapping

In domain modeling, value objects represent:

- measurements (money, distance, time duration),
- identifiers as values (when they are immutable and state-based),

- coordinates, ranges, and intervals,
- configuration snapshots,
- small immutable records used as facts.

These are ideal as value types because:

- they have clear equality semantics,
- they are easily copied and stored,
- they improve correctness by encoding invariants.

8.1.10 A Practical Checklist: "Is This a Value?"

Use this checklist before committing to value semantics:

1. **Is equality state-based?** (not address-based)
2. **Does copying preserve meaning?** (no hidden aliasing)
3. **Is the type safely movable?** (moved-from remains valid)
4. **Are invariants enforced by construction?**
5. **Does it own its resources via RAII or borrow explicitly?**
6. **Can it be replaced freely without changing what it represents?**

8.1.11 Design Guidance: How to Make Values "Strong" in C++

1) Prefer immutability where feasible

Immutability strengthens value semantics and simplifies reasoning.

2) Make invalid states unrepresentable

Use constructors and private members to enforce invariants.

3) Keep interfaces small and intention-revealing

Values should expose operations that preserve invariants.

4) Avoid shared ownership inside a value type

If you use `shared_ptr`, you are likely expressing identity semantics. If you need sharing, be explicit about it and do not label the type as a value.

5) Use views carefully

Types like `string_view` and `span` are *non-owning*. They can be used inside value types only when:

- the referenced data is guaranteed to outlive the value object,
- or the value object is itself a view type (explicitly non-owning).

8.1.12 Summary

An object is a "value" when:

- it is defined by state rather than persistent identity,
- copying produces an equivalent independent object,
- equality compares meaningful state,
- resource management is explicit and safe (RAII),

- the object is freely replaceable in computation.

In Modern C++, strong value types are a cornerstone of long-lived systems: they improve correctness, reduce coupling, simplify ownership, and make code easier to refactor and evolve.

8.2 Identity, Mutability, and Aliasing

8.2.1 Why This Triangle Defines Real Software Failures

Most long-lived system failures that appear as "random bugs" are rooted in three interacting forces:

- **Identity**: the continuity of an entity across time (the thing is the same even when its state changes).
- **Mutability**: state can change after construction.
- **Aliasing**: multiple names/references/handles can access the same underlying object.

Individually, each is manageable. Together, they are the primary source of:

- unexpected side effects,
- invalidation bugs and dangling references,
- data races and concurrency defects,
- invariant violations that are hard to reproduce,
- security-relevant state corruption.

In Modern C++ design, the difference between robust and fragile systems is often:

> **Do we control identity, mutability, and aliasing explicitly—or do we let them happen implicitly?**

8.2.2 Identity Types: What Makes an Object "The Same" Across Time

An **identity type** represents an entity whose meaning is not determined only by its current state. It remains "the same thing" even as it changes.
Typical identity types in software:

- a user session,
- a connection object,
- a device driver instance,
- a cache entry,
- an in-memory aggregate that evolves with events,
- a GUI widget,
- a game entity,
- an actor with a mailbox.

Identity implies continuity. Therefore:

- copying usually does *not* mean duplicating the entity,
- equality is rarely full state equality (often identity-based or key-based),
- stable addresses and stable handles often matter.

8.2.3 Mutability: The Operational Reason for Identity

Identity is typically needed because the object is mutable across time.
If something never changes and is defined by state, it is naturally a value. When something must evolve:

- state transitions matter,
- invariants must hold across transitions,
- observers may need to see the evolving entity.

This is where design becomes engineering:

- decide who is allowed to mutate,
- decide how the mutation is synchronized,
- decide what aliases exist and what they can do.

8.2.4 Aliasing: The Real Problem Most Systems Fail to Model

Aliasing means there exist multiple access paths to the same object:

- two pointers to the same allocation,
- multiple references into a container,
- `shared_ptr` copies,
- iterators and views,
- global registries returning borrowed handles.

Aliasing is not a moral failure; it is sometimes required. But uncontrolled aliasing produces:

- action-at-a-distance (mutation through one alias surprises another),
- lifetime confusion (who keeps the object alive?),
- invalidation bugs (container reallocation invalidates views/iterators),
- concurrency hazards (two threads mutate through different aliases).

8.2.5 Identity + Mutability + Aliasing: The Most Dangerous Combination

The most dangerous object category is:

> **A mutable identity type with shared aliasing.**

Because:

- identity requires continuity,
- mutability requires state transitions,
- aliasing means those transitions can occur from many places.

This combination is the primary engine of non-local reasoning and unpredictable behavior.

8.2.6 Design Goal: Make Aliasing Explicit and Constrained

A healthy design does not attempt to eliminate aliasing everywhere. It attempts to **control** it. Practical strategies:

- keep identity objects behind stable boundaries (repositories, registries, subsystems),
- restrict mutation to a small set of methods and enforce invariants there,
- expose read-only views broadly, but restrict write access,
- prefer unique ownership with explicit borrowing,
- avoid shared ownership unless shared lifetime is truly required.

8.2.7 Aliasing and Ownership in Modern C++: What the Types Mean

Exclusive ownership: `std::unique_ptr`

- one owner,
- clear lifetime root,
- aliasing possible through borrowed references, but ownership remains single.

Shared ownership: `std::shared_ptr`

- multiple owners,
- shared lifetime graph,
- aliasing is the default,
- mutation becomes much harder to reason about.

Borrowing: references, pointers, `span`, `string_view`

- non-owning,
- safe only under explicit lifetime guarantees,
- extremely powerful but easily misused.

8.2.8 Concrete Failure Mode 1: Shared Ownership Used as a Convenience

When `shared_ptr` is used "to avoid thinking about lifetime," a mutable identity object becomes globally aliasable and thus globally mutable.

Bug shape: action-at-a-distance

```
#include <memory>
#include <string>

struct Session {
  std::string token;
  bool admin{};
};

void elevate(std::shared_ptr<Session> s) {
  s->admin = true; // mutation through one alias
}

void check(const std::shared_ptr<Session>& s) {
  // Someone elsewhere may have mutated it.
  if (s->admin) {
    // behavior changes non-locally
  }
}
```

If `Session` is identity + mutable, widespread `shared_ptr` means:

- no clear authority over mutation,
- no clear invariants enforcement point,
- concurrency and security hazards.

Correction: make mutation authority explicit

A typical correction is to:

- centralize mutation in a manager/service,
- expose read-only views to the rest of the system,
- keep ownership rooted and local.

8.2.9 Concrete Failure Mode 2: Borrowed Views Stored as If They Were Owned

Borrowed views are aliasing tools, not ownership tools. Storing them inside long-lived objects without lifetime guarantees creates dangling references.

Bug: storing `string_view` into an identity object

```
#include <string_view>

class UserProfile {
public:
  void set_name(std::string_view n) { name_ = n; } // BUG: name_ may dangle
  std::string_view name() const noexcept { return name_; }

private:
  std::string_view name_{};
};
```

Correction: own the state if the identity must persist

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

class UserProfile {
```

```
public:
  void set_name(std::string_view n) { name_.assign(n.data(), n.size()); }
  std::string_view name() const noexcept { return name_; }

private:
  std::string name_{};
};
```

The rule:

> **If the object outlives the input, it must own the data or hold a proven stable handle.**

8.2.10 Concrete Failure Mode 3: Container Invalidation and Aliasing

Many aliasing bugs are not about pointers to heap allocations. They are about:

- references/iterators/views into containers,
- invalidated by reallocation or erase operations.

Bug shape: store pointer/reference to vector element

```
#include <vector>

struct Node { int v{}; };

void f() {
  std::vector<Node> nodes;
  nodes.push_back(Node{1});
  Node* p = &nodes[0];        // alias into vector storage
```

```
  nodes.push_back(Node{2});   // may reallocate, p may become dangling

  (void)p->v; // use-after-reallocation
}
```

Corrections

Options include:

- store indices instead of pointers when stable indexing is acceptable,
- reserve capacity to prevent reallocation when size is known,
- use node-based containers when stable addresses are required,
- use owning indirection (e.g., `unique_ptr<Node>`) inside the container to stabilize pointees.

8.2.11 Identity Types: How to Model Them Safely in Modern C++

Rule 1: Separate identity from value state when possible

Often the best design is:

- an identity object owns a value state snapshot,
- modifications replace the state (copy-on-write or transactional update),
- readers see values, not mutable internals.

Rule 2: Centralize mutation

Define a clear authority:

- one service or aggregate root controls updates,
- invariants are enforced there,
- external code cannot partially mutate internals.

Rule 3: Provide narrow mutation APIs

Avoid public writable fields and wide "set everything" interfaces. Prefer intent-level operations:

- `activate()`,
- `renew()`,
- `invalidate()`,
- `apply(event)`.

Rule 4: Use borrowing for observation, not for lifetime management

Borrowed references are excellent for:

- short-lived observation,
- read-only access,
- local computations.

They are dangerous as stored state.

Rule 5: Be explicit about thread-safety

Aliasing across threads is a contract issue. If an identity object may be shared across threads:

- either make it immutable and replace it atomically,

- or protect its mutation with synchronization,
- or confine it to a single thread (actor-style ownership).

8.2.12 A Safe Architectural Pattern: Identity Behind Handles

Rather than exposing raw pointers/references widely, expose *handles*:

- IDs that are values,
- opaque tokens,
- stable indices validated by a manager.

This converts:

- identity and lifetime management into a dedicated subsystem responsibility,
- aliasing into explicit handle usage.

Example: handle-based access

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

struct SessionId {
  std::string v;
  friend bool operator==(const SessionId& a, const SessionId& b) noexcept { return
  ↪  a.v == b.v; }
};

struct Session {
```

```
  std::string token;
  bool admin{};
};

class SessionStore {
public:
  std::optional<Session> get(SessionId id) const {
    auto it = sessions_.find(id.v);
    if (it == sessions_.end()) return std::nullopt;
    return it->second; // returns a value snapshot
  }

  void set_admin(SessionId id, bool admin) {
    auto it = sessions_.find(id.v);
    if (it != sessions_.end()) it->second.admin = admin; // centralized mutation
  }

private:
  std::unordered_map<std::string, Session> sessions_;
};
```

This design:

- avoids leaking aliasing of the internal `Session`,
- centralizes mutation,
- returns value snapshots for observation.

8.2.13 Practical Checklist: Designing Identity Types Without Aliasing Damage

1. **Do we truly need identity?** If state-based equality is enough, prefer a value.
2. **Who is allowed to mutate?** Name the authority; enforce invariants there.
3. **How many aliases exist?** Minimize and make them explicit (handles, borrowed refs).
4. **Is lifetime rooted?** Identify the owner that guarantees object existence.
5. **Are we storing borrowed views?** If yes, prove lifetime or switch to owning storage.
6. **Can container invalidation break aliases?** If yes, redesign storage or referencing strategy.
7. **Is concurrency involved?** If yes, define thread-safety as part of the contract.

8.2.14 Summary

Identity types exist because systems model entities that persist and evolve across time. Mutability gives those entities life, but aliasing multiplies the number of ways the system can be wrong. Robust Modern C++ design controls the triangle:

- keep identity behind explicit boundaries and handles,
- centralize mutation and enforce invariants,
- prefer values and snapshots for observation,
- use borrowing carefully and never store views without proven lifetimes,
- treat shared ownership as a deliberate semantic choice, not a convenience.

The result is a system where identity is preserved, mutation is safe, and aliasing is controlled rather than accidental.

8.3 Designing Value-Centric APIs

8.3.1 Why Value-Centric APIs Are the Default in Modern C++

A value-centric API is one where the primary interaction model uses **values**:

- inputs are passed as values or read-only views,
- outputs are returned by value,
- invariants are encoded in types,
- ownership is explicit and local,
- mutation is either avoided or tightly controlled.

In long-lived C++ systems, value-centric APIs are a default choice because they:

- reduce aliasing and non-local side effects,
- make lifetime behavior deterministic (RAII),
- improve testability (pure functions and stable contracts),
- improve composability (algorithms and containers work naturally),
- limit concurrency hazards (values can be transferred safely).

A practical rule:

> **If the caller does not need to observe continuous identity over time, design the API around values.**

8.3.2 Core Principles of Value-Centric API Design

Principle 1: State-Based Meaning and Substitutability

Values are defined by state. Therefore:

- equal values are substitutable,
- copying is semantically valid,
- returning a value is safe and self-contained.

Principle 2: Make Invalid States Unrepresentable

A value-centric API uses types to enforce invariants:

- strong types instead of raw primitives,
- constructors enforce constraints,
- narrow operations preserve invariants.

Principle 3: Ownership Is Explicit

Value-centric APIs avoid ambiguous lifetime rules:

- values own their resources,
- views are explicitly non-owning,
- owning pointers are used only when identity semantics are required.

Principle 4: Prefer Immutability (Or Controlled Mutation)

Values are easiest to reason about when immutable or effectively immutable. When mutation exists, keep it:

- local,
- explicit,
- and validated (invariant-preserving).

8.3.3 Practical API Shapes That Promote Value Semantics

1) Pass-by-value for ownership transfer

If the callee needs to take ownership, accept a value and move from it.

```
#include <string>
#include <utility>

class Mail {
public:
  explicit Mail(std::string body) : body_(std::move(body)) {}
  const std::string& body() const noexcept { return body_; }
private:
  std::string body_;
};

class Mailbox {
public:
  void push(Mail m) { // caller transfers ownership of the value
    store(std::move(m));
```

```
  }
private:
  void store(Mail) {}
};
```

Benefits:

- one clear ownership model,
- caller chooses to move or copy,
- no dangling pointers.

2) Pass-by-`const&` for large read-only values

Use `const&` when:

- the function does not need ownership,
- the object may be large,
- the caller controls lifetime for the duration of the call.

```
#include <vector>

struct Point { double x{}, y{}; };

double polyline_length(const std::vector<Point>& pts);
```

3) Views for non-owning, short-lived access

Use views (`string_view`, `span`) for:

- parameters,

- parsing,
- reading data without copying,

but do not store them unless lifetime is guaranteed by the object itself.

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

bool parse_header(std::string_view line);
std::size_t checksum(std::span<const std::byte> bytes) noexcept;
```

4) Return-by-value as the normal output model

Return-by-value:

- is safe (no lifetime traps),
- composes well,
- allows move optimizations,
- avoids “out parameters” that hide mutation.

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

std::string normalize_path(std::string_view p);
```

8.3.4 Use `std::optional` and `std::expected`-style Results Instead of Sentinel Values

Value-centric APIs avoid:

- special sentinel values,
- partially-initialized outputs,
- ambiguous error reporting.

Optional result for “maybe”

```
#include <optional>
#include <string_view>

struct UserId { int v{}; };

std::optional<UserId> parse_user_id(std::string_view s);
```

Expected-style result for “value or error”

If your environment has `std::expected` available, it is naturally value-centric. If not, you can still model the concept with a small domain result type.

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

struct Error {
  std::string msg;
};
```

```
template <class T>
class Result {
public:
  static Result ok(T v) { return Result(std::move(v)); }
  static Result fail(Error e) { return Result(std::move(e)); }

  bool has_value() const noexcept { return has_; }
  const T& value() const { return v_; }
  const Error& error() const { return e_; }

private:
  explicit Result(T v) : has_(true), v_(std::move(v)) {}
  explicit Result(Error e) : has_(false), e_(std::move(e)) {}

  bool has_{false};
  T v_{};
  Error e_{};
};

Result<int> parse_port(std::string_view s);
```

This style forces:

- explicit handling of failure,
- no hidden mutation to out-parameters,
- stable, testable contracts.

8.3.5 Strong Types: Make APIs Speak the Domain

Value-centric APIs often become self-documenting when primitives are replaced by domain types.

Example: prevent parameter confusion

```
#include <string>
#include <utility>

class UserName {
public:
  explicit UserName(std::string v) : v_(std::move(v)) {
    if (v_.empty()) throw std::invalid_argument("empty user name");
  }
  const std::string& value() const noexcept { return v_; }
private:
  std::string v_;
};

class Email {
public:
  explicit Email(std::string v) : v_(std::move(v)) {
    if (v_.find('@') == std::string::npos) throw
    ↪  std::invalid_argument("invalid email");
  }
  const std::string& value() const noexcept { return v_; }
private:
  std::string v_;
};
```

```
struct UserProfile {
  UserName name;
  Email email;
};
```

Now the API cannot accidentally swap `UserName` and `Email`.

8.3.6 Snapshot and Immutable Data Transfer Objects

For systems that manage identity internally, a value-centric API often returns **snapshots**:

- a stable value representing a moment-in-time state,
- safe to transfer across threads,
- safe to store and compare.

Example: snapshot return from an identity store

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

struct AccountSnapshot {
  int id{};
  long long balance_cents{};
  bool locked{};
};

class AccountStore {
public:
  std::optional<AccountSnapshot> get_snapshot(int id) const;
```

```
};
```

This avoids leaking aliasing of internal mutable identity objects.

8.3.7 Minimizing Hidden Aliasing in API Surfaces

Value-centric APIs avoid designs where copying unexpectedly shares mutable state.
Common pitfalls:

- returning raw pointers or references to internal state,
- storing `string_view` or `span` beyond the call boundary,
- returning iterators into internal containers that may invalidate,
- using `shared_ptr` as the default return type for "objects."

Safer alternatives:

- return values or snapshots,
- return owning containers,
- return stable IDs/handles,
- use iterators only for caller-owned containers.

8.3.8 Value-Centric APIs and Concurrency: Transfer, Do Not Share

A major reason to prefer values is concurrency.

Transfer ownership across threads

A value can be moved to another thread safely (subject to its own invariants). This encourages:

- message passing,
- work queues with move-only tasks,
- elimination of shared mutable state.

```
#include <string>
#include <utility>

struct Job {
  std::string payload;
};

void enqueue(Job j); // transfers ownership into a queue
```

8.3.9 Value-Centric APIs at Boundaries: Keep the Core Clean

Large systems benefit from a boundary strategy:

- **Core:** value-centric types, pure transformations, explicit invariants.
- **Edges:** adapters that translate to/from frameworks, I/O, databases, protocols.

This keeps:

- frameworks from infecting the domain model,
- ownership and lifetimes deterministic in the core,
- testing cheap and meaningful.

8.3.10 Checklist: Designing a Value-Centric API

Use this checklist when reviewing an API:

1. **Is the primary data model value-based?** (copyable, comparable, substitutable)
2. **Are invariants encoded in types and constructors?**
3. **Does the API avoid exposing internal references/pointers?**
4. **Are views used only for short-lived reading, not stored?**
5. **Are outputs returned by value or via explicit result types?**
6. **Is ownership transfer explicit (value parameters, `unique_ptr`)?**
7. **Does the API support concurrency by transfer rather than sharing?**
8. **Are error semantics explicit (optional/expected-style) rather than sentinels?**

8.3.11 Summary

Value-centric APIs are a cornerstone of long-lived Modern C++ systems because they:

- minimize aliasing and non-local side effects,
- make ownership and lifetime explicit and deterministic,
- enable safe concurrency through transfer semantics,
- improve correctness by encoding invariants in types,
- produce stable, testable, composable contracts.

Design with values by default, and introduce identity objects only when continuous identity and mutation across time are truly required.

8.4 When Identity Is Unavoidable

8.4.1 Identity Is Not a Preference; It Is a Constraint

In Modern C++ design, we prefer values because they are:

- copyable and replaceable,
- easy to reason about,
- naturally compatible with RAII and deterministic lifetime,
- safer under concurrency through transfer rather than sharing.

Yet some problems **cannot** be modeled correctly as pure values. Identity becomes unavoidable when the domain requires **continuity**: the object must remain "the same entity" across time, even as its state changes.
A precise engineering statement:

> **Identity is unavoidable when the correctness of the system depends on tracking a single evolving entity rather than a sequence of equivalent states.**

8.4.2 The Concrete Conditions That Force Identity

Identity is required when at least one of the following is true.

Condition 1: Long-Lived Mutable State Must Be Observed Over Time

If multiple parts of the system must observe the evolution of the same entity:

- sessions and authentication state,
- connection lifecycle and negotiated capabilities,

- device driver runtime state,
- UI widgets and their event state,
- game entities with continuous existence.

If you copy such an object as a value, you create *two histories*. That violates the domain model.

Condition 2: External Resources Impose Stable Handles

Identity is unavoidable when an object is fundamentally a handle to something external:

- file descriptors and sockets,
- OS handles (process, thread, mutex, shared memory),
- GPU resources,
- database connections and prepared statements,
- memory-mapped regions,
- network sessions.

The object represents:

- a resource that has one lifecycle,
- one ownership root,
- and must not be duplicated by copying unless duplication is explicitly defined.

Condition 3: Referential Integrity and Graph Structures

When the system contains a graph of objects where relationships must remain stable:

- object graphs with cycles,
- entity relationships (parent/child references),
- runtime dependency graphs and registries.

Pure values struggle here because copying a node must decide what happens to references:

- deep copy the entire graph,
- or produce broken references,
- or introduce hidden sharing.

Identity is the natural representation: stable nodes with controlled aliasing.

Condition 4: Event-Driven Systems and Message Routing

If an entity has:

- a mailbox,
- a subscription list,
- callbacks registered against it,
- or observers attached to it,

then identity is unavoidable. Observers need to refer to the same entity consistently.

Condition 5: Security, Authorization, and Audit Trails

When security decisions depend on an entity's evolving state:

- privileges granted/revoked,
- session expiry and renewal,
- rate limits and quotas,
- lockouts,

copying state as values can create inconsistent authoritative state. Identity objects often form the authoritative state boundary.

Condition 6: Shared Mutable State Is Architecturally Required

Ideally, we avoid shared mutable state. But some systems must share:

- caches,
- registries,
- process-wide pools,
- telemetry sinks,
- central schedulers.

These are identity-heavy components; the system must coordinate around *one* instance or a set of known instances.

8.4.3 The First Rule of Identity: Do Not Pretend It Is a Value

If a type has identity semantics, pretending it is a value is a design bug. This typically shows up as:

- default copy constructors that duplicate the handle incorrectly,
- equality that compares state when identity is what matters,
- accidental sharing via `shared_ptr` used as a default.

A strong guideline:

> **If copying would duplicate an entity incorrectly, then copying must be disabled or must be explicitly defined.**

8.4.4 Identity-Friendly C++ Design: The Standard Patterns

Once identity is unavoidable, the goal becomes:

> **Keep identity local, keep aliasing controlled, and make ownership explicit.**

Pattern 1: Move-only identity objects (the default)

Many identity objects should be move-only:

- file/socket handles,
- lock guards,
- unique resources,
- owning managers.

```
#include <utility>

class Socket {
public:
  Socket() = default;
  explicit Socket(int fd) : fd_(fd) {}

  Socket(const Socket&) = delete;
  Socket& operator=(const Socket&) = delete;

  Socket(Socket&& other) noexcept : fd_(std::exchange(other.fd_, -1)) {}
  Socket& operator=(Socket&& other) noexcept {
    if (this != &other) {
      close();
      fd_ = std::exchange(other.fd_, -1);
    }
    return *this;
  }

  ~Socket() { close(); }

  int native_handle() const noexcept { return fd_; }

private:
  void close() noexcept {
    if (fd_ != -1) {
      // close(fd_) would be here
      fd_ = -1;
    }
```

```
  }

  int fd_{-1};
};
```

This makes identity explicit:

- there is exactly one owner of the socket at a time,
- copying is forbidden because it would violate the resource lifecycle.

Pattern 2: Handle/ID as a value, entity as identity behind a store

This is one of the most robust large-system patterns:

- expose IDs (values) to most of the system,
- keep the mutable identity objects behind a store/manager,
- centralize mutation and invariant enforcement.

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

struct SessionId {
  std::string v;
  friend bool operator==(const SessionId& a, const SessionId& b) noexcept { return
  ↪  a.v == b.v; }
};

struct Session {
```

```
  std::string token;
  bool admin{};
};

class SessionStore {
public:
  std::optional<Session> snapshot(SessionId id) const {
    auto it = sessions_.find(id.v);
    if (it == sessions_.end()) return std::nullopt;
    return it->second; // value snapshot
  }

  void grant_admin(SessionId id) {
    auto it = sessions_.find(id.v);
    if (it != sessions_.end()) it->second.admin = true; // centralized mutation
  }

private:
  std::unordered_map<std::string, Session> sessions_;
};
```

The system gets:

- value-centric observation (snapshots),
- identity-centric mutation (inside the store),
- controlled aliasing (no raw pointers to sessions).

Pattern 3: Identity with explicit borrowing, not shared ownership

When runtime polymorphism is needed, prefer:

- ownership rooted in a composition root,
- dependencies borrowed as references,
- explicit lifetime assumptions.

```cpp
#include <string_view>

struct Handler {
  virtual ~Handler() = default;
  virtual void on_message(std::string_view msg) = 0;
};

class Router {
public:
  explicit Router(Handler& h) : h_(h) {}
  void deliver(std::string_view msg) { h_.on_message(msg); }

private:
  Handler& h_; // borrowed; Router does not own the identity object
};
```

This prevents “everyone owns everything” graphs.

Pattern 4: Polymorphic value semantics (rare, explicit clone)

Sometimes you must store heterogeneous identity-backed behavior while keeping value-like containers. This requires explicit cloning and is a specialized tool.

```cpp
#include <memory>
```

```cpp
struct Command {
  virtual ~Command() = default;
  virtual void exec() = 0;
  virtual std::unique_ptr<Command> clone() const = 0;
};

class CommandValue {
public:
  explicit CommandValue(std::unique_ptr<Command> p) : p_(std::move(p)) {}

  CommandValue(const CommandValue& other) : p_(other.p_->clone()) {}
  CommandValue& operator=(const CommandValue& other) {
    if (this != &other) p_ = other.p_->clone();
    return *this;
  }

  CommandValue(CommandValue&&) noexcept = default;
  CommandValue& operator=(CommandValue&&) noexcept = default;

  void exec() { p_->exec(); }

private:
  std::unique_ptr<Command> p_;
};
```

This keeps semantics honest:

- copying means cloning (creating a new entity instance),
- not sharing identity by accident.

8.4.5 Identity and Concurrency: Choose One of Three Models

Identity objects under concurrency must choose a model explicitly.

Model A: Single-thread confinement (actor-style ownership)

- identity lives on one thread,
- other threads communicate via messages,
- avoids shared mutable state.

Model B: Immutable snapshots with atomic replacement

- readers see values,
- updates replace the snapshot,
- avoids shared mutation, uses controlled publication.

Model C: Shared mutable identity with synchronization

- explicit locks or lock-free structures protect invariants,
- hardest to reason about,
- should be reserved for when A/B are not feasible.

A key rule:

> **If identity is shared across threads, thread-safety must be part of the type's contract, not a caller convention.**

8.4.6 Smells That Indicate Identity is Uncontrolled

- pervasive `shared_ptr<T>` for mutable objects,
- public setters and writable fields for identity objects,
- functions returning raw pointers/references to internal entities,
- storing `string_view`/`span` inside long-lived entities,
- "manager of managers" patterns that hide ownership roots.

8.4.7 A Checklist: "Identity is Unavoidable, Is It Designed Correctly?"

1. **Why identity?** Can you state the continuity requirement precisely?
2. **Copy semantics:** is copying disabled or explicitly defined (clone)?
3. **Ownership root:** where is the authoritative owner?
4. **Aliasing:** how many aliases exist and what can they do?
5. **Mutation authority:** who is allowed to mutate, and where are invariants enforced?
6. **Handles vs pointers:** can the system use IDs/handles instead of raw access?
7. **Concurrency model:** confinement, snapshot replacement, or synchronization?
8. **Boundary strategy:** is identity confined behind a store/facade and kept out of value-centric APIs?

8.4.8 Summary

Identity is unavoidable when correctness requires continuity:

- evolving entities must remain the same across time,
- external resources impose a unique lifecycle and stable handle,
- graphs, routing, observers, and registries require stable references,
- security and authorization rely on authoritative evolving state.

When identity is unavoidable, Modern C++ design responds with discipline:

- disable copying or define it explicitly,
- root ownership clearly (often move-only),
- centralize mutation and enforce invariants,
- control aliasing via handles, snapshots, and narrow APIs,
- define concurrency semantics as part of the contract.

The goal is not to eliminate identity; it is to prevent identity from infecting the whole architecture with uncontrolled aliasing and unpredictable lifetimes.

Ownership as a First-Class Design Concept

9.1 Ownership vs Lifetime

9.1.1 Why This Distinction Matters

In Modern C++ design, many defects that appear as:

- dangling references,
- double frees,
- leaks,
- unexpected mutation,
- shutdown-order crashes,
- data races,

are not caused by syntax mistakes, but by a design mistake:

> **Confusing ownership with lifetime.**

They are related concepts, but they are not the same.

9.1.2 Definitions: Two Different Questions

Ownership answers: "Who is responsible for releasing the resource?"

Ownership is a responsibility contract:

- exactly which object (or set of objects) is responsible for the resource,
- who must ensure correct destruction or release,
- who decides when the resource may be destroyed.

Lifetime answers: "How long does the resource remain valid for use?"

Lifetime is a validity interval:

- when the resource begins to exist (becomes usable),
- when it stops being valid (destroyed or invalidated),
- what operations may occur while it is alive.

A key principle for long-lived systems:

> **Ownership must be explicit; lifetime must be provable.**

9.1.3 The Relationship: Ownership Influences Lifetime, But Does Not Define It

Ownership often determines when destruction occurs, so it influences lifetime. But lifetime can extend beyond a particular owner (or appear to), and that is where design must be precise:

- A borrower can use a resource without owning it (lifetime must outlast the borrow).

- A non-owning view can point into owned storage (view lifetime is constrained).
- Shared ownership keeps an object alive while any owner remains, but does not define *safe access* rules (mutability, thread-safety).

9.1.4 The Four Fundamental Roles in an Ownership Design

Most robust systems can classify every reference into one role:

1. **Owner**: responsible for destruction/release.
2. **Borrower**: may use temporarily; must not outlive the owner.
3. **Observer**: reads state but does not prolong lifetime and must not mutate (often a const-borrow).
4. **Handle**: indirect reference where validity is mediated by a manager (IDs, tokens, indices).

Clarity improves when these roles are visible in types and APIs.

9.1.5 Ownership Models in Modern C++

Model A: Exclusive Ownership (Single Root)

Exclusive ownership means there is exactly one owner responsible for release. This is typically represented by:

- values (owning their subresources by RAII),
- `std::unique_ptr<T>` for heap ownership,
- move-only resource wrappers (file descriptors, sockets, mutex handles).

This model is preferred because:

- lifetime is easy to prove (tree-shaped ownership),
- teardown order is deterministic,
- aliasing does not imply shared ownership.

Model B: Shared Ownership (Graph Lifetime)

Shared ownership means multiple owners keep an object alive. This is represented by:

- `std::shared_ptr<T>` (+ `std::weak_ptr<T>` for breaking cycles).

Shared ownership is legitimate

9.2 Unique Ownership Models

9.2.1 What "Unique Ownership" Means in System Design

Unique ownership is an architectural contract:

> **At any time, exactly one owner is responsible for the lifetime and destruction of a resource.**

This contract is the foundation of predictable lifetime reasoning in Modern C++ because it:

- prevents double free by construction,
- eliminates ownership cycles,
- makes shutdown order deterministic,
- limits aliasing and therefore reduces non-local side effects,
- aligns naturally with RAII and move semantics,

- scales in large systems because ownership forms a tree (or a forest) rather than a graph.

In long-lived systems, unique ownership is not only about memory. It applies equally to:

- file descriptors and sockets,
- OS handles,
- locks and synchronization primitives,
- database connections,
- GPU resources,
- device interfaces,
- exclusive access to hardware or a subsystem.

9.2.2 The Core Unique-Ownership Representations in Modern C++

Unique ownership is expressed through three primary shapes.

Shape 1: Value ownership (RAII members)

A type owns sub-resources as members. Copy/move semantics follow the members. This supports the "Rule of Zero" most of the time.

Shape 2: Move-only resource wrapper

A type owns a non-copyable resource (handle, fd) and is therefore move-only.

Shape 3: Heap ownership via `std::unique_ptr`

Ownership of a dynamic object is exclusive and transferable by move.

9.2.3 Model 1: Unique Ownership by Value (The Default Model)

Intent

Use value members when:

- the owned components have correct RAII semantics,
- you want deterministic teardown order,
- you want maximum locality and minimum allocation.

Example: subsystem composed by value

```
#include <string>
#include <utility>

class Config {
public:
  explicit Config(std::string file) : file_(std::move(file)) {}
private:
  std::string file_;
};

class Metrics {
public:
  void start() {}
  void stop() noexcept {}
};

class Server {
public:
```

```
  explicit Server(std::string config_path)
    : cfg_(std::move(config_path)) {}

  void start() {
    metrics_.start();
    // start networking, etc.
  }

  ~Server() { metrics_.stop(); }

private:
  Config  cfg_;      // uniquely owned by value
  Metrics metrics_;  // uniquely owned by value
};
```

Design properties

- no heap required,
- ownership is self-evident,
- teardown order is the reverse of member declaration.

9.2.4 Model 2: Move-Only Resource Wrappers (Unique Ownership for Handles)

Intent

When a resource cannot be meaningfully copied (fd, handle), the owning type must be move-only.

Example: generic unique-handle wrapper

```
#include <utility>

template <class Handle, class Deleter>
class UniqueHandle {
public:
  UniqueHandle() = default;
  explicit UniqueHandle(Handle h, Deleter d = {}) : h_(h), del_(d) {}

  UniqueHandle(const UniqueHandle&) = delete;
  UniqueHandle& operator=(const UniqueHandle&) = delete;

  UniqueHandle(UniqueHandle&& other) noexcept
    : h_(std::exchange(other.h_, Handle{})), del_(std::move(other.del_)) {}

  UniqueHandle& operator=(UniqueHandle&& other) noexcept {
    if (this != &other) {
      reset();
      h_ = std::exchange(other.h_, Handle{});
      del_ = std::move(other.del_);
    }
    return *this;
  }

  ~UniqueHandle() { reset(); }

  Handle get() const noexcept { return h_; }

  Handle release() noexcept { return std::exchange(h_, Handle{}); }
```

```
  void reset(Handle nh = Handle{}) noexcept {
    if (h_ != Handle{}) del_(h_);
    h_ = nh;
  }

private:
  void reset() noexcept { reset(Handle{}); }

  Handle  h_{};
  Deleter del_{};
};
```

Design properties

- unique ownership is encoded by deleted copy operations,
- resource release is guaranteed by destructor,
- ownership can be transferred safely via move,
- the wrapper makes resource management composable and testable.

9.2.5 Model 3: `std::unique_ptr` (Exclusive Ownership of Dynamic Objects)

When `unique_ptr` is the right tool

Use `std::unique_ptr<T>` when:

- the object must have a stable address,
- the object is large or non-movable,

- you need runtime polymorphism (base pointer),
- you want to hide implementation details (PIMPL),
- you need optional presence without indirection hacks.

Example: ownership transfer by value parameter

```
#include <memory>
#include <utility>

struct Task {
  virtual ~Task() = default;
  virtual void run() = 0;
};

class Scheduler {
public:
  void submit(std::unique_ptr<Task> t) {
    queue_.push_back(std::move(t)); // Scheduler becomes unique owner
  }

private:
  std::vector<std::unique_ptr<Task>> queue_;
};
```

The API communicates ownership clearly:

- caller gives up ownership by moving the pointer,
- scheduler owns the task until execution/destruction.

9.2.6 Model 4: Unique Ownership with Borrowed Access (Owner + Borrowers)

Unique ownership does not forbid aliasing; it forbids *shared ownership*. A common correct model:

- one unique owner,
- many temporary borrowers (references, pointers, views),
- lifetime is guaranteed by the owner.

Example: service owns, collaborators borrow

```
#include <memory>
#include <string_view>

struct Logger { void info(std::string_view); };

class Service {
public:
  explicit Service(std::unique_ptr<Logger> log) : log_(std::move(log)) {}

  Logger& logger() noexcept { return *log_; } // borrowed access

private:
  std::unique_ptr<Logger> log_; // unique ownership
};
```

Rules for this model

- borrowers must not outlive the owning object,

- do not store the borrowed reference beyond the owner's lifetime,
- do not expose mutation broadly if the object is an identity type.

9.2.7 Model 5: Unique Ownership at the Composition Root

In large systems, most unique ownership is anchored at the composition root:

- **root** creates and owns long-lived components,
- subsystems accept references to collaborators (borrowing),
- shutdown order is controlled and testable.

Example: composition root wiring

```
#include <memory>

struct Repo { virtual ~Repo() = default; };
struct SqlRepo : Repo {};

struct App {
  explicit App(Repo& r) : repo_(r) {}
  Repo& repo_;
};

int main() {
  auto repo = std::make_unique<SqlRepo>(); // unique owner at the root
  App app{*repo};                          // app borrows
}
```

This yields:

- fewer `shared_ptr` graphs,
- deterministic shutdown,
- simple tests (swap the owned object at the root).

9.2.8 Unique Ownership and API Design: How to Make It Unambiguous

Use these API conventions.

Convention 1: Take by value for ownership transfer

```
class Pipeline {
public:
  void add_stage(std::unique_ptr<Task> stage); // ownership transfer
};
```

Convention 2: Return raw pointer/reference only for borrowing

If you return `T*` or `T&`, document and enforce that it is non-owning. Prefer returning `const T&` for observation.

Convention 3: Avoid returning `unique_ptr` unless you are transferring ownership

Returning a `unique_ptr` means the caller becomes the new unique owner.

Convention 4: Prefer values for data, `unique_ptr` for identity objects

Value objects should be passed by value/const ref and returned by value. Identity objects that require stable lifetime can be uniquely owned via `unique_ptr`.

9.2.9 Common Pitfalls in Unique Ownership Models

Pitfall 1: "unique_ptr everywhere"

Overusing heap allocation harms:

- locality,
- simplicity,
- performance.

Prefer values unless you truly need indirection.

Pitfall 2: Borrowers escaping the owner

Storing references, pointers, `string_view`, or `span` inside long-lived objects without proven lifetimes creates dangling use.

Pitfall 3: Custom deleters that hide real invariants

If the deleter must perform complex logic, consider a dedicated RAII type that encodes the invariant explicitly.

Pitfall 4: Moving breaks implicit assumptions

A moved-from object must remain valid but unspecified. If clients assume moved-from state, the type is unsafe to use as a value-like component. Design move semantics deliberately.

9.2.10 Checklist: Is a Unique Ownership Model Correct Here?

1. **Is there exactly one responsible owner at any time?**
2. **Is copying disabled when duplication is invalid?**

3. **Is transfer of ownership explicit (moves are visible)?**
4. **Are borrowers guaranteed not to outlive the owner?**
5. **Is the indirection necessary (stable address, polymorphism, PIMPL)?**
6. **Is shutdown order deterministic and tested?**
7. **Is concurrency handled by transfer or confinement rather than sharing?**

9.2.11 Summary

Unique ownership models are the backbone of robust Modern C++ systems:

- they encode responsibility clearly,
- they make lifetime reasoning provable,
- they reduce aliasing damage and eliminate ownership cycles,
- they align with RAII and move semantics.

Use:

- value ownership by default,
- move-only wrappers for non-copyable resources,
- `std::unique_ptr` for dynamic ownership, polymorphism, and stable addresses,
- composition roots to anchor ownership and keep subsystems borrowing.

In long-lived systems, unique ownership is not an optimization; it is a design discipline that prevents entire classes of lifetime and mutation failures.

9.3 Shared Ownership (And Its Real Cost)

9.3.1 Shared Ownership Is a Lifetime Policy, Not a Design Shortcut

Shared ownership means:

> **An object remains alive as long as at least one owner remains.**

In Modern C++, shared ownership is typically expressed through:

- `std::shared_ptr<T>` for shared owning references,
- `std::weak_ptr<T>` for non-owning references that can observe without extending lifetime.

The most important conceptual correction:

> **Shared ownership solves "who keeps it alive?" It does not solve "who may mutate it?" and it does not automatically solve "is this safe across threads?"**

Shared ownership is legitimate and sometimes necessary. But it has real costs: semantic, architectural, performance, and operational.

9.3.2 When Shared Ownership Is Actually Appropriate

Use shared ownership only when the lifetime requirement is inherently shared, such as:

- **Asynchronous completion:** work continues after the initiating scope ends, and several continuations must keep the state alive.
- **Fan-out graphs:** multiple independent subsystems must retain access with no single clear owner (rare in well-layered designs).

- **Caching with shared state:** a shared, read-mostly object is referenced by many clients, and rebuilding is expensive.
- **Observer/listener relationships:** when an object must outlive multiple subscriptions, and teardown ordering is complex.
- **Plugin or component registries:** multiple clients hold long-lived references to a registered object and release independently.

Even in these cases, the best designs typically constrain shared ownership to boundaries and keep mutability controlled.

9.3.3 The Real Cost of Shared Ownership

Cost 1: A Graph Lifetime Model (Harder Reasoning)

Unique ownership yields a tree of lifetimes; shared ownership yields a graph. Graphs introduce:

- uncertain destruction order,
- teardown that depends on dynamic reference patterns,
- shutdown behavior that changes with subtle control-flow differences,
- hard-to-reproduce crashes during shutdown.

Cost 2: Aliasing by Default (Action-at-a-Distance)

Copies of `shared_ptr` create more aliases. If the object is mutable, every alias becomes a potential mutation path. This produces:

- non-local state changes,
- hidden coupling across unrelated modules,
- invariant violations not enforced at a single authority point.

Cost 3: Shared Ownership Often Hides Ownership Boundaries

Many teams reach for `shared_ptr` because they cannot state:

- who owns the object,
- and how long it should live.

That is not a memory-management issue; it is a design issue. The cost is architectural: boundaries become unclear, responsibilities diffuse, and the system becomes resistant to refactoring.

Cost 4: Cycles and Leaks

Shared ownership graphs can form cycles. A cycle means:

- reference counts never reach zero,
- destructors never run,
- resources leak permanently.

`std::weak_ptr` exists to break cycles, but using it correctly requires discipline.

Cost 5: Performance Overhead

Shared ownership introduces overhead in:

- reference count updates (often atomic operations),
- control block allocation,
- cache-line contention on hot shared counts,
- potential extra indirection and less locality.

Even when the pointed object is small, the ownership machinery is not free.

Cost 6: Concurrency Confusion

A critical misconception:

> **`shared_ptr` is thread-safe for its reference count management, not for the shared object.**

If multiple threads access the same object through `shared_ptr`:

- the object itself is not automatically synchronized,
- mutable shared state still requires a thread-safety strategy,
- data races remain possible and common.

Cost 7: Testing and Determinism

Shared ownership increases:

- nondeterminism in teardown,
- hidden dependencies on destruction side-effects,
- flaky tests due to shutdown order differences.

9.3.4 The Two Meanings of "Shared": Do Not Mix Them

Shared ownership often gets conflated with shared access. They are different.

Shared lifetime

- multiple owners keep the object alive.

Shared mutability

- multiple aliases can mutate the object.

You can have shared lifetime with **immutable state**, which is far safer. The dangerous case is **shared lifetime + shared mutability**.

9.3.5 Safe Pattern: Shared Ownership of Immutable (or Effectively Immutable) State

When shared ownership is necessary, prefer making the shared object immutable after construction.

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

class Dictionary {
public:
  explicit Dictionary(std::string data) : data_(std::move(data)) {}
  std::string_view data() const noexcept { return data_; }
private:
  std::string data_; // no mutation after construction
};

std::shared_ptr<const Dictionary> load_dictionary();
```

Benefits:

- shared lifetime is safe,
- readers do not race on mutation,

- invariants hold after construction,
- the object becomes a stable shared asset.

9.3.6 Safe Pattern: Weak Observers to Break Cycles and Prevent Ownership Pollution

Use `weak_ptr` to avoid cycles and to avoid accidental lifetime extension.

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

struct Node {
  std::vector<std::weak_ptr<Node>> edges; // non-owning references
};

auto a = std::make_shared<Node>();
auto b = std::make_shared<Node>();
a->edges.push_back(b);
b->edges.push_back(a); // no cycle of ownership
```

Discipline:

- `shared_ptr` defines ownership edges,
- `weak_ptr` defines observation edges.

9.3.7 Danger Pattern: Shared Ownership as "Default Return Type"

Returning `shared_ptr<T>` widely tends to:

- spread ownership across the architecture,

- eliminate a clear composition root,
- make shutdown behavior dependent on arbitrary reference patterns.

Example: how ownership boundaries get erased

```
#include <memory>

struct Service {};

std::shared_ptr<Service> get_service(); // Who owns it? How long should it live?
```

A better approach in many systems:

- the composition root owns the service,
- consumers borrow (reference/pointer),
- or use a handle/ID if lifetime must be decoupled.

9.3.8 Shared Ownership with Mutable Identity Objects: The High-Risk Case

If a shared-owned object is mutable and represents identity, the system faces:

- action-at-a-distance,
- unclear mutation authority,
- data races unless synchronized,
- invariants that are hard to enforce centrally.

If you must do this, enforce:

- narrow mutation API,
- internal synchronization (or external strict protocol),
- explicit thread-safety contract,
- minimal distribution of owning references.

9.3.9 Architecture Strategy: Confine `shared_ptr` to "Async Edges"

A robust long-lived-system strategy:

- keep the core value-centric and uniquely owned,
- use `shared_ptr` only at asynchronous edges where lifetime cannot be rooted in a stack frame,
- expose non-owning views/handles outward where possible.

9.3.10 Alternatives to Shared Ownership (Often Better)

Before choosing shared ownership, evaluate alternatives:

Alternative 1: Unique ownership + explicit borrowing

- one owner (composition root),
- references for dependencies,
- deterministic teardown.

Alternative 2: Handles/IDs + manager

- values as identifiers,
- central store owns identity objects,
- snapshots for observation.

Alternative 3: Message passing / actor confinement

- transfer values across threads,
- avoid shared mutable state entirely.

Alternative 4: Shared immutable state

- share read-only assets safely,
- rebuild/replace when updates are needed.

9.3.11 Checklist: Using Shared Ownership Without Losing Control

1. **Why is the lifetime inherently shared?** If you cannot justify it, do not use it.
2. **Is the shared object immutable?** If not, what is the mutation authority?
3. **How do you prevent cycles?** Identify ownership edges and observer edges.
4. **Is the design thread-safe?** Reference counting safety is not object safety.
5. **Is `shared_ptr` confined to a boundary?** Avoid spreading it everywhere.
6. **Is shutdown deterministic enough?** If teardown order matters, shared graphs are risky.
7. **Can a handle or snapshot work instead?** Prefer value-centric APIs for most callers.

9.3.12 Summary

Shared ownership is a legitimate tool when lifetime must be shared, especially across asynchronous boundaries. But its real costs are significant:

- lifetime becomes a graph (harder reasoning, less deterministic teardown),
- aliasing spreads (action-at-a-distance),
- cycles can leak permanently,
- performance overhead exists (ref-counting, contention, indirection),
- it does not provide thread-safety for the shared object,
- it often hides missing architectural decisions about ownership boundaries.

The disciplined long-lived-system approach is:

- prefer unique ownership and explicit borrowing,
- allow shared ownership only when the lifetime requirement is truly shared,
- prefer sharing immutable state,
- use `weak_ptr` to break cycles,
- confine shared ownership to well-defined boundaries rather than letting it infect the entire design.

9.4 Ownership Graphs in Large Systems

9.4.1 Why "Ownership Graph" Is the Correct Mental Model

In small programs, ownership can feel local:

- a few objects on the stack,
- a few RAII wrappers,
- straightforward destruction order.

In large systems, ownership is not a set of isolated decisions. It is a **graph** (often a tree with carefully controlled cross-edges):

- nodes are objects/resources,
- edges represent responsibility for lifetime (who keeps whom alive),
- cross-edges represent borrowing/observation without ownership.

A system remains robust when:

> **The ownership graph is intentionally designed, mostly acyclic, and aligned with architectural boundaries.**

A system becomes fragile when:

> **Ownership becomes accidental, cyclic, or distributed across layers without clear authority.**

9.4.2 Two Graphs You Must Not Confuse

Large systems always contain at least two different graphs:

1) The ownership graph (lifetime graph)

- "Which objects are responsible for keeping other objects alive?"
- ideally mostly a tree (or forest) rooted at a composition root.

2) The reference graph (aliasing graph)

- "Which objects can access other objects?"
- can be dense; must be controlled by API design and borrowing rules.

A core discipline:

> **The reference graph may be large; the ownership graph must remain simple.**

9.4.3 The Composition Root: Where Ownership Must Be Anchored

A large system needs one or more **composition roots**:

- the place where long-lived objects are constructed,
- the place where ownership is explicit,
- the place where shutdown order is deterministic and testable.

In services and applications, the composition root is typically:

- `main()`,
- a bootstrapper,
- an application container.

In libraries, the composition root is often the library consumer.

9.4.4 The Ideal Shape: A Tree With Borrowed Cross-Edges

The stable shape for long-lived systems is:

- ownership is a tree (unique ownership) with clear roots,

- cross-component access uses borrowing (references/pointers/views),
- shared ownership is localized to specific boundaries (often async edges).

This yields:

- predictable destruction order,
- easy lifetime proofs,
- minimal risk of cycles and leaks,
- clear reasoning during failure and shutdown.

9.4.5 A Practical Ownership Taxonomy for Large Systems

In a large architecture, most nodes fall into one of these categories.

Category A: Process-lifetime singletons (but not global variables)

These are long-lived infrastructure components owned by the root:

- logging sinks,
- configuration store,
- thread pools,
- metrics registry,
- DB connection pool.

They should be:

- owned by the composition root,

- passed as dependencies (borrowing),
- not accessed via uncontrolled globals.

Category B: Request-scoped ownership trees

Many systems have a request/transaction scope:

- request context,
- per-request caches,
- per-request authorization state,
- per-request trace spans.

These objects are:

- created at request entry,
- destroyed at request completion,
- provide a natural and provable lifetime boundary.

Category C: Entity stores and registries (identity managers)

Identity objects are often owned by a store:

- sessions,
- connections,
- actors,
- live device instances.

Clients should prefer:

- handles/IDs and snapshots,
- controlled mutation through the store,
- minimal exposure of raw pointers to the entities.

Category D: Async boundary state

When work outlives a stack frame, lifetime may need sharing:

- callback state,
- completion tokens,
- promise/future shared state.

This is where shared ownership is often legitimate, but must be constrained.

9.4.6 Visualizing Ownership Graphs as Edges and Contracts

Use this disciplined vocabulary:

- **owning edge**: destruction of parent destroys child (value member, `unique_ptr`)
- **borrowing edge**: access without extending lifetime (reference, raw pointer, view)
- **observing edge**: read-only borrowing, often `const&` or a snapshot
- **weak edge**: observation without ownership in a shared lifetime model (`weak_ptr`)
- **handle edge**: indirect access through an ID validated by a manager

Robust systems document these edges through type choices.

9.4.7 Example: A Healthy Ownership Graph (Tree + Borrowing)

Composition root owns infrastructure; services borrow

```
#include <memory>
#include <string_view>

struct Logger {
  void info(std::string_view);
};

struct DbPool {};

class UserRepo {
public:
  UserRepo(DbPool& db, Logger& log) : db_(db), log_(log) {}
private:
  DbPool& db_;   // borrowed
  Logger& log_;  // borrowed
};

class App {
public:
  App(Logger& log, DbPool& db)
    : repo_(db, log), log_(log) {}

private:
  UserRepo repo_; // owned by value (part of App's tree)
  Logger&  log_;  // borrowed
};
```

```
int main() {
  auto log = std::make_unique<Logger>(); // root owns
  auto db  = std::make_unique<DbPool>(); // root owns

  App app{*log, *db}; // app borrows infrastructure, owns internal subtree
}
```

Properties:

- ownership is clear and mostly a tree,
- no global lifetime ambiguity,
- destruction order is predictable.

9.4.8 Failure Shape 1: Ownership Distributed by Convenience (`shared_ptr` Everywhere)

If all components share-own each other, the ownership graph becomes:

- dense,
- cyclic,
- hard to reason about,
- prone to leaks and shutdown surprises.

Illustration of accidental cycle

```
#include <memory>
```

```
struct A { std::shared_ptr<void> b; };
struct B { std::shared_ptr<void> a; };

auto a = std::make_shared<A>();
auto b = std::make_shared<B>();
a->b = b;
b->a = a; // cycle -> lifetime never ends
```

Fix pattern:

- choose one ownership direction,
- replace the opposite edge with `weak_ptr` or a handle/ID.

9.4.9 Failure Shape 2: Borrowers Escape Their Owners (Dangling Lifetimes)

Even with a tree ownership model, lifetime fails when borrowing edges escape:

- storing a reference in a longer-lived object,
- storing `string_view`/`span` without lifetime guarantee,
- capturing `this` in async callbacks that outlive the object.

Async escape example

```
#include <functional>

struct Executor {
  void post(std::function<void()> f);
};
```

```
class Controller {
public:
  explicit Controller(Executor& ex) : ex_(ex) {}

  void start_async() {
    ex_.post([this] { do_work(); }); // may outlive *this
  }

private:
  void do_work() {}
  Executor& ex_;
};
```

Corrective strategies:

- confine lifetime (cancel tasks before destruction),
- use a shared state object at the async boundary,
- or use handle-based routing through a manager.

9.4.10 Large-System Strategy: "Simple Ownership, Rich Access"

A stable strategy for large codebases:

- Make ownership simple and mostly acyclic (unique ownership roots).
- Allow rich access patterns through borrowing and handles.
- Keep shared ownership localized to async boundaries and shared immutable assets.
- Keep identity objects behind stores/registries; expose IDs and snapshots.

This preserves:

- performance (better locality, fewer atomic ref-count operations),
- determinism (predictable teardown),
- testability (inject dependencies at the root),
- maintainability (clear boundaries and fewer hidden couplings).

9.4.11 How to Engineer Ownership Graphs Deliberately

Step 1: Draw the lifetime roots

Identify:

- process-lifetime roots,
- request-lifetime roots,
- session/entity-lifetime roots,
- async-lifetime roots.

Step 2: Classify each edge

For every relationship, decide:

- ownership edge,
- borrow edge,
- weak edge,
- handle edge.

Step 3: Enforce with types

- ownership: value members, `unique_ptr`, move-only wrappers,
- borrow: references and non-owning pointers,
- weak observation: `weak_ptr`,
- indirect access: IDs/handles + manager.

Step 4: Make teardown a first-class scenario

Test:

- shutdown order,
- cancellation of asynchronous work,
- destruction of registries and caches,
- error paths that exit early.

9.4.12 Checklist: Ownership Graph Quality in a Large System

1. **Can you name the composition roots?**
2. **Is the ownership graph mostly acyclic?**
3. **Is shared ownership localized to justified boundaries?**
4. **Are identity objects kept behind stores/registries with controlled access?**
5. **Do borrowed edges have provable lifetimes?**
6. **Are async callbacks prevented from outliving owners?**

7. **Are container invalidation rules respected (no escaping references)?**
8. **Is shutdown deterministic and tested?**

9.4.13 Summary

Ownership graphs determine whether a large C++ system is predictable or fragile. The long-lived-system strategy is:

- anchor ownership in explicit composition roots,
- keep the ownership graph simple (prefer trees),
- allow rich access via borrowing and handles without spreading ownership,
- confine shared ownership to async boundaries and shared immutable assets,
- manage identity behind stores and expose values (IDs/snapshots) to the rest of the system.

When ownership is engineered as a first-class concept, lifetime bugs become rare, shutdown becomes deterministic, and architecture becomes replaceable rather than entangled.

RAII as a System-Level Design Tool

10.1 RAII Beyond Memory

10.1.1 RAII as a General Resource Protocol

RAII (Resource Acquisition Is Initialization) is often introduced as "automatic memory management." In Modern C++ engineering, that description is incomplete.
A more accurate system-level definition:

> **RAII is the discipline of binding every resource to an object whose constructor establishes the resource invariant and whose destructor releases the resource deterministically.**

Two consequences matter at architecture scale:

- **Determinism:** release happens at a known point (scope end / owner destruction), not "eventually."
- **Exception safety:** cleanup is implicit even when control flow exits early (errors, exceptions, returns).

RAII is therefore not "about memory." It is about **all resources**:

- files and sockets,

- mutexes and locks,
- transactions and database sessions,
- mapped memory and shared memory,
- GPU/driver resources,
- temporary configuration changes,
- logging/telemetry scopes,
- thread joins and task lifetimes,
- security privileges and capabilities.

10.1.2 Why RAII Is a System-Level Design Tool

At system scale, resource leaks are not limited to memory leaks. The most damaging leaks often involve:

- leaked file descriptors (exhaustion under load),
- leaked locks (deadlocks),
- uncommitted transactions (stalls, data corruption, contention),
- unreleased OS handles (resource exhaustion),
- missing thread joins (terminate on destruction or runaway threads),
- unbalanced privilege changes (security and policy violations),
- unbalanced temporary state changes (global configuration drift).

RAII addresses these by enforcing a universal invariant:

> **Every acquire has a matching release, and the pairing is enforced by object lifetime, not by programmer discipline.**

10.1.3 The Anatomy of a Correct RAII Type

A correct RAII type follows these principles:

1) Establish an invariant in the constructor

After construction, the object is either:

- fully valid and owns the resource, or
- construction fails and no partially-owned resource escapes.

2) Release in the destructor (no-throw)

Destructors must not throw. Release must be best-effort and safe.

3) Define ownership semantics explicitly

Most RAII wrappers are:

- **move-only** (exclusive ownership),
- or **non-copyable** if copying is not meaningful.

4) Provide explicit escape hatches

Sometimes you must transfer ownership to a lower-level API:

- provide `release()` to relinquish ownership,

- provide `get()` to observe the handle,
- provide `reset()` to replace the resource safely.

10.1.4 RAII for Files and OS Handles

Design intent

Files and handles are resources with scarcity and lifecycle. They should be represented as move-only RAII objects.

Generic unique-handle wrapper (system-level primitive)

```
#include <utility>

template <class Handle, class Deleter>
class UniqueHandle {
public:
  UniqueHandle() = default;
  explicit UniqueHandle(Handle h, Deleter d = {}) : h_(h), del_(d) {}

  UniqueHandle(const UniqueHandle&) = delete;
  UniqueHandle& operator=(const UniqueHandle&) = delete;

  UniqueHandle(UniqueHandle&& other) noexcept
    : h_(std::exchange(other.h_, Handle{})), del_(std::move(other.del_)) {}

  UniqueHandle& operator=(UniqueHandle&& other) noexcept {
    if (this != &other) {
      reset();
      h_ = std::exchange(other.h_, Handle{});
```

```
      del_ = std::move(other.del_);
    }
    return *this;
  }

  ~UniqueHandle() { reset(); }

  Handle get() const noexcept { return h_; }

  Handle release() noexcept { return std::exchange(h_, Handle{}); }

  void reset(Handle nh = Handle{}) noexcept {
    if (h_ != Handle{}) del_(h_);
    h_ = nh;
  }

private:
  Handle  h_{};
  Deleter del_{};
};
```

This pattern scales across OS APIs:

- the system cannot leak handles silently,
- destruction order is deterministic,
- ownership transfer is explicit via move/release.

10.1.5 RAII for Locking and Concurrency Control

Why RAII is mandatory for locks

Lock/unlock must be balanced under all control paths:

- early returns,
- exceptions,
- multiple exit points.

RAII makes lock release automatic.

Example: scoped lock guard (pattern)

```
#include <mutex>

class LockGuard {
public:
  explicit LockGuard(std::mutex& m) : m_(m) { m_.lock(); }
  LockGuard(const LockGuard&) = delete;
  LockGuard& operator=(const LockGuard&) = delete;
  ~LockGuard() { m_.unlock(); }
private:
  std::mutex& m_;
};
```

In production code, standard library guards are preferred, but the design principle matters:

- acquisition in constructor,
- release in destructor,
- non-copyable guard type.

10.1.6 RAII for Transactions (Commit, Rollback, and Failure Semantics)

Transaction scope is a resource

A transaction is not merely a function call; it is a stateful protocol:

- begin,
- perform operations,
- commit or rollback.

If an exception happens mid-way, rollback must occur deterministically.

Example: transactional scope guard

```
#include <utility>

struct DbTx {
  void begin();
  void commit();
  void rollback() noexcept;
};

class TxScope {
public:
  explicit TxScope(DbTx& tx) : tx_(tx) { tx_.begin(); }

  TxScope(const TxScope&) = delete;
  TxScope& operator=(const TxScope&) = delete;

  TxScope(TxScope&&) = delete;
```

```
  TxScope& operator=(TxScope&&) = delete;

  void commit() {
    tx_.commit();
    committed_ = true;
  }

  ~TxScope() {
    if (!committed_) tx_.rollback(); // rollback on scope exit unless committed
  }

private:
  DbTx& tx_;
  bool committed_{false};
};
```

This makes the failure contract explicit:

- **commit is explicit**,
- **rollback is automatic on failure paths**.

10.1.7 RAII for Temporary State Changes (Restore-On-Exit)

Many system bugs come from temporary global or thread-local state changes that are not restored:

- changing log levels,
- changing locale/timezone formatting,
- changing “current directory”,

- enabling/disabling features,
- setting thread name or priority.

RAII models these as “restore-on-exit” resources.

Example: restore-on-exit guard

```
#include <utility>

class ScopedFlag {
public:
  explicit ScopedFlag(bool& flag, bool new_value)
    : flag_(flag), old_(std::exchange(flag_, new_value)) {}

  ScopedFlag(const ScopedFlag&) = delete;
  ScopedFlag& operator=(const ScopedFlag&) = delete;

  ~ScopedFlag() { flag_ = old_; }

private:
  bool& flag_;
  bool  old_;
};
```

This pattern is system-level:

- it prevents configuration drift,
- it makes reversible changes safe under failure paths.

10.1.8 RAII for Threads and Asynchronous Work

The resource is the running thread

A running thread is a resource that must be:

- joined,
- or detached under a deliberate policy,
- and usually canceled/stopped under controlled shutdown.

A robust design uses RAII to enforce completion policy.

Example: join-on-destruction wrapper

```
#include <thread>
#include <utility>

class JoiningThread {
public:
  JoiningThread() = default;

  explicit JoiningThread(std::thread t) : t_(std::move(t)) {}

  JoiningThread(const JoiningThread&) = delete;
  JoiningThread& operator=(const JoiningThread&) = delete;

  JoiningThread(JoiningThread&&) noexcept = default;
  JoiningThread& operator=(JoiningThread&&) noexcept = default;

  ~JoiningThread() {
```

```
    if (t_.joinable()) t_.join();
  }

  std::thread& get() noexcept { return t_; }

private:
  std::thread t_;
};
```

This is not about memory; it is about enforcing a shutdown protocol.

10.1.9 RAII and Exception Safety: The Architectural Guarantee

RAII is the foundation of exception-safe systems because it makes cleanup unconditional. In system design terms:

- error handling becomes local,
- cleanup becomes implicit,
- invariants are preserved across failure paths.

A large system becomes manageable when:

> **Every subsystem can fail and unwind safely without requiring manual cleanup conventions across the entire codebase.**

10.1.10 RAII at the Boundary: Adapters Around Unsafe APIs

Many C libraries and OS APIs are non-RAII:

- manual acquire/release functions,

- global state toggles,
- error codes and early returns.

A system-level RAII approach:

- wraps such APIs in small RAII types,
- exports safe value-centric or move-only interfaces upward,
- confines unsafe patterns to narrow boundary modules.

10.1.11 Common RAII Design Mistakes (System-Level View)

Mistake 1: Destructors that throw

Destructors must not throw. Release logic must be safe and best-effort.

Mistake 2: Copyable ownership wrappers

If copying duplicates ownership incorrectly, delete copy operations. Most RAII wrappers should be move-only.

Mistake 3: Leaking raw handles

If raw handles must be exposed:

- provide read-only access via `get()`,
- require explicit `release()` to transfer ownership.

Mistake 4: Mixing ownership and borrowing without clarity

Returning pointers/references to internal resources without a lifetime contract reintroduces lifetime bugs.

Mistake 5: RAII used only locally, while architecture remains unmanaged

If ownership roots and shutdown protocols are unclear, local RAII wrappers will not save the system from global teardown failures.

10.1.12 Checklist: RAII Beyond Memory

1. **What is the resource?** (fd, lock, transaction, privilege, thread, mapping, setting)
2. **Where is acquisition?** (constructor establishes invariant)
3. **Where is release?** (destructor releases deterministically, no-throw)
4. **Is copying meaningful?** If not, delete copy operations.
5. **Is move transfer correct and explicit?** (ownership transfer should be visible)
6. **Does the wrapper prevent protocol violations?** (unbalanced begin/commit/rollback, lock/unlock)
7. **Is the wrapper confined to boundaries when wrapping unsafe APIs?**
8. **Does the design remain correct under early return and exceptions?**

10.1.13 Summary

RAII is a system-level design tool because it generalizes to *all* resources:

- it makes release deterministic and reliable under failure paths,
- it encodes protocols (lock/unlock, begin/commit/rollback, set/restore) into types,
- it strengthens architecture by making ownership and shutdown behavior explicit.

When applied beyond memory, RAII becomes a universal strategy for building long-lived systems that remain stable under load, errors, and evolution.

10.2 Resources as Design Units

10.2.1 From “Objects” to “Resources” as the Primary Design Primitive

A long-lived system is not fundamentally a set of classes. It is a set of **resources** that must be:

- acquired,
- used under rules,
- shared or transferred under constraints,
- and released correctly under all control paths.

Therefore, a system-level design discipline is:

> **Treat resources as first-class design units: explicitly model their lifetime, ownership, protocol, and failure semantics as part of the type system and architecture.**

This is what elevates RAII from “memory management” to architectural engineering.

10.2.2 What Counts as a Resource in Modern C++ System Design

A resource is anything that:

- is acquired through an operation that can fail,
- must be released or unwound to avoid system degradation,
- has usage constraints (a protocol),
- is scarce or externally managed,

- and whose lifetime must be correct for safety, performance, or correctness.

Examples across system layers:

- **OS resources:** file descriptors, sockets, handles, processes, threads.
- **Synchronization resources:** mutex locks, read locks, semaphores, permits.
- **Data resources:** database transactions, prepared statements, cursors.
- **Memory resources:** heap allocations, arenas, memory mappings, shared memory segments.
- **Hardware resources:** GPU buffers, device contexts, DMA mappings, exclusive peripherals.
- **Temporal resources:** timeouts, leases, watchdog feeding windows.
- **Policy resources:** elevated privileges, impersonation contexts, security tokens.
- **Behavioral resources:** logging scopes, tracing spans, temporary feature flags.

10.2.3 Why "Resources" Are Better Units Than "Classes"

Classes describe structure. Resources describe:

- **invariants** (what must always be true),
- **protocol** (what operations are valid in which states),
- **ownership** (who is responsible),
- **lifetime** (when validity begins/ends),
- **failure modes** (what happens on error),

- **release semantics** (how to unwind).

A class can be a poor design unit if:

- it mixes multiple resources and multiple protocols,
- it hides lifetime assumptions,
- it permits partially-initialized valid-looking states,
- it spreads resource release responsibility across many functions.

A resource-centric design unit has one dominant responsibility:

> **enforce correct usage and deterministic release of one resource protocol.**

10.2.4 The Resource Contract: Five Questions Every Resource Type Must Answer

To treat resources as design units, every resource type must answer:

1. **Acquire:** How is the resource obtained, and how can that fail?
2. **Own:** Who owns it, and is ownership exclusive or shared?
3. **Use:** What is the valid protocol (allowed operations and sequencing)?
4. **Release:** How is it released, and is release guaranteed no-throw?
5. **Transfer:** Can ownership be moved, copied, lent, or observed?

The quality of the answers determines whether the system is robust.

10.2.5 RAII as the Universal Mechanism for Resource Units

RAII provides the uniform mapping:

- acquisition in construction,
- release in destruction,
- valid state invariant during lifetime,
- move-only ownership transfer when appropriate.

This maps resource correctness from "programmer discipline" to **type-level enforcement**.

10.2.6 Designing a Resource Unit: The Minimal RAII Skeleton

A generic skeleton for a resource unit typically includes:

- a handle/state,
- deleted copy operations when duplication is invalid,
- move operations for ownership transfer,
- destructor releasing deterministically,
- `get()/release()/reset()` as explicit escape hatches.

```
#include <utility>

template <class Handle, class Deleter>
class Resource {
public:
  Resource() = default;
```

```
  explicit Resource(Handle h, Deleter d = {}) : h_(h), del_(d) {}

  Resource(const Resource&) = delete;
  Resource& operator=(const Resource&) = delete;

  Resource(Resource&& other) noexcept
    : h_(std::exchange(other.h_, Handle{})), del_(std::move(other.del_)) {}

  Resource& operator=(Resource&& other) noexcept {
    if (this != &other) {
      reset();
      h_ = std::exchange(other.h_, Handle{});
      del_ = std::move(other.del_);
    }
    return *this;
  }

  ~Resource() { reset(); }

  Handle get() const noexcept { return h_; }

  Handle release() noexcept { return std::exchange(h_, Handle{}); }

  void reset(Handle nh = Handle{}) noexcept {
    if (h_ != Handle{}) del_(h_);
    h_ = nh;
  }

private:
```

```
  Handle  h_{};
  Deleter del_{};
};
```

This is not just a coding pattern; it is a design commitment:

- ownership is explicit,
- release is guaranteed,
- transfer is intentional.

10.2.7 Resources With Protocol: "Acquire/Use/Release" Is Not Enough

Many resources require more than "close on destruction." They have an internal protocol with states.
Examples:

- transactions: begin → commit/rollback,
- locks: lock → unlock,
- temporary settings: set → restore,
- leases: acquire → renew/expire/release.

A resource unit should encode protocol correctness such that invalid sequences are hard to express.

Example: transaction resource with explicit commit and automatic rollback

```
struct TxApi {
  void begin();
```

```
  void commit();
  void rollback() noexcept;
};

class Transaction {
public:
  explicit Transaction(TxApi& api) : api_(api) { api_.begin(); }

  Transaction(const Transaction&) = delete;
  Transaction& operator=(const Transaction&) = delete;

  void commit() {
    api_.commit();
    committed_ = true;
  }

  ~Transaction() {
    if (!committed_) api_.rollback(); // protocol enforced by lifetime
  }

private:
  TxApi& api_;
  bool committed_{false};
};
```

The resource unit encapsulates:

- the protocol,
- the failure behavior,
- the invariant that rollback happens unless committed.

10.2.8 Resources and Boundaries: Where to Place the Resource Units

Resource units should live at architectural boundaries:

- OS and platform integration layer,
- database driver and persistence layer,
- networking layer,
- concurrency primitives layer,
- security and identity layer.

The core domain should ideally depend on:

- values,
- pure transformations,
- stable contracts,

not on raw OS handles or manual protocols.
This yields a stable dependency direction:

- core is policy and meaning,
- boundary is mechanics and resource control.

10.2.9 Resource Units and Ownership Graphs

Treating resources as design units improves ownership graphs because:

- each resource has a clear owner,
- transfers are explicit and move-based,

- borrowed access is distinguished from owning access,
- shared ownership becomes rare and localized.

This tends to produce:

- mostly-tree ownership graphs,
- deterministic teardown order,
- fewer shutdown crashes and leaks.

10.2.10 Resource Units and Error Handling Strategy

A resource-centric approach pairs naturally with explicit error handling:

- constructors enforce “valid or fail,”
- operations return explicit status (or throw within a clearly defined policy),
- destructors do best-effort cleanup without throwing.

A reliable system-level rule:

> **If acquisition can fail, represent that explicitly at construction boundaries; never allow an invalid resource unit to exist silently.**

10.2.11 The Hidden Resource: Invariants and “Permission to Use”

Some resources are not physical handles but *capabilities*:

- a lock is a permission to access a critical section,
- a transaction is a permission to mutate a database atomically,

- an elevated security token is a permission to perform privileged operations,
- a lease is permission to hold a role for a time window.

Modeling these as resource units has major benefits:

- permission is granted and revoked deterministically,
- accidental misuse becomes harder,
- security and correctness become type-guided.

10.2.12 Common Mistakes When Resources Are Not Treated as Design Units

Mistake 1: Multiple resources in one type without clear invariants

A "manager" class that owns:

- sockets, locks, transactions, buffers,

often becomes a bug magnet. Split into resource units with single protocols.

Mistake 2: Manual protocol enforcement scattered across functions

If `lock()` and `unlock()` are called in different places, protocol correctness depends on human discipline. Use a guard type.

Mistake 3: Silent invalid states

Allowing default-constructed "half-valid" resource objects encourages undefined behavior. Prefer:

- explicit empty state with clear checks,
- or factories that return optional/expected results,
- or types that cannot exist unless valid.

Mistake 4: Shared ownership as a protocol substitute

Using `shared_ptr` to avoid deciding ownership boundaries spreads resource responsibility across the architecture.

10.2.13 Checklist: Resource-Unit Design Review

1. **What is the resource?** (scarce handle, protocol permission, external state)
2. **Is the resource unit single-purpose?** One dominant protocol and invariant.
3. **Is acquisition explicit and failure visible?**
4. **Is release deterministic and no-throw?**
5. **Are copy/move semantics correct for the resource?**
6. **Do APIs distinguish owning vs borrowing?**
7. **Is the resource unit placed at a boundary layer?**
8. **Does the design simplify the ownership graph rather than densify it?**

10.2.14 Summary

Treating resources as design units is a system-level discipline:

- it shifts correctness from "remember to clean up" to type-enforced invariants,
- it encodes protocols into lifetimes (RAII),
- it makes ownership graphs simpler and teardown deterministic,
- it keeps dangerous mechanics at boundaries and preserves a value-centric core.

In long-lived Modern C++ systems, this approach is one of the most effective ways to eliminate whole classes of operational failures that have nothing to do with memory, yet can take production systems down.

10.3 Deterministic Cleanup in Complex Systems

10.3.1 Why Deterministic Cleanup Is a System Requirement

In complex systems, failures are rarely caused by "one missing delete". They are caused by cleanup that is:

- incomplete,
- out of order,
- non-deterministic,
- or skipped on uncommon control paths (errors, timeouts, partial initialization).

Deterministic cleanup is the property that:

> **Resources are released in a predictable, testable order at well-defined points, across all control paths.**

This matters because real systems accumulate resources:

- file descriptors,
- sockets,
- locks,
- transactions,

- background tasks and threads,
- buffers and mappings,
- registrations (callbacks, observers),
- temporary configuration state.

When cleanup becomes non-deterministic, production systems fail in ways that are:

- load-dependent,
- time-dependent,
- shutdown-dependent,
- and extremely hard to reproduce.

RAII is the foundation, but deterministic cleanup in complex systems is an architectural topic, not a local coding trick.

10.3.2 The Three Cleanup Dimensions: Local, Subsystem, System

Deterministic cleanup must be engineered at three levels.

Level 1: Local scope cleanup (basic RAII)

- locks released at scope end,
- temporary state restored,
- file closed on exit,
- partial work rolled back.

Level 2: Subsystem teardown (component lifetime)

- stop accepting new work,
- drain queues,
- cancel outstanding operations,
- flush buffers,
- detach observers,
- release resource pools.

Level 3: System shutdown (ownership graph teardown)

- deterministic destruction order of major components,
- termination of background threads before dependencies disappear,
- avoidance of global static destruction hazards,
- predictable end-to-end shutdown even under partial failures.

If only Level 1 exists, the system still fails at Level 2 and Level 3.

10.3.3 The Deterministic Cleanup Contract

A robust complex system typically enforces this contract:

1. **Acquisition is explicit and minimal.**
2. **Ownership is rooted.** There is a known owner for every long-lived resource.
3. **Teardown is staged.** Stop $\rightarrow$ drain/cancel $\rightarrow$ release.

4. **Cleanup is idempotent.** Calling cleanup twice is safe.

5. **Destructors do not throw.** Cleanup is best-effort and safe.

10.3.4 Staged Teardown: Stop, Drain, Release

Complex systems fail when teardown is "just destruct everything." A disciplined teardown is staged:

Stage A: Stop (no new work)

- close acceptors,
- stop scheduling new tasks,
- flip an atomic "stopping" flag,
- reject new requests with a stable error path.

Stage B: Drain/Cancel (resolve in-flight work)

- drain queues,
- cancel timers and outstanding I/O,
- wait for in-flight operations to complete,
- join threads or stop event loops.

Stage C: Release (free resources deterministically)

- release pools,
- close handles,

- detach observers,
- destroy objects in a known dependency-safe order.

RAII integrates naturally: each stage can be encoded as an object lifetime boundary.

10.3.5 Encoding Teardown Stages in Types

Example: a component with explicit stop/drain and deterministic destruction

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

class Worker {
public:
  Worker() : running_(true), t_([this]{ run(); }) {}

  Worker(const Worker&) = delete;
  Worker& operator=(const Worker&) = delete;

  void stop() noexcept { running_.store(false); } // Stage A: stop

  void drain() {
    if (t_.joinable()) t_.join();                 // Stage B: drain
  }

  ~Worker() {
    stop();
    drain();                                      // Stage C: release via teardown
  }
```

```
private:
  void run() {
    while (running_.load()) {
      // do work
    }
  }

  std::atomic<bool> running_{false};
  std::thread t_;
};
```

Key point:

- the destructor enforces shutdown, but the stages are explicit and testable.

10.3.6 Deterministic Cleanup Under Partial Initialization

A common production failure:

- complex startup acquires multiple resources,
- resource acquisition fails halfway,
- cleanup code is incomplete or wrong,
- the process leaks handles or leaves global state in a bad configuration.

RAII eliminates partial-initialization cleanup bugs by construction:

- each acquired resource is immediately bound to an owner object,
- if later acquisition fails, stack unwinding cleans up earlier resources automatically.

Example: startup that cannot leak on failure paths

```
#include <stdexcept>
#include <utility>

struct HandleDeleter { void operator()(int) noexcept {} };

class UniqueFd {
public:
  explicit UniqueFd(int fd = -1) : fd_(fd) {}
  UniqueFd(const UniqueFd&) = delete;
  UniqueFd& operator=(const UniqueFd&) = delete;

  UniqueFd(UniqueFd&& o) noexcept : fd_(std::exchange(o.fd_, -1)) {}
  UniqueFd& operator=(UniqueFd&& o) noexcept {
    if (this != &o) { reset(); fd_ = std::exchange(o.fd_, -1); }
    return *this;
  }

  ~UniqueFd() { reset(); }

  int get() const noexcept { return fd_; }
  void reset(int n = -1) noexcept { if (fd_ != -1) HandleDeleter{}(fd_); fd_ = n; }

private:
  int fd_{-1};
};

struct App {
  App() {
```

```
    UniqueFd log_fd(open_log());
    UniqueFd sock(open_socket());
    init_tls(); // may throw

    // Once everything succeeds, move into members
    log_ = std::move(log_fd);
    sock_ = std::move(sock);
  }

private:
  static int open_log() { return 1; }
  static int open_socket() { return 2; }
  static void init_tls() { /* may throw */ }

  UniqueFd log_{};
  UniqueFd sock_{};
};
```

If `init_tls()` fails:

- `log_fd` and `sock` destruct deterministically,
- no manual cleanup branch is required.

10.3.7 Deterministic Cleanup and Callbacks: The Hidden Leak

A complex system often "leaks" through registrations:

- event listeners,
- callbacks in a dispatcher,

- subscriptions in a pub/sub bus,
- timers and scheduled jobs.

These are resources: they must be unregistered deterministically.

Pattern: scoped registration token

```
#include <utility>

struct Dispatcher {
  int subscribe(void(*)(int));
  void unsubscribe(int id) noexcept;
};

class Subscription {
public:
  Subscription(Dispatcher& d, void(*cb)(int))
    : d_(d), id_(d_.subscribe(cb)) {}

  Subscription(const Subscription&) = delete;
  Subscription& operator=(const Subscription&) = delete;

  Subscription(Subscription&& o) noexcept
    : d_(o.d_), id_(std::exchange(o.id_, -1)) {}

  Subscription& operator=(Subscription&&) = delete;

  ~Subscription() {
    if (id_ != -1) d_.unsubscribe(id_);
  }
```

```
private:
  Dispatcher& d_;
  int id_{-1};
};
```

This makes subscription lifetime:

- explicit,
- bounded,
- and safe under exceptions and early returns.

10.3.8 Deterministic Cleanup and Deadlock Avoidance

Locks are resources. Deterministic cleanup includes deterministic *release order* to prevent deadlocks:

- acquire locks in a consistent global order,
- release is naturally reverse order in RAII-based scopes,
- avoid holding locks across calls into unknown code.

RAII helps, but system design must enforce:

- lock hierarchy rules,
- limited lock scope,
- clear ownership of synchronization policy.

10.3.9 Deterministic Cleanup and Shutdown Order (The Ownership Graph Problem)

Shutdown failures often come from:

- background threads still running while dependencies are destroyed,
- callbacks firing after the target object is gone,
- log/metrics sinks destroyed while other components still report.

This is not solved by "using RAII" alone. It is solved by:

- anchoring ownership in a composition root,
- maintaining a mostly-tree ownership graph,
- explicit teardown stages: stop/drain before destroy.

Rule: stop/drain before destroying dependencies

If component A uses component B from a background thread:

- A must be stopped and drained before B is destroyed.

In practice:

- join workers before destroying the services they call,
- cancel timers before destroying callback targets,
- shutdown event loops before destroying objects referenced by posted tasks.

10.3.10 Idempotent Cleanup: The Production-Grade Requirement

In complex systems, cleanup is often triggered from multiple paths:

- normal shutdown,
- error path shutdown,
- partial initialization failure,
- explicit restart logic.

Therefore cleanup operations should be **idempotent**:

- calling `stop()` twice is safe,
- calling `drain()` twice is safe,
- destroying after stop/drain is safe.

A robust pattern:

- internal state flags,
- `exchange` to consume tokens once,
- nulling handles after release.

10.3.11 Checklist: Deterministic Cleanup in Complex Systems

1. **Have you identified all resources, including registrations and background work?**
2. **Is every resource bound to an owner object immediately upon acquisition?**
3. **Is teardown staged (stop → drain/cancel → release)?**

4. **Is shutdown order derived from the ownership graph (dependencies destroyed last)?**
5. **Are destructors no-throw and best-effort?**
6. **Are cleanup operations idempotent?**
7. **Can callbacks outlive their targets? If yes, how is that prevented?**
8. **Are there tests that exercise early-failure startup and shutdown under load?**

10.3.12 Summary

Deterministic cleanup in complex systems is an architectural property. RAII provides the mechanism, but the system must be designed to:

- bind each resource to an owner immediately,
- encode protocols and teardown stages in types,
- maintain a simple ownership graph,
- stop and drain concurrent activity before destroying dependencies,
- and guarantee safe cleanup across all control paths, including partial initialization failures.

When cleanup is deterministic and staged, production failures move from "mysterious" to "engineered," and the system becomes refactorable without fear of hidden lifetime landmines.

10.4 RAII vs Garbage Collection Design Tradeoffs

10.4.1 Two Different Resource Models, Two Different System Behaviors

RAII and Garbage Collection (GC) are often compared as if they are both "memory management." At a system-design level, they are different runtime models that shape:

- latency behavior,
- failure modes,
- predictability and shutdown,
- concurrency and scalability,
- integration with OS resources,
- and the overall architecture of long-lived systems.

A precise separation:

- **RAII** is *deterministic resource management* bound to scope/lifetime.
- **GC** is *automatic memory reclamation* driven by reachability, typically non-deterministic in time.

This yields the first design insight:

> **RAII is a general resource protocol; GC is primarily a memory reclamation strategy.**

10.4.2 What RAII Guarantees That GC Does Not

1) Deterministic release

RAII releases resources at a known point:

- scope exit,
- owner destruction,
- or explicit reset/move boundaries.

This matters for non-memory resources:

- file descriptors,
- locks,
- transactions,
- sockets,
- mapped memory,
- temporary configuration changes.

2) Cleanup correctness on all control paths

With RAII, early returns and exceptions cannot skip cleanup, because cleanup is bound to destruction.

3) Predictable shutdown and teardown

RAII supports staged teardown:

- stop,
- drain,
- release,

with predictable ordering derived from ownership graphs.

4) Explicit ownership semantics

RAII encourages:

- unique ownership (tree lifetimes),
- explicit borrowing,
- controlled aliasing.

10.4.3 What GC Provides That RAII Does Not

1) Automatic reclamation of unreachable memory

GC reduces certain classes of memory leaks caused by forgotten deallocation, by reclaiming unreachable objects.

2) Simpler programming model for cyclic object graphs

GC naturally reclaims cycles without special handling (where pure reference counting would leak).

3) Some forms of developer productivity at the application layer

GC can reduce cognitive load in certain application domains where:

- object graphs are large and dynamic,
- lifetimes are hard to root,
- and strict determinism is not required.

But this is not free: GC changes the failure and latency profile.

10.4.4 The Core Tradeoff: Determinism vs Reachability-Based Reclamation

The fundamental tradeoff is not “manual vs automatic.” It is:

- **RAII:** deterministic release at well-defined program points.
- **GC:** reclamation based on reachability, at times chosen by the runtime.

Consequences:

- RAII prioritizes predictability and strong resource protocols.
- GC prioritizes convenience of memory lifetime (but not necessarily other resources).

10.4.5 Latency and Throughput: The System-Level Perspective

RAII latency profile

RAII shifts costs to:

- construction/destruction points,
- explicit scope boundaries,
- and explicit transfers (moves).

Benefits:

- latency is usually more predictable,
- worst-case pauses are mostly under programmer control,
- cleanup does not require global heap scanning.

GC latency profile

GC introduces runtime-managed pauses and background work:

- stop-the-world phases (in some collectors),
- concurrent marking/sweeping overhead,
- write barriers and read barriers (depending on collector),
- heap growth and compaction behavior.

Even modern collectors that reduce pauses still:

- impose overhead on allocations and pointer writes,
- create non-local performance effects under load,
- can produce tail-latency spikes if not carefully tuned.

A design rule:

> **If tail latency is a first-order requirement, deterministic release models are usually easier to engineer and validate than reachability-based reclamation.**

10.4.6 Memory Fragmentation and Locality

RAII

RAII does not inherently solve fragmentation; it allows:

- explicit allocators,
- arenas,
- pool allocation,

- object layout control,
- and locality-oriented data structures.

GC

Collectors may:

- compact memory (improving fragmentation),
- but also move objects (requiring pointer updates),
- and impose barriers and metadata overhead.

In systems requiring stable addresses (FFI, OS handles, memory-mapped structures), moving GC can be problematic, leading to pinning and reduced compaction effectiveness.

10.4.7 Non-Memory Resources: The Major Architectural Difference

GC primarily manages **memory**. But complex systems are constrained by many resources besides memory:

- descriptor limits,
- lock protocols,
- transaction boundaries,
- network session lifetimes,
- temporary state changes.

These resources must be released deterministically. In GC languages, this is typically achieved via:

- explicit `close()` methods,
- `try/finally` constructs,
- structured “with” or “using” blocks,
- explicit cancellation protocols.

Observation:

> **For non-memory resources, disciplined systems in GC languages often reintroduce RAII-like structured cleanup patterns.**

10.4.8 Failure Modes and Operational Risk

RAII failure modes

- leaks or double frees when ownership is unclear,
- use-after-free when borrowing escapes owner lifetime,
- shutdown order problems when ownership graphs are poorly designed.

These are design issues and are mitigated by:

- explicit ownership models,
- move-only resource wrappers,
- value-centric APIs,
- staging teardown and testing shutdown paths.

GC failure modes

- memory retained by unintended references (logical leaks),
- throughput loss due to high allocation rates,
- tail-latency spikes due to GC activity,
- pauses correlated with heap size and reference graph complexity,
- operational tuning burdens (heap sizing, GC mode selection, pause targets).

These are not “bugs” in code, but runtime behavior tradeoffs that become operational constraints.

10.4.9 Concurrency: Sharing vs Transfer

RAII-oriented concurrency

RAII pairs naturally with:

- message passing using move-only job payloads,
- explicit ownership transfer between threads,
- confinement of mutable identity to a thread (actor-like designs),
- minimal shared mutable state.

GC-oriented concurrency

GC makes sharing object graphs across threads easy at the language level, but:

- does not remove the need for synchronization on mutable state,
- may add overhead through barriers and collector coordination,
- may make memory access patterns less predictable under heavy concurrency.

10.4.10 Expressing Intent: “Owning” vs “Borrowing”

A practical engineering difference:

- RAII encourages encoding owning vs non-owning relationships in types (`unique_ptr`, references, views).
- GC systems often encode ownership implicitly through reachability, which can obscure lifetime boundaries and responsibility.

This influences architecture: explicitness supports refactoring and long-term maintenance.

10.4.11 Hybrid Architectures: RAII at the Core, GC at the Edge

Many real systems adopt a hybrid approach:

- performance-critical cores in RAII languages (C/C++/Rust),
- application orchestration and UI in GC languages,
- boundaries enforced through clear ownership and data transfer formats.

A robust boundary rule:

> **Cross-language boundaries should exchange values (immutable data, buffers) rather than shared mutable object graphs.**

10.4.12 Concrete Design Guidance: Choosing the Model

Prefer RAII-first designs when

- deterministic cleanup of non-memory resources is critical,
- predictable latency and tail behavior are requirements,

- memory and resource budgets are tight,
- stable shutdown ordering matters,
- integration with OS/hardware/FFI requires stable addresses and precise lifetimes,
- the system must remain maintainable under heavy concurrency and performance constraints.

Prefer GC-first designs when

- the domain is dominated by complex object graphs and rapid feature iteration,
- deterministic resource release is not the primary constraint (or is handled explicitly),
- operational teams can accept and tune GC behavior,
- the product can tolerate occasional latency variability.

10.4.13 A C++ Lens: RAII Does Not Mean "Manual Everything"

A common misreading: "RAII requires manual cleanup everywhere." In practice:

- RAII *eliminates* manual cleanup by localizing resource release into destructors,
- the programming model becomes structured and predictable,
- the remaining challenge is architectural: designing ownership graphs and lifetimes well.

10.4.14 Illustration: RAII for a Non-Memory Resource vs GC-Style Explicit Close

RAII wrapper

```
struct FileApi {
```

```
  int open(const char* path);
  void close(int fd) noexcept;
};

class File {
public:
  File(FileApi& api, const char* path) : api_(api), fd_(api_.open(path)) {}

  File(const File&) = delete;
  File& operator=(const File&) = delete;

  File(File&& o) noexcept : api_(o.api_), fd_(std::exchange(o.fd_, -1)) {}
  File& operator=(File&& o) noexcept {
    if (this != &o) { reset(); fd_ = std::exchange(o.fd_, -1); }
    return *this;
  }

  ~File() { reset(); }

private:
  void reset() noexcept {
    if (fd_ != -1) api_.close(fd_);
    fd_ = -1;
  }

  FileApi& api_;
  int fd_{-1};
};
```

Explicit close style (conceptually common in GC environments)

```
void process_file(FileApi& api, const char* path) {
  int fd = api.open(path);
  try {
    // use fd
  } catch (...) {
    api.close(fd);
    throw;
  }
  api.close(fd);
}
```

The contrast:

- RAII enforces pairing by lifetime.
- Explicit close requires discipline across all control paths.

10.4.15 Checklist: Evaluating RAII vs GC for a Long-Lived System

1. **Is deterministic release of non-memory resources critical?**
2. **Is tail latency a strict requirement?**
3. **Do you need stable addresses and tight OS integration?**
4. **Can you accept runtime tuning and GC-driven variability?**
5. **Is memory retention by accidental references a significant risk?**
6. **Will your architecture benefit from explicit ownership boundaries?**
7. **Can you confine GC behavior to non-critical layers while keeping a RAII core?**

10.4.16 Summary

RAII and GC represent different system contracts:

- **RAII** provides deterministic cleanup for all resources and supports predictable shutdown and latency.
- **GC** provides automatic reclamation of unreachable memory and simplifies certain graph-heavy programming models, at the cost of runtime-managed overhead and potential latency variability.

For long-lived systems that must remain stable under load, failures, and evolution, RAII functions as a system-level design tool because it makes cleanup deterministic and binds resource protocols to lifetimes. GC can be highly effective in application domains, but it shifts key constraints into runtime behavior, operational tuning, and non-deterministic reclamation timing.

Part IV

Interfaces, APIs, and Contracts

Designing Stable Interfaces

11.1 API vs ABI

11.1.1 Why the Distinction Is Not Academic

In long-lived C++ systems, interface stability is not a vague goal; it is an operational constraint. Breaking an interface can mean:

- forced full rebuilds across a monorepo,
- binary incompatibility across independently deployed components,
- plugin failures at runtime,
- undefined behavior due to layout or calling convention mismatch,
- subtle data corruption across shared library boundaries.

Therefore we must separate two different promises:

- **API stability**: source-level compatibility.
- **ABI stability**: binary-level compatibility.

Confusing them creates systems that compile but fail in production.

11.1.2 Definitions

API (Application Programming Interface)

An API is the **source-level contract** a programmer writes against:

- function signatures,
- class public members and methods,
- templates and concepts requirements,
- namespaces and header-visible declarations,
- documented preconditions/postconditions and error behavior.

If you change the API, client code may no longer compile or may change behavior.

ABI (Application Binary Interface)

An ABI is the **binary-level contract** between separately compiled units:

- name mangling,
- calling conventions,
- object layout (size, alignment, field offsets),
- vtable layout and RTTI structures,
- exception handling mechanism,
- binary representation of standard library types,
- struct/class padding and member ordering,

- symbol visibility and linkage conventions.

If you change the ABI, code may still compile, but binaries may fail to link, load, or behave correctly at runtime.

11.1.3 The Practical Matrix: API Compatibility vs ABI Compatibility

There are four real-world states:

1. **API compatible, ABI compatible**:
    - clients compile and existing binaries keep working.
2. **API breaking, ABI compatible** (rare and usually artificial):
    - existing binaries may work,
    - but client source fails to compile against the new headers.
3. **API compatible, ABI breaking** (dangerous):
    - client source compiles,
    - but prebuilt binaries/plugins crash or corrupt data at runtime.
4. **API breaking, ABI breaking**:
    - both source and binary compatibility are broken.

The third case is the one that causes the most production incidents.

11.1.4 What Typically Belongs to the ABI Surface in C++

If you expose these across shared-library boundaries, they become ABI commitments:

- concrete classes with data members,
- virtual functions (vtable and RTTI),
- inline functions in headers (codegen becomes part of the client),
- templates instantiated in client code,
- exceptions crossing module boundaries,
- STL types in public signatures (depending on platform/toolchain constraints),
- enums with unspecified underlying type,
- `std::function` and other type-erased wrappers (layout and allocator interplay),
- any type with alignment-sensitive layout and padding.

A strong rule for long-lived systems:

> **If you expose C++ object layout across a boundary, you have committed to ABI stability for that layout.**

11.1.5 Common ABI Breaks That Look "Harmless" at the Source Level

1) Adding a data member to a public class

This changes size and layout. Existing binaries that allocate or embed the type break.

```
// v1
struct Widget {
  int id;
};

// v2: source-compatible for some uses, ABI-breaking
struct Widget {
  int id;
  int flags; // changes sizeof(Widget), offsets, alignment
};
```

2) Reordering members or changing packing/alignment

Offsets change, ABI breaks.

3) Adding/removing virtual functions or changing inheritance

Virtual dispatch layout changes, ABI breaks. Even reordering virtual functions can change vtable indices.

4) Changing exception specifications or throwing across boundaries

Exception propagation across boundaries depends on compatible runtime and ABI conventions. Even when it “works,” it is a fragile contract.

5) Changing the underlying type of an enum or the width of integer types

Binary representation changes.

6) Changing inlining and header-only code behavior

Inlining is not only about performance; it is also an ABI and deployment issue:

- the client embeds code generated from headers,
- replacing the library binary alone might not change client behavior.

11.1.6 The C Boundary: Why C APIs Are Often Used for ABI Stability

Many long-lived systems expose a C ABI for stability because:

- C name mangling is stable and simpler,
- calling conventions are standardized per platform,
- struct layout is simpler (still a commitment but more predictable),
- no exceptions, RTTI, or vtables are involved,
- language interoperability improves (plugins, other languages).

This does not mean the system must be written in C. It means the boundary is expressed in a stable ABI form.

Example: stable C ABI with opaque handles

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct engine_handle engine_handle;

engine_handle* engine_create();
void engine_destroy(engine_handle*);

int engine_process(engine_handle*, const char* input, char* out_buf, int out_cap);
```

```
#ifdef __cplusplus
}
#endif
```

Here:

- clients never see object layout,
- ABI stability is tractable,
- implementation can evolve internally.

11.1.7 C++ ABI Stability Strategy: Hide Layout, Expose Behavior

If you must expose C++ at the boundary, you must avoid exposing layout. Two common strategies:

Strategy A: PIMPL (Pointer to Implementation)

Public class becomes a stable handle; internal layout can change.

```
#include <memory>
#include <string_view>

class Engine {
public:
  Engine();
  ~Engine();

  Engine(Engine&&) noexcept;
  Engine& operator=(Engine&&) noexcept;
```

```
  Engine(const Engine&) = delete;
  Engine& operator=(const Engine&) = delete;

  int process(std::string_view input);

private:
  struct Impl;
  std::unique_ptr<Impl> p_; // ABI surface: pointer size is stable
};
```

Rules for ABI stability with PIMPL:

- keep inline methods minimal,
- avoid exposing STL types if your ABI policy requires it (or standardize toolchain),
- avoid exceptions crossing the boundary; return error codes or expected-like results.

Strategy B: Pure abstract interface + factory functions

Expose only a stable vtable contract and avoid concrete layout. This still commits to the vtable layout, so it must be versioned and stable.

```
#include <string_view>

struct IEngine {
  virtual ~IEngine() = default;
  virtual int process(std::string_view input) = 0;
};

IEngine* make_engine();     // factory
```

```
void destroy_engine(IEngine*);
```

This approach is common in plugin architectures but requires strict control over:

- compiler/toolchain ABI compatibility,
- virtual interface evolution (versioning),
- exception policy across boundaries.

11.1.8 Versioning: You Cannot Avoid It

A stable interface policy needs versioning rules.

API versioning

- deprecate before removing,
- maintain backwards-compatible overloads,
- keep semantics stable (not just signatures).

ABI versioning

- avoid ABI breaks in published binary interfaces,
- if ABI must change, bump the ABI version and support parallel loading if required,
- prefer additive extension points (new functions, new interfaces) over mutation of existing layouts.

11.1.9 Where ABI Stability Matters Most

ABI stability is primarily a concern when you have:

- shared libraries distributed independently from clients,
- plugins loaded at runtime,
- multiple teams building separately and deploying binaries independently,
- long-term support releases where client rebuild is not feasible.

In a monorepo where everything rebuilds together, ABI stability may be less critical than:

- build time and header dependencies,
- source-level churn,
- test stability.

But once you cross organizational or deployment boundaries, ABI becomes a hard constraint.

11.1.10 A Practical C++ Rule Set for Stable Interfaces

Rule 1: Decide your boundary policy explicitly

Choose one:

- C ABI boundary,
- C++ with PIMPL,
- pure virtual interface with strict versioning,
- header-only (API-only, no ABI promise) with full rebuild expectation.

Rule 2: Minimize the ABI surface

- hide data members,
- avoid exposing templates across binary boundaries,
- avoid exposing STL types unless toolchain is standardized and controlled,
- keep inline functions minimal.

Rule 3: Do not let exceptions cross module boundaries

Use:

- error codes,
- status objects,
- expected-like return types,
- or translate exceptions at the boundary into stable errors.

Rule 4: Design extension points

Prefer:

- adding new functions,
- adding new interfaces,
- versioned factories,

over changing existing type layouts.

11.1.11 Checklist: "Is This an API Change or an ABI Change?"

1. Will existing client source compile without change?
2. Will existing compiled binaries (plugins) continue to link/load?
3. Did you change any:
    - public class layout,
    - virtual functions/inheritance,
    - inline function bodies,
    - template-visible implementation,
    - exception behavior across the boundary,
    - STL types or allocator assumptions in signatures?
4. Can you hide the change behind PIMPL or a C ABI layer?
5. Do you need to bump an ABI version?

11.1.12 Summary

- **API** is the source-level interface: what client code writes against.
- **ABI** is the binary-level contract: what separately built components must agree on.

For long-lived C++ systems, ABI stability requires deliberate boundary design:

- keep the ABI surface small,
- avoid exposing layout and templates across binaries,
- prefer stable C ABI boundaries or PIMPL,

- version interfaces deliberately,
- and treat ABI changes as deployment events, not simple refactors.

11.2 Binary Compatibility Considerations

11.2.1 Binary Compatibility Is an Operational Contract

Binary compatibility means that **existing compiled clients** (applications, plugins, shared-library consumers) continue to:

- link successfully (static or dynamic),
- load successfully (dynamic loader),
- and run correctly (no UB caused by layout/calling mismatches),

when the provider library is updated without recompiling the client.
This is a different promise than source compatibility. A system that "still compiles" may still be binary-incompatible and fail in production.

11.2.2 What "Binary Compatibility" Actually Depends On

At a C++ boundary, binary compatibility depends on the stability of:

- **Symbol identities:** names, mangling, visibility, linkage.
- **Calling conventions:** parameter passing, return conventions, variadic rules.
- **Type representation:** size, alignment, padding, member offsets, enum underlying types.
- **Polymorphism machinery:** vtable layout, RTTI structures, dynamic_cast behavior.

- **Exception runtime:** unwinding ABI, personality routines, typeinfo.
- **Allocator and new/delete pairing:** who allocates and who frees across boundary.
- **Standard library ABI:** representation of `std::string`, `std::vector`, etc. under the chosen toolchain and mode.
- **Code generation assumptions:** inlining, visibility, LTO, optimization differences that affect boundary expectations.

11.2.3 Binary Compatibility Surfaces: What You Expose Becomes Your ABI

A practical engineering rule:

> **Every public exported symbol and every type used in exported signatures is part of your ABI surface.**

Therefore, binary compatibility starts with controlling what is exported:

- exported functions,
- exported global objects (avoid),
- exported classes (especially concrete, non-PIMPL),
- exported inline functions and templates (codegen in client).

11.2.4 The Most Common Binary Breaks in C++ Libraries

1) Changing object layout

Any change that affects:

- `sizeof(T)`,

- `alignof(T)`,
- member offsets,
- padding,

can break clients that:

- allocate the type,
- embed it,
- pass it by value,
- or access fields through compiled offsets (even if the source still compiles).

```
// ABI surface (v1)
struct PacketHeader {
  unsigned id;
  unsigned len;
};

// v2: adding a field breaks ABI for by-value usage, arrays, and embedding
struct PacketHeader {
  unsigned id;
  unsigned len;
  unsigned flags; // ABI break: size/alignment/offset changes
};
```

2) Modifying virtual interfaces

ABI hazards include:

- adding/removing/reordering virtual functions,

- changing base classes,
- adding virtual inheritance,
- changing the destructor virtuality.

Clients compile vtable slot indices and RTTI expectations into their binaries. A harmless-looking change in headers can become a runtime call into the wrong function.

3) Changing function signatures subtly

ABI breaks can occur even if the function name looks the same at source level:

- changing parameter types,
- changing return types,
- changing default arguments (API behavior changes; ABI may remain but semantics break),
- changing `noexcept` in ways that affect calling and unwinding expectations,
- changing calling convention attributes.

4) Exposing STL types across boundaries without strict toolchain control

Even when compilers appear compatible, differences in:

- standard library implementation,
- debug iterator modes,
- ABI toggles,
- allocator types and propagation,

can cause mismatch when `std::string`, `std::vector`, etc. cross module boundaries.

5) Crossing allocation/deallocation across boundaries incorrectly

If one module allocates and another deallocates, binary compatibility requires:

- consistent allocator/new/delete runtime,
- consistent CRT configuration,
- consistent rules for who owns freeing responsibility.

A safer pattern:

- the module that allocates also provides the deallocation function,
- or the boundary uses opaque handles and explicit destroy functions.

11.2.5 Binary Compatibility Design Strategies

Strategy A: C ABI boundary with opaque handles

This is the most stable and toolchain-independent approach.

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct engine_handle engine_handle;

engine_handle* engine_create();
void engine_destroy(engine_handle*);

int engine_process(engine_handle*, const char* in, char* out, int out_cap);
```

```
#ifdef __cplusplus
}
#endif
```

Properties:

- no C++ name mangling,
- no object layout exposure,
- no exceptions across boundary,
- predictable compatibility rules.

Strategy B: PIMPL to hide layout while keeping a C++ API

PIMPL makes the public class a stable shell.

```
#include <memory>
#include <string_view>

class Engine {
public:
  Engine();
  ~Engine();

  Engine(Engine&&) noexcept;
  Engine& operator=(Engine&&) noexcept;

  Engine(const Engine&) = delete;
  Engine& operator=(const Engine&) = delete;
```

```
  int process(std::string_view input);

private:
  struct Impl;
  std::unique_ptr<Impl> p_; // stable-size pointer member
};
```

Binary-compatibility discipline with PIMPL:

- keep the public class layout unchanged (do not add members),
- avoid inline implementations that depend on `Impl`,
- translate exceptions at the boundary (return status).

Strategy C: Versioned interfaces (plugin style)

If using abstract interfaces, you must version them. The safe pattern is:

- keep old interface versions,
- add new interfaces rather than mutating existing ones,
- negotiate versions at runtime.

```
struct IEngineV1 {
  virtual ~IEngineV1() = default;
  virtual int process(const char* input) = 0;
};

struct IEngineV2 : IEngineV1 {
  virtual int process2(const char* input, int flags) = 0; // additive
};
```

11.2.6 A Binary Compatibility Policy: What You Must Freeze

A long-lived library that promises binary compatibility typically freezes:

- exported symbol names (and their signatures),
- class layouts of exported concrete types,
- vtable layout of exported interfaces (or versions them),
- exception behavior across boundaries (usually: do not throw across boundary),
- allocator responsibilities (who frees),
- calling conventions and visibility rules,
- toolchain requirements (compiler + standard library + mode).

Without an explicit policy, "binary compatibility" becomes undefined.

11.2.7 Toolchain and Build-Mode Compatibility (Often Overlooked)

Binary compatibility requires more than source headers:

- consistent compiler family and version constraints (where required),
- consistent standard library implementation and ABI mode,
- consistent debug/release or iterator-debug modes,
- consistent CRT linkage and settings (especially on some platforms),
- consistent struct packing pragmas and compile flags that affect ABI.

A practical discipline:

> **If clients and library are built with different ABI-affecting modes, binary compatibility is not guaranteed even if the API is identical.**

11.2.8 Symbol Visibility and ODR: Controlling What Becomes ABI

Binary compatibility improves when you:

- export only what must be exported,
- hide internal symbols,
- avoid exporting templates and inline-heavy headers as part of the “stable ABI” promise.

Header-only code increases:

- rebuild cost,
- semantic drift (clients embed old code),
- and reduces the usefulness of binary replacement as a deployment technique.

11.2.9 Exception and Error-Model Rules at Binary Boundaries

A robust binary boundary typically enforces:

- no exceptions cross the boundary,
- errors are returned as status codes or structured status objects,
- logging/telemetry is internal or explicitly injected as an interface.

Reason:

- exception runtimes and unwind ABIs can be toolchain-sensitive,
- even when compatible, exceptions couple module lifetimes and error semantics tightly.

11.2.10 Ownership Rules Across Binary Boundaries

Cross-boundary ownership must be explicit and stable. Common safe rules:

- **Creator destroys:** the module that creates an object must also provide `destroy()`.
- **Opaque handles:** clients do not delete or free provider-owned objects.
- **No STL containers across boundary:** pass buffers with size/ptr pairs, or provide accessor APIs.

Example: creator-destroys rule

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct widget widget;

widget* widget_create();
void widget_destroy(widget*);

int widget_get_id(const widget*);

#ifdef __cplusplus
}
#endif
```

11.2.11 Designing for Additive Evolution Without ABI Break

To evolve a binary-stable library:

- add new functions rather than changing existing signatures,
- add new interface versions rather than mutating a vtable,
- add new opaque types rather than modifying exported struct layouts,
- use feature-query functions to detect capabilities at runtime.

Example: feature query

```
#ifdef __cplusplus
extern "C" {
#endif

int engine_has_feature(int feature_id);

#ifdef __cplusplus
}
#endif
```

This allows:

- old clients to keep working,
- new clients to use new features conditionally.

11.2.12 Checklist: Binary Compatibility Review

1. Did you change any exported symbol signature, name, or visibility?
2. Did you change any public type layout, alignment, or packing?
3. Did you add/reorder virtual functions or change inheritance?

4. Did you expose or alter STL types in boundary signatures?
5. Could exceptions cross the boundary under any path?
6. Is allocation/freeing responsibility explicit and unchanged?
7. Are toolchain and build modes constrained and documented?
8. Are changes additive (new functions/interfaces) rather than mutating existing ABI surfaces?

11.2.13 Summary

Binary compatibility is a strict promise: existing binaries must keep working with a new library build. In C++, this requires controlling the ABI surface and enforcing a deliberate boundary policy:

- prefer C ABI with opaque handles for maximum stability,
- or use PIMPL to hide layout while keeping a C++ API,
- version interfaces rather than mutating vtables,
- avoid exceptions and allocator crossing at boundaries,
- minimize exported symbols and avoid exposing fragile layouts and templates.

A binary-stable design is not achieved by caution alone; it is achieved by architecture that treats ABI as a first-class contract.

11.3 Header-Only vs Compiled Interfaces

11.3.1 The Real Choice: "Where Does the Code Live?"

In C++, an "interface" is not only a set of declarations. It also implies:

- where the implementation is compiled,
- who owns code generation,
- what must be rebuilt when it changes,
- and what can be replaced at deployment time.

Therefore, the practical design choice is:

- **Header-only interface:** implementation is visible to clients and compiled into every translation unit that uses it.
- **Compiled interface:** implementation is compiled into a library; clients see declarations and link to the binary.

This is not merely a performance decision. It determines:

- build times and incremental rebuild cost,
- ABI commitments and deployment strategy,
- optimization boundaries (inlining vs link-time),
- and the long-term evolution of the system.

11.3.2 Header-Only Interfaces: What They Really Mean

A header-only library typically includes:

- function definitions marked `inline`, `constexpr`, or templates,
- template-heavy components,
- policy-based and generic algorithms,
- small utilities where inlining is expected.

When you ship header-only code, you are shipping:

- the API (source contract),
- and the **implementation** (code generation instructions).

Clients compile the implementation into their binaries, which creates unique engineering properties.

11.3.3 Compiled Interfaces: What They Really Mean

A compiled interface typically means:

- headers expose declarations (and limited inlines),
- implementation lives in `.cpp` (or module implementation partitions),
- clients link against a static or shared library.

When you ship a compiled library, you ship:

- headers (API),
- and a binary (ABI + implementation).

This enables:

- binary replacement,
- controlled symbol exports,
- stable deployment artifacts.

11.3.4 The Key Engineering Dimensions

Dimension 1: Build-Time and Incremental Rebuild Cost

Header-only:

- changing a header forces recompilation of all dependent translation units,
- heavy templates amplify compile time and memory usage,
- incremental builds degrade as dependency graphs grow.

Compiled:

- implementation changes often require recompiling only the library,
- client rebuilds can be avoided when headers are stable,
- supports large-scale incremental development and faster CI cycles.

A core scaling rule:

> **In large codebases, compile-time is a first-class performance metric.**

Dimension 2: ABI and Deployment Model

Header-only:

- usually implies no ABI promise, because there is no separately deployed binary to remain compatible,
- updates require recompiling clients to pick up changes,
- old client binaries embed the old implementation indefinitely.

Compiled:

- enables ABI-stable distribution when designed appropriately,
- supports patching and security updates without rebuilding clients,
- allows separate versioning of implementation and API.

Dimension 3: Optimization and Performance

Header-only:

- enables aggressive inlining at call sites,
- improves optimization for small generic functions,
- may lead to code bloat due to repeated instantiations.

Compiled:

- may inhibit inlining across boundaries unless LTO is used,
- but can still achieve strong optimization through:
  - link-time optimization,

  - profile-guided optimization,
  - carefully chosen inline boundaries.

A mature perspective:

> **Header-only is not automatically faster; it is automatically more inlinable.**

Dimension 4: Encapsulation and Change Control

Header-only:

- exposes implementation details, intentionally or accidentally,
- makes refactoring harder because changes ripple through client builds,
- increases the risk of ODR issues if not carefully designed.

Compiled:

- allows true encapsulation of implementation,
- supports PIMPL and boundary hardening,
- reduces accidental coupling and protects invariants.

Dimension 5: ODR and One-Definition-Rule Risk

Header-only increases exposure to:

- multiple definition issues,
- inconsistent macro configurations across translation units,
- subtle behavior differences due to conditional compilation.

Compiled libraries concentrate definitions into a single place and reduce ODR surface.

11.3.5 Concrete Example: Header-Only Utility vs Compiled Library Core

Header-only: small function templates (appropriate)

```
#pragma once
#include <type_traits>

template <class T>
constexpr bool is_power_of_two(T v) {
  static_assert(std::is_unsigned_v<T>);
  return v != 0 && (v & (v - 1)) == 0;
}
```

This is appropriate because:

- it is small and generic,
- it benefits from compile-time evaluation and inlining,
- it avoids heavy dependencies and minimizes rebuild impact.

Compiled: stable service boundary (appropriate)

```
// engine.hpp
#pragma once
#include <string_view>

class Engine {
public:
  Engine();
  ~Engine();
```

```
  Engine(Engine&&) noexcept;
  Engine& operator=(Engine&&) noexcept;

  Engine(const Engine&) = delete;
  Engine& operator=(const Engine&) = delete;

  int process(std::string_view input);

private:
  struct Impl;
  Impl* p_; // could be unique_ptr in real design; shown as raw pointer for emphasis
};
```

The interface is stable while implementation can evolve in `engine.cpp`. This reduces rebuilds and supports binary replacement.

11.3.6 Where Header-Only Shines (Legitimate Use Cases)

Header-only is often the best choice when:

- **Generic programming requires templates** (containers, algorithms, policies).
- **Small utilities** where compilation cost is negligible.
- **Compile-time computation** (`constexpr`, metaprogramming).
- **Performance-critical micro-abstractions** where inlining is essential.
- **Cross-platform distribution simplicity** where shipping a binary per platform is expensive.

But the constraint is:

Header-only scales well only when dependency footprint and change frequency are controlled.

11.3.7 Where Compiled Interfaces Win (Long-Lived System Reality)

Compiled interfaces are usually superior when:

- **Large teams and large codebases:** minimizing rebuild cascades is essential.
- **Frequent implementation evolution:** you need to change internals without recompiling the world.
- **ABI-stable distribution:** plugins, shared libraries, independent deployments.
- **Security updates:** patching binaries without forcing client rebuilds.
- **Encapsulation:** protecting invariants and reducing coupling.
- **Limiting template bloat:** reducing repeated instantiations in clients.

11.3.8 The Hidden Cost of Header-Only: "Behavior Is Frozen into Clients"

With header-only code:

- every client binary contains its own copy of the implementation,
- updating the library headers does not update deployed binaries unless clients rebuild,
- bug fixes and security fixes require client rebuild and redeploy.

This is a critical operational property that affects long-lived systems.

11.3.9 A Mature Hybrid Pattern: Thin Headers, Compiled Core

A common stable strategy is:

- keep headers small, stable, and low-dependency,
- inline only what must be inline,
- compile the heavy implementation into libraries,
- expose templates only for genuinely generic layers.

This produces:

- predictable build scaling,
- better encapsulation,
- controlled ABI surface,
- strong optimization options (LTO when needed).

11.3.10 Modern C++ Modules and Interface Strategy (Conceptual View)

Modules shift the physical dependency model:

- reduce textual inclusion costs,
- reduce macro leakage,
- improve build scalability for interface consumption.

However, the core design tradeoffs remain:

- where implementation lives,

- what becomes part of the public surface,
- what must be rebuilt when it changes,
- what ABI promise is required.

Modules improve the physical coupling story, but do not remove the need for explicit API/ABI policy.

11.3.11 Decision Checklist: Choosing Header-Only vs Compiled

1. **Is the library fundamentally template-based and generic?** If yes, header-only may be natural.
2. **Is rebuild time a major constraint?** If yes, prefer compiled cores.
3. **Do you need binary replacement without client rebuilds?** If yes, compiled interface is required.
4. **Is ABI stability a requirement (plugins, shared libs)?** If yes, minimize header-only exposure.
5. **Will the implementation change frequently?** If yes, keep it compiled.
6. **Are you controlling toolchains and build modes across all consumers?** If no, avoid exposing fragile ABI surfaces.
7. **Can you keep headers small and dependency-light?** If no, header-only will not scale.

11.3.12 Summary

Header-only and compiled interfaces are different system strategies:

- **Header-only** maximizes inlining and generic flexibility but increases rebuild cascades, ODR risk, and freezes behavior into client binaries.
- **Compiled interfaces** improve encapsulation, build scalability, and enable binary replacement, but require careful ABI policy and may need LTO for cross-boundary optimization.

For long-lived systems, a disciplined hybrid is often ideal:

- keep the public surface thin and stable,
- keep heavy implementation compiled,
- use header-only templates only where generic programming genuinely requires them,
- and treat build-time and deployment constraints as first-class design requirements.

11.4 Designing for Long-Term Evolution

11.4.1 Long-Term Evolution Is a Design Constraint, Not a Hope

In long-lived systems, interfaces are not written once. They are maintained under:

- new requirements,
- new performance constraints,
- changing security expectations,
- new platforms and toolchains,
- independent deployments and plugins,
- organizational scaling and multiple teams.

Therefore, stable interface design must answer:

> **How can this interface grow for years without forcing continuous rewrites, breaking downstream users, or turning the codebase into a compatibility trap?**

The engineering goal is not to avoid change, but to make change *safe, additive, and bounded*.

11.4.2 The Three Surfaces You Must Manage

Long-term evolution requires controlling three different surfaces:

1) Semantic contract (behavior)

- preconditions and postconditions,
- performance expectations (big-O, allocations, latency),
- error behavior and recovery,
- thread-safety and reentrancy,
- lifetime and ownership rules.

2) API surface (source contract)

- signatures, types, and overload sets,
- configuration knobs and defaults,
- header exposure and template visibility.

3) ABI surface (binary contract)

- exported symbols,
- object layouts,
- vtables and RTTI,
- calling conventions and runtime assumptions.

Most interface failures happen because only the API surface was considered while semantics and ABI drifted silently.

11.4.3 Principle: Prefer Additive Evolution Over Mutating Change

The most reliable long-term strategy is:

> **Add new capabilities without changing the meaning or binary shape of existing ones.**

Practically:

- add new functions instead of changing signatures,
- add new types instead of extending public layouts,
- add new interface versions instead of mutating vtables,
- add new options rather than changing default behavior.

11.4.4 Make the Default Path Stable and Predictable

Defaults are a large part of your contract. Changing defaults is often a breaking semantic change even when source code compiles.
A long-lived interface should:

- keep defaults conservative and stable,
- introduce new defaults behind explicit version gates or opt-in flags,
- preserve old behavior for existing call sites.

11.4.5 Design for Extension Points Up Front

You cannot predict future requirements, but you can design where change will attach.
Common extension mechanisms:

Option objects (stable configuration growth)

Instead of adding parameters repeatedly, use an options struct that can grow.

```
#include <cstdint>

struct ProcessOptions {
  std::uint32_t version = 1;   // allows evolution
  std::uint32_t flags   = 0;   // future toggles
  int timeout_ms        = 0;   // 0 = default policy
};

class Engine {
public:
  int process(const char* input, const ProcessOptions& opt);
```

```
};
```

Design constraints:

- options struct is part of the contract (document fields and defaults),
- growth must preserve meaning of existing fields,
- versioning enables deterministic interpretation over time.

Callbacks and hooks (but with explicit lifetime contracts)

Hooks allow extension without changing core code. But they require strict rules:

- when callbacks run,
- thread-safety requirements,
- ownership of captured state,
- cancellation and teardown ordering.

Capability queries (feature discovery for backward compatibility)

Allow new features to be used conditionally without breaking old clients.

```
enum class Feature : int {
  fast_path = 1,
  experimental_mode = 2
};

class Engine {
public:
  bool has(Feature f) const noexcept;
};
```

11.4.6 Hide Representation; Expose Behavior

Representation changes far more frequently than behavior. Therefore:

- avoid exposing concrete layouts across boundaries,
- minimize public data members,
- use PIMPL or opaque handles for binary-stable distribution,
- keep inline functions minimal when ABI or deployment matters.

PIMPL as an evolution tool

```
#include <memory>
#include <string_view>

class Engine {
public:
  Engine();
  ~Engine();

  Engine(Engine&&) noexcept;
  Engine& operator=(Engine&&) noexcept;

  Engine(const Engine&) = delete;
  Engine& operator=(const Engine&) = delete;

  int process(std::string_view input);

private:
  struct Impl;
```

```
  std::unique_ptr<Impl> p_; // implementation can evolve independently
};
```

This allows:

- changing internal data structures,
- adding caches,
- changing algorithms,
- improving performance,

with minimal interface disruption.

11.4.7 Stability Requires Versioning Rules

You need explicit versioning rules for:

- API: how deprecation and removal are managed,
- ABI: when binary breaks are allowed and how they are communicated,
- behavior: how semantic changes are introduced.

Deprecation is a migration tool, not a warning label

A correct deprecation policy includes:

- replacement API exists before deprecation,
- deprecation produces visible diagnostics,
- a defined support window,
- clear removal criteria tied to released versions.

11.4.8 Contracts: Specify What Must Never Change

Long-term evolution is enabled by explicit contracts. For each stable interface, define:

- **Ownership rules:** who owns returned objects, how long references remain valid.
- **Thread-safety contract:** thread-safe, conditionally safe, or not safe; required external synchronization.
- **Error model:** exceptions vs status returns; which errors are recoverable.
- **Performance envelope:** allocation behavior, complexity expectations, latency goals.
- **Determinism:** whether results depend on ordering, timing, locale, or environment.

If these are not specified, change becomes accidental and breaks downstream users silently.

11.4.9 Avoid "Semantic Drift" Through Explicit Behavior Guarantees

Semantic drift is when code compiles and links, but behavior changes unexpectedly. Examples:

- new version throws on inputs that used to return errors,
- a function becomes blocking,
- new caching changes ordering or timing,
- default timeouts change.

Long-term interfaces prevent semantic drift by:

- keeping stable semantics for existing functions,
- introducing new semantics under new function names or explicit options,
- using version fields and feature flags for opt-in behavior.

11.4.10 Design the Error Boundary as a Stability Boundary

Error handling is part of the interface contract. For long-lived systems:

- avoid letting exceptions cross module boundaries in binary-stable designs,
- map internal failures to stable error codes or status objects,
- keep error domains extensible (new error codes can be added without reinterpreting old ones).

Extensible status type (conceptual pattern)

```
#include <cstdint>

struct Status {
  std::uint32_t domain = 0; // subsystem
  std::uint32_t code   = 0; // stable code space
  const char*   msg    = nullptr; // optional, non-owning view
};

class Engine {
public:
  Status process(const char* input) noexcept;
};
```

11.4.11 Prefer Value-Centric Interfaces for Stability

Value-centric APIs evolve more safely because:

- values are copied/moved explicitly,
- lifetimes are local and predictable,

- aliasing is reduced,
- internal representation can change behind the value boundary.

When identity is required, stabilize it through:

- handles/IDs,
- controlled stores,
- explicit lifetime rules.

11.4.12 Minimize "Surface Area" and "Policy Exposure"

Long-term stability improves when:

- the interface is small and orthogonal,
- each function has a single responsibility,
- configuration is consolidated into option objects,
- policies are explicit and injectable rather than hidden in globals.

Excess surface area creates:

- more compatibility promises,
- more migration burden,
- more ways to break users accidentally.

11.4.13 Evolution-Friendly Techniques in Modern C++

Technique 1: Non-member, non-friend interfaces where possible

This reduces coupling and allows alternative implementations.

Technique 2: Type erasure to decouple compile-time dependency

Type erasure can provide stable runtime polymorphism without exposing templates in the interface surface.

Technique 3: Concepts to stabilize requirements, not implementations

Concepts can clarify what a generic interface expects, but the interface must still be kept small to avoid rebuild storms.

Technique 4: Modules to reduce physical coupling

Modules reduce textual inclusion and macro leakage, improving build scalability. But they do not remove the need for stable interface policy.

11.4.14 Checklist: Is This Interface Designed to Evolve for 10+ Years?

1. Can new features be added without changing existing signatures?
2. Can implementation change without changing representation visible to clients?
3. Are defaults stable and backward-compatible?
4. Is there a defined versioning and deprecation policy?
5. Are ownership, lifetime, and thread-safety contracts explicit?
6. Is the ABI surface minimal and controlled (or explicitly not promised)?
7. Can new error codes be added without redefining old behavior?
8. Are there feature queries or options for opt-in semantics?

11.4.15 Summary

Designing for long-term evolution requires engineering the interface as a contract that can grow safely:

- manage semantic, API, and ABI surfaces explicitly,
- prefer additive evolution over mutation,
- stabilize defaults and avoid semantic drift,
- hide representation and minimize ABI surface,
- use option objects, capability queries, and versioning to attach new behavior without breaking old users,
- keep ownership, lifetime, error, and concurrency rules explicit.

The result is an interface that does not merely survive change, but makes change systematic, testable, and safe for years of continuous evolution.

Error Handling as Design

12.1 Error Codes, Exceptions, and Expected

12.1.1 Error Handling Is Part of the Interface Contract

In long-lived systems, error handling is not an implementation detail. It is a **design decision** that shapes:

- API clarity and misuse resistance,
- performance and latency behavior,
- testability and observability,
- binary boundaries and deployment safety,
- concurrency behavior and shutdown correctness.

An interface that does not define its error model precisely is incomplete. A stable system requires that error behavior is:

- explicit at boundaries,
- consistent across subsystems,

- predictable under failure,
- and easy to propagate without losing context.

12.1.2 Three Models, Three Different Contracts

C++ supports three mainstream error-reporting models that are often mixed:

1. **Error codes** (`bool`/`int`/`enum`/`std::error_code`):
    - caller must check and branch.
2. **Exceptions**:
    - failures unwind the stack; normal path stays uncluttered.
3. **`std::expected<T, E>`** (C++23):
    - return either a value `T` or an error `E` in a type-safe way.

These are not interchangeable; each encodes a different promise about control flow and responsibility.

12.1.3 A Design Taxonomy of Failures

Before choosing a mechanism, classify failures:

1) Programmer errors (contract violations)

Examples:

- invalid state usage,
- null pointer passed where prohibited,

- violating preconditions,
- API misuse.

These are usually handled by:

- assertions and contracts (where available),
- defensive checks at boundaries,
- terminating or failing fast in critical components.

2) Operational errors (expected in real environments)

Examples:

- file not found,
- network timeout,
- permission denied,
- resource exhaustion,
- transient service failures.

These must be represented and propagated reliably.

3) Recoverable domain errors

Examples:

- invalid user input,
- business rule violation,
- concurrency conflict that can be retried.

4) Unrecoverable runtime failures

Examples:

- heap corruption detection,
- invariant collapse,
- impossible state after validation.

These are typically fatal or trigger controlled shutdown.
A key rule:

> **Choose an error mechanism per failure class and boundary, not per function based on taste.**

12.1.4 Error Codes: Strengths, Weaknesses, and Proper Use

Strengths

- explicit control flow (no hidden unwinding),
- stable across ABI boundaries (especially C boundaries),
- predictable in environments where exceptions are restricted,
- easy to map to OS-level error reporting.

Weaknesses

- easy to ignore (unchecked return values),
- can clutter the normal path,
- encourages ad-hoc conventions (`-1`, `0`, magic enums),
- context propagation can be poor unless designed explicitly.

Prefer structured error codes over raw integers

A robust pattern is to return an error-domain type such as `std::error_code` or a project-specific error type. This makes error identity stable and composable.

```
#include <system_error>
#include <string_view>

std::error_code read_config(std::string_view path) noexcept;

// caller
if (auto ec = read_config("app.conf")) {
  // log ec, branch, retry, or fail startup
}
```

Design discipline:

- error codes should be **checked** at boundaries,
- prefer **noexcept** on error-code APIs to guarantee no hidden failures,
- document whether errors are retryable and what context is preserved.

12.1.5 Exceptions: Strengths, Weaknesses, and Proper Use

Strengths

- separate normal path from error path (cleaner code),
- propagate across multiple layers without repeated branching,
- naturally support composition and strong invariants (RAII + unwinding),
- can carry rich diagnostic context.

Weaknesses

- control flow is non-local (harder to reason about in some codebases),
- boundaries become sensitive (FFI, plugins, mixed runtimes),
- exception safety requires disciplined invariants and RAII everywhere,
- some environments restrict exceptions (embedded, kernel-like constraints, policy).

Exception policy must be explicit

In a long-lived system, define:

- which layers may throw,
- which layers must not throw (`noexcept` boundaries),
- how exceptions are translated into stable status at module boundaries.

Translate exceptions at boundaries

A stable boundary pattern: catch everything at subsystem edges and convert to a stable error representation.

```
#include <exception>
#include <system_error>

enum class AppErrc { ok = 0, unknown = 1, bad_input = 2 };

struct Status {
  AppErrc code{};
  const char* msg{}; // non-owning; keep ownership rules explicit
```

```
};

Status run_request() noexcept {
  try {
    // normal logic, may throw internally
    return {AppErrc::ok, nullptr};
  } catch (const std::exception& e) {
    return {AppErrc::unknown, e.what()};
  } catch (...) {
    return {AppErrc::unknown, "non-std exception"};
  }
}
```

This preserves:

- internal convenience of exceptions,
- external stability and predictability.

12.1.6 `std::expected`: Strengths, Weaknesses, and Proper Use

What `expected` represents

`std::expected<T,E>` expresses:

- **success returns a value** `T`,
- **failure returns an error** `E`,

without throwing and without losing type information.

Strengths

- forces error handling (cannot accidentally use `T` without checking),
- composes well across layers (especially domain and validation errors),
- keeps control flow explicit while preserving a value-oriented style,
- works well with `noexcept`-friendly policies.

Weaknesses

- can add syntactic overhead if not structured well,
- requires designing error types (`E`) carefully for stability,
- may encourage over-returning detailed errors where a simpler status is enough,
- does not automatically capture stack context (that is a separate observability concern).

A disciplined `expected` error type

Prefer small, stable, comparable error representations. Avoid embedding heavy objects unless justified.

```
#include <expected>
#include <string_view>

enum class ParseErrc {
  ok = 0,
  empty,
  invalid_char,
  overflow
```

```
};

struct ParseError {
  ParseErrc code{};
  std::size_t position{}; // stable, cheap context
};

std::expected<int, ParseError> parse_int(std::string_view s) noexcept;
```

Caller usage:

```
auto r = parse_int("123");
if (!r) {
  // handle r.error()
} else {
  int v = *r;
}
```

12.1.7 Choosing Between the Three: A Practical Policy

A long-lived C++ system typically benefits from a policy such as:

Policy A: Exceptions inside, status at boundaries

- Core implementation may throw for operational failures.
- Public API boundaries return `Status`/`error_code`/`expected` and are `noexcept`.
- All exceptions are caught and translated at the boundary.

Policy B: Expected everywhere for recoverable failures

- `expected` for validation/domain/operational errors that are common and recoverable.
- Exceptions reserved for truly exceptional invariants or programmer errors.

Policy C: Error codes for ABI/FFI-critical layers

- OS boundary, plugin boundary, C API boundary uses error codes/status and never throws.
- Higher layers may use `expected` for clarity.

The correct choice depends on:

- ABI constraints,
- latency/tail requirements,
- exception policy constraints,
- the ratio of success to failure in hot paths,
- and how much context must be preserved.

12.1.8 Performance and Latency: What Actually Matters

Error codes and expected

- The cost is explicit branching and returning an error object.
- Inlining and data layout matter: keep errors small and cheap to move.
- For hot paths, minimize error object construction on success.

Exceptions

- Normal path can be fast and clean.
- Throwing is typically expensive and should not be used for expected frequent failures.
- Using exceptions for common control flow (e.g., validation) tends to produce poor performance and unclear intent.

A robust performance principle:

> **Use exceptions for rare failures; use expected/error codes when failures are part of normal operation or validation.**

12.1.9 Preserving Context Without Breaking Stability

Error handling must balance:

- stable identity (error code/category),
- actionable context (position, endpoint, subsystem, retry hint),
- and low overhead.

A stable approach:

- keep **error identity** small and stable (enum/code),
- attach **context** as data fields, not as ad-hoc strings,
- let logging/telemetry be responsible for human-readable formatting.

12.1.10 Compositional Patterns With `expected`

A common problem: avoiding nested `if (!r)` blocks. Use small helper functions to linearize control flow.

Example: explicit propagation helper

```
#include <expected>
#include <utility>

template <class T, class E>
std::expected<T, E> propagate(std::expected<T, E> r) {
  return r;
}

template <class T, class E>
T value_or_throw(std::expected<T, E> r) {
  if (!r) throw r.error();
  return *r;
}
```

In production, choose one global policy:

- either stay `noexcept` with `expected` propagation,
- or convert to exceptions at a boundary (not both everywhere).

12.1.11 Designing No-Fail APIs: When `noexcept` Is a Contract

Some functions must not fail:

- destructors,
- cleanup paths,
- logging best-effort sinks,
- fail-safe shutdown actions.

For these:

- use `noexcept`,
- return status only if the caller can act on it,
- otherwise record diagnostics internally.

12.1.12 A Boundary Rule for Long-Lived Systems

To avoid fragile behavior across modules:

> **Do not let exceptions cross ABI boundaries. Translate to stable status (`error_code`/`Status`/`expected`) at the boundary.**

This rule improves:

- binary compatibility robustness,
- plugin safety,
- operational predictability during failures,
- and uniform error reporting across languages and runtimes.

12.1.13 Worked Example: One Operation, Three Error Models

Consider a parse-and-open step that can fail.

Model 1: error code

```
#include <system_error>
#include <string_view>

std::error_code parse_and_open(std::string_view spec) noexcept;
```

Model 2: exceptions (internal layer)

```
#include <string_view>

void parse_and_open_or_throw(std::string_view spec); // throws on failure
```

Model 3: expected

```
#include <expected>
#include <string_view>

enum class OpenErrc { invalid_spec, not_found, permission, io };

struct OpenError {
  OpenErrc code{};
  std::size_t pos{};
};

std::expected<void, OpenError> parse_and_open(std::string_view spec) noexcept;
```

Design interpretation:

- error codes: simplest ABI boundary, weakest enforcement.
- exceptions: clean call sites when failures are rare, but require policy boundaries.
- expected: explicit, enforced handling with structured context.

12.1.14 Checklist: Selecting the Error Model for an Interface

1. Is this a public boundary (plugins/FFI/shared libraries) where exceptions must not cross?
2. Are failures common (validation/normal operation) or rare (exceptional)?

3. Do callers need structured context to recover or retry?
4. Is `noexcept` part of the contract for this layer?
5. Is the performance budget sensitive to tail latency and frequent failures?
6. Will the error identity need to remain stable across versions?
7. Can callers realistically handle every error, or should some be fatal at this layer?

12.1.15 Summary

Error handling in Modern C++ is a design contract:

- **Error codes** are boundary-friendly and explicit but easy to ignore; prefer structured codes (`std::error_code` or stable enums) over magic integers.
- **Exceptions** keep the normal path clean and compose well with RAII, but require a clear policy and should not cross ABI boundaries.
- **`std::expected`** provides a type-safe value-or-error contract that forces handling and works well for recoverable and frequent failures, especially under `noexcept` policies.

A long-lived system succeeds when it defines a consistent policy:

- classify failures,
- choose mechanisms per layer and boundary,
- preserve stable error identity and structured context,
- and translate errors at system edges into a uniform, operationally reliable contract.

12.2 Error Propagation Strategies

12.2.1 Propagation Is the Real Problem

Detecting an error is usually easy:

- a file failed to open,
- a socket timed out,
- an invariant was violated,
- a parse failed.

The hard part in long-lived systems is **propagation**:

- carrying the error through layers without losing meaning,
- preserving enough context to debug and recover,
- avoiding duplicated handling at every layer,
- maintaining stable contracts across module boundaries,
- preventing error reporting from degenerating into inconsistent ad-hoc conventions.

A system with weak propagation strategies becomes fragile:

- failures turn into generic “unknown error”,
- logs lose the causal chain,
- retries happen blindly or never happen,
- shutdown paths leak resources or deadlock.

12.2.2 The Propagation Contract: Identity, Context, and Actionability

For propagation to remain stable over years, every error needs:

1. **Stable identity** (what category/code is this?)
2. **Structured context** (where and why did it fail?)
3. **Actionability hint** (retryable? fatal? requires user input?)

If propagation carries only a string, systems drift into:

- inconsistent wording,
- unreliable parsing,
- and lost semantics.

12.2.3 Layering: Who Detects, Who Enriches, Who Handles?

A robust architecture assigns roles per layer:

Low-level boundary layers (OS, networking, DB drivers)

- detect errors close to the source,
- capture raw codes and minimal context,
- do not decide global policy.

Mid-level service layers

- map low-level errors into domain-relevant categories,
- add structured context (operation, key, endpoint, correlation id),
- decide retry/backoff policies where appropriate.

Top-level boundaries (API endpoints, CLI, UI, plugin boundary)

- translate to stable outward representation (status codes, protocol errors),
- log once with full context,
- present minimal safe messages to external consumers.

A guiding rule:

> **Errors should be handled once, near a boundary where a decision can be made. Other layers should propagate and enrich, not "handle."**

12.2.4 Strategy 1: Return Status / Error Codes With Explicit Propagation

This strategy is boundary-friendly and common in systems that require:

- `noexcept` APIs,
- ABI stability,
- predictable control flow.

Key requirements:

- forbid ignoring results (use `[[nodiscard]]`),
- centralize conversion from raw OS errors to project status,
- attach structured context fields.

A structured status type

```
#include <cstdint>
#include <string_view>

enum class Severity : std::uint8_t { info, warning, error, fatal };
enum class Action   : std::uint8_t { none, retry, backoff, user_fix, abort };

struct Status {
  std::uint32_t domain = 0;   // subsystem
  std::uint32_t code   = 0;   // stable code within domain
  Severity severity    = Severity::error;
  Action action_hint   = Action::none;

  // stable structured context (keep small and cheap)
  std::uint64_t op_id  = 0;   // correlation id
  std::uint32_t line   = 0;   // optional: source location captured by macro
  std::string_view msg = {};  // non-owning; logging owns formatting
};

[[nodiscard]] constexpr bool ok(const Status& s) noexcept {
  return s.code == 0 && s.severity != Severity::fatal;
}
```

Propagation pattern

```
[[nodiscard]] Status read_config() noexcept;
[[nodiscard]] Status connect_db() noexcept;
[[nodiscard]] Status start_service() noexcept;
```

```
[[nodiscard]] Status boot_system() noexcept {
  if (auto s = read_config(); !ok(s)) return s;
  if (auto s = connect_db();  !ok(s)) return s;
  if (auto s = start_service();!ok(s)) return s;
  return {};
}
```

This yields:

- explicit control flow,
- uniform early return,
- stable outward contract.

12.2.5 Strategy 2: `std::expected` (Value-or-Error) Propagation

This strategy is ideal when:

- failures are part of normal operation (validation, parsing),
- you want enforced handling,
- you want value-centric composition without exceptions.

Expected with a stable error type

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

enum class Errc : std::uint32_t {
  ok = 0,
```

```
  not_found,
  invalid_input,
  permission,
  timeout,
  io,
  internal
};

struct Error {
  Errc code{Errc::ok};
  std::uint32_t domain{0};
  std::uint64_t op_id{0};
  std::string_view what{}; // non-owning diagnostic
};

template <class T>
using Result = std::expected<T, Error>;
```

Linear propagation with early returns

```
Result<int> parse_port(std::string_view s) noexcept;
Result<void> open_socket(int port) noexcept;

Result<void> start_server(std::string_view port_text) noexcept {
  auto p = parse_port(port_text);
  if (!p) return std::unexpected(p.error());

  auto r = open_socket(*p);
  if (!r) return std::unexpected(r.error());
```

```
  return {};
}
```

Design discipline:

- keep `Error` small and stable,
- attach context progressively (domain/op_id),
- avoid converting errors to strings early.

Enrichment without losing original cause

```
Result<void> open_socket(int port) noexcept;

Result<void> start_listen(int port) noexcept {
  auto r = open_socket(port);
  if (!r) {
    auto e = r.error();
    e.domain = 7;                // networking domain
    e.what   = "open_socket";     // stable step identifier
    return std::unexpected(e);
  }
  return {};
}
```

12.2.6 Strategy 3: Exceptions With Boundary Translation

This strategy is common when:

- failures are rare in the normal path,
- internal layers benefit from clean code,

- RAII and invariants are strong,
- boundary policy forbids exceptions crossing outward.

Exception propagation inside

```
#include <stdexcept>
#include <string_view>

int parse_port_or_throw(std::string_view);
void open_socket_or_throw(int port);

void start_server_or_throw(std::string_view port_text) {
  int port = parse_port_or_throw(port_text);
  open_socket_or_throw(port);
}
```

Boundary translation to stable status

```
#include <exception>
#include <string_view>

struct Status { int code{}; std::string_view msg{}; };

Status start_server(std::string_view port_text) noexcept {
  try {
    start_server_or_throw(port_text);
    return {0, {}};
  } catch (const std::exception& e) {
    return {1, e.what()};
  } catch (...) {
```

```
    return {1, "unknown failure"};
  }
}
```

Key rule:

> **Throw inside, translate at the boundary. Never allow exceptions to cross ABI boundaries.**

12.2.7 Strategy 4: Error Mapping (From Low-Level to Domain Errors)

Low-level errors are rarely meaningful to domain callers. Propagation must include **mapping**:

- OS error → transport error,
- transport error → service error,
- service error → domain error.

A good mapping preserves:

- stable domain-facing categories,
- while retaining a chain to the underlying cause for diagnostics.

Simple mapping pattern (error code style)

```
enum class DbErrc { ok=0, unavailable, auth, timeout, protocol, internal };

DbErrc map_os_to_db(int os_errno) noexcept {
  switch (os_errno) {
    case 13: return DbErrc::auth;        // example: permission
    case 110:return DbErrc::timeout;     // example: timed out
```

```
    default:return DbErrc::internal;
  }
}
```

Design note:

- mapping tables should be centralized to prevent inconsistent behavior across the codebase,
- mapping should not destroy the original raw code; retain it for debugging when needed.

12.2.8 Strategy 5: Cause Chains Without Heap-Heavy Exceptions

Complex failures often involve a causal chain:

- high-level operation failed because low-level step failed.

In a no-exceptions or expected-based system, preserve causal chains using:

- stable codes + a small “cause” field,
- or a bounded array of step identifiers,
- or an error stack captured in telemetry rather than in returned objects.

Bounded cause chain (small, stable)

```
#include <array>
#include <cstdint>

struct ErrorChain {
  std::uint32_t code = 0;
  std::uint32_t domain = 0;
  std::array<std::uint32_t, 4> steps{}; // bounded
```

```
  std::uint8_t count = 0;

  void push_step(std::uint32_t step) noexcept {
    if (count < steps.size()) steps[count++] = step;
  }
};
```

12.2.9 Propagation and Logging: Log Once, With Full Context

A recurring production anti-pattern:

- every layer logs the same error,
- log volume explodes,
- root cause becomes hard to find.

A robust policy:

> **Propagate errors upward with context. Log once at a boundary where the error is handled or reported externally.**

Exceptions:

- you may log at the source if the information will be lost (e.g., raw OS errno),
- but the general approach should prevent duplication.

12.2.10 Propagation and Retry Policy

Propagation should carry whether the error is:

- retryable immediately,

- retryable with backoff,
- non-retryable (user input required),
- fatal (shutdown required).

This avoids the dangerous pattern:

- callers retry blindly on errors that are not retryable,
- or fail fast on errors that could be recovered.

12.2.11 Propagation Across Concurrency Boundaries

Errors do not automatically propagate across:

- threads,
- tasks,
- callbacks,
- futures/promises,
- event loops.

Therefore:

- asynchronous APIs should return a **result object** that carries success/error,
- cancellation should be part of the error model,
- teardown paths must handle “in-flight” failures deterministically.

A result-carrying async completion signature (conceptual)

```
#include <expected>
#include <string_view>

struct Error { int code{}; std::string_view what{}; };

template <class T>
using Result = std::expected<T, Error>;

using Completion = void(*)(Result<int>);

void async_query(const char* sql, Completion cb) noexcept;
```

12.2.12 Propagation and ABI Boundaries

At ABI boundaries:

- do not throw across the boundary,
- return stable codes or status objects,
- avoid exposing unstable types in signatures,
- ensure ownership of error message storage is clear (prefer non-owning views or caller-provided buffers).

12.2.13 A Practical Unified Policy for Large Systems

A common, stable policy that scales:

- **Inside subsystems:**

  - use exceptions for truly rare operational failures if permitted,
  - or use `expected` for recoverable/normal failures.

- **Across subsystem boundaries:**
  - translate to `Result/Status` types and mark boundaries `noexcept`.

- **At the system boundary:**
  - map to protocol-level errors (HTTP codes, RPC statuses, CLI exit codes),
  - log once with correlation id and full structured context.

This ensures:

- predictable outward behavior,
- rich internal diagnostics,
- and stable evolution of error contracts.

12.2.14 Checklist: Error Propagation Design Review

1. Does every error have stable identity (domain + code)?
2. Is structured context preserved (operation, position, endpoint, id)?
3. Is the error actionable (retry hint, fatal vs recoverable)?
4. Are errors handled once at a boundary, not logged at every layer?
5. Are exceptions translated at ABI boundaries?
6. Is mapping from low-level to domain errors centralized and consistent?
7. Do async/concurrent APIs propagate errors explicitly?
8. Are error types small, cheap, and stable across versions?

12.2.15 Summary

Error propagation strategies determine whether a system is debuggable and evolvable. A stable long-lived C++ system:

- defines a clear propagation contract (identity + context + actionability),
- assigns responsibilities by layer (detect, enrich, handle),
- uses one or two primary mechanisms (`Status`, `expected`, exceptions) under an explicit policy,
- translates at boundaries, especially ABI boundaries,
- avoids duplicate logging and preserves causal information.

When propagation is engineered, failures remain observable, recoverable when appropriate, and do not collapse into generic "unknown error" behavior over time.

12.3 Designing Failure-Aware APIs

12.3.1 Failure-Aware Means the API Makes Failure Explicit

A failure-aware API does not merely *handle* errors internally. It *communicates* failure correctly as part of the contract, so that callers can:

- detect failure reliably,
- respond appropriately (retry, fallback, abort, user correction),
- preserve diagnostics and causal context,
- maintain system invariants under partial failure.

A system becomes fragile when failures are:

- hidden behind default values,
- converted into silent partial success,
- or expressed inconsistently across the surface area.

Failure-aware API design makes failure **unignorable, classifiable, and actionable**.

12.3.2 The Failure-Aware Contract: Five Required Properties

A long-lived, failure-aware interface should define:

1. **What can fail** (inputs, environment, resources, concurrency).
2. **How failure is reported** (status/expected/exceptions; and whether `noexcept` is guaranteed).
3. **What remains true after failure** (invariants, object state, rollback guarantees).
4. **Whether retry is valid** (retryable vs non-retryable; idempotency).
5. **What diagnostics are preserved** (stable codes + structured context).

If any is unspecified, callers either guess or over-defend, both of which harm long-term maintainability.

12.3.3 Principle 1: Make Failure Visible in the Type System

Modern C++ gives several tools to force correct usage:

Use `std::expected` for value-or-error results

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

enum class Errc : std::uint32_t {
  ok = 0,
  invalid_input,
  not_found,
  permission,
  timeout,
  io,
  conflict,
  internal
};

struct Error {
  Errc code{Errc::ok};
  std::uint32_t domain{0};    // stable subsystem id
  std::uint64_t op_id{0};     // correlation id for tracing
  std::string_view step{};    // stable step identifier
};

template <class T>
using Result = std::expected<T, Error>;
```

This prevents accidental use of the value without checking for error.

Use `[[nodiscard]]` on failure-returning functions

```
[[nodiscard]] Result<int> parse_port(std::string_view) noexcept;
```

Use explicit types instead of sentinel values

Sentinel values silently destroy failure semantics.
Bad:

```
int read_value(); // returns -1 on error (ambiguous)
```

Better:

```
[[nodiscard]] Result<int> read_value() noexcept;
```

12.3.4 Principle 2: Separate Programmer Errors from Operational Failures

A failure-aware API distinguishes:

- **programmer errors** (precondition violations): typically assertions/contracts or immediate failure,
- **operational failures** (I/O, network, resource exhaustion): reported as recoverable errors.

This prevents a destructive ambiguity:

- treating misuse as a normal error encourages sloppy callers,
- treating operational failures as fatal makes systems brittle.

12.3.5 Principle 3: Define Post-Failure Object State and Guarantees

After failure, what is the state of objects?
A failure-aware API specifies one of these guarantees:

No side effects

- on failure, nothing changes (strongest).

Strong guarantee

- either success, or state remains unchanged.

Basic guarantee

- invariants preserved, but state may change (still usable).

No guarantee (rare, dangerous)

- failure leaves object invalid (should be avoided or tightly contained).

Example: explicit guarantee documented by behavior

```
class Store {
public:
  // Strong guarantee: on failure, no record is modified.
  [[nodiscard]] Result<void> put(std::string_view key, std::string_view value)
  ↪  noexcept;
};
```

12.3.6 Principle 4: Encode Ownership and Lifetime Rules for Failure Paths

Many failures create partial results, borrowed views, or temporary buffers. Failure-aware APIs prevent lifetime bugs by:

- returning values (moveable types) rather than borrowed pointers,
- using RAII handles for resources acquired during the operation,

- avoiding implicit ownership transfers.

Prefer caller-owned buffers when crossing boundaries

This avoids hidden allocations and makes failure behavior explicit.

```
#include <cstddef>
#include <expected>

enum class BufErrc { ok=0, too_small, io, internal };

struct BufError { BufErrc code{}; std::size_t needed{}; };

std::expected<std::size_t, BufError>
read_into(void* out, std::size_t cap) noexcept;
```

Caller knows:

- how much was written on success,
- whether a retry with larger buffer is valid and how large it must be.

12.3.7 Principle 5: Make Retry Semantics Explicit (Idempotency Matters)

Failure-aware APIs should state:

- is retry safe?
- is the operation idempotent?
- can the operation partially succeed?

Without this, callers implement unsafe retries that create:

- duplicated side effects,
- repeated writes,
- inconsistent states.

Add actionability hints to errors

```
#include <cstdint>

enum class Action : std::uint8_t { none, retry, backoff, user_fix, abort };

struct Error {
  Errc code{Errc::ok};
  Action hint{Action::none};
  std::uint64_t op_id{0};
};
```

Now propagation carries:

- stable identity (code),
- and a consistent interpretation hint (action).

12.3.8 Principle 6: Avoid "Boolean APIs" for Rich Failure Domains

Returning `bool` often destroys diagnostics.
Bad:

```
bool connect(); // why did it fail?
```

Better:

```
[[nodiscard]] Result<void> connect() noexcept;
```

If you truly only need yes/no, document it explicitly and keep it narrow. But for most operational tasks, callers need reason and action.

12.3.9 Principle 7: Design Error Identity as Stable Domains and Codes

Large systems evolve. Therefore, error identity must be:

- stable across versions,
- composable across subsystems,
- and easy to map to external protocols.

A disciplined approach:

- each subsystem has a stable `domain` id,
- each domain defines stable `code` values,
- context fields are additive and optional.

This enables:

- backward-compatible addition of new codes,
- stable dashboards and alerting,
- consistent user-visible behavior.

12.3.10 Principle 8: “Noexcept at Boundaries” as a Deployment Tool

If an API is a boundary:

- shared library boundary,
- plugin boundary,
- C ABI boundary,
- RPC boundary,

then it should typically be:

- `noexcept`,
- and return a stable error representation.

This prevents:

- exceptions escaping into unknown runtimes,
- ABI fragility,
- uncontrolled termination in production.

12.3.11 Principle 9: “Handle Once, Propagate Many”

A failure-aware API and architecture follow this policy:

- low layers: detect and report precisely,
- mid layers: enrich and map to domain,
- boundary layer: handle, log once, translate outward.

This avoids:

- duplicate logs,
- lost causal chains,
- ad-hoc handling in deep layers.

12.3.12 Worked Example: Failure-Aware Parse + Open + Serve

Types

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

enum class Errc : std::uint32_t { ok=0, invalid_input, permission, timeout, io,
↪  internal };
enum class Action : std::uint8_t { none, retry, backoff, user_fix, abort };

struct Error {
  Errc code{Errc::ok};
  Action hint{Action::none};
  std::uint32_t domain{0};
  std::uint64_t op_id{0};
  std::string_view step{};
};

template <class T>
using Result = std::expected<T, Error>;
```

API

```
Result<int> parse_port(std::string_view text) noexcept;

Result<void> open_socket(int port) noexcept;     // no side effects on failure

Result<void> start_accept_loop() noexcept;       // returns retry/backoff hints
```

Composition with consistent propagation

```
Result<void> boot_server(std::string_view port_text) noexcept {
  auto p = parse_port(port_text);
  if (!p) return std::unexpected(p.error());

  auto s = open_socket(*p);
  if (!s) {
    auto e = s.error();
    e.step = "open_socket";
    e.domain = 7;
    return std::unexpected(e);
  }

  auto r = start_accept_loop();
  if (!r) {
    auto e = r.error();
    e.step = "start_accept_loop";
    e.domain = 7;
    return std::unexpected(e);
  }
```

```
  return {};
}
```

Properties:

- failures are explicit in the return type,
- errors carry stable identity and action hints,
- each step can enrich context without losing identity,
- callers can implement retry/backoff safely.

12.3.13 Checklist: Failure-Aware API Design Review

1. Are failures unignorable (`[[nodiscard]]`, `expected`/Status rather than sentinels)?
2. Is error identity stable (domain + code) and independent of strings?
3. Does the API specify post-failure state guarantees (strong/basic/no side effects)?
4. Are ownership and lifetime rules explicit for success and failure paths?
5. Are retry semantics and idempotency documented or encoded (action hints)?
6. Are boundaries `noexcept` with translation from exceptions if needed?
7. Is logging performed once at a handling boundary, not repeatedly in deep layers?
8. Can new error codes and context fields be added without breaking old callers?

12.3.14 Summary

Failure-aware APIs treat failure as a first-class part of interface design:

- they make failure explicit in types (`expected`/Status + `[[nodiscard]]`),
- preserve stable error identity and structured context,
- define post-failure state guarantees and lifetime rules,
- encode actionability (retry/backoff/user-fix) rather than forcing callers to guess,
- translate exceptions at boundaries and keep boundaries `noexcept`,
- and evolve safely by adding new codes and context without breaking existing clients.

This approach produces interfaces that remain understandable, debuggable, and maintainable under years of growth and real-world failure conditions.

12.4 Making Errors Unignorable

12.4.1 Why "Unchecked Errors" Are a System Bug

In production systems, the most damaging failure mode is not the error itself, but the caller **continuing as if nothing happened**. Unchecked failures create:

- silent data corruption (partial writes, invalid states),
- cascading failures (one error triggers many follow-up faults),
- security vulnerabilities (bypassed checks, degraded invariants),
- observability collapse (no logs at the handling boundary),

- undefined behavior due to invalid assumptions.

A long-lived C++ system must therefore design APIs such that:

> **Ignoring an error is difficult, visible, and preferably impossible.**

12.4.2 Two Kinds of "Ignorable" Failures

It is essential to separate:

1) Errors that must be handled

Examples:

- I/O failures,
- authentication failures,
- parsing/validation failures,
- resource exhaustion,
- transaction commit failures,
- network timeouts.

2) Results that may be intentionally ignored

Examples:

- best-effort logging,
- telemetry sampling drop,
- optional cache prefetch.

Failure-aware design makes the first category unignorable while keeping the second category explicitly best-effort.

12.4.3 Mechanism 1: `[[nodiscard]]` as a Contract Signal

The simplest and most effective tool is `[[nodiscard]]`. It expresses:

- this return value is meaningful,
- ignoring it is suspicious,
- treat it as an error at compile time (with warnings enabled).

Use `[[nodiscard]]` on:

- `bool` / `int` status codes,
- `std::error_code`,
- `std::expected`,
- custom `Status`/`Result` types,
- resource-acquisition results.

Apply to functions

```
#include <system_error>
#include <string_view>

[[nodiscard]] std::error_code load_config(std::string_view path) noexcept;
```

Apply to types

If a type represents success/failure, mark the type itself `[[nodiscard]]`.

```
#include <cstdint>
#include <string_view>

enum class Errc : std::uint32_t { ok=0, io, invalid_input, timeout, internal };

struct [[nodiscard]] Status {
  Errc code{Errc::ok};
  std::string_view msg{};
};

constexpr bool ok(Status s) noexcept { return s.code == Errc::ok; }
```

Then any function returning `Status` becomes naturally “must-check”.

12.4.4 Mechanism 2: Prefer `expected` Over Sentinel Values

Sentinel values are the easiest errors to ignore because they look like normal data. A failure-aware API avoids:

- returning `-1` or `nullptr` without context,
- returning empty strings/vectors on failure without distinction,
- encoding failure into a valid range of the return type.

Instead, return a discriminated result:

- `std::expected<T,E>` for value-or-error,

- or `Status` for operation-only results.

```
#include <expected>
#include <string_view>

enum class ParseErrc { ok=0, empty, invalid_char, overflow };

struct ParseError {
  ParseErrc code{ParseErrc::ok};
  std::size_t pos{};
};

[[nodiscard]] std::expected<int, ParseError>
parse_int(std::string_view s) noexcept;
```

If the caller ignores the result, `[[nodiscard]]` triggers a diagnostic; if the caller uses it, they must branch.

12.4.5 Mechanism 3: Make the Success Path Ergonomic Without Allowing Silent Failure

A common reason errors are ignored is that handling is verbose. Design the API so correct handling is easy.

Provide explicit helpers for branching

```
#include <expected>

template <class T, class E>
[[nodiscard]] constexpr bool has_value(const std::expected<T,E>& r) noexcept {
```

```
  return r.has_value();
}

template <class T, class E>
[[nodiscard]] constexpr const E& error_of(const std::expected<T,E>& r) noexcept {
  return r.error();
}
```

This encourages consistent usage patterns across the codebase.

Do not hide error checks behind implicit conversions

Avoid designing result types that implicitly convert to `bool` in ways that allow accidental misuse. Prefer explicit methods like `ok()`, `has_value()`, `error()`.

12.4.6 Mechanism 4: Strengthen the Type System With Narrow Result Types

If a function can fail, reflect it in the signature. Good patterns:

- `Result<T>` or `expected<T,E>`,
- distinct error enums per subsystem,
- typed handles for resources.

Bad patterns:

- returning `int` where values and errors overlap,
- returning `void` and logging internally while caller proceeds,
- returning a pointer with unclear ownership and null meaning.

Typed handles avoid “null means error”

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

struct SocketHandle {
  int fd = -1;
};

enum class SockErrc : std::uint32_t { ok=0, permission, timeout, io };

struct SockError { SockErrc code{SockErrc::ok}; };

[[nodiscard]] std::expected<SocketHandle, SockError>
open_socket(int port) noexcept;
```

Now the failure is explicit and cannot be confused with a valid handle.

12.4.7 Mechanism 5: Enforce Checking Through Project-Wide Policy

Compiler attributes alone are not enough if builds ignore warnings. A failure-aware organization configures:

- warnings for nodiscard results,
- warnings treated as errors in CI,
- static analysis checks for unchecked results,
- code review rules for boundary error handling.

At the API level, you can reinforce policy by:

- returning types that are hard to ignore (`expected`/Status),
- avoiding APIs that "swallow" failures,
- ensuring boundary functions return stable results and are `noexcept`.

12.4.8 Mechanism 6: Design "Must-Use" RAII Guards for Critical Operations

Some operations must be completed or rolled back. RAII can make failures unignorable by tying cleanup to lifetime.

Transaction guard pattern

```
#include <expected>

enum class TxErrc { ok=0, io, conflict, internal };
struct TxError { TxErrc code{TxErrc::ok}; };

struct [[nodiscard]] Transaction {
  bool committed = false;

  std::expected<void, TxError> commit() noexcept {
    committed = true;
    return {};
  }

  ~Transaction() {
    if (!committed) {
      // rollback best-effort; destructor must not throw
```

```
    }
  }
};
```

The type is `[[nodiscard]]`, so a transaction cannot be created and silently dropped without warnings.

12.4.9 Mechanism 7: Split Best-Effort APIs From Must-Succeed APIs

If an API is genuinely best-effort, express that clearly so callers are not forced into meaningless checks.

Best-effort logging

```
void log_best_effort(const char* msg) noexcept; // explicitly ignorable
```

Must-handle operation

```
struct [[nodiscard]] Status { int code{}; };

[[nodiscard]] Status persist_record() noexcept; // must be handled
```

This separation prevents two harmful extremes:

- forcing checks on operations where callers can do nothing,
- allowing ignoring failures where callers must react.

12.4.10 Mechanism 8: Prevent Partial Success From Looking Like Success

Some functions can return partial results:

- short reads/writes,

- partial parsing,
- best-effort operations.

Failure-aware APIs do not represent partial success as full success. They encode it explicitly.

Read APIs: return bytes + status

```
#include <expected>
#include <cstddef>

enum class ReadErrc { ok=0, eof, timeout, io };
struct ReadError { ReadErrc code{ReadErrc::ok}; };

struct ReadOk { std::size_t bytes = 0; bool eof = false; };

[[nodiscard]] std::expected<ReadOk, ReadError>
read_some(void* out, std::size_t cap) noexcept;
```

Now:

- `eof` is not an error but still a control-flow event,
- timeouts and I/O failures are explicit errors,
- and callers cannot accidentally assume “all bytes were read”.

12.4.11 Mechanism 9: Use Explicit Boundary Functions for Handling

Deep layers should propagate; boundaries should handle.
A strong policy:

- internal functions return `Status`/`expected`,

- boundary functions convert errors to external protocol representation and log once.

```
#include <expected>
#include <string_view>

struct Error { int code{}; std::string_view msg{}; };
template <class T> using Result = std::expected<T, Error>;

Result<void> do_work() noexcept;

int main() {
  auto r = do_work();
  if (!r) {
    // handle once here: log + exit code mapping
    return 1;
  }
  return 0;
}
```

This ensures errors are not ignored by “accidentally falling through.”

12.4.12 Checklist: Making Errors Unignorable

1. Are failure-returning functions and types marked `[[nodiscard]]`?
2. Are failures represented by `expected`/Status rather than sentinels?
3. Are implicit conversions avoided (no silent `bool` tricks)?
4. Are partial results represented explicitly (bytes + status, eof flags)?
5. Are best-effort operations separated from must-handle operations?

6. Are boundary layers responsible for handling and logging once?
7. Are build settings enforcing diagnostics (warnings-as-errors in CI)?
8. Are error identities stable and actionable (domain/code + hints)?

12.4.13 Summary

Making errors unignorable is a foundational requirement for long-lived systems:

- use `[[nodiscard]]` on error-bearing types and functions,
- avoid sentinel values; prefer `std::expected` or structured `Status`,
- make correct handling ergonomic and consistent,
- encode partial success explicitly,
- separate best-effort APIs from must-handle APIs,
- enforce checking through tooling and CI policy,
- handle errors once at boundaries while propagating with stable identity and context.

A failure-aware system is not one that never fails, but one that makes failure visible, structured, and impossible to ignore.

Contracts, Invariants, and Preconditions

13.1 Expressing Assumptions Explicitly

13.1.1 Assumptions Are Hidden Contracts

Every function and every type in C++ carries assumptions:

- input ranges,
- lifetime and ownership,
- ordering constraints,
- thread-safety expectations,
- performance expectations,
- representation assumptions (alignment, endianness, encoding),
- valid states and invariants.

When these assumptions are not expressed explicitly, they become **latent bugs**. The program compiles and appears correct, until:

- a caller violates an undocumented precondition,

- a future refactor weakens an invariant,
- concurrency changes introduce a race,
- performance regressions appear because a complexity assumption was false.

Long-lived systems therefore treat assumptions as part of the public contract.

13.1.2 Three Places Where Assumptions Must Be Expressed

Assumptions should be expressed in three complementary forms:

1) In the type system (strongest)

- use types that prevent invalid states,
- constrain generic code with concepts,
- separate ownership from borrowing in signatures.

2) In runtime checks (defensive boundaries)

- validate external inputs,
- assert invariants internally,
- fail fast on programmer error.

3) In documentation-level contract text (semantic promise)

- define preconditions and postconditions,
- define error behavior and guarantees,
- define thread-safety and performance constraints.

A mature design uses all three, with priority given to the type system.

13.1.3 Assumptions as Preconditions, Postconditions, and Invariants

To make assumptions explicit, classify them:

Preconditions

What must be true *before* a function is called. Examples:

- pointer must be non-null,
- index must be in range,
- buffer must be writable,
- object must be in “open” state,
- caller must hold a lock.

Postconditions

What will be true *after* successful completion. Examples:

- output buffer contains valid data,
- returned handle owns a resource,
- object remains valid and usable,
- operation is idempotent under certain inputs.

Invariants

What must be true *always* for a type (outside of construction/destruction and narrow mutation windows). Examples:

- size field matches buffer size,
- tree remains ordered,
- refcount never becomes negative,
- mutex protects specific data members.

13.1.4 The Core Rule: "Make Invalid States Unrepresentable"

The most reliable way to express assumptions is to design types that cannot represent invalid states. Examples:

- avoid raw integers for semantic values (ports, IDs, sizes),
- use strong types to prevent mixing units,
- encode optionality explicitly,
- separate owning pointers from borrowed views.

Strong types for semantic constraints

```
#include <cstdint>

struct Port {
  std::uint16_t value;

  static constexpr bool valid(std::uint32_t v) noexcept {
```

```
    return v > 0 && v <= 65535;
  }
};
```

Even stronger: a validated constructor that enforces the precondition once at the boundary, then treats the type as trusted internally.

13.1.5 Expressing Assumptions in Function Signatures

A signature should reveal:

- whether a parameter is input/output,
- whether it can be null,
- whether it is optional,
- whether ownership is transferred.

Avoid ambiguous pointer semantics

Ambiguous:

```
void set_buffer(char* p); // can p be null? who owns it? how long must it live?
```

Clearer:

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

void set_buffer(std::span<char> writable) noexcept;
```

Here the assumption is explicit:

- non-null (span has pointer+size; empty span is valid),
- writable,
- lifetime: the caller provides the storage and must keep it alive for the required duration (documented).

Optionality should be explicit

```
#include <optional>
#include <string_view>

void set_tag(std::optional<std::string_view> tag) noexcept;
```

Ownership should be explicit

```
#include <memory>

struct Engine {};

std::unique_ptr<Engine> make_engine(); // ownership transferred to caller

void use_engine(const Engine&);        // borrowing, no ownership
```

13.1.6 Concepts: Expressing Compile-Time Assumptions in Generic Code

Generic code must state what it requires. Without explicit constraints, templates accept too much and fail with obscure errors.

A concept as an explicit assumption

```
#include <concepts>
```

```
template <class T>
concept Hashable =
  requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
  };
```

Using the concept:

```
#include <unordered_set>

template <Hashable T>
void add_to_set(std::unordered_set<T>& s, const T& v) {
  s.insert(v);
}
```

Now the assumption is explicit and enforced at compile time.

13.1.7 Runtime Checks: Defensive Validation at Boundaries

Not all assumptions can be enforced by types. External inputs (files, network, UI, user text) must be validated at runtime.
Rule:

> **Validate at boundaries; trust internally under maintained invariants.**

Boundary validation example

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

```
enum class Errc { ok=0, invalid_input };

struct Error { Errc code{Errc::ok}; };

struct Port { std::uint16_t value; };

std::expected<Port, Error> parse_port(std::string_view text) noexcept {
  unsigned v = 0;
  for (char c : text) {
    if (c < '0' || c > '9') return std::unexpected(Error{Errc::invalid_input});
    v = v * 10u + unsigned(c - '0');
    if (v > 65535u) return std::unexpected(Error{Errc::invalid_input});
  }
  if (v == 0u) return std::unexpected(Error{Errc::invalid_input});
  return Port{static_cast<std::uint16_t>(v)};
}
```

Once validated, internal code can rely on the stronger assumption.

13.1.8 Assertions: Expressing Internal Assumptions and Invariants

Assertions are appropriate for programmer errors and invariant checks. They are not a replacement for validating external input.

Local invariant assertion

```
#include <cassert>
#include <vector>

struct Buffer {
```

```
  std::vector<char> data;
  std::size_t used = 0;

  void append(const char* p, std::size_t n) {
    assert(p != nullptr);
    assert(used <= data.size());
    if (used + n > data.size()) data.resize(used + n);
    std::copy(p, p + n, data.begin() + used);
    used += n;
    assert(used <= data.size());
  }
};
```

This makes internal assumptions executable and testable.

13.1.9 Designing Types Around Invariants

A long-lived system benefits from types that:

- establish invariants in constructors,
- preserve invariants in mutators,
- document invariant meaning and scope.

Example: enforcing invariant with encapsulation

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

class RingBuffer {
```

```
public:
  explicit RingBuffer(std::size_t cap)
    : buf_(cap), head_(0), tail_(0), size_(0) {
    assert(cap > 0);
  }

  bool empty() const noexcept { return size_ == 0; }
  bool full()  const noexcept { return size_ == buf_.size(); }

  void push(char c) {
    assert(!full()); // precondition for push
    buf_[tail_] = c;
    tail_ = (tail_ + 1) % buf_.size();
    ++size_;
    assert(size_ <= buf_.size()); // invariant
  }

private:
  std::vector<char> buf_;
  std::size_t head_, tail_, size_;
};
```

The invariant is expressed through:

- encapsulation (private fields),
- assertions in critical mutators,
- explicit precondition documentation (via assert and comments).

13.1.10 Explicit Assumptions About Concurrency

Concurrency assumptions must be explicit, or they become races.
A failure mode:

- an API is assumed to be thread-safe by callers,
- but it is not designed that way,
- leading to data races and undefined behavior.

A contract should specify one of:

- thread-safe (all public methods safe concurrently),
- conditionally safe (safe for const operations only, or with external lock),
- not thread-safe.

Encoding the lock requirement in the API (conceptual)

```
#include <mutex>

class Store {
public:
  void put_locked(std::unique_lock<std::mutex>& lock, int k, int v) {
    (void)lock; // lock must be held by contract
    // modify internal state
  }

private:
  std::mutex m_;
};
```

This forces the caller to provide evidence (a lock object) that the assumption is satisfied.

13.1.11 Performance Assumptions as Contracts

Long-lived systems break when performance assumptions are implicit. For key APIs, specify:

- complexity (amortized/upper bound),
- allocation behavior,
- blocking vs non-blocking behavior,
- latency constraints (especially in realtime or services).

This can be expressed via:

- separate functions for blocking vs non-blocking operations,
- option objects controlling timeouts and budgets,
- explicit naming conventions (e.g., `try_`, `blocking_`).

13.1.12 An Interface Checklist for Explicit Assumptions

1. Are preconditions expressible by types (strong types, spans, optionals, ownership types)?
2. Are generic requirements explicit (concepts) rather than implicit template errors?
3. Are boundary validations present for external inputs?
4. Are invariants encapsulated and checked (assertions/tests)?
5. Is thread-safety policy explicit and enforceable?
6. Are lifetime and ownership rules explicit in signatures and return types?
7. Are performance and blocking assumptions documented by naming and API shape?

13.1.13 Summary

Expressing assumptions explicitly is a primary tool for building long-lived C++ systems:

- encode assumptions in the type system whenever possible,
- validate inputs at boundaries,
- assert invariants internally,
- constrain generic code with concepts,
- make concurrency, lifetime, and performance assumptions explicit in API shape.

When assumptions are explicit, systems become:

- easier to reason about,
- harder to misuse,
- safer under refactoring,
- and more stable under years of evolution.

13.2 Runtime vs Compile-Time Contracts

13.2.1 Contracts Are Executable Assumptions

A contract is an explicit statement of an assumption that matters for correctness:

- preconditions (what must be true before calling),
- postconditions (what will be true after success),
- invariants (what must always hold for a type or module),

- and sometimes progress/temporal constraints (e.g., lock must be held, operation must complete within a budget).

C++ systems express contracts in two principal ways:

- **compile-time contracts**: enforced by the compiler and type system,
- **runtime contracts**: checked during execution (assertions, validation, contract checks).

A long-lived system uses both:

> **Use compile-time contracts to prevent invalid programs; use runtime contracts to prevent invalid executions.**

13.2.2 What Compile-Time Contracts Can Guarantee

Compile-time contracts are best when the property depends on *types* and *structure* rather than values. They prevent entire classes of misuse before execution.

1) Type-based validity

Examples:

- "this function requires an integral type",
- "this range must be input-iterable",
- "this policy must provide `acquire()` and `release()`".

2) Ownership and lifetime intent (partially)

Types can express ownership intent:

- `std::unique_ptr<T>` means unique ownership,

- `std::shared_ptr<T>` means shared ownership,
- references and `std::span` mean borrowing,
- `std::optional<T>` makes optionality explicit.

Compile-time contracts cannot prove real lifetimes in general, but they can prevent many API-level mistakes (e.g., passing ownership when borrowing is required).

3) Structural interface requirements (concepts)

Concepts allow precise compile-time constraints on templates.

```
#include <concepts>
#include <cstddef>

template <class T>
concept ByteReadable =
  requires(T t, void* dst, std::size_t n) {
    { t.read(dst, n) } -> std::same_as<std::size_t>;
  };
```

Using the concept:

```
#include <span>

template <ByteReadable R>
std::size_t read_exact(R& r, std::span<std::byte> out) {
  return r.read(out.data(), out.size());
}
```

Now misuse (passing an incompatible type) fails at compile time with a direct diagnostic.

4) Constant-expression constraints

When the required property is known at compile time, enforce it with:

- `static_assert`,
- `consteval` validation,
- constraints on non-type template parameters.

```
#include <cstddef>

template <std::size_t N>
struct AlignedBuffer {
  static_assert(N % 64 == 0, "N must be a multiple of 64 bytes");
  alignas(64) unsigned char data[N];
};
```

13.2.3 What Runtime Contracts Can Guarantee

Runtime contracts are best when the property depends on *values*, *environment*, or *dynamic state*. They detect invalid executions in real deployments.

1) Value constraints

Examples:

- index in range,
- pointer non-null at runtime,
- input string is valid UTF-8,
- numeric value within a domain range,

- buffer capacity sufficient.

2) State constraints

Examples:

- “socket must be open”,
- “transaction must not be committed twice”,
- “mutex must be held”,
- “object must be initialized”.

3) Environmental constraints

Examples:

- file exists and is readable,
- network reachable,
- permissions granted,
- system resources available.

4) Cross-cutting constraints (security and safety boundaries)

Examples:

- authentication required,
- authorization policy check,
- input sanitation and decoding.

13.2.4 Runtime Checking Tools: Validation vs Assertions vs Termination

Runtime contracts come in different forms depending on who is at fault and what recovery means.

Boundary validation (recoverable)

Used for external inputs and operational failures. It produces errors, not termination.

```
#include <expected>
#include <string_view>

enum class Errc { ok=0, invalid_input };

struct Error { Errc code{Errc::ok}; };

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

Assertions (programmer errors / invariant checks)

Used when violating the condition indicates a bug in the program. Assertions may be compiled out in release builds depending on policy.

```cpp
#include <cassert>
#include <vector>

void push_value(std::vector<int>& v, int x) {
  assert(x >= 0);   // programmer guarantee
  v.push_back(x);
}
```

Fail-fast termination (invariant collapse)

For critical invariants where continuing would be unsafe, a fail-fast policy can be chosen. This is common in safety/security-critical components.

```cpp
#include <cstdlib>
#include <cstdio>

[[noreturn]] inline void fail_fast(const char* msg) noexcept {
  std::fputs(msg, stderr);
  std::fputs("\n", stderr);
  std::abort();
}
```

13.2.5 Choosing Compile-Time vs Runtime: A Design Matrix

Prefer compile-time when

- the rule depends on types, interfaces, or constant parameters,
- the rule can be proven without executing the program,
- violating the rule is always a programming error.

Prefer runtime when

- the rule depends on input values, external environment, or dynamic state,
- the rule is about recoverable failures or user input,
- the system must provide diagnostics and continue.

Use both when the rule has two layers

A common pattern:

- compile-time: enforce structural constraints (types and interfaces),
- runtime: validate value constraints (ranges and state).

13.2.6 A Combined Example: Buffer Write Contract

Goal: write `N` bytes to a buffer.
Compile-time contract:

- the buffer element type must be byte-like,
- the buffer must be writable (`std::span<std::byte>`).

Runtime contract:

- the buffer must have enough capacity.

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

enum class Errc { ok=0, too_small };
```

```cpp
struct Error { Errc code{Errc::ok}; std::size_t needed{}; };

std::expected<void, Error>
write_bytes(std::span<std::byte> out, std::span<const std::byte> in) noexcept {
  if (out.size() < in.size()) {
    return std::unexpected(Error{Errc::too_small, in.size()});
  }
  for (std::size_t i = 0; i < in.size(); ++i) out[i] = in[i];
  return {};
}
```

The signature expresses:

- byte semantics and borrowing (`span`),
- explicit failure channel (`expected`),
- and the runtime precondition (capacity) is checked and reported.

13.2.7 Runtime Contracts and Performance Policy

Runtime checks have a cost, but the cost is context-dependent:

- boundary validation is non-negotiable for security and correctness,
- internal invariants can be checked selectively (debug builds, sampling, critical points),
- performance-critical hot paths may require specialized checked/unchecked variants.

A disciplined pattern:

- provide a checked API for general use,
- optionally provide an “unchecked” internal API for proven-safe call sites (kept private),
- keep the unchecked path inaccessible to general callers.

13.2.8 Contracts Across ABI Boundaries

Across ABI boundaries, runtime contracts should be:

- explicit (`noexcept` + stable status return),
- defensive (validate inputs; never assume well-formedness),
- non-throwing (exceptions must not escape).

Compile-time contracts still help at build time, but they cannot protect mixed-language callers at runtime.

13.2.9 Checklist: Contract Placement Review

1. Can the rule be enforced by types or concepts? If yes, do it at compile time.
2. Does the rule depend on values or environment? If yes, validate at runtime.
3. Is violation a programmer error? Use assertions/fail-fast, not recoverable errors.
4. Is violation caused by external inputs? Use recoverable validation with structured errors.
5. Are invariants localized and encapsulated so they can be checked effectively?
6. Are hot-path checks balanced with policy (checked public API, private unchecked fast path)?

13.2.10 Summary

Runtime and compile-time contracts are complementary:

- **Compile-time contracts** prevent invalid programs by enforcing interface and type constraints through concepts, `static_assert`, and strong types.

- **Runtime contracts** prevent invalid executions by validating dynamic state and external inputs through explicit checks, assertions, and fail-fast policies where appropriate.

Long-lived C++ systems succeed when contracts are expressed at the strongest possible level:

- encode structure in the type system,
- validate values at boundaries,
- assert invariants internally,
- and define a consistent policy for what is recoverable versus what is a bug.

13.3 Defensive Design Without Paranoia

13.3.1 The Goal: Robustness With Clear Responsibility

Defensive design is often misunderstood as "add checks everywhere." In long-lived C++ systems, that approach becomes:

- noisy code,
- duplicated validation,
- inconsistent error handling,
- reduced performance in hot paths,
- and ultimately **lower** confidence, because responsibilities are unclear.

The correct goal is:

> **Be defensive at boundaries, strict about contracts, and simple inside trusted zones.**

This is “defensive design without paranoia”:

- validate untrusted inputs and environment,
- enforce contracts for programmer errors,
- preserve invariants by construction,
- and avoid redundant checks that hide real bugs.

13.3.2 A Practical Mental Model: Trust Zones

A scalable design divides the system into trust zones:

Zone A: Untrusted boundary

Examples:

- network requests,
- files and configuration,
- user input,
- IPC and plugin boundaries,
- external library callbacks.

Rule:

> **Assume nothing. Validate everything.**

Zone B: Trusted core

Examples:

- internal domain logic,
- algorithms operating on validated data,
- invariants maintained by encapsulated types.

Rule:

> **Assume validated inputs and preserved invariants. Do not revalidate everywhere.**

Zone C: Critical invariants and safety barriers

Examples:

- memory management and ownership transitions,
- concurrency boundaries and lock discipline,
- persistence and transaction boundaries,
- security-sensitive operations.

Rule:

> **Fail fast if invariants break; continuing is unsafe.**

This zoning prevents both extremes:

- naive optimism (trusting untrusted inputs),
- paranoid duplication (rechecking trusted invariants everywhere).

13.3.3 Defensive Design Starts With Clear Contracts

The most effective defense is **clarity**:

- preconditions: what callers must guarantee,
- postconditions: what the callee guarantees on success,
- invariants: what is always preserved for a type,
- error model: how failures are reported and propagated.

A defensive system without clear contracts becomes an inconsistent pile of ad-hoc checks.

13.3.4 Principle 1: Validate at the Boundary, Normalize Once

Boundary validation should:

- reject malformed inputs,
- normalize accepted inputs into canonical internal forms,
- convert external ambiguity into explicit internal structure.

Once normalized, the trusted core should operate on the structured representation.

Example: normalize input into a strong type

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

enum class Errc { ok=0, invalid_input };
```

```cpp
struct Error { Errc code{Errc::ok}; };

struct Port {
  std::uint16_t value{};
};

[[nodiscard]] std::expected<Port, Error>
parse_port(std::string_view s) noexcept {
  if (s.empty()) return std::unexpected(Error{Errc::invalid_input});
  unsigned v = 0;
  for (char c : s) {
    if (c < '0' || c > '9') return std::unexpected(Error{Errc::invalid_input});
    v = v * 10u + unsigned(c - '0');
    if (v > 65535u) return std::unexpected(Error{Errc::invalid_input});
  }
  if (v == 0u) return std::unexpected(Error{Errc::invalid_input});
  return Port{static_cast<std::uint16_t>(v)};
}
```

After this boundary step, internal code can accept `Port` without repeating numeric checks.

13.3.5 Principle 2: Use Types to Eliminate Entire Categories of Checks

Defensive design improves when invalid states cannot be represented:

- `std::span` instead of pointer+length pairs,
- strong IDs instead of raw integers,
- `std::optional` instead of null-as-state,
- ownership types (`unique_ptr`/`shared_ptr`) instead of ambiguous raw pointers.

Example: span removes many pointer assumptions

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

void fill(std::span<std::byte> out) noexcept {
  // out may be empty, but never an invalid (dangling) pointer by itself;
  // lifetime remains a contract with the caller.
  for (std::size_t i = 0; i < out.size(); ++i) out[i] = std::byte{0};
}
```

Here, defensive complexity is reduced by signature design.

13.3.6 Principle 3: Separate Programmer Errors From Operational Failures

Paranoia happens when:

- code treats programmer misuse as recoverable,
- or treats operational failures as programmer faults.

Use two different mechanisms:

- **programmer errors:** assertions/contracts/fail-fast,
- **operational failures:** explicit error returns (`expected`/Status/error_code).

Example: assertion for misuse, expected for runtime failures

```
#include <cassert>
#include <expected>

enum class Errc { ok=0, io };
```

```cpp
struct Error { Errc code{Errc::ok}; };

class File {
public:
  explicit File(int fd) : fd_(fd) { assert(fd_ >= 0); } // programmer guarantee

  [[nodiscard]] std::expected<std::size_t, Error>
  read(void* dst, std::size_t n) noexcept {
    if (n == 0) return 0u;
    // operational failures returned as Error
    return std::unexpected(Error{Errc::io});
  }

private:
  int fd_;
};
```

This prevents “catch everything and continue” behavior that hides bugs.

13.3.7 Principle 4: Avoid Redundant Checks That Hide Real Bugs

Repeatedly checking the same condition at every layer causes:

- code duplication,
- inconsistent behavior (some layers reject, others accept),
- wasted CPU cycles,
- and a false sense of safety.

Instead, define ownership of validation:

- boundary layer validates,
- core assumes validated inputs,
- invariant checks exist at key guard points, not everywhere.

A strong heuristic:

> **If the same check appears in three places, the design is missing a trust boundary or a strong type.**

13.3.8 Principle 5: "Make Errors Unignorable" Instead of "Check More"

Defensive design improves when failures cannot be silently ignored:

- `[[nodiscard]]` return types,
- `std::expected` / `Status` instead of sentinels,
- clear boundary handling and one-time logging.

This produces robustness without paranoid revalidation.

13.3.9 Principle 6: Provide Checked Public APIs and Keep Unchecked Internals Private

In performance-sensitive systems:

- checked public API ensures correctness at boundaries,
- private unchecked functions can exist for proven-safe call sites.

The key rule:

> **Unchecked variants must be inaccessible to general callers.**

Example: checked wrapper + private unchecked core

```cpp
#include <expected>
#include <span>
#include <cstddef>

enum class Errc { ok=0, invalid_input };
struct Error { Errc code{Errc::ok}; };

class Copier {
public:
  [[nodiscard]] std::expected<void, Error>
  copy(std::span<std::byte> dst, std::span<const std::byte> src) noexcept {
    if (dst.size() < src.size()) return std::unexpected(Error{Errc::invalid_input});
    copy_unchecked(dst, src);
    return {};
  }

private:
  static void copy_unchecked(std::span<std::byte> dst, std::span<const std::byte>
  ↪  src) noexcept {
    for (std::size_t i = 0; i < src.size(); ++i) dst[i] = src[i];
  }
};
```

This design provides:

- safety at the boundary,
- performance inside trusted zones,
- clear responsibility for validation.

13.3.10 Principle 7: Defensive Concurrency Requires Explicit Synchronization Contracts

Many “paranoid” checks are attempts to compensate for unclear concurrency rules. A defensive interface states explicitly:

- thread-safe: safe for concurrent calls,
- conditionally safe: safe only under external synchronization,
- not thread-safe: single-threaded usage required.

When concurrency assumptions are explicit, the code does not need scattered defensive locks.

13.3.11 Principle 8: “Fail Fast” Only Where Continuing Is Unsafe

Fail-fast is appropriate when:

- invariants are broken (internal logic bug),
- memory corruption is suspected,
- security boundaries are violated,
- continuing would produce corrupted outputs.

Fail-fast is not appropriate for:

- normal operational errors (timeouts, not found),
- user input errors,
- expected external failures.

13.3.12 Operational Discipline: Observability Over Overchecking

Many defensive checks exist only because the system is hard to debug. Instead of scattering checks:

- attach correlation ids at boundaries,
- propagate structured error identity and context,
- log once at handling boundaries,
- export metrics for failure rates and retry counts.

A debuggable system needs fewer paranoid checks because it can *prove* what happens under failure.

13.3.13 Checklist: Defensive Without Paranoia

1. Are trust boundaries defined (untrusted inputs vs trusted core)?
2. Is input validation done once and normalized into strong internal types?
3. Are programmer errors separated from operational failures?
4. Are errors unignorable (`[[nodiscard]]`, `expected`/Status) rather than silently swallowed?
5. Are invariants enforced by encapsulation and checked at key guard points?
6. Are unchecked fast paths private and reachable only from proven-safe call sites?
7. Are concurrency assumptions explicit rather than compensated by random locking?
8. Is observability strong enough to reduce the need for defensive guessing?

13.3.14 Summary

Defensive design without paranoia is a disciplined contract strategy:

- validate and normalize at boundaries,
- express assumptions through types and concepts,
- enforce invariants by construction and encapsulation,
- separate programmer faults from operational failures,
- make failures unignorable rather than adding endless checks,
- keep unchecked performance paths private,
- and apply fail-fast only where continuing is unsafe.

This produces systems that are robust, maintainable, and performant without degenerating into noisy, duplicated defensive code.

13.4 Designing for Correctness

13.4.1 Correctness Is an Architectural Property

Correctness is not a feature you add at the end with tests. In long-lived C++ systems, correctness emerges from design choices that:

- prevent invalid states,
- make assumptions explicit,
- make failures observable and unignorable,

- preserve invariants under refactoring,
- and limit the blast radius of inevitable mistakes.

A system that relies on “developer discipline” instead of structural correctness mechanisms will degrade over time as teams grow, components evolve, and requirements shift.

13.4.2 The Correctness Stack: From Types to Runtime to Process

A practical model is a layered correctness stack:

Layer 1: Type-level correctness (prevent invalid programs)

- strong types and semantic wrappers,
- ownership expressed in types,
- concepts for generic code constraints,
- compile-time validation via `static_assert` and `consteval`.

Layer 2: Runtime correctness (prevent invalid executions)

- boundary validation of inputs and environment,
- invariant checking and assertions for programmer errors,
- explicit error channels and propagation policies.

Layer 3: System correctness (prevent integration failures)

- stable APIs and clear contracts between subsystems,
- deterministic resource management (RAII),

- well-defined concurrency policies,
- well-defined ABI boundaries and translation rules.

Layer 4: Engineering correctness (prevent regression over time)

- warnings-as-errors and static analysis,
- consistent code review rules around contracts and invariants,
- test strategy aligned with failure modes (unit, integration, property-based).

This section focuses on design-level choices that materially raise correctness.

13.4.3 Principle 1: Make Invalid States Unrepresentable

The strongest correctness strategy is to encode invariants in the type system so incorrect states cannot exist.

Replace raw primitives with semantic types

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

enum class Errc : std::uint32_t { ok=0, invalid_input };

struct Error { Errc code{Errc::ok}; };

struct UserId {
  std::uint64_t value{};
};
```

```
struct Port {
  std::uint16_t value{};
};

[[nodiscard]] std::expected<Port, Error>
parse_port(std::string_view s) noexcept;
```

Benefits:

- prevents mixing unrelated integers (e.g., port vs user id),
- turns hidden assumptions into explicit construction,
- centralizes validation into one place.

Use `std::span` to make buffer assumptions explicit

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

void process(std::span<const std::byte> input,
             std::span<std::byte> output) noexcept;
```

Now:

- size is explicit,
- null pointer ambiguity is reduced,
- and range-based correctness becomes natural.

13.4.4 Principle 2: Express Ownership and Lifetime in the API

Many correctness failures in C++ are lifetime failures. Design APIs so ownership intent is obvious and enforceable.

Prefer values and RAII handles

```
#include <memory>
#include <string_view>

struct Connection {};

std::unique_ptr<Connection> connect(std::string_view endpoint); // ownership
↪  transfer

void use(const Connection& c) noexcept; // borrowing
```

Avoid ambiguous raw-pointer ownership

Ambiguous:

```
Connection* connect(); // who deletes? can it be null? when does it expire?
```

Correctness improves when ownership is explicit and the destructor enforces cleanup.

13.4.5 Principle 3: Separate Trusted Core From Untrusted Boundaries

Correctness is easier when the system has trust zones:

- boundary: validate and normalize,
- core: assume invariants hold, keep logic simple and deterministic.

Boundary validation returns structured failure

```
#include <expected>
#include <string_view>

enum class Errc { ok=0, invalid_input };
struct Error { Errc code{Errc::ok}; };

struct Request {
  std::string_view user;
  std::string_view payload;
};

[[nodiscard]] std::expected<Request, Error>
parse_request(std::string_view raw) noexcept;
```

Once validated, internal code accepts `Request` and does not re-parse or re-check basic structure everywhere.

13.4.6 Principle 4: Choose a Single, Coherent Error Model Per Boundary

Correctness collapses when error handling is inconsistent:

- half the functions throw,
- others return `bool`,
- others return null pointers,
- some log internally and continue.

A correctness-oriented design chooses a policy:

- **inside subsystems:** exceptions for rare failures or `expected` for recoverable/frequent failures,
- **across boundaries:** `noexcept` + stable status/expected.

Make failures unignorable

```
#include <expected>

struct Error { int code{}; };

template <class T>
using Result = std::expected<T, Error>;

[[nodiscard]] Result<void> save_record() noexcept;
```

13.4.7 Principle 5: Define Postconditions and Failure Guarantees

Correctness requires that callers know what remains true after failure.
A robust API documents and enforces:

- strong guarantee (no state change on failure),
- or basic guarantee (invariants preserved, state may change).

Example: strong guarantee by design

```
#include <expected>
#include <string_view>

enum class Errc { ok=0, io, conflict };
struct Error { Errc code{Errc::ok}; };
```

```
class Store {
public:
  // Strong guarantee: if this fails, the store is unchanged.
  [[nodiscard]] std::expected<void, Error>
  put(std::string_view key, std::string_view value) noexcept;
};
```

Correctness improves because recovery logic becomes deterministic.

13.4.8 Principle 6: Invariants by Construction and Encapsulation

Types that expose internal representation force callers to maintain invariants. Correctness requires:

- private representation,
- invariant-preserving operations,
- narrow mutation points.

Example: invariant-preserving class

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

class IndexSet {
public:
  explicit IndexSet(std::size_t n) : bits_(n, false) {}

  void set(std::size_t i) {
```

```
    assert(i < bits_.size()); // precondition for internal callers
    bits_[i] = true;
  }

  bool test(std::size_t i) const {
    assert(i < bits_.size());
    return bits_[i];
  }

  std::size_t size() const noexcept { return bits_.size(); }

private:
  std::vector<bool> bits_;
};
```

In production, boundary callers validate indices; internal code asserts invariants and stays simple.

13.4.9 Principle 7: Concurrency Correctness Requires Explicit Policy

Correctness in concurrent systems is impossible without explicit rules.
Every interface that can be called concurrently must specify:

- thread-safe,
- conditionally safe (requires external synchronization),
- or single-threaded.

Encoding external synchronization as part of the contract

```
#include <mutex>
```

```
class Cache {
public:
  void put_locked(std::unique_lock<std::mutex>& lock, int k, int v) {
    (void)lock; // lock held is a precondition
    // mutate internal state
  }

private:
  std::mutex m_;
};
```

Correctness improves because the API makes the assumption visible.

13.4.10 Principle 8: Deterministic Resource Management as a Correctness Tool

Resource correctness is about more than memory:

- file descriptors,
- sockets,
- mutexes,
- transactions,
- temporary privileges,
- timers and registrations.

RAII makes cleanup deterministic and reduces correctness reliance on “remembering to call cleanup.”

Scope-bound resource guard

```
#include <utility>

class ScopeExit {
public:
  explicit ScopeExit(void(*fn)() noexcept) noexcept : fn_(fn) {}
  ~ScopeExit() { if (fn_) fn_(); }
  ScopeExit(ScopeExit&& o) noexcept : fn_(std::exchange(o.fn_, nullptr)) {}
  ScopeExit(const ScopeExit&) = delete;
  ScopeExit& operator=(const ScopeExit&) = delete;

private:
  void(*fn_)() noexcept;
};
```

This supports correctness for non-memory resources and rollback-like patterns.

13.4.11 Principle 9: Make Observability Part of Correctness

Correctness in long-lived systems includes the ability to *prove* what happened. Design for:

- correlation ids propagated through operations,
- stable error identity (domain + code),
- logging once at boundaries with full context,
- metrics for failure rates and retries.

Without observability, teams add paranoid checks instead of understanding failure modes.

13.4.12 Principle 10: "Simple Core, Strict Boundary" Beats "Defensive Everywhere"

A correctness-oriented design is strict where it must be strict:

- validate external inputs,
- enforce contracts and invariants,
- fail fast on programmer errors,
- propagate operational failures explicitly.

Then the core remains:

- simpler,
- more testable,
- and less error-prone.

13.4.13 Checklist: Designing for Correctness

1. Are invalid states prevented by types (strong types, spans, optionals)?
2. Is ownership and lifetime intent explicit in signatures (values, RAII handles)?
3. Are trust boundaries defined and is validation centralized?
4. Is the error model consistent, explicit, and unignorable (`expected`/Status + `[[nodiscard]]`)?
5. Are post-failure guarantees specified (strong/basic) and enforced?
6. Are invariants encapsulated and preserved by construction?

7. Is the concurrency policy explicit and enforceable?
8. Are resources managed deterministically (RAII for all resources)?
9. Is observability designed in (stable codes, correlation ids, boundary logging)?

13.4.14 Summary

Designing for correctness in Modern C++ is an architectural discipline:

- encode assumptions in types and constraints,
- validate untrusted inputs and normalize at boundaries,
- preserve invariants through encapsulation and construction,
- choose a coherent error model and make failures unignorable,
- specify postconditions and failure guarantees,
- define concurrency rules explicitly,
- manage all resources deterministically with RAII,
- and design observability so failures are understood, not guessed.

When correctness is designed in, tests become confirmation rather than the only line of defense, and the system remains reliable under years of evolution.

Part V

Compile-Time Design and Metaprogramming

Type Systems as Design Languages

14.1 Types as Constraints

14.1.1 From "Type Checking" to "Design Checking"

In long-lived C++ systems, the type system is not merely a mechanism that prevents nonsensical operations (e.g., adding a pointer to a string). In modern C++ (C++20 and later), types can be deliberately shaped to encode *design constraints*: what is allowed, what is forbidden, what must be present, and what must be impossible.
A constraint-oriented type design aims to achieve the following goals:

- **Make illegal states unrepresentable:** if a state must not exist at runtime, it should not be constructible at compile time.
- **Move validation left:** errors become compile-time diagnostics rather than production incidents.
- **Express intent in APIs:** the signature becomes a contract that documents and enforces requirements.
- **Control complexity:** reduce defensive checks, reduce branching, reduce runtime configuration ambiguity.

This section treats the type system as a *design language*: a way to state invariants and compatibility rules so that compilers can continuously verify them.

14.1.2 Kinds of Constraints Expressible with Types

Modern C++ offers multiple layers where constraints can be encoded:

1. **Representation constraints:** prevent mixing units, ids, handles, and domain-specific scalars.
2. **Construction constraints:** ensure objects can only be created in valid states.
3. **Protocol constraints:** ensure a type supports an operation set (interface-like requirements).
4. **Semantic constraints (partially enforceable):** enforce what can be checked (syntax, exception guarantees, value categories), and document what cannot be proven (meaning, business rules).
5. **Configuration constraints:** restrict template parameters so only coherent policy combinations are possible.
6. **Performance and safety constraints:** enable overloads or algorithms only when properties like triviality, layout, or noexcept are satisfied.

14.1.3 Strong Types: Domain Constraints by Construction

A *strong type* is a wrapper that prevents accidental mixing of values that share the same primitive representation. This is one of the highest ROI techniques for long-lived systems.

```
#include <cstdint>
#include <compare>
```

```
template<class T, class Tag>
class Strong {
public:
  using value_type = T;

  constexpr explicit Strong(T v) : v_(v) {}

  constexpr T value() const noexcept { return v_; }

  friend constexpr auto operator<=>(Strong, Strong) = default;

private:
  T v_;
};

struct UserIdTag {};
struct OrderIdTag {};
struct MetersTag {};
struct SecondsTag {};

using UserId  = Strong<std::uint64_t, UserIdTag>;
using OrderId = Strong<std::uint64_t, OrderIdTag>;
using Meters  = Strong<double, MetersTag>;
using Seconds = Strong<double, SecondsTag>;

constexpr UserId  u{42};
constexpr OrderId o{42};

// auto x = (u == o);     // error: different strong types (desired)
// auto y = u + o;        // error: no operator+ (desired)
```

Design effect. Strong typing transforms entire classes of runtime bugs into compile-time errors: wrong-parameter-order bugs, unit confusion, and accidental id reuse.

14.1.4 Types as Units: Constraint Composition

For physical units or dimensional analysis, constraints can be encoded as type-level exponents. This enforces correct unit arithmetic at compile time.

```
#include <type_traits>

template<int L, int T>
struct Dim { static constexpr int len = L; static constexpr int tim = T; };

template<class D>
struct Quantity {
  double v{};
  constexpr explicit Quantity(double x) : v(x) {}
};

using Length = Quantity<Dim<1,0>>;
using Time   = Quantity<Dim<0,1>>;
using Speed  = Quantity<Dim<1,-1>>;

template<class D1, class D2>
using DimAdd = Dim<D1::len + D2::len, D1::tim + D2::tim>;

template<class D1, class D2>
constexpr auto operator*(Quantity<D1> a, Quantity<D2> b) {
  return Quantity<DimAdd<D1,D2>>(a.v * b.v);
}

template<class D1, class D2>
using DimSub = Dim<D1::len - D2::len, D1::tim - D2::tim>;
```

```
template<class D1, class D2>
constexpr auto operator/(Quantity<D1> a, Quantity<D2> b) {
  return Quantity<DimSub<D1,D2>>(a.v / b.v);
}

constexpr Speed v = Length(100.0) / Time(10.0);
// constexpr Length bad = Length(1.0) / Time(2.0); // error: dimensions don't match
```

Design effect. The **type algebra** becomes a constraint system: only dimensionally-correct expressions are representable.

14.1.5 Concepts and Requires: Protocol Constraints with First-Class Language Support

Before C++20, SFINAE and type traits were used to conditionally enable templates. While powerful, SFINAE often produced unreadable diagnostics and entangled implementation logic with enablement logic.
C++20 introduced `concepts`, `requires`-clauses, and `requires`-expressions as a first-class constraint system.

Named Constraints with Concepts

A `concept` is a named compile-time predicate over types/values. It is used to express requirements directly in template interfaces.

```
#include <concepts>
#include <utility>

template<class T>
concept Addable =
```

```
  requires(T a, T b) {
    { a + b } -> std::same_as<T>;
  };

template<Addable T>
T add(T a, T b) {
  return a + b;
}
```

Key properties of concepts-based constraints:

- Constraints are part of the **interface**, not hidden meta-programming.
- Violations produce **constraint-failure diagnostics** rather than deep substitution traces.
- Constraints participate in **overload resolution** and template partial ordering via "more constrained" selection.

Fine-Grained Constraint Expressions

A `requires`-expression can state multiple kinds of requirements:

- **Simple requirement:** an expression must be valid.
- **Type requirement:** a nested type must exist.
- **Compound requirement:** an expression must be valid and satisfy type/noexcept constraints.
- **Nested requirement:** an arbitrary boolean constraint must hold.

```
#include <concepts>
#include <type_traits>
```

```cpp
template<class T>
concept BufferLike =
  requires(T b, std::size_t n) {
    typename T::value_type;                 // type requirement
    { b.size() } -> std::convertible_to<std::size_t>;   // compound requirement (type)
    { b.data() } -> std::same_as<typename T::value_type*>;

    b.resize(n);                            // simple requirement
    requires std::is_trivially_copyable_v<typename T::value_type>; // nested requirement
  };
```

Design effect. This expresses a *design protocol* (a minimal interface) without inheritance, virtual dispatch, or runtime checks.

14.1.6 Constraints as an Overload-Resolution Language

Constraints are not just validation; they are a *selection language*: the compiler uses them to choose the best candidate among overloads/templates.

Constrained Overloads

```cpp
#include <concepts>
#include <string>
#include <string_view>

template<class T>
concept StringLike =
  std::convertible_to<T, std::string_view>;

void log(StringLike auto s) {
  // optimized string-like path
}
```

```
template<class T>
void log(const T& x) {
  // fallback path
}
```

When both are viable, the constrained overload can be preferred if it is *more constrained* in the ordering rules.

Constrained Specializations

Constraints also provide structured control over partial specializations.

```
#include <concepts>
#include <type_traits>

template<class T>
struct Serializer;

template<class T>
  requires std::is_integral_v<T>
struct Serializer<T> {
  static T read(/*...*/) { /*...*/ return {}; }
};

template<class T>
  requires std::is_floating_point_v<T>
struct Serializer<T> {
  static T read(/*...*/) { /*...*/ return {}; }
};
```

Design effect. This prevents incoherent “catch-all” specializations and encourages explicit protocol partitioning.

14.1.7 Constraints and Error Messages: Designing Diagnostics

Constraints are part of your public API. Therefore, concepts should be written to fail in ways that teach.

Prefer Named Concepts

Prefer:

```
template<class T>
concept Hashable =
  requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
  };

template<Hashable K, class V>
class HashMap { /* ... */ };
```

over anonymous, complex boolean expressions in the template header. A named concept yields clearer diagnostics and can be reused consistently.

Make Requirements Minimal and Layered

Instead of a single monolithic concept, create a hierarchy:

```
#include <concepts>

template<class T>
concept Semiregular = std::copyable<T> && std::default_initializable<T>;

template<class T>
concept Comparable =
  requires(const T& a, const T& b) { { a < b } -> std::convertible_to<bool>; };
```

```
template<class T>
concept OrderedValue = Semiregular<T> && Comparable<T>;
```

Design effect. When a user fails a higher-level concept, they can often see which lower-level concept failed.

14.1.8 Constraints vs Runtime Validation

Not everything can be proven at compile time. A robust design separates:

- **Static constraints:** type properties, operation availability, invariants of representation.
- **Dynamic constraints:** values from user input, I/O, configuration, network, sensors, clocks.

Use types to guarantee *structure*, then use runtime validation to guarantee *data*.

The "Validated Type" Pattern

A validated type is constructed only through a function that performs runtime checks, after which the type represents a trusted state for the rest of the program.

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

class NonEmptyString {
public:
  static std::optional<NonEmptyString> make(std::string s) {
    if (s.empty()) return std::nullopt;
    return NonEmptyString(std::move(s));
```

```
  }

  std::string_view view() const noexcept { return s_; }

private:
  explicit NonEmptyString(std::string s) : s_(std::move(s)) {}
  std::string s_;
};
```

Design effect. Downstream code can treat `NonEmptyString` as a *proof-carrying value*.

14.1.9 Policy Types: Constraints on Configuration Space

Templates often expose configuration via policy types. Without constraints, policy parameters create an unbounded configuration space and fragile combinations.

Constrained Policies

```
#include <concepts>

struct ThreadSafe {};
struct NotThreadSafe {};

template<class P>
concept ThreadPolicy =
  std::same_as<P, ThreadSafe> || std::same_as<P, NotThreadSafe>;

template<ThreadPolicy P>
class Cache {
public:
  void put(/*...*/) {
    if constexpr (std::same_as<P, ThreadSafe>) {
      // lock and update
```

```
    } else {
      // fast path
    }
  }
};
```

Design effect. Invalid policy types become compile-time errors. The configuration space becomes explicit and reviewable.

14.1.10 Constraints and Performance Guarantees

Constraints can enable more aggressive optimizations safely:

- If a type is **trivially copyable**, you can use bulk operations and avoid per-element logic.
- If a callable is **noexcept**, you can commit state transitions earlier without rollback paths.
- If iterators meet **random access** constraints, algorithms can use indexing and vectorization.

```
#include <type_traits>
#include <cstring>

template<class T>
void move_block(T* dst, const T* src, std::size_t n) {
  if constexpr (std::is_trivially_copyable_v<T>) {
    std::memmove(dst, src, n * sizeof(T));
  } else {
    for (std::size_t i = 0; i < n; ++i) dst[i] = src[i];
  }
}
```

Design effect. The type system becomes a *safe optimization switch*.

14.1.11 The Metaprogramming View: Constraints as Compile-Time Proofs

In metaprogramming, a constraint can be treated as a proof obligation:

- **Preconditions:** "this template is only instantiated if `C<T>` holds."
- **Lemmas:** helper concepts that factor reasoning (e.g., `Semiregular`, `Comparable`).
- **Conclusions:** overload selection and specialization become compile-time consequences.

A useful mental model: *the compiler is a proof engine* that checks whether types meet the obligations stated by your concepts.

14.1.12 Practical Design Rules

1. **Constrain at boundaries:** apply concepts at public templates and extension points, not deep inside private code.
2. **Prefer semantic names:** use `Sortable`, `BufferLike`, `Hashable` rather than `HasBeginEnd`.
3. **Keep concepts minimal:** require the smallest protocol that supports the algorithm.
4. **Avoid "boolean soup":** factor complex constraints into layered concepts.
5. **Constrain policy parameters:** keep configuration spaces finite and coherent.
6. **Use strong types for domain values:** prevent mixing identifiers, units, and handles.
7. **Accept runtime validation where needed:** types encode structure; runtime checks encode uncertain data.

14.1.13 Summary: Types as an Architectural Medium

When types act as constraints, they stop being passive annotations and become active architectural components:

- They specify **what** is allowed (protocols and invariants).
- They guide **how** code is selected (overloads, specialization ordering).
- They improve **why** the API is correct (self-documenting intent, enforced rules).
- They reduce **where** bugs can hide (invalid states eliminated at compile time).

In compile-time design, types are not merely a representation of data; they are the language in which your system's rules are written.

14.2 Encoding Rules in Types

14.2.1 Why Encode Rules in Types?

Long-lived systems fail in predictable ways: values are passed in the wrong order, operations are applied in invalid states, IDs are mixed across domains, units are confused, lifetimes are violated, and "optional" states leak into places that assume validity.
Modern C++ enables a disciplined response: *encode rules in types*. A rule encoded in types is enforced by the compiler through:

- **construction rules** (what can exist),
- **operation rules** (what can be done),
- **composition rules** (what can be combined),

- **lifetime rules** (how resources are owned and shared),
- **protocol rules** (what interfaces are required),
- **capability rules** (which features are available).

The design intent is not "more templates"; the intent is **less ambiguity**. When rules are encoded in types, the API becomes a design boundary: callers must satisfy constraints up front, and misuse becomes a compile-time error.

14.2.2 Rule Categories: What Types Can Encode Reliably

Not every rule can be proven at compile time. The following categories are strong candidates for type-level encoding:

1. **Domain separation:** prevent mixing conceptually different values that share a representation (e.g., `UserId` vs `OrderId`, meters vs seconds).
2. **State machines:** prevent calling an operation in the wrong phase (e.g., connect–authenticate–use; open–read/write–close).
3. **Ownership and lifetime:** prevent use-after-free and double-free through RAII and ownership types.
4. **Capabilities:** enable operations only when a type guarantees the capability (e.g., random-access iteration).
5. **Configuration coherence:** ensure policy combinations are valid.
6. **Valid-by-construction values:** guarantee invariants after runtime validation.

14.2.3 Encoding Domain Rules with Strong Types

Strong types encode the rule: “these values must never be mixed.”

```
#include <cstdint>
#include <compare>

template<class Rep, class Tag>
class Strong {
public:
  using rep_type = Rep;

  constexpr explicit Strong(Rep v) : v_(v) {}
  constexpr Rep value() const noexcept { return v_; }

  friend constexpr auto operator<=>(Strong, Strong) = default;

private:
  Rep v_;
};

struct UserIdTag {};
struct ProductIdTag {};
struct RequestIdTag {};

using UserId    = Strong<std::uint64_t, UserIdTag>;
using ProductId = Strong<std::uint64_t, ProductIdTag>;
using RequestId = Strong<std::uint64_t, RequestIdTag>;

void load_user(UserId);
void load_product(ProductId);

void demo() {
  UserId u{10};
```

```
  ProductId p{10};

  load_user(u);
  load_product(p);

  // load_user(p); // error: cannot convert ProductId to UserId (rule enforced)
}
```

Design rule. If two concepts are different in your domain model, they should be different types, even if both are `uint64_t` at runtime.

14.2.4 Encoding Unit Rules with Dimensional Types

Units are a special case of domain separation, with additional algebraic structure.

```
#include <type_traits>

template<int L, int T, int M>
struct Dim { static constexpr int l=L, t=T, m=M; };

template<class D>
struct Quantity {
  double v{};
  constexpr explicit Quantity(double x) : v(x) {}
};

using Length = Quantity<Dim<1,0,0>>;
using Time   = Quantity<Dim<0,1,0>>;
using Mass   = Quantity<Dim<0,0,1>>;
using Speed  = Quantity<Dim<1,-1,0>>;

template<class D1, class D2>
using DimSub = Dim<D1::l - D2::l, D1::t - D2::t, D1::m - D2::m>;
```

```
template<class D1, class D2>
constexpr auto operator/(Quantity<D1> a, Quantity<D2> b) {
  return Quantity<DimSub<D1,D2>>(a.v / b.v);
}

constexpr Speed v = Length(100.0) / Time(10.0);
// constexpr Length bad = Length(1.0) / Time(2.0); // error: dimensions mismatch
```

Design effect. The rule "do not divide length and assign to length" is enforced structurally.

14.2.5 Encoding State Rules with Typestate

Many production defects are state violations: using an object before initialization, calling `send` before `connect`, or committing before validation. Typestate encodes the allowed operations per state.

A Minimal Typestate Pattern

```
#include <string>
#include <utility>

struct Disconnected {};
struct Connected {};
struct Authenticated {};

class Socket {
public:
  Socket() = default;

  // Transition: Disconnected -> Connected
  static Socket connect(std::string endpoint) {
```

```
    Socket s;
    s.endpoint_ = std::move(endpoint);
    s.connected_ = true;
    return s;
  }

  const std::string& endpoint() const noexcept { return endpoint_; }

private:
  std::string endpoint_;
  bool connected_{false};
};

template<class State>
class Client;

template<>
class Client<Disconnected> {
public:
  Client() = default;

  Client<Connected> connect(std::string endpoint) && {
    Socket s = Socket::connect(std::move(endpoint));
    return Client<Connected>(std::move(s));
  }
};

template<>
class Client<Connected> {
public:
  Client<Authenticated> authenticate(std::string token) && {
    // perform handshake using token ...
    (void)token;
```

```
    return Client<Authenticated>(std::move(sock_));
  }

private:
  friend class Client<Disconnected>;
  explicit Client(Socket s) : sock_(std::move(s)) {}
  Socket sock_;
};

template<>
class Client<Authenticated> {
public:
  void send(std::string payload) {
    // safe: by type, this can only be called after authentication
    (void)payload;
  }

private:
  friend class Client<Connected>;
  explicit Client(Socket s) : sock_(std::move(s)) {}
  Socket sock_;
};

void demo() {
  Client<Disconnected> c;
  auto a = std::move(c).connect("srv").authenticate("token");
  a.send("data");

  // Client<Disconnected>{}.send("data");  // error: send not available
  // Client<Connected>{}.send("data");     // error: send not available
}
```

Design rule. Represent phases as types, and expose only state-valid operations. This shifts protocol mistakes from runtime to compile time.

14.2.6 Encoding Ownership Rules with RAII and Move-Only Types

Ownership is a rule: "exactly one entity releases this resource." The most robust encoding is RAII + move-only semantics.

```
#include <utility>
#include <cstdio>

class FileHandle {
public:
  explicit FileHandle(const char* path, const char* mode)
    : f_(std::fopen(path, mode)) {}

  ~FileHandle() { if (f_) std::fclose(f_); }

  FileHandle(const FileHandle&) = delete;
  FileHandle& operator=(const FileHandle&) = delete;

  FileHandle(FileHandle&& other) noexcept : f_(std::exchange(other.f_, nullptr)) {}
  FileHandle& operator=(FileHandle&& other) noexcept {
    if (this != &other) {
      if (f_) std::fclose(f_);
      f_ = std::exchange(other.f_, nullptr);
    }
    return *this;
  }

  std::FILE* get() const noexcept { return f_; }
  explicit operator bool() const noexcept { return f_ != nullptr; }
```

```
private:
  std::FILE* f_{nullptr};
};
```

Design effect. Copy is forbidden, move transfers ownership, destruction releases resource. The type encodes the lifetime rule without relying on conventions.

14.2.7 Encoding "Valid-Only" Rules with Validated Types

Some rules depend on runtime data (user input, files, network). The common failure is letting unvalidated data masquerade as valid. A validated type encodes: "if you have this type, validation already happened."

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

class PortNumber {
public:
  static std::optional<PortNumber> parse(int p) {
    if (p < 1 || p > 65535) return std::nullopt;
    return PortNumber(p);
  }

  int value() const noexcept { return p_; }

private:
  explicit PortNumber(int p) : p_(p) {}
  int p_{};
};

class NonEmptyString {
```

```
public:
  static std::optional<NonEmptyString> make(std::string s) {
    if (s.empty()) return std::nullopt;
    return NonEmptyString(std::move(s));
  }
  std::string_view view() const noexcept { return s_; }

private:
  explicit NonEmptyString(std::string s) : s_(std::move(s)) {}
  std::string s_;
};
```

Design rule. Prefer **parse/validate constructors** that return `std::optional` (or an error-aware type) instead of allowing arbitrary construction followed by scattered checks.

14.2.8 Encoding Protocol Rules with Concepts

Protocol rules describe what operations must exist and what their semantics must be compatible with. C++20 concepts encode these requirements at the API boundary.

```
#include <concepts>
#include <cstddef>

template<class T>
concept ContiguousBuffer =
  requires(T b) {
    typename T::value_type;
    { b.size() } -> std::convertible_to<std::size_t>;
    { b.data() } -> std::same_as<typename T::value_type*>;
  };

template<ContiguousBuffer B>
void fill_zero(B& b) {
```

```
  for (std::size_t i = 0; i < b.size(); ++i) b.data()[i] = {};
}
```

Design effect. Instead of documenting "must provide `size()` and `data()`", the type system enforces it.

14.2.9 Encoding Capability Rules with Iterator and Callable Constraints

Algorithms can encode which capabilities they require. This prevents accidental use of inefficient paths or invalid iterator categories.

```
#include <concepts>
#include <iterator>
#include <vector>

template<std::random_access_iterator It>
auto middle(It first, It last) {
  return first + (last - first) / 2;
}

void demo(std::vector<int>& v) {
  auto it = middle(v.begin(), v.end()); // ok
}
```

Design rule. Constrain algorithms by the weakest concept that still expresses correctness and performance intent.

14.2.10 Encoding Configuration Rules with Policy Types

Configuration rules often live in documentation and are violated silently. Policy types can be constrained so only coherent combinations are expressible.

```
#include <concepts>
#include <type_traits>

struct ThreadSafe {};
struct NotThreadSafe {};

struct LruEviction {};
struct NoEviction {};

template<class P>
concept ThreadPolicy =
  std::same_as<P, ThreadSafe> || std::same_as<P, NotThreadSafe>;

template<class E>
concept EvictionPolicy =
  std::same_as<E, LruEviction> || std::same_as<E, NoEviction>;

template<ThreadPolicy TP, EvictionPolicy EP>
class Cache {
public:
  void put(/*...*/) {
    if constexpr (std::same_as<TP, ThreadSafe>) {
      // locking path
    }
    if constexpr (std::same_as<EP, LruEviction>) {
      // LRU management path
    }
  }
};
```

Design effect. Policy misuse becomes a compile-time error, and the configuration space becomes explicit.

14.2.11 Encoding "Non-Nullable" Rules in Interfaces

Nullability is a rule: "this parameter must always refer to a valid object." Encoding this rule reduces defensive checks and clarifies responsibilities.

```
#include <functional>

struct Service {
  void run() {}
};

void execute(std::reference_wrapper<Service> svc) {
  svc.get().run(); // cannot be null
}
```

Design note. In modern C++, prefer references (or reference wrappers) for non-null requirements. Reserve raw pointers for *optional* and *observing* relationships.

14.2.12 Encoding Error-Handling Rules with Type-Based Results

Error-handling is a rule: "callers must acknowledge failure." If a function can fail, representing it as `bool` or sentinel values can be ignored. A result type encodes acknowledgment in the interface.

```
#include <variant>
#include <string>

struct ParseError { std::string message; };

template<class T>
using Result = std::variant<T, ParseError>;
```

```
Result<int> parse_int(const std::string& s) {
  try {
    return std::stoi(s);
  } catch (...) {
    return ParseError{"invalid integer"};
  }
}

int use(const std::string& s) {
  auto r = parse_int(s);
  if (auto* v = std::get_if<int>(&r)) return *v;
  // error path is explicit
  return 0;
}
```

Design rule. Do not allow "failure" to look like a valid value. Encode error states in types.

14.2.13 Tradeoffs: Where Type-Encoding Can Go Wrong

Encoding rules in types is powerful, but it must remain an engineering tool:

- **Over-constraining:** making concepts too strict reduces composability and blocks legitimate use.
- **Leaky semantics:** types can enforce syntax, but cannot prove business meaning.
- **Complexity inflation:** heavy template metaprogramming can harm readability and build times.
- **Brittle abstractions:** encoding unstable domain rules at the type level can make evolution difficult.

A practical guideline is to encode rules that are:

- **stable** across versions,
- **structural** (domain separation, protocol membership, ownership),
- **high-impact** when violated (security, safety, correctness),
- **cheap** to maintain relative to their bug-prevention value.

14.2.14 Design Checklist: Encoding Rules in Types

1. Identify the rule and classify it: domain, state, ownership, protocol, capability, configuration, validation.
2. Decide if it is **static** (type-checkable) or **dynamic** (value-checkable).
3. Prefer **strong types** to prevent mixing domain values.
4. Use **typestate** to make state violations impossible.
5. Use **RAII + move-only** types to encode ownership.
6. Use **validated types** for runtime-checked invariants.
7. Use **concepts** to encode protocols and capabilities at API boundaries.
8. Constrain policy parameters to keep configurations coherent.
9. Keep constraints minimal and diagnostics-friendly.

14.2.15 Summary

Encoding rules in types turns a C++ codebase into a continuously verified design model. It reduces the burden of runtime checks, limits the space of invalid programs, and makes architectural intent explicit in every interface. In modern C++, **the type system is not only**

about safety and correctness; it is an engineering medium for maintaining coherence, evolvability, and trust across years of change.

14.3 Preventing Invalid States

14.3.1 The Design Objective

A large C++ system does not usually fail because an algorithm is mathematically wrong. It fails because a program reaches a state that the design never intended to exist:

- a "configured" object that is missing mandatory fields,
- a socket used before authentication,
- a resource handle copied and released twice,
- a measurement with the wrong unit,
- a "nullable" dependency that is assumed non-null,
- a partially-initialized value that leaks beyond a boundary,
- a function that returns "success" and "failure" using the same type.

Preventing invalid states means shifting the burden from scattered runtime checks to *design-level enforcement*. In modern C++, the strongest enforcement lever is: **make invalid states unrepresentable** (or at least unconstructible outside validation).
This section presents a toolbox of type-driven techniques that encode invariants in interfaces, object construction, and compile-time constraints, while keeping the design evolvable.

14.3.2 A Practical Taxonomy of Invalid States

Before choosing a technique, classify the invalid states you want to prevent:

1. **Representation invalidity:** values from different domains share the same representation and are mixed.
2. **State-machine invalidity:** operations called in the wrong phase.
3. **Nullability invalidity:** a dependency required by design is modeled as optional.
4. **Lifetime invalidity:** ownership and lifetime are not encoded; use-after-free or double-release becomes possible.
5. **Value invalidity:** data must satisfy constraints (ranges, formats), but is used before validation.
6. **Configuration invalidity:** template/policy combinations are incoherent or unsupported.
7. **Error-channel invalidity:** failure is encoded as a valid value and is silently ignored.

Each class maps to a small set of proven patterns.

14.3.3 Make Illegal Combinations Impossible: Strong Types

The simplest invalid state is accidental mixing of semantically distinct values. Strong types remove that state from the program.

```
#include <cstdint>
#include <compare>

template<class Rep, class Tag>
class Strong {
```

```
public:
  using rep_type = Rep;
  constexpr explicit Strong(Rep v) : v_(v) {}
  constexpr Rep value() const noexcept { return v_; }
  friend constexpr auto operator<=>(Strong, Strong) = default;
private:
  Rep v_;
};

struct UserIdTag {};
struct SessionIdTag {};
using UserId    = Strong<std::uint64_t, UserIdTag>;
using SessionId = Strong<std::uint64_t, SessionIdTag>;

struct Session { SessionId id; };
struct User    { UserId id; };

void load_user(UserId);
void resume_session(SessionId);

void demo(User u, Session s) {
  load_user(u.id);
  resume_session(s.id);
  // resume_session(u.id); // error: invalid state (wrong domain) is unrepresentable
}
```

Design rule. If two values must not be interchangeable in your architecture, they must not share the same type.

14.3.4 Prevent Partially-Initialized Objects: Construction as a Gate

A common invalid state is "object exists but is not valid". The primary tool is to make construction the *only* way to obtain an instance, and ensure construction cannot succeed without establishing invariants.

Private Constructor + Factory that Validates

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

class NonEmptyString {
public:
  static std::optional<NonEmptyString> make(std::string s) {
    if (s.empty()) return std::nullopt;
    return NonEmptyString(std::move(s));
  }

  std::string_view view() const noexcept { return s_; }

private:
  explicit NonEmptyString(std::string s) : s_(std::move(s)) {}
  std::string s_;
};
```

Design effect. The invalid state "empty string treated as a name" cannot occur once the type exists.

Enforce Mandatory Fields: Builder that Cannot Finish Early

When multiple fields are mandatory, a naive builder allows an invalid intermediate state. A typestate builder encodes the rule: “you cannot call `build()` until required fields exist.”

```
#include <string>
#include <utility>

struct NameSet {};
struct HostSet {};
struct PortSet {};

struct AppConfig {
  std::string name;
  std::string host;
  int port;
};

template<class Name, class Host, class Port>
class ConfigBuilder;

template<>
class ConfigBuilder<void, void, void> {
public:
  ConfigBuilder<NameSet, void, void> name(std::string n) && {
    b_.name = std::move(n);
    return ConfigBuilder<NameSet, void, void>(std::move(b_));
  }
private:
  AppConfig b_{};
  template<class, class, class> friend class ConfigBuilder;
};

template<>
```

```
class ConfigBuilder<NameSet, void, void> {
public:
  ConfigBuilder<NameSet, HostSet, void> host(std::string h) && {
    b_.host = std::move(h);
    return ConfigBuilder<NameSet, HostSet, void>(std::move(b_));
  }
private:
  explicit ConfigBuilder(AppConfig b) : b_(std::move(b)) {}
  AppConfig b_;
  template<class, class, class> friend class ConfigBuilder;
};

template<>
class ConfigBuilder<NameSet, HostSet, void> {
public:
  ConfigBuilder<NameSet, HostSet, PortSet> port(int p) && {
    b_.port = p;
    return ConfigBuilder<NameSet, HostSet, PortSet>(std::move(b_));
  }
private:
  explicit ConfigBuilder(AppConfig b) : b_(std::move(b)) {}
  AppConfig b_;
  template<class, class, class> friend class ConfigBuilder;
};

template<>
class ConfigBuilder<NameSet, HostSet, PortSet> {
public:
  AppConfig build() && { return std::move(b_); }
private:
  explicit ConfigBuilder(AppConfig b) : b_(std::move(b)) {}
  AppConfig b_;
  template<class, class, class> friend class ConfigBuilder;
```

```
};

inline auto config() { return ConfigBuilder<void, void, void>{}; }

void demo() {
  AppConfig c = config()
    .name("service-a")
    .host("127.0.0.1")
    .port(8080)
    .build();

  // auto bad = config().host("x").build(); // error: cannot host before name; cannot build
  ↪  early
}
```

Design rule. If "missing required fields" is a real production failure mode, treat it as a type-level problem.

14.3.5 Prevent Wrong-Phase Calls: Typestate for Protocols

Many systems have phases: connect → authenticate → use. Encoding these phases in the type system removes the invalid state "used too early."

```
#include <string>
#include <utility>

struct Disconnected {};
struct Connected {};
struct Authenticated {};

template<class State>
class Client;
```

```
template<>
class Client<Disconnected> {
public:
  Client() = default;
  Client<Connected> connect(std::string endpoint) && {
    endpoint_ = std::move(endpoint);
    return Client<Connected>(std::move(endpoint_));
  }
private:
  std::string endpoint_;
};

template<>
class Client<Connected> {
public:
  Client<Authenticated> authenticate(std::string token) && {
    (void)token; // handshake...
    return Client<Authenticated>(std::move(endpoint_));
  }
private:
  explicit Client(std::string endpoint) : endpoint_(std::move(endpoint)) {}
  std::string endpoint_;
  friend class Client<Disconnected>;
};

template<>
class Client<Authenticated> {
public:
  void send(std::string payload) {
    (void)payload; // safe by construction
  }
private:
  explicit Client(std::string endpoint) : endpoint_(std::move(endpoint)) {}
```

```
  std::string endpoint_;
  friend class Client<Connected>;
};

void demo() {
  Client<Disconnected> c;
  auto a = std::move(c).connect("srv").authenticate("token");
  a.send("data");

  // Client<Connected> x; x.send("data"); // error: invalid state not representable
}
```

Design note. Typestate is most appropriate for *non-trivial protocols* with safety/security implications. For small objects, keep the design simpler.

14.3.6 Prevent Null-Related Invalid States: Model Non-Null Explicitly

If a dependency is mandatory, modeling it as a nullable pointer creates an invalid state: "the system can exist with a missing required component."
Prefer references for mandatory dependencies and reserve raw pointers for optional/observing relationships.

Constructor Injection with References

```
struct Logger {
  void info(const char*) {}
};

class Service {
public:
  explicit Service(Logger& log) : log_(log) {}
  void run() { log_.info("running"); }
```

```
private:
  Logger& log_; // cannot be null
};
```

Design effect. The rule "Service requires Logger" is encoded structurally.

14.3.7 Prevent Lifetime Invalid States: RAII and Move-Only Ownership

Ownership mistakes create the most catastrophic invalid states in systems software. The primary tool is RAII with explicit copy/move policies.

```
#include <utility>
#include <cstdio>

class File {
public:
  static File open(const char* path, const char* mode) {
    return File(std::fopen(path, mode));
  }

  ~File() { if (f_) std::fclose(f_); }

  File(const File&) = delete;
  File& operator=(const File&) = delete;

  File(File&& other) noexcept : f_(std::exchange(other.f_, nullptr)) {}
  File& operator=(File&& other) noexcept {
    if (this != &other) {
      if (f_) std::fclose(f_);
      f_ = std::exchange(other.f_, nullptr);
    }
    return *this;
  }
```

```
  std::FILE* get() const noexcept { return f_; }
  explicit operator bool() const noexcept { return f_ != nullptr; }

private:
  explicit File(std::FILE* f) : f_(f) {}
  std::FILE* f_{nullptr};
};
```

Design rule. If the system must guarantee "exactly one release," encode it in the type: delete copy, allow move, release in destructor.

14.3.8 Prevent Invalid Values: Encode Ranges and Formats

When values must satisfy rules (ranges, formats, monotonicity), the invalid state is: "raw value used before validation."

Range-Limited Types

```
#include <optional>
#include <cstdint>

class Percent {
public:
  static std::optional<Percent> make(int x) {
    if (x < 0 || x > 100) return std::nullopt;
    return Percent(static_cast<std::uint8_t>(x));
  }
  std::uint8_t value() const noexcept { return v_; }
private:
  explicit Percent(std::uint8_t v) : v_(v) {}
  std::uint8_t v_{};
```

```
};
```

Design effect. If a function accepts `Percent`, it is impossible to pass 200.

Parsing as a Boundary: "Untrusted" vs "Trusted" Types

A robust boundary design separates:

- **untrusted representations** (strings, raw bytes, loosely-typed JSON),
- **trusted domain types** (validated, invariant-holding values).

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

class Email {
public:
  static std::optional<Email> parse(std::string s) {
    // Minimal structural check (domain may enforce stricter rules):
    auto at = s.find('@');
    if (at == std::string::npos || at == 0 || at + 1 >= s.size()) return std::nullopt;
    return Email(std::move(s));
  }

  std::string_view view() const noexcept { return s_; }

private:
  explicit Email(std::string s) : s_(std::move(s)) {}
  std::string s_;
};
```

Design rule. Never let parsing rules leak across the codebase. Parse once at the boundary; propagate a trusted type internally.

14.3.9 Prevent Incoherent Template Configurations: Constrain Policies with Concepts

Template-based designs can create invalid configurations if any type is accepted. Constrain policy parameters to a finite, explicit set.

```
#include <concepts>

struct ThreadSafe {};
struct NotThreadSafe {};

template<class P>
concept ThreadPolicy =
  std::same_as<P, ThreadSafe> || std::same_as<P, NotThreadSafe>;

template<ThreadPolicy P>
class Queue {
public:
  void push(int) {
    if constexpr (std::same_as<P, ThreadSafe>) {
      // lock-based path
    } else {
      // fast path
    }
  }
};
```

Design effect. The invalid state “Queue configured with an unknown policy” cannot be represented.

14.3.10 Prevent Silent Failure: Make Errors Non-Ignorable

A frequent invalid state is “program continues as if success happened.” Encode failure with a distinct type so it must be acknowledged.

```
#include <variant>
#include <string>

struct Error { std::string msg; };

template<class T>
using Result = std::variant<T, Error>;

Result<int> load_value() {
  return Error{"I/O failure"};
}

int use() {
  auto r = load_value();
  if (auto* v = std::get_if<int>(&r)) return *v;
  // error path is explicit and structurally visible
  return 0;
}
```

Design note. This technique scales: errors become part of the design language rather than comments.

14.3.11 When Invalid States Cannot Be Fully Removed

Some invalid states depend on external reality: network partitions, disk corruption, time drift, concurrency races, hardware faults. Type-level techniques do not eliminate these; they *make assumptions explicit*.

The practical objective is:

- remove invalid states that come from **programmer confusion** (types can fix),
- isolate invalid states that come from **environment uncertainty** (runtime handling must exist),
- ensure boundaries convert **untrusted** inputs into **trusted** domain types.

14.3.12 Engineering Checklist

1. Identify invalid states from incidents, code reviews, and interface misuse.
2. Encode domain separation with strong types (IDs, units, handles).
3. Encode protocol phases with typestate where order matters.
4. Encode ownership with RAII and explicit copy/move rules.
5. Encode mandatory dependencies as references; use pointers only for optional observing links.
6. Introduce validated types for runtime rules (ranges, formats, parsing).
7. Constrain template/policy parameters with concepts.
8. Make failure structurally visible via result types rather than sentinel values.

14.3.13 Summary

Preventing invalid states is a design strategy, not a micro-optimization. Modern C++ provides a coherent set of mechanisms to do it: **strong types** prevent mixing domains, **validated types** prevent using untrusted values, **typestate** prevents wrong-phase calls, **RAII and move-only**

ownership prevent lifetime corruption, **concepts** prevent incoherent configurations and enforce protocols, and **typed error channels** prevent silent failure.
The payoff is architectural: fewer runtime checks, fewer defensive branches, stronger invariants at module boundaries, and a system that is easier to evolve safely over years.

14.4 Expressive APIs with Strong Types

14.4.1 Strong Types as an API Design Instrument

In long-lived systems, most defects are not "algorithmic mistakes" but *interface mistakes*: wrong argument ordering, wrong units, wrong identifiers, ambiguous ownership, and hidden preconditions. A strong-type API design treats the type system as a documentation-and-enforcement layer.
A **strong type** is a type that is not implicitly interchangeable with other types, even when they share the same runtime representation. The goal is to make an API *self-describing* and *compiler-verified*:

- **Intent becomes visible in signatures:** the API reads like a design spec.
- **Misuse becomes ill-formed:** wrong calls fail at compile time.
- **Refactoring becomes safer:** constraints travel with types, not comments.
- **Invariants become local:** fewer runtime checks and fewer defensive branches.

14.4.2 What "Expressive" Means in C++ APIs

An expressive API achieves clarity without sacrificing performance:

1. **Names express domain roles** (e.g., `UserId`, `Port`, `Timeout`).

2. **Types encode rules** (non-null, validated, unit-correct, state-correct).
3. **Conversions are explicit** (no silent narrowing, no implicit unit mixing).
4. **Ownership is visible** (`unique_ptr` vs raw pointer vs reference).
5. **Overloads are constrained** (concepts express what a template accepts).

Strong types are one of the most direct ways to achieve this.

14.4.3 A Minimal Strong Type Wrapper

A robust strong-type wrapper should:

- be **explicitly constructible**,
- expose the underlying representation only deliberately,
- support comparison (and optionally hashing) as needed,
- remain a **zero-cost abstraction** when inlined.

```
#include <compare>
#include <cstdint>
#include <type_traits>

template<class Rep, class Tag>
class Strong {
public:
  using rep_type = Rep;
  using tag_type = Tag;

  constexpr explicit Strong(Rep v) noexcept(std::is_nothrow_move_constructible_v<Rep>)
    : v_(std::move(v)) {}
```

```
  constexpr Rep value() const noexcept(std::is_nothrow_copy_constructible_v<Rep>) { return
  ↪  v_; }

  friend constexpr auto operator<=>(Strong, Strong) = default;

private:
  Rep v_;
};
```

This type blocks accidental mixing because `Tag` distinguishes the domain.

14.4.4 Eliminating "Same-Representation" Bugs

IDs are the classic example: many IDs are `uint64_t` in memory, yet mixing them is a domain error that can silently corrupt systems.

```
#include <cstdint>

struct UserIdTag {};
struct OrderIdTag {};
struct SessionIdTag {};

using UserId    = Strong<std::uint64_t, UserIdTag>;
using OrderId   = Strong<std::uint64_t, OrderIdTag>;
using SessionId = Strong<std::uint64_t, SessionIdTag>;

struct User { UserId id; };
struct Order { OrderId id; };

void load_user(UserId);
void load_order(OrderId);
```

```
void demo(User u, Order o) {
  load_user(u.id);
  load_order(o.id);

  // load_user(o.id); // error: cannot convert OrderId to UserId
}
```

API impact. The signature `load_user(UserId)` is clearer than `load_user(uint64_t)` and cannot be miscalled.

14.4.5 Named Parameters Without Language Support

C++ does not have named parameters, so wrong-argument ordering remains a real hazard. Strong types provide a disciplined alternative.

```
#include <string>

struct HostTag {};
struct ApiKeyTag {};
struct TimeoutMsTag {};

using Host      = Strong<std::string, HostTag>;
using ApiKey    = Strong<std::string, ApiKeyTag>;
using TimeoutMs = Strong<int, TimeoutMsTag>;

class Client {
public:
  Client(Host host, ApiKey key, TimeoutMs timeout)
    : host_(std::move(host)), key_(std::move(key)), timeout_(timeout) {}

private:
  Host host_;
  ApiKey key_;
```

```
  TimeoutMs timeout_;
};

void demo() {
  Client c(Host{"api.service"}, ApiKey{"secret"}, TimeoutMs{5000});

  // Client bad(ApiKey{"secret"}, Host{"api.service"}, TimeoutMs{5000}); // error: argument
  ↪  order mismatch
}
```

Design note. This is often the cleanest way to enforce "named-parameter" semantics in performance-critical code.

14.4.6 Strong Types for Units and Time

Unit confusion is a recurring class of production failures. Time and sizes are especially error-prone because they are frequently represented as integers.

Time as a Type, Not an Integer

Prefer `std::chrono` duration types in APIs, not raw integers.

```
#include <chrono>

class Backoff {
public:
  explicit Backoff(std::chrono::milliseconds base) : base_(base) {}

  std::chrono::milliseconds next(int attempt) const {
    return base_ * attempt;
  }

private:
```

```
  std::chrono::milliseconds base_;
};

void demo() {
  Backoff b(std::chrono::milliseconds{200});
  auto t = b.next(5);
  (void)t;
}
```

API impact. The unit is encoded by the type, preventing ambiguous "is this seconds or milliseconds?" arguments.

Separating Domain Units

When `std::chrono` does not cover your domain units (e.g., meters vs seconds), a strong-type unit wrapper prevents mixing.

```
struct MetersTag {};
struct SecondsTag {};
using Meters  = Strong<double, MetersTag>;
using Seconds = Strong<double, SecondsTag>;

struct SpeedTag {};
using MetersPerSecond = Strong<double, SpeedTag>;

MetersPerSecond speed(Meters d, Seconds t) {
  return MetersPerSecond{ d.value() / t.value() };
}
```

14.4.7 Encoding Validation: "Trusted" Values Only

A strong-type API can also enforce that values were validated. Instead of accepting raw strings and performing scattered checks, accept a validated type.

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

class NonEmptyString {
public:
  static std::optional<NonEmptyString> make(std::string s) {
    if (s.empty()) return std::nullopt;
    return NonEmptyString(std::move(s));
  }
  std::string_view view() const noexcept { return s_; }

private:
  explicit NonEmptyString(std::string s) : s_(std::move(s)) {}
  std::string s_;
};

class UserName {
public:
  static std::optional<UserName> parse(std::string s) {
    auto v = NonEmptyString::make(std::move(s));
    if (!v) return std::nullopt;
    return UserName(std::move(*v));
  }

  std::string_view view() const noexcept { return name_.view(); }

private:
  explicit UserName(NonEmptyString n) : name_(std::move(n)) {}
  NonEmptyString name_;
};

void create_user(UserName);
```

```
void boundary(std::string raw) {
  if (auto u = UserName::parse(std::move(raw))) {
    create_user(std::move(*u)); // inside system: trusted-only
  }
}
```

API impact. The function `create_user(UserName)` is an architectural statement: "this layer never receives unvalidated names."

14.4.8 Ownership and Nullability: Strong Types in Resource APIs

Expressive APIs must communicate ownership and lifetime:

- use **references** for mandatory dependencies,
- use **raw pointers** for optional observing references,
- use **smart pointers** for ownership transfer or shared ownership.

Mandatory Dependency

```
struct Logger {
  void info(const char*) {}
};

class Service {
public:
  explicit Service(Logger& log) : log_(log) {}
  void run() { log_.info("run"); }
private:
  Logger& log_; // cannot be null
};
```

Ownership Transfer

```
#include <memory>

struct Task { void exec() {} };

class Scheduler {
public:
  void submit(std::unique_ptr<Task> t) {
    // Scheduler now owns t
    if (t) t->exec();
  }
};
```

API impact. Ownership rules are not comments. They are part of the type system.

14.4.9 Constrained Templates: Expressive Generic APIs

Generic APIs become expressive when they state their requirements in the type system. C++20 concepts allow template interfaces to encode protocol constraints directly.

```
#include <concepts>
#include <string_view>

template<class T>
concept StringLike = std::convertible_to<T, std::string_view>;

void log_line(StringLike auto s) {
  // takes std::string, std::string_view, const char*, etc.
  (void)s;
}
```

Design effect. Instead of "works for anything that happens to compile," the API communicates the intended input family.

14.4.10 Interoperation: Exposing the Underlying Representation Safely

Strong types should not make integration painful. Two techniques preserve expressiveness while keeping safety:

Explicit Unwrap at Boundaries

```cpp
#include <cstdint>

struct FdTag {};
using FileDescriptor: Strong<int, FdTag>; // conceptual illustration

// In practice, keep unwrap explicit:
int native_fd(/*FileDescriptor*/ auto fd) {
  return fd.value();
}
```

Controlled Conversion Functions

Prefer named conversion functions over implicit conversion operators. Implicit conversions reintroduce ambiguity and can break overload resolution.

14.4.11 Design Guidelines for Strong-Type APIs

1. **Start at boundaries:** public interfaces, service APIs, persistence boundaries, network boundaries.

2. **Strong-type anything that can be swapped:** IDs, units, durations, sizes, ports, paths, endpoints.

3. **Keep construction explicit:** avoid implicit conversion to/from the underlying representation.

4. **Separate untrusted and trusted:** parse/validate at input boundaries, then use strong domain types internally.

5. **Use standard vocabulary types:** `std::chrono` for time, `std::span` for views, `std::string_view` for non-owning text.

6. **Encode ownership in types:** references for mandatory dependencies, smart pointers for ownership.

7. **Avoid "tag explosion":** introduce strong types where the risk/impact is high and stable.

14.4.12 Summary

Expressive APIs are a key mechanism for making systems long-lived: they reduce accidental misuse, improve readability, and localize invariants. Strong types are one of the most effective tools for achieving expressiveness in modern C++, because they transform domain rules into compiler-verified constraints.
A strong-type API does not merely accept values; it accepts *meaning*.

Templates as Architectural Tools

15.1 Template Abstraction Layers

15.1.1 Templates as Architecture, Not Syntax

In mature C++ systems, templates are not merely a way to write "generic containers." They are an architectural mechanism that lets you build *abstraction layers* whose rules are enforced at compile time.

A **template abstraction layer** is a design boundary where:

- variability is expressed as **type parameters** and **policy parameters**,
- compatibility is expressed as **constraints** (concepts/traits),
- behavior selection is performed at compile time (overload resolution, partial ordering, specialization),
- runtime cost is minimized (inlining, monomorphization, constant folding),
- the public API can remain stable while implementation details evolve behind generic interfaces.

The architectural question is not "should we use templates?" It is: **where should genericity live, and where should it stop?**

15.1.2 The Layering Model

A robust architecture typically uses three conceptual layers:

1. **Core layer (domain invariants):** stable types, strong invariants, minimal dependencies.
2. **Generic infrastructure layer (templates):** algorithms, containers, adaptors, policies.
3. **Boundary layer (runtime polymorphism / type erasure / dynamic loading):** plugin points, ABI-stable interfaces, late binding.

Templates dominate the infrastructure layer because they provide:

- **static polymorphism** (compile-time dispatch),
- **expressive constraints** (what is accepted and why),
- **zero-cost customization** (policies, traits, tags),
- **composability** (adapters on adapters).

But templates should often *end* at the boundary layer, to prevent:

- ABI instability across shared libraries,
- excessive recompilation due to template instantiation churn,
- exponential type growth in public interfaces,
- unreadable diagnostics when errors cross module boundaries.

15.1.3 Layer 1: The Minimal Concept Layer

Any template layer must start by stating: *what do we require from types?* In C++20+, the cleanest way is to define concepts that represent architectural roles.

```
#include <concepts>
#include <string_view>

template<class T>
concept LogSink =
  requires(T s, std::string_view msg) {
    { s.write(msg) } -> std::same_as<void>;
  };

template<class T>
concept Clock =
  requires(T c) {
    typename T::time_point;
    { c.now() } -> std::same_as<typename T::time_point>;
  };
```

Architectural effect. Concepts turn "conventions" into **compile-time interface contracts**. They are the foundation of template abstraction layers because they make roles explicit.

15.1.4 Layer 2: Policy-Based Design (Compile-Time Configuration)

A common use of templates as architecture is policy-based design: encode variation in **policy types** so that configuration is explicit, reviewable, and testable.

A Policy-Driven Component

```
#include <concepts>
#include <string_view>
```

```
struct NoLock {
  struct guard { explicit guard(NoLock&) {} };
};

struct MutexLock {
  struct guard { explicit guard(MutexLock&) {/* lock */} ~guard() {/* unlock */} };
};

template<class P>
concept LockPolicy =
  requires(P p) {
    typename P::guard;
    { typename P::guard{p} };
  };

template<LockPolicy Lock, class Sink>
  requires requires(Sink s, std::string_view m) { s.write(m); }
class Logger {
public:
  explicit Logger(Sink sink) : sink_(std::move(sink)) {}

  void log(std::string_view msg) {
    typename Lock::guard g(lock_);
    sink_.write(msg);
  }

private:
  Lock lock_{};
  Sink sink_;
};
```

Architectural effect. The same component can serve:

- embedded/low-latency builds (`NoLock`),
- server builds (`MutexLock`),

without runtime branching or configuration flags that drift out of sync.

Constrain the Policy Space

Unconstrained policy parameters create a "configuration chaos" layer. Always constrain policy parameters so invalid configurations are unrepresentable.

```
#include <concepts>

struct JsonFormat { static constexpr int id = 1; };
struct TextFormat { static constexpr int id = 2; };

template<class F>
concept FormatPolicy =
  std::same_as<F, JsonFormat> || std::same_as<F, TextFormat>;

template<FormatPolicy F>
struct Encoder {
  static constexpr int format_id = F::id;
};
```

15.1.5 Layer 3: Traits and Customization Points (Extensibility Without Inheritance)

A template layer must be extensible, but extensibility should not require modifying the generic code. Two dominant patterns:

- **traits**: type-based metadata and behavior selection,

- **customization points**: function-based extension (often via ADL) with a controlled fallback.

Traits as Architectural Metadata

```
#include <type_traits>
#include <cstddef>

template<class T>
struct storage_traits {
  static constexpr bool contiguous = false;
  static constexpr std::size_t alignment = alignof(T);
};

template<class T>
inline constexpr bool is_contiguous_v = storage_traits<T>::contiguous;

template<class T>
constexpr bool use_fast_path() {
  return std::is_trivially_copyable_v<T> && is_contiguous_v<T>;
}
```

Traits let you build a generic layer that can adapt to new types without changing core algorithms.

Controlled Customization Point with Fallback

```
#include <concepts>
#include <string_view>

namespace detail {
  void write_default(auto& sink, std::string_view msg) { sink.write(msg); }
}
```

```
template<class Sink>
concept HasWrite =
  requires(Sink s, std::string_view msg) {
    { write(s, msg) } -> std::same_as<void>; // ADL customization point
  };

struct write_cpo {
  template<class Sink>
  void operator()(Sink& sink, std::string_view msg) const {
    if constexpr (HasWrite<Sink>) {
      write(sink, msg);            // user-provided overload via ADL
    } else {
      detail::write_default(sink, msg); // fallback
    }
  }
};

inline constexpr write_cpo write_to{};
```

Architectural effect. The generic layer remains closed for modification but open for extension: new sinks can customize `write` without inheriting from a base class.

15.1.6 Layer 4: Static Polymorphism (CRTP) for Architectural Roles

When you need polymorphic behavior but want to avoid virtual dispatch and heap allocation, static polymorphism (often CRTP) becomes a layer-building tool.

```
#include <string_view>

template<class Derived>
class Processor {
public:
  void process(std::string_view input) {
```

```
    // shared pre/post logic
    static_cast<Derived*>(this)->process_impl(input);
  }
};

class Tokenizer : public Processor<Tokenizer> {
public:
  void process_impl(std::string_view input) {
    (void)input;
    // tokenize...
  }
};

class Parser : public Processor<Parser> {
public:
  void process_impl(std::string_view input) {
    (void)input;
    // parse...
  }
};
```

Architectural effect. A common framework layer (shared logic) can exist without a runtime base class. This is useful inside a module where ABI boundaries are not a concern.

15.1.7 Layer 5: Algorithm Layers Over Views (Ranges and `std::span`)

A critical architectural move is to separate:

- **data ownership** (containers, storage),
- **data access** (views: `std::span`, iterator pairs, range views),
- **algorithms** (templates constrained by iterator/range concepts).

This reduces coupling and improves testability.

Span-Based API Layer

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

template<class T>
std::size_t checksum(std::span<const T> data) {
  std::size_t h = 14695981039346656037ull;
  for (const auto& x : data) {
    h ^= static_cast<std::size_t>(x);
    h *= 1099511628211ull;
  }
  return h;
}
```

Architectural effect. The algorithm layer accepts *any* contiguous source without taking ownership: vector, array, string, memory-mapped region (wrapped), and more.

15.1.8 Where Template Layers Must Stop: Type Erasure Boundaries

Template layering is extremely powerful inside a build unit or a module, but crossing dynamic boundaries (plugins, shared libraries, stable ABI) often requires **type erasure**.
The architectural rule is:

> Use templates to *implement* variability; use type erasure to *export* stable runtime interfaces.

A Minimal Type-Erased Interface

```
#include <memory>
```

```
#include <string_view>
#include <utility>

class AnySink {
public:
  template<class Sink>
  AnySink(Sink s)
    : self_(std::make_unique<Model<Sink>>(std::move(s))) {}

  void write(std::string_view msg) { self_->write(msg); }

private:
  struct Concept {
    virtual ~Concept() = default;
    virtual void write(std::string_view) = 0;
  };

  template<class Sink>
  struct Model final : Concept {
    explicit Model(Sink s) : sink_(std::move(s)) {}
    void write(std::string_view msg) override { sink_.write(msg); }
    Sink sink_;
  };

  std::unique_ptr<Concept> self_;
};
```

Architectural effect. Internally you can have a rich template ecosystem of sinks and policies, but externally you expose a stable, non-templated interface.

15.1.9 Composing Layers: A Worked Architectural Example

A realistic design composes multiple template layers:

1. **Concept layer:** defines what "sink" means.
2. **Policy layer:** lock policy, formatting policy, buffering policy.
3. **Customization layer:** ADL customization point `write`.
4. **Boundary layer:** type erasure `AnySink` for ABI stability.

This yields:

- high performance internally (compile-time dispatch),
- stable runtime interfaces externally,
- extensibility without modifying core templates,
- clearer testing surfaces (test via concepts and small policies).

15.1.10 Engineering Pitfalls and How to Avoid Them

Template abstraction layers can also harm architecture if misused:

- **Template leakage into public APIs:** exporting templates across module boundaries increases rebuild cost and complicates ABI evolution. Prefer non-templated facades at boundaries.
- **Unbounded instantiation growth:** too many combinations of policies/types can explode compile times and binary size. Constrain policy spaces and keep combinations finite.

- **Overly clever metaprogramming:** if the design requires deep template machinery to understand basic behavior, the layer becomes fragile. Prefer simple concepts and straightforward policies.
- **Poor diagnostics:** constraints written as "boolean soup" create unreadable errors. Prefer named concepts and layered requirements.
- **Accidental semantic constraints:** encoding business rules in templates can reduce evolvability. Encode stable structural rules in types; keep volatile rules in runtime configuration.

15.1.11 Design Checklist: Building Template Abstraction Layers

1. Define roles as **concepts** with minimal required operations.
2. Keep the generic layer closed for modification: enable extension via **traits** and **customization points**.
3. Use **policy types** for compile-time configuration; constrain the policy space.
4. Separate **ownership** from **views** from **algorithms** (span/ranges).
5. Stop templates at dynamic boundaries: provide **type-erased** or non-templated facades.
6. Measure and manage: compile time, binary size, and instantiation counts are architectural concerns.

15.1.12 Summary

Template abstraction layers are a core architectural tool in modern C++. They enable:

- **static enforcement** of interface roles via concepts,

- **compile-time configuration** via policies,
- **extensibility** via traits and customization points,
- **performance** via static polymorphism and inlining,
- **clean boundaries** via type erasure where ABI stability matters.

Used with discipline, templates become a design language for architecture: they define what can vary, what must hold, and where complexity is allowed to live.

15.2 Generic Programming vs Metaprogramming

15.2.1 Why This Distinction Matters Architecturally

In modern C++ design, the word "templates" is overloaded: sometimes it means writing algorithms that work for many types, and sometimes it means computing *about types* at compile time. These are related but not identical disciplines.

Generic programming is primarily about *abstraction over values and operations*: writing code once so it works for a family of types that satisfy an interface. It is the foundation of the standard library algorithms, iterators, ranges, and containers.

Metaprogramming is primarily about *computation over types and compile-time values*: using the compiler as an evaluation engine to build types, select implementations, and enforce rules before runtime.

Architecturally, confusing them leads to two recurring failures:

- treating every generic design problem as a type-computation problem (overengineering),
- treating every compile-time constraint as a runtime concern (underengineering).

The right mental model:

Generic programming defines *what your program does* for many types.
Metaprogramming defines *how your program is formed* for many types.

15.2.2 Core Definitions

Generic Programming

Generic programming is the discipline of designing algorithms and data structures in terms of **requirements** (interfaces) rather than concrete types.
Key properties:

- **Type-parameterized behavior:** the same code works for many types.
- **Semantic requirements:** correct behavior depends on meaning (ordering, equality, iterator validity).
- **Composable abstractions:** algorithms compose through common concepts (iterators/ranges).
- **Primary output:** reusable runtime behavior.

Metaprogramming

Metaprogramming uses compile-time evaluation to:

- create or transform types,
- choose implementations,
- derive properties,
- enforce structural constraints,
- generate or configure code paths.

Key properties:

- **Type-level computation:** transformation of type information.
- **Compile-time branching:** selection using `if constexpr` or specialization.
- **Reflection-like patterns:** limited introspection via traits and detection idioms.
- **Primary output:** better-formed, better-optimized, or restricted programs.

15.2.3 The Shared Mechanism: Templates, Different Goals

Both disciplines often use templates, but their architectural intent differs:

- Generic programming uses templates for **parametric polymorphism**.
- Metaprogramming uses templates as a **compile-time computation system**.

A practical way to tell them apart:

> If the template is mostly about accepting different types while executing the same algorithm, it is generic programming. If the template is mostly about computing types or selecting implementations before runtime, it is metaprogramming.

15.2.4 Generic Programming: Writing Algorithms Over Requirements

Concept-Constrained Generic Algorithms (C++20+)

Modern C++ generic programming is increasingly expressed using concepts, where requirements are explicit and diagnostics are improved.

```
#include <concepts>
#include <iterator>
```

```
template<std::input_iterator It, class T>
requires std::equality_comparable_with<std::iter_value_t<It>, T>
bool contains(It first, It last, const T& value) {
  for (; first != last; ++first) {
    if (*first == value) return true;
  }
  return false;
}
```

This is generic programming because:

- the algorithm is primarily about runtime behavior (searching),
- the template expresses the type family it can operate on,
- it depends on semantic requirements (equality comparable).

Value-Category and Range-Friendly Design

Generic code should avoid unnecessary constraints and avoid taking ownership. It commonly uses views such as `std::span` or range interfaces.

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

template<class T>
std::size_t count_nonzero(std::span<const T> xs) {
  std::size_t n = 0;
  for (const auto& x : xs) {
    if (x != T{}) ++n;
  }
  return n;
}
```

Architectural effect. The algorithm is reusable across ownership domains (vector, array, mapped memory), and the type system states only what is necessary.

15.2.5 Metaprogramming: Shaping Programs at Compile Time

Compile-Time Selection with `if constexpr`

`if constexpr` is metaprogramming in a pragmatic, maintainable form: it selects code paths based on compile-time properties.

```
#include <type_traits>
#include <cstring>
#include <cstddef>

template<class T>
void move_block(T* dst, const T* src, std::size_t n) {
  if constexpr (std::is_trivially_copyable_v<T>) {
    std::memmove(dst, src, n * sizeof(T));
  } else {
    for (std::size_t i = 0; i < n; ++i) dst[i] = src[i];
  }
}
```

This is metaprogramming because:

- a type trait is evaluated at compile time,
- the program structure changes (two distinct implementations),
- the result is optimized without runtime branching.

Type Construction and Transformation

Metaprogramming often builds types from other types.

```
#include <type_traits>

template<class T>
using remove_cvref_t = std::remove_cv_t<std::remove_reference_t<T>>;

template<class T>
constexpr bool is_plain_value_v = std::is_same_v<T, remove_cvref_t<T>>;
```

Specialization as Compile-Time Decision Tables

Partial specialization can encode compile-time decision tables. This can be powerful but must be constrained to avoid complexity.

```
#include <type_traits>

template<class T, bool = std::is_integral_v<T>>
struct parse_impl;

template<class T>
struct parse_impl<T, true> {
  static T parse(const char*) { return {}; } // integral parse path
};

template<class T>
struct parse_impl<T, false> {
  static T parse(const char*) { return {}; } // non-integral path
};

template<class T>
T parse(const char* s) {
  return parse_impl<T>::parse(s);
}
```

Architectural effect. The dispatch is compile-time, not runtime, but specialization-based systems can become fragile if used as a general programming language.

15.2.6 Where They Meet: "Metaprogramming in Service of Generic Programming"

In well-designed libraries, metaprogramming typically exists to *support* generic programming. Common examples:

- **Detection and constraints:** detect whether a type supports an operation and constrain accordingly.
- **Optimization selection:** choose fast paths based on type traits.
- **Adapter composition:** shape types that adapt one interface to another.
- **Policy wiring:** build concrete types from configuration policies.

Detection Used to Constrain a Generic API

```
#include <concepts>
#include <string_view>

template<class T>
concept HasToString =
  requires(const T& x) {
    { x.to_string() } -> std::convertible_to<std::string_view>;
  };

template<class T>
std::string_view stringify(const T& x) {
  if constexpr (HasToString<T>) {
    return x.to_string();
```

```
  } else {
    return "<no to_string>";
  }
}
```

This combines:

- metaprogramming (`if constexpr` + concept predicate),
- to enable a generic interface that works across diverse types.

15.2.7 Architectural Decision Guidance

Prefer Generic Programming When

- you are implementing reusable algorithms and data-structure behavior,
- the behavior is stable and depends on a small set of requirements,
- you want composability with standard conventions (iterators/ranges),
- runtime behavior is the main deliverable.

Prefer Metaprogramming When

- you must enforce structural constraints at compile time,
- you need compile-time selection of implementations for safety or performance,
- you are encoding policies and configuration into types,
- you are building a library framework where correctness depends on compile-time properties.

Avoid Heavy Metaprogramming When

- the compile-time logic becomes harder to understand than the runtime behavior,
- diagnostics become a recurring burden for users,
- the compile-time computation leaks into public APIs unnecessarily,
- compilation time and binary size become unacceptable.

15.2.8 Cost Model: Performance, Build Times, and Complexity

Architecturally, templates have a real cost model:

- **Generic programming cost:** multiple instantiations can increase compile time and code size, but often yields high runtime performance due to inlining and specialization.
- **Metaprogramming cost:** compile-time computation can dominate build time, and complex specialization graphs can harm maintainability.

A long-lived system should treat:

- compilation time,
- diagnostic quality,
- binary size,

as first-class architectural constraints, not afterthoughts.

15.2.9 A Design Checklist for Teams

1. Write the generic algorithm first with minimal constraints.
2. Add concepts to express the intended requirements.
3. Introduce metaprogramming only to:
    - enforce invariants,
    - select safe/fast paths,
    - adapt interfaces in a controlled manner.
4. Prefer `if constexpr` and named concepts over deep specialization networks.
5. Keep templates out of ABI-stable boundaries; use type erasure or non-templated facades when needed.
6. Continuously measure compile time and instantiation growth.

15.2.10 Summary

Generic programming and metaprogramming share mechanisms but differ in intent.

- Generic programming is about writing reusable runtime behavior over requirements.
- Metaprogramming is about compile-time computation that shapes and restricts programs.

In architecture, metaprogramming should usually serve generic programming: it should make generic code safer, clearer, and faster, not turn the codebase into a compile-time puzzle.
A disciplined design uses generic programming as the default, and introduces metaprogramming strategically where compile-time enforcement or selection delivers high leverage in safety, correctness, and performance.

15.3 Concepts as Design Documentation

15.3.1 Concepts Are "Executable Documentation"

Traditional documentation describes what an API *expects*. Concepts describe what an API *accepts*, and the compiler enforces it.
In long-lived C++ systems, the most expensive design failures often come from **implicit** requirements: "this function expects an iterator," "this type must be sortable," "this value must be hashable," or "this policy must provide a lock guard."
When these expectations are written only in prose:

- they are easy to ignore,
- they drift from the implementation,
- they yield poor diagnostics when violated,
- they create brittle extension points that fail late.

C++20 concepts turn such requirements into a **first-class part of the interface**. For architecture, this means concepts become a form of **design documentation that cannot lie**.

15.3.2 From Accidental Genericity to Intended Genericity

A template without constraints is a promise the author did not explicitly define. It may accept surprising types, reject reasonable ones, or fail deep in the body with unreadable diagnostics.
Concepts enable **intended genericity**:

- define the role a type must play,
- constrain the surface area of accepted types,
- make error messages focus on *why the type does not satisfy the role*.

15.3.3 The Vocabulary Layer: Naming Architectural Roles

The first step in using concepts as documentation is to define a vocabulary for architectural roles in your system.

A Role Concept

```
#include <concepts>
#include <string_view>

template<class Sink>
concept LogSink =
  requires(Sink s, std::string_view msg) {
    { s.write(msg) } -> std::same_as<void>;
  };
```

Documentation value. This concept states, precisely and compactly:

> A `LogSink` is any type that supports `write(string_view)` and returns `void`.

It documents:

- the required operation,
- the required parameter type,
- the required return type.

Concepts as "Glossary Entries"

In architecture, naming matters. A concept is a glossary entry that can be imported into every generic interface consistently:

- `LogSink`, `Clock`, `Serializer`, `Transport`, `Repository`

- each communicates a stable role in the architecture
- each prevents drift: if the role changes, the concept changes and breaks mismatched code immediately

15.3.4 Concepts Clarify API Boundaries

Concepts should appear at the boundary of templates where users interact with your library or module.

Constrained Function Template

```
#include <concepts>
#include <string_view>

template<class Sink>
concept LogSink =
  requires(Sink s, std::string_view msg) {
    { s.write(msg) } -> std::same_as<void>;
  };

template<LogSink Sink>
void log_line(Sink& sink, std::string_view msg) {
  sink.write(msg);
}
```

Documentation value. The signature documents the requirement directly: `log_line` does not accept "any type"; it accepts `LogSink`.

Constrained Class Template

```
#include <string_view>
```

```
template<LogSink Sink>
class Logger {
public:
  explicit Logger(Sink sink) : sink_(std::move(sink)) {}

  void info(std::string_view msg) { sink_.write(msg); }

private:
  Sink sink_;
};
```

Documentation value. The architecture becomes explicit: `Logger` is parameterized by a sink role, not by an arbitrary type.

15.3.5 Concepts Improve Diagnostics and Reduce Support Burden

A major practical advantage of concepts is that they move failures to the call site and explain them in terms of missing requirements.
Compare two failure modes:

- **unconstrained template:** fails deep inside the implementation with a long substitution trace,
- **constrained template:** fails at the boundary because the constraint is not satisfied.

This has architectural impact:

- fewer bug reports caused by template instantiation noise,
- clearer extension requirements for third-party teams,
- more predictable integration and upgrade paths.

15.3.6 Layering Requirements: Minimal, Composable Concepts

Good concepts are rarely monolithic. They are typically layered from minimal requirements upward.

Layered Concepts

```
#include <concepts>

template<class T>
concept Semiregular =
  std::copyable<T> && std::default_initializable<T>;

template<class T>
concept Comparable =
  requires(const T& a, const T& b) {
    { a < b } -> std::convertible_to<bool>;
  };

template<class T>
concept OrderedValue =
  Semiregular<T> && Comparable<T>;
```

Documentation value. If `OrderedValue` fails, the compiler can report whether `Semiregular` or `Comparable` failed, guiding the user to the exact missing capability.

15.3.7 Documenting Performance Intent with Concepts

Concepts can document not only correctness requirements, but also performance assumptions.

Iterator Category as Performance Documentation

```
#include <concepts>
```

```
#include <iterator>

template<std::random_access_iterator It>
It middle(It first, It last) {
  return first + (last - first) / 2;
}
```

Documentation value. The algorithm documents:

> This function requires random access iterators (because it performs arithmetic).

Without the concept, the function might compile for weaker iterators but fail in confusing ways, or force slower patterns.

15.3.8 Documenting Safety and Exception Guarantees

Concepts can document structural safety rules and enable safer overloads.

Noexcept as a Constraint

```
#include <concepts>
#include <utility>

template<class F, class... Args>
concept NothrowInvocable =
  requires(F f, Args... args) {
    { std::invoke(f, args...) } noexcept;
  };

template<class F>
requires NothrowInvocable<F&>
void commit_with_no_rollback(F&& action) {
  std::forward<F>(action)();
}
```

Documentation value. This documents a design rule:

> We only commit without rollback if the action is nothrow.

15.3.9 Concepts as Architectural Guards on Policy Parameters

Policy-based templates are an architectural tool, but they are dangerous without constraints. Concepts document the policy requirements and prevent invalid policies from entering the system.

```
#include <concepts>

template<class P>
concept LockPolicy =
  requires(P p) {
    typename P::guard;
    { typename P::guard{p} };
  };

struct NoLock {
  struct guard { explicit guard(NoLock&) {} };
};

template<LockPolicy L>
class ProtectedCounter {
public:
  void inc() {
    typename L::guard g(lock_);
    ++value_;
  }
private:
  L lock_{};
  int value_{0};
};
```

Documentation value. The concept defines what "lock policy" means in your architecture. A policy that fails to provide `guard` will not compile, and the error points to the missing requirement.

15.3.10 Avoiding Bad Concepts: Common Documentation Failures

Not all concepts document well. The following mistakes reduce clarity:

- **Boolean soup:** complex unnamed requirements in a `requires` clause. Prefer named concepts with semantic names.
- **Overly strict requirements:** requiring more than the algorithm needs harms composability. Keep concepts minimal.
- **Accidental semantics:** concepts can check syntax and types, but not full business meaning. Document semantic expectations in prose, but enforce structure in code.
- **Leaking implementation details:** concepts that expose private helper types couple users to internals. Keep role concepts stable.

15.3.11 Practical Guidelines for Using Concepts as Documentation

1. **Treat concepts as vocabulary:** define role concepts that match architectural language.
2. **Constrain public templates:** constraints belong at module boundaries.
3. **Keep concepts minimal:** require only what is necessary for correctness and performance intent.
4. **Layer concepts:** build higher-level roles from smaller, reusable concepts.
5. **Name concepts semantically:** `Sortable`, `Hashable`, `LogSink` communicate purpose.

6. **Use concepts to guide overload sets:** prefer more constrained overloads for better behavior and diagnostics.

7. **Separate structure from meaning:** concepts enforce structural contracts; prose covers business semantics.

15.3.12 Summary

Concepts turn template interfaces into **design documentation that compiles**. They improve clarity, diagnostics, and maintainability by making requirements explicit, layered, and reusable across a system.
In architectural terms, concepts provide:

- a shared vocabulary for roles,
- enforceable contracts at generic boundaries,
- a mechanism to encode performance and safety intent,
- a disciplined way to keep template abstraction layers coherent over time.

A long-lived C++ codebase should treat concepts not as an optional feature, but as a primary tool for documenting and enforcing design intent.

15.4 Reducing Template Complexity

15.4.1 Template Complexity Is an Architectural Cost

Templates are one of C++'s strongest architectural tools, but they impose real costs:

- longer compile times and heavier incremental rebuilds,

- larger binaries from many instantiations,
- difficult diagnostics that slow integration and onboarding,
- fragile designs that resist refactoring,
- accidental coupling across modules through template leakage.

Reducing template complexity is not about avoiding templates; it is about **shaping template usage** so that genericity remains a net gain for long-lived systems.
A disciplined approach is guided by three questions:

1. **Where is genericity needed?** (internal layers vs public boundaries)
2. **How many instantiations will exist?** (type/policy combinatorics)
3. **How readable are constraints and diagnostics?** (concepts and layering)

15.4.2 Common Sources of Template Complexity

Template complexity typically comes from a small set of patterns:

- **Unconstrained templates:** errors appear deep in bodies; requirements are implicit.
- **Over-templatized public APIs:** user code becomes tied to your implementation choices.
- **Combinatorial policy explosion:** many policy parameters multiply instantiations.
- **Deep specialization graphs:** partial specializations used as a general programming language.
- **Meta-heavy idioms:** detection idioms and trait cascades replacing simple constraints.
- **Header-only everything:** every change triggers widespread rebuilds.

Reducing complexity means addressing these sources systematically.

15.4.3 Prefer Concepts Over SFINAE and Ad-Hoc Traits

SFINAE works, but it can obscure intent and amplify diagnostics. C++20 concepts provide a direct, readable constraint language.

Replace SFINAE Enablement with Concepts

```
#include <concepts>
#include <string_view>

template<class T>
concept StringLike = std::convertible_to<T, std::string_view>;

void log_line(StringLike auto s) {
  (void)s;
}
```

Complexity reduction.

- constraints move to the signature,
- failure occurs at the boundary,
- requirements become readable documentation.

Name Concepts to Improve Diagnostics

```
#include <concepts>
#include <cstddef>

template<class T>
concept Hashable =
  requires(const T& x) {
    { std::hash<T>{}(x) } -> std::convertible_to<std::size_t>;
```

```
  };

template<Hashable K, class V>
class HashMap { /* ... */ };
```

A named concept is easier to understand than a large boolean expression in a `requires` clause.

15.4.4 Make Templates Shallow: Layer and Localize Genericity

A powerful rule:

> Keep templates near the algorithmic core, but keep module boundaries simple.

Expose Non-Templated Facades

Instead of exporting deep templates, export a stable facade that uses templates internally.

```
#include <memory>
#include <string_view>
#include <utility>

class AnySink {
public:
  template<class Sink>
  AnySink(Sink s)
    : self_(std::make_unique<Model<Sink>>(std::move(s))) {}

  void write(std::string_view msg) { self_->write(msg); }

private:
  struct Concept {
    virtual ~Concept() = default;
    virtual void write(std::string_view) = 0;
```

```
  };

  template<class Sink>
  struct Model final : Concept {
    explicit Model(Sink s) : sink_(std::move(s)) {}
    void write(std::string_view msg) override { sink_.write(msg); }
    Sink sink_;
  };

  std::unique_ptr<Concept> self_;
};
```

Complexity reduction. The template explosion stays inside the module. Users see one stable class with predictable build impact and ABI behavior.

15.4.5 Reduce Instantiation Count: Control the Combinatorics

Template complexity often becomes a *counting problem*: how many distinct types will instantiate a template?

Constrain Policy Spaces

Instead of allowing any type as a policy, restrict the set:

```
#include <concepts>

struct ThreadSafe {};
struct NotThreadSafe {};

template<class P>
concept ThreadPolicy =
  std::same_as<P, ThreadSafe> || std::same_as<P, NotThreadSafe>;
```

```
template<ThreadPolicy P>
class Queue { /* ... */ };
```

Complexity reduction. Policy types become finite and auditable. This prevents accidental instantiation blowups from unexpected policy types.

Prefer Runtime Configuration When Appropriate

Not every variability dimension should be compile-time. If a choice is:

- rarely on a hot path,
- likely to change frequently,
- high in combinatorial impact,

then a runtime branch or configuration object may be a better architectural trade.

Use `if constexpr` to Consolidate Variants

Instead of multiple specializations, prefer a single template with compile-time branching.

```
#include <type_traits>
#include <cstring>
#include <cstddef>

template<class T>
void move_block(T* dst, const T* src, std::size_t n) {
  if constexpr (std::is_trivially_copyable_v<T>) {
    std::memmove(dst, src, n * sizeof(T));
  } else {
    for (std::size_t i = 0; i < n; ++i) dst[i] = src[i];
  }
}
```

Complexity reduction. One template instantiation, one place to read, fewer specialization edges, and usually better diagnostics.

15.4.6 Prefer "Normal C++" Over Template Cleverness

A recurring rule for long-lived codebases:

> If a non-template solution is clear and fast enough, prefer it.

Techniques that reduce template cleverness:

- use `std::span` and range-based APIs instead of type-heavy container templates at boundaries,
- prefer overloads over meta-logic when there are only a few cases,
- prefer small policy classes over complex type-trait systems.

15.4.7 Avoid Specialization Networks as Control Flow

Partial specialization is powerful, but large specialization networks become a fragile, implicit control-flow graph.

Problem Pattern

```
#include <type_traits>

template<class T, bool = std::is_integral_v<T>>
struct impl;

template<class T>
struct impl<T, true> { static void run() {/*...*/} };
```

```
template<class T>
struct impl<T, false> { static void run() {/*...*/} };

template<class T>
void f() { impl<T>::run(); }
```

This is acceptable for simple binary branching. But as the number of dimensions grows, specialization graphs become hard to reason about.

Preferred Pattern: Single Template + `if constexpr`

```
#include <type_traits>

template<class T>
void f() {
  if constexpr (std::is_integral_v<T>) {
    // integral path
  } else {
    // non-integral path
  }
}
```

Complexity reduction. The decision logic is explicit and localized.

15.4.8 Keep Error Messages Human: Design for Diagnostics

Diagnostics are an architectural concern because they shape developer time.

Write Minimal Concepts and Layer Them

```
#include <concepts>

template<class T>
```

```
concept Semiregular = std::copyable<T> && std::default_initializable<T>;

template<class T>
concept Comparable =
  requires(const T& a, const T& b) {
    { a < b } -> std::convertible_to<bool>;
  };

template<class T>
concept OrderedValue = Semiregular<T> && Comparable<T>;
```

When `OrderedValue` fails, the user sees which smaller concept failed.

Avoid "Catch-All" Templates

Catch-all templates accept too much and fail late. Prefer constrained overload sets.

```
#include <concepts>
#include <string_view>

template<class T>
concept StringLike = std::convertible_to<T, std::string_view>;

void print(StringLike auto s) { (void)s; }

template<class T>
void print(const T& x) { (void)x; } // fallback
```

Complexity reduction. Constrained overloads keep intent clear and failures localized.

15.4.9 Compilation Firebreaks: Separate Interface and Implementation

A major source of complexity is header churn. Even in template-heavy systems, you can create firebreaks:

- keep stable, constrained template interfaces in headers,
- move non-templated implementation details to `.cpp`,
- limit includes by forward declaring and using pimpl/type erasure at boundaries,
- reduce transitive template instantiations by depending on views (`span`) rather than containers.

Explicit Instantiation (When Appropriate)

For a small set of known types, explicit instantiation can reduce rebuild costs and binary duplication.

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

// header:
template<class T>
std::size_t checksum(const std::vector<T>&);

// source:
template<class T>
std::size_t checksum(const std::vector<T>& v) { /*...*/ return v.size(); }

// explicit instantiations:
template std::size_t checksum<int>(const std::vector<int>&);
template std::size_t checksum<double>(const std::vector<double>&);
```

Design note. This is most effective for library boundaries with a known set of instantiations.

15.4.10 A Practical Strategy for Teams

Reducing template complexity is best treated as a continuous discipline:

1. **Constrain everything public:** concepts in signatures, no unconstrained extension points.
2. **Keep template depth shallow:** avoid nesting generic abstractions without strong reason.
3. **Control instantiation growth:** constrain policy sets and avoid excessive combinatorics.
4. **Prefer `if constexpr` over specialization graphs:** explicit decision logic improves readability.
5. **Use type erasure at boundaries:** keep ABI stable and rebuild cost predictable.
6. **Measure compilation:** track compile time and binary size as architectural metrics.

15.4.11 Summary

Template complexity is a predictable risk in large C++ systems. The modern approach reduces complexity by making requirements explicit (concepts), limiting instantiation growth (constrained policies), favoring localized compile-time branching (`if constexpr`), and stopping templates at boundaries (type erasure or non-templated facades).
In architectural terms:

- templates should amplify clarity and reuse,
- not amplify rebuild time, binary size, and cognitive load.

A disciplined template architecture treats the type system as a design tool, but it also treats developer time as a first-class resource.

Compile-Time Execution and Design Validation

16.1 constexpr and consteval Design

16.1.1 Compile-Time Execution as a Design Tool

Modern C++ treats compile-time execution as a first-class design mechanism. With `constexpr` and `consteval`, you can do more than "precompute constants": you can **validate designs**, **encode invariants**, and **eliminate entire classes of runtime failure modes**.
This section uses a practical architectural lens:

- **`constexpr`** means: *this computation can run at compile time* (and also may run at runtime).
- **`consteval`** means: *this computation must run at compile time* (a compile-time-only function).

The engineering value is that the compiler becomes a validation engine: it rejects invalid configurations, inconsistent rules, and unsafe assumptions before the program is deployed.

16.1.2 Mental Models and Guarantees

`constexpr`: Dual-Use Functions

A `constexpr` function is eligible for constant evaluation when called in a constant-evaluated context. Otherwise, it behaves like a normal function.
Design implication:

- you can write one function that supports both compile-time and runtime use,
- you can unit-test it at runtime and also rely on it for compile-time validation.

`consteval`: Immediate Functions

A `consteval` function is an *immediate function*: every call must be evaluated at compile time.
Design implication:

- use it to enforce that certain values *cannot* be produced at runtime,
- use it to construct types/configurations that must be validated before the binary exists.

16.1.3 Design Targets for Compile-Time Execution

The most effective uses of `constexpr`/`consteval` in architecture typically fall into:

1. **Configuration validation:** prevent invalid parameter sets.
2. **Protocol and rule validation:** ensure tables, mappings, and state definitions are consistent.
3. **Strong typing and units:** enforce correctness of domain rules at compile time.
4. **Deterministic initialization:** build precomputed tables and hashed keys.
5. **API hardening:** force compile-time construction of safe tokens and identifiers.

16.1.4 Compile-Time Validation with `static_assert`

The simplest form of design validation is `static_assert`. It documents and enforces invariants at compile time.

```
#include <cstddef>
#include <type_traits>

template<class T>
struct StoragePolicy {
  static constexpr std::size_t alignment = alignof(T);
  static constexpr bool trivial = std::is_trivially_copyable_v<T>;

  static_assert(alignment >= alignof(void*),
  ↪  "T alignment must be pointer-aligned or stronger");
};
```

Design effect. A policy that violates a fundamental assumption is rejected immediately, rather than producing subtle runtime bugs.

16.1.5 A `constexpr` Parser: Dual-Use Rule Checking

A clean architectural pattern is to implement validation logic once and use it:

- at compile time for configuration constants,
- at runtime for external input.

Example: `constexpr` Port Validation

```
#include <cstdint>
#include <optional>
```

```
constexpr bool is_valid_port(int p) noexcept {
  return p >= 1 && p <= 65535;
}

struct Port {
  std::uint16_t v{};

  static constexpr std::optional<Port> make(int p) noexcept {
    if (!is_valid_port(p)) return std::nullopt;
    return Port{static_cast<std::uint16_t>(p)};
  }
};

constexpr Port default_port() {
  auto p = Port::make(8080);
  // compile-time validation via constant evaluation when used in constexpr context
  if (!p) throw "invalid port"; // will be a compile-time error if evaluated as constant
  return *p;
}
```

Design note. The "throw in constexpr" technique is useful for constant evaluation failures. In constant evaluation, such a throw makes the expression non-constant, causing compilation to fail when a constant is required.

16.1.6 `consteval` for Non-Negotiable Rules

Use `consteval` when a value must be validated before the program is formed.

Example: Compile-Time Only Token Construction

```
#include <array>
#include <cstddef>
```

```
struct Token {
  std::array<char, 16> bytes{};
};

consteval Token make_token(const char (&s)[17]) {
  Token t{};
  for (std::size_t i = 0; i < 16; ++i) {
    char c = s[i];
    if (c == '\0') {
      // forbid short literals to ensure full length
      throw "token must be exactly 16 characters";
    }
    t.bytes[i] = c;
  }
  if (s[16] != '\0') {
    throw "token literal must end at position 16";
  }
  return t;
}

constexpr Token API_TOKEN = make_token("0123456789ABCDEF");
// constexpr Token BAD = make_token("short"); // error: must be const-evaluated and
↪  validated
```

Design effect. The rule is absolute: tokens cannot be created from runtime strings, and invalid literals are rejected during compilation.

16.1.7 Compile-Time Tables and Design Consistency Checks

Systems frequently rely on lookup tables: enums to strings, opcode tables, message IDs, state transitions. Errors in these tables are common and expensive. `constexpr` enables building and

validating them at compile time.

Example: Enum-to-String Table with Validation

```
#include <array>
#include <string_view>

enum class Status : int {
  ok = 0,
  failed = 1,
  timeout = 2,
  _count
};

constexpr std::array<std::string_view, static_cast<int>(Status::_count)> status_names = {
  "ok",
  "failed",
  "timeout"
};

constexpr bool validate_status_table() {
  return status_names.size() == static_cast<int>(Status::_count);
}

static_assert(validate_status_table(), "status_names must match Status::_count");

constexpr std::string_view to_string(Status s) {
  return status_names[static_cast<int>(s)];
}
```

Design effect. The mapping is validated structurally. The table becomes a compile-time artifact, reducing runtime initialization risk.

16.1.8 Using `constexpr` to Encode Deterministic Identifiers

Architectures often need stable identifiers: command codes, message types, metric names. A `constexpr` hash can generate compile-time IDs that remain stable across runs.

Example: `constexpr` FNV-1a (Deterministic, Not Cryptographic)

```
#include <cstdint>
#include <string_view>

constexpr std::uint64_t fnv1a_64(std::string_view s) noexcept {
  std::uint64_t h = 14695981039346656037ull;
  for (char c : s) {
    h ^= static_cast<std::uint8_t>(c);
    h *= 1099511628211ull;
  }
  return h;
}

constexpr std::uint64_t metric_id = fnv1a_64("requests.total");
static_assert(metric_id != 0, "unexpected");
```

Design note. Such hashes are useful for compile-time keys and switch-like dispatch tables, but they are not a substitute for cryptographic hashing.

16.1.9 `constexpr` Objects and Invariants

A compile-time constructible type is often a higher-quality design: it implies deterministic initialization, no hidden runtime dependencies, and typically clearer ownership and invariants.

Example: A `constexpr`-friendly Value Type

```
#include <cstdint>
```

```
struct Rgb {
  std::uint8_t r{}, g{}, b{};

  constexpr Rgb(std::uint8_t rr, std::uint8_t gg, std::uint8_t bb) : r(rr), g(gg), b(bb) {}
  static constexpr Rgb black() { return {0,0,0}; }
};

constexpr Rgb background = Rgb::black();
```

Design effect. Types that are usable in constant evaluation tend to have:

- value semantics,
- explicit invariants,
- predictable construction.

16.1.10 Architectural Pattern: Compile-Time Policy Validation

Policy-based designs can validate their configurations at compile time.

```
#include <concepts>
#include <type_traits>

struct ThreadSafe {};
struct NotThreadSafe {};

template<class P>
concept ThreadPolicy =
  std::same_as<P, ThreadSafe> || std::same_as<P, NotThreadSafe>;

template<ThreadPolicy P, int Capacity>
```

```
class RingBuffer {
  static_assert(Capacity > 0, "Capacity must be positive");
  static_assert((Capacity & (Capacity - 1)) == 0, "Capacity must be a power of two");

public:
  static constexpr int capacity = Capacity;

  void push(int) {
    if constexpr (std::same_as<P, ThreadSafe>) {
      // locking path
    } else {
      // no-lock path
    }
  }
};
```

Design effect. The buffer cannot be instantiated with invalid capacity, and policy selection remains explicit and analyzable.

16.1.11 Guidelines: Choosing Between `constexpr` and `consteval`

- Use **`constexpr`** when the function/type should be usable both at compile time and runtime. This is the default for reusable validation logic and value-type operations.
- Use **`consteval`** when the value must be proven and fixed at compile time: keys, tokens, compile-time-only configuration, structural checks that must not depend on runtime input.
- Use **`static_assert`** for invariants that should always hold for any instantiation.
- Prefer **simple constant evaluation** over heavy template metaprogramming when the problem is computational.

16.1.12 Failure Modes and How to Avoid Them

- **Overusing consteval:** forcing compile-time evaluation for values that naturally come from runtime inputs makes APIs awkward and can push users toward unsafe workarounds.
- **Hiding dynamic assumptions:** compile-time evaluation cannot validate external reality (I/O, network, clocks). Do not confuse compile-time proof with runtime truth.
- **Compile-time bloat:** large constant-evaluated computations can increase build time. Keep constant evaluation focused on high-leverage validation and stable tables.
- **Ambiguous diagnostics:** write validation logic with clear failure points (e.g., `static_assert` messages) to keep error reporting actionable.

16.1.13 Design Checklist

1. Identify invariants that should never fail in production (sizes, ranges, table consistency, protocol definitions).
2. Implement validation in `constexpr` functions where runtime use is also meaningful.
3. Promote critical, non-negotiable construction paths to `consteval`.
4. Use `static_assert` for structural invariants and policy constraints.
5. Prefer constant evaluation for computations that would otherwise require complex type-level metaprogramming.
6. Keep compile-time computations stable and measure build impact.

16.1.14 Summary

`constexpr` and `consteval` turn the compiler into a design validation engine. Used correctly, they:

- eliminate invalid configurations before deployment,
- enforce invariants structurally and consistently,
- produce deterministic tables and identifiers,
- reduce runtime checks and initialization complexity,
- improve architectural clarity by making rules executable.

In long-lived C++ systems, compile-time execution is not a novelty feature. It is a systematic way to encode, validate, and preserve design intent across years of evolution.

16.2 Compile-Time State Machines

16.2.1 Why State Machines Belong in the Type System

Many real production failures are not "wrong calculations"; they are **protocol violations**: calling operations in the wrong order, using a component before initialization, skipping validation, or reaching an undefined phase.
A **compile-time state machine** encodes the permitted states and transitions in the type system so that illegal transitions are *ill-formed*. Instead of relying on runtime flags and defensive checks, the compiler enforces the protocol.
Architecturally, this yields:

- **No invalid states at call sites:** operations that are not legal in a state simply do not exist.

- **Stronger invariants:** a value of type `State<X>` is a proof that the protocol reached state `X`.
- **Clearer APIs:** state transitions appear in function signatures.
- **Reduced runtime branching:** protocol correctness is not "checked"; it is constructed.

The key design principle:

> When the order of operations matters for correctness, safety, or security, encode it as a state machine and let types enforce it.

16.2.2 Two Compile-Time Approaches

Typestate (Type-Level States)

The most common pattern uses **distinct types** to represent states: `Disconnected`, `Connected`, `Authenticated`, etc. Transitions consume a value in one state and produce a value in another.

Constexpr-Validated State Machines

Another approach uses **`constexpr` transition tables** and compile-time checks to validate that the state machine definition itself is consistent: no missing transitions, no invalid destinations, no unreachable states (when required), and no conflicting event mappings.
In practice, long-lived systems often use both:

- typestate to enforce correctness at usage sites,
- constexpr validation to enforce correctness of the machine definition.

16.2.3 Typestate: Enforcing Protocol Order by Construction

A Minimal Protocol: Connect → Authenticate → Use

```
#include <string>
```

```
#include <string_view>
#include <utility>

struct Disconnected {};
struct Connected {};
struct Authenticated {};

class Transport {
public:
  static Transport connect(std::string endpoint) {
    Transport t;
    t.endpoint_ = std::move(endpoint);
    t.connected_ = true;
    return t;
  }

  void send(std::string_view payload) {
    (void)payload;
    // send over wire...
  }

  const std::string& endpoint() const noexcept { return endpoint_; }

private:
  std::string endpoint_;
  bool connected_{false};
};

template<class State>
class Client;

template<>
class Client<Disconnected> {
```

```
public:
  Client() = default;

  Client<Connected> connect(std::string endpoint) && {
    Transport t = Transport::connect(std::move(endpoint));
    return Client<Connected>(std::move(t));
  }
};

template<>
class Client<Connected> {
public:
  Client<Authenticated> authenticate(std::string token) && {
    (void)token;
    // perform auth handshake using token...
    return Client<Authenticated>(std::move(t_));
  }

private:
  friend class Client<Disconnected>;
  explicit Client(Transport t) : t_(std::move(t)) {}
  Transport t_;
};

template<>
class Client<Authenticated> {
public:
  void request(std::string_view payload) {
    // only valid after authentication
    t_.send(payload);
  }

private:
```

```
  friend class Client<Connected>;
  explicit Client(Transport t) : t_(std::move(t)) {}
  Transport t_;
};

void demo() {
  Client<Disconnected> c;
  auto a = std::move(c).connect("srv").authenticate("token");
  a.request("GET /");

  // Client<Disconnected>{}.request("x"); // error: request not available
  // Client<Connected>{}.request("x");    // error: request not available
}
```

Design effect. Illegal transitions are eliminated. A caller cannot accidentally send a request before authentication because the operation is not part of the type.

Encoding Resource State: Open → Read/Write → Close

For resource protocols, typestate prevents use-after-close patterns.

```
#include <cstddef>
#include <utility>

struct Closed {};
struct Open {};

class FileImpl {
public:
  static FileImpl open(/*path*/) { return FileImpl(true); }
  void write(const void*, std::size_t) {}
  void close() {}
```

```
private:
  explicit FileImpl(bool ok) : ok_(ok) {}
  bool ok_{false};
};

template<class State>
class File;

template<>
class File<Closed> {
public:
  static File<Open> open(/*path*/) {
    return File<Open>(FileImpl::open());
  }
};

template<>
class File<Open> {
public:
  void write(const void* p, std::size_t n) { impl_.write(p, n); }

  File<Closed> close() && {
    impl_.close();
    return File<Closed>{};
  }

private:
  friend class File<Closed>;
  explicit File(FileImpl impl) : impl_(std::move(impl)) {}
  FileImpl impl_;
};

void demo_file() {
```

```
  auto f = File<Closed>::open();
  f.write("abc", 3);
  auto closed = std::move(f).close();
  (void)closed;
  // f.write("x", 1); // error: f is moved-from; protocol completed
}
```

Design effect. The open/closed protocol is encoded structurally. This is particularly valuable when the protocol is safety-critical.

16.2.4 Ergonomics: Avoiding Typestate Friction

Typestate can introduce friction: more types, more moves, and potentially more verbose call chains. The design goal is to preserve correctness without harming usability.
Practical techniques:

- Provide fluent chaining (return next state by value).
- Limit typestate to protocols where violations are expensive.
- Hide state tags behind factory helpers (`make_client()`), so callers do not mention types explicitly.
- Keep states small and value-like (cheap to move).

A Facade That Hides the State Template

```
#include <utility>

struct Disconnected {};
struct Connected {};
struct Authenticated {};
```

```
template<class S> class Client;

inline Client<Disconnected> make_client() { return Client<Disconnected>{}; }

// (Client specializations omitted for brevity; same as earlier)
```

Design effect. Callers see `make_client()` and fluent transitions, not template machinery.

16.2.5 Compile-Time Validation of a State Machine Definition

Typestate enforces usage correctness, but the *definition* of the state machine can still be wrong: missing transitions, wrong next state indices, duplicated events, etc.
A strong design validates the transition table at compile time using `constexpr`.

A Constexpr Transition Table

We model:

- **states** as indices,
- **events** as indices,
- transitions as a table `next[state][event] = new_state` (or `-1` for invalid).

```
#include <array>
#include <cstddef>

constexpr int invalid_state = -1;

template<std::size_t States, std::size_t Events>
struct TransitionTable {
  std::array<std::array<int, Events>, States> next{};
};
```

```
enum class S : std::size_t { disconnected=0, connected=1, authed=2, _count };
enum class E : std::size_t { connect=0, auth=1, send=2, close=3, _count };

constexpr TransitionTable<static_cast<std::size_t>(S::_count),
                          static_cast<std::size_t>(E::_count)>
make_table() {
  TransitionTable<3,4> t{};

  // initialize all transitions to invalid
  for (auto& row : t.next) {
    for (auto& cell : row) cell = invalid_state;
  }

  // disconnected
  t.next[0][0] = 1; // connect -> connected

  // connected
  t.next[1][1] = 2; // auth -> authed
  t.next[1][3] = 0; // close -> disconnected

  // authed
  t.next[2][2] = 2; // send -> authed (self-loop)
  t.next[2][3] = 0; // close -> disconnected

  return t;
}

constexpr auto table = make_table();

constexpr bool validate_table() {
  // rule 1: all destinations are either invalid or a valid state index
  for (std::size_t s = 0; s < table.next.size(); ++s) {
```

```
    for (std::size_t e = 0; e < table.next[s].size(); ++e) {
      int d = table.next[s][e];
      if (d == invalid_state) continue;
      if (d < 0) return false;
      if (static_cast<std::size_t>(d) >= table.next.size()) return false;
    }
  }
  // rule 2: must be able to connect from disconnected
  if (table.next[0][0] == invalid_state) return false;

  return true;
}

static_assert(validate_table(), "State machine transition table is invalid");
```

Design effect. Your protocol definition becomes a compile-time artifact: an invalid table cannot ship.

16.2.6 Using the Validated Table for Runtime Dispatch

Sometimes you need runtime-driven state machines (parsers, protocols, device drivers), but you still want compile-time validation of the transition definition.

```
#include <cstddef>

struct Machine {
  std::size_t state{0};

  bool step(std::size_t event) {
    int ns = table.next[state][event];
    if (ns == invalid_state) return false;
    state = static_cast<std::size_t>(ns);
    return true;
```

```
  }
};
```

Architectural value. Compile-time validation ensures the table is coherent. Runtime logic remains simple and data-driven.

16.2.7 When Compile-Time State Machines Are Worth It

Typestate and compile-time validation are high-leverage when:

- protocol violations are security-critical (authentication, authorization flows),
- state order is safety-critical (device initialization, actuator enabling),
- runtime checks are too easy to bypass or too costly to repeat everywhere,
- you have many call sites and want to enforce correct usage uniformly.

They may be excessive when:

- the protocol is trivial and violations are low impact,
- the API must remain extremely simple for external users,
- the state machine evolves rapidly and would cause large refactoring churn.

16.2.8 Engineering Guidelines

1. Encode phases as types when order matters and misuse is costly.
2. Expose only operations valid for the current state.
3. Make transitions explicit: consume `State<A>` and produce `State<B>`.

4. Validate transition tables at compile time when using table-driven machines.
5. Hide state templates behind simple factories to keep ergonomics acceptable.
6. Stop typestate at module boundaries if ABI stability is required; use a non-templated facade at boundaries.

16.2.9 Summary

Compile-time state machines elevate protocols to first-class design artifacts. Typestate prevents illegal usage by making invalid operations unrepresentable, while constexpr validation ensures the machine definition is internally consistent.
In long-lived systems, this approach:

- removes entire categories of runtime checks,
- makes APIs self-documenting and misuse-resistant,
- strengthens security and safety protocols,
- reduces the long-term cost of maintaining correct sequencing rules.

When used strategically, compile-time state machines turn protocol correctness from a convention into a property enforced by the compiler.

16.3 Static Reflection and Design Introspection

16.3.1 Reflection as an Architectural Capability

Static reflection is compile-time introspection: the program can ask questions about its own types, members, functions, and declarations during compilation, and use the answers to validate designs or generate code.

From an architectural standpoint, reflection is not a "metaprogramming trick"; it is an enforcement mechanism:

- It can **reduce boilerplate** (serialization, logging, mapping, adapters).
- It can **improve correctness** by validating invariants about structure and APIs.
- It can **improve evolvability** by keeping generated code consistent with the true source of truth (the declarations).
- It can **shift failures left**: structural mismatches become compile-time errors.

The key promise is that *design intent becomes mechanically checkable*.

16.3.2 Modern C++ Reflection Model (C++26 Draft Direction)

The current direction of standardized static reflection in C++ uses three foundational ideas:

1. **A reflection value type** (a single opaque "meta handle") representing program entities.
2. **Core language syntax to obtain reflection values** (reflect an entity, reflect a value/object/function).
3. **Splicing syntax** to inject reflected entities back into code (turn meta information into declarations/expressions/types).

This creates a closed loop:

$$\text{program entity} \rightarrow \text{reflection value} \rightarrow \text{metafunctions} \rightarrow \text{validated or generated code.}$$

16.3.3 Reflection Values and the `std::meta::info` Handle

The design centers on a dedicated reflection value type (commonly presented as `std::meta::info`) that represents a compile-time handle to an entity or value.
Important design properties for architecture:

- **Opaque representation:** you do not "peek" into compiler internals; you query via metafunctions.
- **Consteval-only nature:** reflection values are designed for constant evaluation contexts, supporting safe compile-time reasoning without turning runtime into a dynamic reflection system.
- **Stable semantic identity:** reflections can represent "this is *that* entity" (not merely "a string name").

16.3.4 Core Operations: Reflect, Query, and Splice

A practical mental model is to categorize reflection usage into three operations:

1) Reflect

Obtain a reflection value for an entity (type, variable, function, member).

2) Query

Ask compile-time questions using metafunctions such as:

- identifier/name queries,
- membership queries (data members, base classes, parameters),
- property queries (access, linkage, const/volatile, static vs non-static),

- type queries (what is the type of this member?).

3) Splice

Inject the reflected entity into code:

- splice a member name to form a member access expression,
- splice a type into a declaration,
- splice a namespace into a using-directive,
- splice a concept into a template parameter constraint.

16.3.5 A First Example: “Access a Member by Reflected Name”

This example demonstrates the core loop: reflect members, query their identifiers, splice the chosen member.

```
#include <meta>
#include <string_view>

struct S { unsigned i:2, j:6; };

consteval std::meta::info member_named(std::string_view name) {
  for (std::meta::info field : nonstatic_data_members_of(^^S)) {
    if (has_identifier(field) && identifier_of(field) == name) {
      return field;
    }
  }
  return {}; // null reflection (represents "not found")
}

int main() {
```

```
  S s{0, 0};

  // Splice the chosen member into an expression:
  s.[:member_named("j"):] = 42;   // Equivalent intent: s.j = 42;

  // If the reflection is not a usable constant, the program is ill-formed:
  // s.[:member_named("x"):] = 0; // error: no such member
}
```

Architectural insight. This pattern generalizes into schema mapping, GUI binding, scripting gateways, and bridging structured data formats to native members without manual switch statements.

16.3.6 Design Introspection Use Cases

Static reflection becomes a design tool when you use it to **enforce invariants** and **generate consistent glue code**. Common high-leverage domains:

1) Serialization and Schema Evolution

Instead of maintaining parallel "serialization metadata" by hand, reflect the structure directly. The design goal is not "magic serialization" but **single source of truth**.
You can:

- generate field lists,
- enforce that only explicitly annotated members are serialized,
- validate stable field identifiers,
- detect breaking changes at compile time (e.g., removed required fields).

2) Logging, Tracing, and Debug Views

Reflection can generate:

- structured dumps of aggregates,
- stable field-name/value pairs for telemetry,
- compile-time selected redaction rules.

3) API Conformance Checks

In architecture, you often need “this type satisfies our service contract”:

- required methods exist,
- method signatures match,
- overload sets obey naming and parameter rules,
- invariants about visibility and linkage hold.

Concepts can express part of this. Reflection can validate deeper structure and produce better diagnostics.

4) Adapters, Mocks, and Test Harness Generation

Given an interface-like type, reflection can generate:

- forwarding wrappers,
- instrumentation decorators,
- test doubles and mocks where each function is auto-declared consistently.

16.3.7 Introspection with Structural Rules: Compile-Time Validation

A key benefit is enforcing design rules with zero runtime cost.

Example: Validate "No Raw Pointer Data Members" Policy

This example sketches a structural policy check: if the type has non-static data members of raw pointer type, fail compilation.

```
#include <meta>
#include <type_traits>

template<class T>
consteval void enforce_no_raw_pointers() {
  for (std::meta::info m : nonstatic_data_members_of(^^T)) {
    auto mt = type_of(m); // reflection of the member's type
    using M = [:mt:];     // splice reflected type into a real type

    if constexpr (std::is_pointer_v<M>) {
      // Reject design: raw pointer data members often encode unclear ownership/nullability
      ↪ rules.
      static_assert(!std::is_pointer_v<M>,
      ↪ "Raw pointer data member is not allowed by design policy");
    }
  }
}

struct Good {
  int value;
};

struct Bad {
  int* p; // violates policy
```

```
};

consteval void validate() {
  enforce_no_raw_pointers<Good>();
  // enforce_no_raw_pointers<Bad>(); // compile-time error (policy violation)
}
```

Architectural insight. This is how reflection becomes *design governance*: the codebase can mechanically enforce architectural rules that otherwise live in style guides.

16.3.8 Controlled Generation: From Introspection to Synthesized Code

Static reflection is most powerful when paired with safe generation mechanisms. The architectural requirement is that generation must be:

- readable at the call site (clear intent),
- constrained (avoid "anything generates anything"),
- deterministic (build outputs stable across toolchains),
- auditable (generated structure can be reasoned about).

One practical pattern is to generate types by *substituting* template arguments or splicing reflected entities.

Example: Build a Type from a Computed Argument List

```
#include <meta>
#include <utility>
#include <vector>

template<typename T>
```

```
consteval std::meta::info make_integer_seq_refl(T N) {
  std::vector args{^^T};
  for (T k = 0; k < N; ++k) {
    args.push_back(std::meta::reflect_value(k));
  }
  return substitute(^^std::integer_sequence, args);
}

template<typename T, T N>
using make_integer_sequence = [:make_integer_seq_refl<T>(N):];
```

Architectural insight. This demonstrates "compute meta, then materialize type": a common pattern for building safe, validated compile-time constructs without deep template recursion.

16.3.9 Annotations and Intent-Driven Introspection

A recurring architectural need is to distinguish:

- what is *structurally present* (members exist),
- from what is *intended for a specific concern* (serialize this, log that, redact this).

Reflection by itself gives structure; annotations give **intent**. The combination enables designs such as:

- serialize only annotated fields,
- enforce that every public endpoint has documentation tags,
- apply compile-time redaction policies to sensitive members,
- generate stable external schemas from explicitly opted-in elements.

16.3.10 Reflection vs Traditional Metaprogramming: Complexity and Clarity

Reflection changes the cost model:

- Traditional template metaprogramming encodes structure indirectly (traits, specialization, detection).
- Reflection encodes structure directly (ask the compiler for the members, names, parameters, etc.).

Architecturally, this can reduce:

- duplication between types and "metadata" types,
- brittle SFINAE systems,
- macro-heavy registration systems.

But it introduces new responsibilities:

- controlling compile-time cost,
- designing stable introspection APIs,
- ensuring generated code remains understandable and debuggable.

16.3.11 Practical Constraints and Engineering Discipline

Static reflection should be applied with the same discipline as templates and `constexpr`:

1. **Use reflection at boundaries and frameworks:** serialization layers, logging layers, adapter generators, policy validators.

2. **Prefer explicit intent:** do not serialize everything by default; require opt-in via annotations or explicit lists.
3. **Keep rules local and testable:** write small `consteval` validators and use `static_assert` for clear failures.
4. **Avoid hidden generation cascades:** reflection-driven generation should be visible and auditable.
5. **Measure build impact:** reflection can be compile-time heavy; treat compile time as a first-class architectural metric.

16.3.12 Summary

Static reflection turns compile-time execution into **design introspection**: types and declarations become queryable artifacts, and design rules become enforceable code.
In long-lived systems, reflection enables:

- structural validation of architectural policies,
- single-source-of-truth generation for serialization/logging/adapters,
- improved evolvability by removing parallel metadata maintenance,
- earlier, clearer failures for protocol and interface mismatches.

The architectural value is simple:

> When structure and intent are validated at compile time, systems become harder to misuse, easier to evolve, and cheaper to maintain.

16.4 Eliminating Runtime Errors by Design

16.4.1 The Architectural Goal

A long-lived C++ system cannot rely on testing alone to prevent failures. Tests sample behavior; they do not prove the absence of invalid states. The most reliable strategy is to **eliminate entire classes of runtime errors by design**.
In modern C++, "by design" means:

- encode rules in types,
- validate invariants at construction,
- restrict APIs so misuse is ill-formed,
- move checks to compile time when inputs are compile-time known,
- isolate unavoidable runtime uncertainty behind narrow, explicit boundaries.

The design intent is not to write "clever templates." The intent is to build a system where many runtime failures are *unrepresentable*.

16.4.2 A Taxonomy of Runtime Errors You Can Eliminate

Not all runtime errors can be eliminated: I/O failures, network partitions, and hardware faults are part of reality. However, many expensive failures *can* be eliminated or drastically reduced.

1. **Wrong-domain values:** mixing IDs, units, coordinate systems.
2. **Wrong order of operations:** protocol and state machine violations.
3. **Null misuse:** dereferencing optional pointers where non-null is required.

4. **Invalid configuration:** impossible policy combinations, invalid sizes/capacities.
5. **Unvalidated input leaks:** using raw strings/bytes beyond boundaries.
6. **Ownership and lifetime bugs:** double-free, use-after-free, leaks.
7. **Silent error handling:** ignoring failure due to sentinel values.
8. **Ambiguous overload misuse:** calling the wrong function due to implicit conversions.

Each category maps to design techniques that shift failures left.

16.4.3 Technique 1: Strong Types to Eliminate Domain Confusion

When two concepts are different, they must be different types. This removes a large family of runtime bugs caused by argument swapping and accidental mixing.

```
#include <cstdint>
#include <compare>

template<class Rep, class Tag>
class Strong {
public:
  constexpr explicit Strong(Rep v) : v_(v) {}
  constexpr Rep value() const noexcept { return v_; }
  friend constexpr auto operator<=>(Strong, Strong) = default;
private:
  Rep v_;
};

struct UserIdTag {};
struct OrderIdTag {};
using UserId  = Strong<std::uint64_t, UserIdTag>;
using OrderId = Strong<std::uint64_t, OrderIdTag>;
```

```
void load_user(UserId);
void load_order(OrderId);

void demo(UserId u, OrderId o) {
  load_user(u);
  load_order(o);
  // load_user(o); // compile-time error: wrong domain
}
```

Error class eliminated. Wrong-identifier runtime behavior due to swapped IDs becomes impossible.

16.4.4 Technique 2: Encode Units and Time in Types

Time and units are frequent sources of silent runtime faults. Use `std::chrono` for time and typed units for domain-specific measures.

```
#include <chrono>

class RetryPolicy {
public:
  explicit RetryPolicy(std::chrono::milliseconds base_delay)
    : base_(base_delay) {}

  std::chrono::milliseconds delay_for(int attempt) const {
    return base_ * attempt;
  }

private:
  std::chrono::milliseconds base_;
};
```

```
void demo() {
  RetryPolicy p(std::chrono::milliseconds{200});
  auto d = p.delay_for(5);
  (void)d;
}
```

Error class eliminated. Ambiguous “is this seconds or milliseconds?” bugs vanish from interfaces.

16.4.5 Technique 3: Prevent Invalid States (Typestate Protocols)

If order matters, encode it. A caller should not be able to invoke operations that are invalid in the current state.

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

struct Disconnected {};
struct Connected {};
struct Authenticated {};

template<class State>
class Client;

template<>
class Client<Disconnected> {
public:
  Client() = default;

  Client<Connected> connect(std::string endpoint) && {
    endpoint_ = std::move(endpoint);
    return Client<Connected>(std::move(endpoint_));
```

```
  }

private:
  std::string endpoint_;
};

template<>
class Client<Connected> {
public:
  Client<Authenticated> authenticate(std::string token) && {
    (void)token;
    return Client<Authenticated>(std::move(endpoint_));
  }

private:
  friend class Client<Disconnected>;
  explicit Client(std::string endpoint) : endpoint_(std::move(endpoint)) {}
  std::string endpoint_;
};

template<>
class Client<Authenticated> {
public:
  void send(std::string_view payload) { (void)payload; }

private:
  friend class Client<Connected>;
  explicit Client(std::string endpoint) : endpoint_(std::move(endpoint)) {}
  std::string endpoint_;
};

void demo() {
  Client<Disconnected> c;
```

```
  auto a = std::move(c).connect("srv").authenticate("token");
  a.send("data");

  // Client<Disconnected>{}.send("x"); // compile-time error: invalid state
}
```

Error class eliminated. Wrong-phase calls and missing handshake steps are prevented by construction.

16.4.6 Technique 4: Make Construction a Validation Gate

Many runtime errors occur because objects exist in an invalid or partially initialized state. Prevent this by:

- private constructors,
- factory functions that validate,
- returning `std::optional` (or a result type) to force acknowledgment.

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

class NonEmptyString {
public:
  static std::optional<NonEmptyString> make(std::string s) {
    if (s.empty()) return std::nullopt;
    return NonEmptyString(std::move(s));
  }
  std::string_view view() const noexcept { return s_; }

private:
```

```
  explicit NonEmptyString(std::string s) : s_(std::move(s)) {}
  std::string s_;
};

void process_user(NonEmptyString name);

void boundary(std::string raw) {
  if (auto name = NonEmptyString::make(std::move(raw))) {
    process_user(*name);
  } else {
    // explicit invalid-input path
  }
}
```

Error class eliminated. Invalid input no longer flows as if it were valid data.

16.4.7 Technique 5: Eliminate Ownership Bugs with RAII and Move-Only Handles

Ownership is a design rule; encode it in the type system with RAII. Where appropriate, make the handle move-only to prevent double-release.

```
#include <cstdio>
#include <utility>

class File {
public:
  static File open(const char* path, const char* mode) {
    return File(std::fopen(path, mode));
  }

  ~File() { if (f_) std::fclose(f_); }
```

```
  File(const File&) = delete;
  File& operator=(const File&) = delete;

  File(File&& other) noexcept : f_(std::exchange(other.f_, nullptr)) {}
  File& operator=(File&& other) noexcept {
    if (this != &other) {
      if (f_) std::fclose(f_);
      f_ = std::exchange(other.f_, nullptr);
    }
    return *this;
  }

  std::FILE* get() const noexcept { return f_; }
  explicit operator bool() const noexcept { return f_ != nullptr; }

private:
  explicit File(std::FILE* f) : f_(f) {}
  std::FILE* f_{nullptr};
};
```

Error class eliminated. Double-close and many leak patterns are removed structurally.

16.4.8 Technique 6: Make Errors Non-Ignorable (Typed Error Channels)

Sentinel values are easy to ignore. A typed result forces the caller to choose a success or failure path.

```
#include <variant>
#include <string>

struct Error { std::string msg; };
```

```
template<class T>
using Result = std::variant<T, Error>;

Result<int> parse_int(const std::string& s) {
  try {
    return std::stoi(s);
  } catch (...) {
    return Error{"invalid integer"};
  }
}

int use(const std::string& s) {
  auto r = parse_int(s);
  if (auto* v = std::get_if<int>(&r)) return *v;
  return 0; // explicit error handling
}
```

Error class eliminated. Silent failure and unchecked error codes become less likely by interface design.

16.4.9 Technique 7: Constrain Generic APIs with Concepts

Unconstrained templates accept too much and fail late. Concepts move requirements to the signature and improve diagnostics.

```
#include <concepts>
#include <string_view>

template<class T>
concept LogSink =
  requires(T s, std::string_view msg) {
    { s.write(msg) } -> std::same_as<void>;
  };
```

```
template<LogSink Sink>
void log_line(Sink& sink, std::string_view msg) {
  sink.write(msg);
}
```

Error class reduced. Misuse fails at the call site rather than deep inside template instantiation.

16.4.10 Technique 8: Compile-Time Validation with `constexpr` and `consteval`

For configuration and tables that are known at compile time, `constexpr` and `consteval` allow rejecting invalid designs before runtime.

Compile-Time Only Construction for Non-Negotiable Rules

```
#include <array>
#include <cstddef>

struct Token { std::array<char, 8> bytes{}; };

consteval Token make_token(const char (&s)[9]) {
  Token t{};
  for (std::size_t i = 0; i < 8; ++i) {
    if (s[i] == '\0') throw "token must be exactly 8 chars";
    t.bytes[i] = s[i];
  }
  if (s[8] != '\0') throw "token literal must end at 8";
  return t;
}

constexpr Token SERVICE_TOKEN = make_token("ABCD1234");
```

Error class eliminated. Invalid tokens cannot be created at runtime and invalid literals cannot compile.

16.4.11 What Cannot Be Eliminated (And How to Contain It)

Some runtime errors are fundamentally external:

- disk full, permission denied, file corruption,
- network timeouts, partitions, invalid remote peers,
- hardware faults, clock drift, resource exhaustion.

The design goal for these is not elimination, but **containment**:

- represent failure explicitly (typed results),
- centralize boundary handling (I/O layer),
- keep core domain logic free from I/O assumptions,
- validate and convert untrusted inputs into trusted types at boundaries.

16.4.12 A Unified Design Checklist

1. Identify your top runtime failures and classify them (domain, state, ownership, validation, configuration).
2. Make illegal states unrepresentable (strong types, typestate, RAII).
3. Make construction the validation gate (private constructors + factories).
4. Encode time/units and ownership in types.
5. Constrain templates with concepts at public boundaries.

6. Use `constexpr/consteval` for compile-time-known rules and tables.

7. For unavoidable runtime failures, make error paths explicit and localized.

16.4.13 Summary

Eliminating runtime errors by design is the highest-leverage engineering strategy in modern C++ systems. It works by shifting correctness from "checked at runtime" to "constructed correctly":

- types prevent domain confusion,
- typestate prevents protocol violations,
- RAII prevents lifetime corruption,
- validated types prevent untrusted input leaks,
- concepts prevent unintended genericity,
- constexpr/consteval prevent invalid compile-time configurations.

The result is not just fewer bugs. The result is a system whose architecture makes correctness the default and misuse the exception.

Part VI

Concurrency and Parallel Design

Concurrency as a Design Problem

17.1 Concurrency vs Parallelism

17.1.1 Why This Distinction Is a Design Requirement

In software design, **concurrency** and **parallelism** are related but distinct. Confusing them produces architectures that are either:

- complex without performance benefits (concurrency machinery where it is not needed), or
- fast but unsafe (parallel execution without correct synchronization and data ownership rules).

This chapter treats the distinction as a first-class design problem: the goal is to build systems that are **correct under interleaving** (concurrency) and **efficient under multi-core execution** (parallelism) without collapsing the two concerns.

17.1.2 Definitions

Concurrency

Concurrency is the ability of a system to **manage multiple tasks that overlap in time**. Overlapping does not require multiple cores. A single core can be concurrent by interleaving execution (time slicing, event loops, cooperative scheduling).

Design emphasis:

- correctness under nondeterministic ordering,
- coordination (ordering, exclusion, signaling),
- isolation of shared state,
- progress guarantees (no deadlocks, no starvation).

Parallelism

Parallelism is the ability of a system to **execute multiple tasks at the same time**. It requires actual simultaneous execution (multiple cores, SIMD lanes, GPU threads, distributed nodes). Design emphasis:

- throughput and latency improvements,
- workload partitioning (data parallelism, task parallelism),
- minimizing contention and synchronization overhead,
- scalability with hardware resources.

17.1.3 A Useful Mental Model

- Concurrency answers: **How do we structure the program so multiple things can progress?**
- Parallelism answers: **How do we make it run faster by doing multiple things at once?**

A system can be:

- concurrent but not parallel (single-threaded event loop),

- parallel but not truly concurrent in structure (data-parallel batch computation with no complex coordination),
- both concurrent and parallel (server handling many requests across a thread pool).

17.1.4 Concurrency Without Parallelism: The Event Loop

A classic concurrency design uses an event loop: tasks overlap in time but execute on one thread.

```
#include <queue>
#include <functional>

class EventLoop {
public:
  void post(std::function<void()> f) { q_.push(std::move(f)); }

  void run() {
    while (!q_.empty()) {
      auto f = std::move(q_.front());
      q_.pop();
      f(); // tasks are interleaved, not parallel
    }
  }

private:
  std::queue<std::function<void()>> q_;
};
```

Design properties.

- no data races inside the loop (single thread),
- concurrency is expressed as ordering and scheduling of callbacks,
- blocking operations must be avoided or isolated (otherwise the loop stalls).

17.1.5 Parallelism Without Much Concurrency: Data-Parallel Work

A data-parallel problem may not require complex coordination: split input into chunks and compute independently.

```
#include <vector>
#include <thread>
#include <numeric>
#include <cstddef>

long long sum_parallel(const std::vector<int>& v) {
  const std::size_t n = v.size();
  const std::size_t mid = n / 2;

  long long left = 0;
  long long right = 0;

  std::thread t([&]{
    left = std::accumulate(v.begin(), v.begin() + mid, 0LL);
  });

  right = std::accumulate(v.begin() + mid, v.end(), 0LL);
  t.join();

  return left + right;
}
```

Design properties.

- the core issue is partitioning and combining results,
- the coordination requirement is minimal (join),
- shared mutable state is avoided (each worker writes to its own result).

17.1.6 Where the Design Difficulty Actually Lives

Concurrency Difficulty: Shared State and Interleavings

Concurrency becomes difficult when tasks share mutable state:

- the order of reads/writes changes outcomes,
- invariants can be temporarily broken,
- progress can be blocked by locks and waiting.

The core design strategy is to minimize shared mutable state:

- make data immutable where possible,
- confine mutation to a single owner thread,
- use message passing rather than shared memory,
- design explicit ownership and lifetime boundaries.

Parallelism Difficulty: Overheads and Scalability

Parallelism becomes difficult when:

- synchronization overhead dominates computation,
- contention causes cores to wait,
- memory bandwidth and cache coherence limit scaling,
- work is imbalanced (one worker becomes the bottleneck).

The core design strategy is to control overhead:

- choose appropriate granularity (avoid too small tasks),
- minimize shared writes and false sharing,
- use locality-aware partitioning,
- prefer lock-free or wait-free structures only when justified by measurement and correctness proofs.

17.1.7 A Practical Comparison of Design Concerns

- **Concurrency** focuses on **correctness**:
  - ordering constraints,
  - invariants under interleaving,
  - deadlock avoidance,
  - race freedom,
  - cancellation/timeouts and cleanup.
- **Parallelism** focuses on **performance**:
  - partitioning,
  - load balancing,
  - reducing synchronization,
  - exploiting hardware (cores, SIMD),
  - measuring speedup vs overhead.

17.1.8 Implications for C++ Design (Memory Model and Data Races)

C++ defines a formal memory model, and the key practical consequence is:

> A data race on a non-atomic object is undefined behavior.

Therefore, concurrency design must treat race freedom as non-negotiable:

- use mutexes or other synchronization for shared non-atomic data,
- use atomics where lock-free coordination is appropriate,
- use higher-level constructs (futures, thread pools, executors) to reduce error-prone low-level code.

Parallelism does not remove these rules; it amplifies the cost of getting them wrong.

17.1.9 Design Patterns That Separate Concerns

High-quality systems separate concurrency concerns (coordination) from parallelism concerns (execution resources).

Pattern 1: Actor-Style Concurrency

A component owns its state and communicates by messages. This reduces shared mutable state and makes concurrency reasoning local.

```
#include <queue>
#include <mutex>
#include <optional>

template<class Msg>
class Mailbox {
```

```
public:
  void push(Msg m) {
    std::lock_guard<std::mutex> g(mx_);
    q_.push(std::move(m));
  }

  std::optional<Msg> try_pop() {
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    Msg m = std::move(q_.front());
    q_.pop();
    return m;
  }

private:
  std::mutex mx_;
  std::queue<Msg> q_;
};
```

Design effect. The shared state is limited to the mailbox. The actor's internal state can remain single-threaded.

Pattern 2: Parallel Execution as a Service (Thread Pool)

Concurrency expresses tasks; parallelism is provided by an execution service. This keeps the domain logic mostly independent of thread management.

```
#include <vector>
#include <thread>
#include <functional>
#include <queue>
#include <mutex>
#include <condition_variable>
```

```
class ThreadPool {
public:
  explicit ThreadPool(std::size_t n) {
    for (std::size_t i = 0; i < n; ++i) {
      workers_.emplace_back([this] { worker(); });
    }
  }

  ~ThreadPool() {
    {
      std::lock_guard<std::mutex> g(mx_);
      stop_ = true;
    }
    cv_.notify_all();
    for (auto& t : workers_) t.join();
  }

  void submit(std::function<void()> f) {
    {
      std::lock_guard<std::mutex> g(mx_);
      q_.push(std::move(f));
    }
    cv_.notify_one();
  }

private:
  void worker() {
    for (;;) {
      std::function<void()> job;
      {
        std::unique_lock<std::mutex> lk(mx_);
        cv_.wait(lk, [&]{ return stop_ || !q_.empty(); });
```

```
        if (stop_ && q_.empty()) return;
        job = std::move(q_.front());
        q_.pop();
      }
      job();
    }
  }

  std::mutex mx_;
  std::condition_variable cv_;
  std::queue<std::function<void()>> q_;
  std::vector<std::thread> workers_;
  bool stop_{false};
};
```

Design effect. The system can be concurrent in structure (many tasks) while parallelism is managed centrally.

17.1.10 When You Need Concurrency, When You Need Parallelism

Concurrency is the primary requirement when:

- you have multiple independent activities (I/O, timers, user interactions),
- the system must remain responsive while work is ongoing,
- correctness depends on coordination (ordering, cancellation, timeouts),
- you must handle many in-flight requests or sessions.

Parallelism is the primary requirement when:

- you have CPU-heavy work that dominates runtime,

- the work can be partitioned into mostly independent pieces,
- speedup matters and can be measured,
- the bottleneck is throughput rather than coordination.

17.1.11 Engineering Checklist

1. Decide whether your problem is coordination-bound (concurrency) or compute-bound (parallelism).
2. Reduce shared mutable state before adding threads.
3. Prefer structured concurrency tools (tasks, futures, cancellation) over ad-hoc thread creation.
4. Treat data races as catastrophic design errors (undefined behavior).
5. Measure speedup; parallelism without measurement often increases latency and complexity.
6. Separate the concurrency model (how tasks coordinate) from the execution model (where tasks run).

17.1.12 Summary

Concurrency is about managing multiple tasks and ensuring correctness under interleaving.
Parallelism is about executing multiple tasks simultaneously to improve performance.
They can coexist, but they solve different problems. A long-lived C++ architecture treats concurrency as a correctness discipline and parallelism as a measured performance strategy, then composes both through clear boundaries, ownership rules, and disciplined synchronization.

17.2 Data Races as Design Failures

17.2.1 Why Data Races Are Architectural, Not Accidental

In C++, a data race is not merely a bug that "sometimes happens." It is a sign that the architecture allowed an invalid state: two or more threads accessed the same object concurrently, at least one access was a write, and there was no synchronization establishing a happens-before relationship. Treating data races as "implementation details" is one of the most expensive mistakes in systems engineering. A system that can race is a system whose design has not made ownership, sharing, and mutation rules explicit.
Architecturally, a data race is a failure of:

- **ownership design** (who owns which state),
- **confinement design** (where mutation is allowed),
- **protocol design** (when reads/writes are legal),
- **boundary design** (what crosses between threads and how).

17.2.2 The C++ Consequence: Undefined Behavior

A crucial property of the C++ memory model is that:

> A data race on a non-atomic object produces undefined behavior.

This means a data race is not "a wrong value sometimes." It is a complete collapse of language-level reasoning: the compiler is permitted to assume the race does not exist and may transform the program in ways that make failures non-local and non-reproducible.
Therefore, the design goal is not to "handle races". The goal is to **design the race out of the system**.

17.2.3 Definition by Example: What a Data Race Looks Like

The Naive Shared Counter

```
#include <thread>
#include <vector>

int counter = 0;

void worker() {
  for (int i = 0; i < 100000; ++i) {
    ++counter; // data race: unsynchronized read-modify-write
  }
}

int main() {
  std::vector<std::thread> ts;
  for (int i = 0; i < 4; ++i) ts.emplace_back(worker);
  for (auto& t : ts) t.join();
}
```

This program has a data race because:

- multiple threads access `counter`,
- at least one access is a write,
- there is no synchronization.

Why “It Works on My Machine” Is Meaningless

Even if the result appears correct in some runs:

- the compiler may optimize based on assumptions that no data race exists,

- the CPU may reorder operations,
- timing and cache coherence effects can vary across hardware and builds.

Correctness cannot depend on luck.

17.2.4 The Root Cause: Shared Mutable State

Nearly all data races are a symptom of one architectural choice: **shared mutable state without a clear access protocol**.
Therefore, the most effective strategy is to design for:

- **single ownership**,
- **controlled sharing**,
- **explicit synchronization boundaries**,
- **limited mutation surface**.

17.2.5 Design Principle 1: Confinement (Single-Thread Ownership)

The strongest guarantee is confinement: a piece of state is owned and mutated by exactly one thread.

Actor-Style Confinement (Message Passing)

```
#include <queue>
#include <mutex>
#include <optional>
#include <utility>

template<class Msg>
```

```
class Mailbox {
public:
  void push(Msg m) {
    std::lock_guard<std::mutex> g(mx_);
    q_.push(std::move(m));
  }

  std::optional<Msg> try_pop() {
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    Msg m = std::move(q_.front());
    q_.pop();
    return m;
  }

private:
  std::mutex mx_;
  std::queue<Msg> q_;
};

struct Add { int x; };

class CounterActor {
public:
  void on(Add m) { value_ += m.x; } // state mutated in one thread only
  int value() const noexcept { return value_; }

private:
  int value_{0};
};
```

Architectural effect. The shared state is not the counter; it is the mailbox. The counter state remains single-threaded and race-free.

17.2.6 Design Principle 2: Make Sharing Explicit and Narrow

When state must be shared, do not share it "by accident." Create explicit concurrency boundaries.

Guarded State with a Mutex

```cpp
#include <mutex>

class Counter {
public:
  void inc() {
    std::lock_guard<std::mutex> g(mx_);
    ++value_;
  }

  int get() const {
    std::lock_guard<std::mutex> g(mx_);
    return value_;
  }

private:
  mutable std::mutex mx_;
  int value_{0};
};
```

Architectural effect. The rule "all access must hold the lock" is enforced by encapsulation: no external code can touch `value_` without passing through the protocol.

Avoid "Naked Data" in Shared Structures

A common design failure is exposing shared fields publicly or returning references to internal state. Encapsulation must extend to concurrency.

17.2.7 Design Principle 3: Use Atomics for Coordination, Not for General Sharing

Atomics are a precision tool. They are ideal for:

- counters, flags, sequence numbers,
- publication of pointers or indices,
- lock-free coordination patterns where invariants are simple.

They are not ideal for:

- complex multi-field invariants,
- maintaining consistency across many values,
- replacing good architecture with "sprinkled atomics."

Atomic Counter (Correct for This Problem)

```
#include <atomic>

class AtomicCounter {
public:
  void inc() noexcept { value_.fetch_add(1, std::memory_order_relaxed); }
  int get() const noexcept { return value_.load(std::memory_order_relaxed); }

private:
  std::atomic<int> value_{0};
};
```

Design note. Relaxed ordering is often appropriate for statistics counters where only atomicity is required, not cross-thread ordering of other data.

17.2.8 Design Principle 4: Publication and Initialization Are Protocols

A frequent source of races is publishing an object to other threads before it is fully initialized, or accessing it before publication is properly synchronized.
The architectural rule:

> If an object is shared between threads, its publication is part of the design protocol.

Safe publication can be achieved using:

- mutex-protected handoff,
- condition variables,
- `std::future` / `std::promise`,
- atomic pointer publication with acquire-release ordering (when justified).

Safe Handoff with `std::promise/std::future`

```
#include <future>
#include <thread>
#include <string>

std::future<std::string> start_worker() {
  std::promise<std::string> p;
  auto f = p.get_future();

  std::thread([pp = std::move(p)]() mutable {
    std::string result = "ready";
    pp.set_value(std::move(result)); // synchronized handoff
  }).detach();

  return f;
}
```

Architectural effect. The design declares a safe publication mechanism: the result becomes visible only after the promise is satisfied.

17.2.9 Design Principle 5: Data Races Are Often API Design Bugs

Many races occur because APIs allow mutation from multiple threads without making it explicit. Examples of problematic APIs:

- returning non-const references to shared internal objects,
- storing raw pointers to shared objects without lifetime rules,
- accepting callbacks that run on unspecified threads without documentation and constraints,
- allowing "read" operations that actually mutate internal caches without synchronization.

To fix this, the API must:

- state thread-safety guarantees clearly (thread-safe, thread-compatible, thread-hostile),
- enforce access protocols in types and encapsulation,
- separate read-only views from mutable access,
- centralize concurrency controls in a small set of components.

17.2.10 Thread-Safety Levels as Design Contracts

A practical architectural classification:

- **Thread-safe:** safe for concurrent use without external synchronization.
- **Thread-compatible:** safe only if callers synchronize externally.

- **Thread-confined:** must only be used on one designated thread.
- **Thread-hostile:** cannot be used safely across threads (rare and usually indicates redesign needed).

Every public type in a concurrent subsystem should declare which category it belongs to.

17.2.11 Detecting Races Is Not Preventing Them

Dynamic tools can help find races, but architecture prevents them. Relying on detection alone is insufficient because:

- race schedules are nondeterministic and may not manifest under tests,
- production workloads differ from test workloads,
- undefined behavior can hide or amplify symptoms unpredictably.

Use detection tools to validate the design, not to replace it.

17.2.12 A Concurrency Design Checklist (Race Prevention)

1. Identify all shared mutable state and label it as a design risk.
2. Prefer confinement: one thread owns and mutates a state.
3. If sharing is required, enforce a single access protocol:
    - encapsulate state behind a mutex,
    - or use atomics for simple coordination,
    - or use message passing for complex invariants.
4. Make publication explicit: define how objects become visible to other threads.

5. Avoid APIs that leak internal mutable state.
6. Document thread-safety level for every public type and function in the concurrent subsystem.
7. Treat any data race as a design defect requiring refactoring, not patching.

17.2.13 Summary

A data race is not a minor bug. In C++, it is a structural design failure that triggers undefined behavior. The remedy is architectural:

- eliminate shared mutable state where possible,
- confine mutation to a single owner,
- encapsulate shared state behind explicit protocols (locks, atomics, message passing),
- make publication and lifetime rules explicit,
- design APIs that cannot be misused concurrently.

The result is not only correctness. It is a codebase where concurrency becomes predictable, debuggable, and maintainable over years.

17.3 Ownership and Synchronization

17.3.1 Why Ownership Is the Real Concurrency Primitive

In concurrent systems, synchronization primitives (mutexes, atomics, condition variables) do not create correctness on their own. They only *enforce* a rule that must already exist.
The rule is **ownership**:

- Who owns a piece of state?
- Who is allowed to mutate it?
- Under what protocol can other threads observe it?
- When does the ownership transfer or end?

Most data races and deadlocks are not caused by "wrong mutex usage" as such. They are caused by a design that did not establish a consistent ownership model.
A disciplined architecture treats ownership as the primary design tool, and uses synchronization as an *implementation of that model*.

17.3.2 A Working Vocabulary

This section uses the following core terms:

- **Owner:** the component/thread responsible for an object's lifetime and mutation policy.
- **Borrower/Observer:** can read/use a resource without owning it (no lifetime control).
- **Exclusive ownership:** exactly one owner (often `std::unique_ptr` or confinement).
- **Shared ownership:** multiple owners extend lifetime (`std::shared_ptr`) when justified.
- **Confinement:** object is owned and mutated by a single thread.
- **Synchronization boundary:** where state crosses threads; must define visibility and ordering.
- **Publication:** making an object safely visible to other threads.

17.3.3 The Core Rule: No Shared Mutable State Without a Protocol

In C++, a data race on a non-atomic object is undefined behavior. Therefore, the design rule is:

> If state is mutable and shared across threads, you must define a single protocol that governs all access.

This protocol can be implemented via:

- confinement (no sharing),
- a mutex (shared state but serialized access),
- atomics (coordination with well-defined memory ordering),
- message passing (sharing by transferring data, not sharing memory),
- read-mostly patterns (copy-on-write, snapshots, RCU-like designs).

17.3.4 Ownership Models and Their Synchronization Implications

Model A: Thread Confinement (No Synchronization Needed Internally)

If a component owns its state and only one thread touches it, there is no internal shared-state synchronization problem. Coordination happens at the boundary: how messages/events arrive.

```
#include <queue>
#include <mutex>
#include <optional>
#include <utility>

template<class Msg>
class Mailbox {
public:
```

```
  void push(Msg m) {
    std::lock_guard<std::mutex> g(mx_);
    q_.push(std::move(m));
  }

  std::optional<Msg> try_pop() {
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    Msg m = std::move(q_.front());
    q_.pop();
    return m;
  }

private:
  std::mutex mx_;
  std::queue<Msg> q_;
};

struct Add { int x; };

class CounterActor {
public:
  void on(Add m) { value_ += m.x; } // confined mutation
  int value() const noexcept { return value_; }
private:
  int value_{0};
};
```

Design effect. The mailbox is the only shared structure. The actor state remains single-threaded, simplifying correctness.

Model B: Exclusive Ownership With Transfer (Move-Only Handoff)

When ownership must cross threads, prefer **transfer of exclusive ownership**. This reduces coordination complexity because there is only one owner at a time.

```
#include <memory>
#include <queue>
#include <mutex>
#include <optional>
#include <utility>

template<class T>
class Channel {
public:
  void send(T x) {
    std::lock_guard<std::mutex> g(mx_);
    q_.push(std::move(x));
  }

  std::optional<T> try_recv() {
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    T x = std::move(q_.front());
    q_.pop();
    return x;
  }

private:
  std::mutex mx_;
  std::queue<T> q_;
};

struct Job { int id; };
```

```
void demo_transfer() {
  Channel<std::unique_ptr<Job>> ch;
  ch.send(std::make_unique<Job>(Job{1}));

  if (auto j = ch.try_recv()) {
    // exactly one owner exists here
    (void)(*j)->id;
  }
}
```

Design effect. Move-only ownership transfer avoids shared mutation and avoids most lifetime races.

Model C: Shared Ownership (Lifetime) vs Shared Mutation (Synchronization)

`std::shared_ptr` solves *lifetime sharing* but does not automatically solve *safe concurrent mutation* of the object being shared.
Design rule:

> Shared ownership is not shared mutability.

You may share a `shared_ptr<T>` across threads and still race on `T` unless access is synchronized.

```
#include <memory>
#include <mutex>

struct State {
  void set(int v) { value = v; }
  int get() const { return value; }
  int value{0};
};

class SharedState {
```

```
public:
  explicit SharedState(std::shared_ptr<State> s) : s_(std::move(s)) {}

  void set(int v) {
    std::lock_guard<std::mutex> g(mx_);
    s_->set(v);
  }

  int get() const {
    std::lock_guard<std::mutex> g(mx_);
    return s_->get();
  }

private:
  std::shared_ptr<State> s_;
  mutable std::mutex mx_;
};
```

Design effect. Ownership is shared (lifetime), but mutation is serialized by a protocol (mutex).

17.3.5 Safe Publication: Making Objects Visible Across Threads

A common concurrency failure is reading an object from another thread before its initialization or updates become visible. Safe publication is part of ownership transfer.
Practical safe publication mechanisms:

- mutex-protected store and load,
- `std::future`/`std::promise` handoff,
- condition variable signaling after initialization,
- atomic pointer publication with acquire/release semantics when justified.

Handoff With `std::promise`/`std::future`

```
#include <future>
#include <thread>
#include <memory>

struct Config { int value; };

std::future<std::shared_ptr<const Config>> load_config_async() {
  std::promise<std::shared_ptr<const Config>> p;
  auto f = p.get_future();

  std::thread([pp = std::move(p)]() mutable {
    auto cfg = std::make_shared<Config>(Config{42});
    pp.set_value(std::shared_ptr<const Config>(cfg)); // publish fully initialized object
  }).detach();

  return f;
}
```

Design effect. The publication mechanism is explicit: the consumer can only observe the object after it is fully constructed and handed off.

17.3.6 Synchronization Strategies Derived From Ownership

Ownership drives which synchronization strategy is appropriate.

1) Mutexes: Serialize Access to Shared Mutable State

Use a mutex when:

- there are multi-field invariants,
- operations must be atomic as a group,

- simplicity and correctness outweigh lock overhead.

2) Atomics: Coordination for Simple Shared Variables

Use atomics when:

- the shared state is simple (flags, counters, indices),
- you can express invariants without needing multi-variable atomicity,
- you can define appropriate memory ordering.

```
#include <atomic>

class StopFlag {
public:
  void request_stop() noexcept { stop_.store(true, std::memory_order_release); }
  bool stop_requested() const noexcept { return stop_.load(std::memory_order_acquire); }
private:
  std::atomic<bool> stop_{false};
};
```

Design note. Acquire/release establishes an ordering relationship suitable for many stop/publication patterns.

3) Message Passing: Share Data by Moving It

Use message passing when:

- you want to confine complex state to one thread,
- you want explicit ordering via message queues,
- you want to avoid exposing shared mutable structures.

Message passing is the architectural extension of exclusive ownership transfer.

17.3.7 Lock Ordering and Deadlocks as Ownership Violations

Deadlocks often emerge when ownership boundaries are unclear: multiple components lock multiple resources without a global ordering rule.
Architectural defenses:

- define a strict lock acquisition order (by hierarchy or address ordering),
- collapse related resources under one owner (one mutex),
- use scoped locking patterns and minimize lock duration,
- avoid calling unknown code (callbacks, virtuals) while holding locks.

Scoped Locking and Avoiding Re-Entrancy

```
#include <mutex>
#include <functional>

class SafeNotifier {
public:
  void set_callback(std::function<void(int)> cb) {
    std::lock_guard<std::mutex> g(mx_);
    cb_ = std::move(cb);
  }

  void notify(int v) {
    std::function<void(int)> cb_copy;
    {
      std::lock_guard<std::mutex> g(mx_);
      cb_copy = cb_; // copy under lock
    }
    if (cb_copy) cb_copy(v); // invoke without holding lock
  }
```

```
private:
  std::mutex mx_;
  std::function<void(int)> cb_;
};
```

Design effect. The mutex protects internal state, but callbacks are invoked outside the lock, preventing re-entrancy deadlocks and reducing lock hold time.

17.3.8 Read-Mostly Ownership: Immutable Snapshots

For configurations and read-mostly data, prefer immutability: publish an immutable snapshot and replace it atomically.

```
#include <atomic>
#include <memory>

struct Config { int x; };

class ConfigStore {
public:
  void publish(std::shared_ptr<const Config> c) {
    std::atomic_store_explicit(&cfg_, std::move(c), std::memory_order_release);
  }

  std::shared_ptr<const Config> current() const {
    return std::atomic_load_explicit(&cfg_, std::memory_order_acquire);
  }

private:
  std::shared_ptr<const Config> cfg_;
};
```

Design effect. Readers take a shared snapshot with no locks. Writers replace the snapshot by publishing a new immutable object. This combines shared ownership (lifetime) with safe publication.

17.3.9 API Design Rules: Make Thread-Safety a Contract

A concurrency-aware architecture documents and enforces thread-safety contracts:

- **Thread-confined types:** must only be used on one thread; enforce by design (no sharing, no copy).
- **Thread-compatible types:** safe only with external synchronization; document clearly.
- **Thread-safe types:** internally synchronized; encapsulate protocols and avoid leaking internals.

Concrete API design implications:

- avoid exposing internal mutexes or returning references to internal mutable state,
- separate const access (read-only) from mutation access, and specify their thread guarantees,
- prefer value types and immutable objects for cross-thread boundaries,
- prefer move-only handles for resources whose ownership must not be duplicated.

17.3.10 Engineering Checklist

1. Identify all mutable state and assign an **owner**.
2. Decide whether it is **confined**, **transferred**, or **shared**.

3. If shared and mutable, define exactly one synchronization protocol (mutex or equivalent).
4. Treat `shared_ptr` as a lifetime tool, not a synchronization tool.
5. Define how objects are **published** across threads (handoff mechanisms).
6. Establish lock-ordering rules and forbid unknown code execution while holding locks.
7. Prefer immutable snapshots for read-mostly shared data.

17.3.11 Summary

Ownership is the organizing principle of concurrency design. Synchronization primitives are implementations of ownership rules.
A race-free architecture:

- confines mutable state when possible,
- transfers exclusive ownership when crossing threads,
- uses shared ownership only when necessary and pairs it with explicit synchronization,
- defines safe publication protocols,
- minimizes lock scope and forbids re-entrant lock usage patterns.

When ownership is explicit, synchronization becomes simpler, and correctness scales with system size rather than collapsing under complexity.

17.4 Designing Thread-Safe APIs

17.4.1 Thread-Safety Is a Public Contract

Thread-safety is not something you “add later” with a few mutexes. It is a property of the public contract:

- what operations may run concurrently,
- what guarantees exist about visibility and ordering,
- what progress guarantees exist (no deadlocks, bounded blocking),
- what is required from callers (external synchronization, confinement),
- what is forbidden (calling back into user code under a lock, sharing non-thread-safe handles).

A thread-safe implementation behind an unclear contract is still a design failure: users will make wrong assumptions and create races or deadlocks at integration points.
Therefore, designing a thread-safe API begins with specifying **the concurrency model**.
Implementation follows.

17.4.2 Define the Thread-Safety Level Explicitly

A robust codebase classifies each public type into one of these categories:

- **Thread-safe:** operations may be called concurrently without external synchronization.
- **Thread-compatible:** operations require external synchronization if used concurrently.
- **Thread-confined:** the object must be used only on one designated thread.

This classification must be a design artifact, not tribal knowledge.

A Documentation-Friendly Pattern (In Code)

```
struct thread_safe_tag {};
struct thread_compatible_tag {};
struct thread_confined_tag {};

template<class Tag>
struct thread_safety_level { using tag = Tag; };

class Cache : public thread_safety_level<thread_safe_tag> {
  // ...
};

class Parser : public thread_safety_level<thread_confined_tag> {
  // ...
};
```

Design effect. This is not enforcement by itself, but it makes the intended model visible and reviewable.

17.4.3 Choose a Concurrency Model Before Writing the API

There are only a few stable concurrency models that scale in long-lived systems:

1. **Confinement (single-thread ownership):** simplest; message passing at boundaries.
2. **External synchronization:** object is not internally locked; caller controls locks.
3. **Internal synchronization:** object protects its invariants; safe concurrent calls.
4. **Immutable snapshots:** publish immutable state; readers are lock-free.
5. **Actor style:** state is owned by an execution context; API is asynchronous message dispatch.

The API should make the model obvious. If users cannot infer the model from the interface, the design is already risky.

17.4.4 Rule 1: Keep Invariants Behind a Single Synchronization Boundary

Thread-safety means preserving invariants under concurrency. The most reliable way is to:

- encapsulate all mutable state,
- ensure every access that may observe or modify it goes through one protocol.

A Minimal Thread-Safe Counter

```
#include <mutex>

class Counter {
public:
  void inc() {
    std::lock_guard<std::mutex> g(mx_);
    ++value_;
  }

  int get() const {
    std::lock_guard<std::mutex> g(mx_);
    return value_;
  }

private:
  mutable std::mutex mx_;
  int value_{0};
};
```

Design effect. The invariant “value_ is accessed only under mx_” is enforceable by encapsulation.

17.4.5 Rule 2: Avoid Leaking References, Iterators, and Raw Pointers to Internals

A classic API mistake is returning a reference or iterator into a protected container. Even if you lock internally, the reference escapes the lock and becomes unsafe.

Bad: Returning References to Internal State

```
#include <mutex>
#include <vector>

class BadStore {
public:
  int& at(std::size_t i) { // dangerous
    std::lock_guard<std::mutex> g(mx_);
    return data_.at(i); // reference escapes lock
  }
private:
  std::mutex mx_;
  std::vector<int> data_{1,2,3};
};
```

Good: Return Values or Snapshots

```
#include <mutex>
#include <vector>

class Store {
public:
```

```
  int at(std::size_t i) const {
    std::lock_guard<std::mutex> g(mx_);
    return data_.at(i); // copy is safe
  }

  std::vector<int> snapshot() const {
    std::lock_guard<std::mutex> g(mx_);
    return data_; // copy snapshot; safe for iteration outside
  }

private:
  mutable std::mutex mx_;
  std::vector<int> data_{1,2,3};
};
```

Design effect. A snapshot-based API makes iteration safe without holding locks across unknown user code.

17.4.6 Rule 3: Never Call User Code While Holding a Lock

Invoking callbacks, virtual functions, or unknown functors while holding locks is a frequent cause of:

- deadlocks (re-entrancy),
- priority inversion,
- unbounded lock hold time,
- hidden lock ordering cycles.

Pattern: Copy Callback Under Lock, Invoke Outside

```
#include <mutex>
#include <functional>

class Notifier {
public:
  void set_callback(std::function<void(int)> cb) {
    std::lock_guard<std::mutex> g(mx_);
    cb_ = std::move(cb);
  }

  void notify(int v) {
    std::function<void(int)> cb;
    {
      std::lock_guard<std::mutex> g(mx_);
      cb = cb_; // copy or move under lock
    }
    if (cb) cb(v); // invoke outside lock
  }

private:
  std::mutex mx_;
  std::function<void(int)> cb_;
};
```

Design effect. You keep invariants protected while preventing re-entrancy deadlocks.

17.4.7 Rule 4: Make Blocking Behavior Explicit

Thread-safe APIs often hide blocking behavior: a call may block on a lock, a condition variable, or I/O. If this is not explicitly documented in the interface contract, users may call it from latency-sensitive threads (UI thread, real-time thread) and freeze the system.

API design should classify operations as:

- **non-blocking:** bounded time; does not wait for other threads,
- **blocking:** may wait for locks or conditions,
- **potentially unbounded:** may wait on external resources or user actions.

A practical tactic is to split operations:

```
#include <mutex>
#include <queue>
#include <optional>

template<class T>
class Queue {
public:
  void push(T x) {
    std::lock_guard<std::mutex> g(mx_);
    q_.push(std::move(x));
  }

  std::optional<T> try_pop() { // non-blocking
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    T v = std::move(q_.front());
    q_.pop();
    return v;
  }

private:
  std::mutex mx_;
  std::queue<T> q_;
};
```

Design effect. Non-blocking variants are safe for tight loops and low-latency threads.

17.4.8 Rule 5: Separate Lifetime Safety from Data Safety

`std::shared_ptr` provides shared lifetime, not synchronized mutation. Thread-safe APIs must specify:

- whether the object itself is safe to mutate concurrently,
- or only safe to share its lifetime and access through external synchronization.

For read-mostly shared data, immutable snapshots are a strong pattern.

Immutable Snapshot Publication

```
#include <atomic>
#include <memory>

struct Config { int x; };

class ConfigStore {
public:
  void publish(std::shared_ptr<const Config> cfg) {
    std::atomic_store_explicit(&cfg_, std::move(cfg), std::memory_order_release);
  }

  std::shared_ptr<const Config> current() const {
    return std::atomic_load_explicit(&cfg_, std::memory_order_acquire);
  }

private:
  std::shared_ptr<const Config> cfg_;
};
```

Design effect. Readers are lock-free and cannot observe a partially initialized object. Writers publish whole new snapshots.

17.4.9 Rule 6: Define Atomicity at the API Level

Thread safety is not just "no data races." Users also need to know which operations are atomic with respect to each other.
Example questions:

- Is `get()` consistent with `set()`?
- Is `size()` a snapshot or can it change during iteration?
- Are multi-step updates atomic as a group?

If multi-step invariants must be preserved, provide atomic operations:

```
#include <mutex>

class Balance {
public:
  void deposit(int x) {
    std::lock_guard<std::mutex> g(mx_);
    value_ += x;
  }

  bool withdraw_if_possible(int x) { // atomic check+update
    std::lock_guard<std::mutex> g(mx_);
    if (value_ < x) return false;
    value_ -= x;
    return true;
  }
```

```
  int snapshot() const {
    std::lock_guard<std::mutex> g(mx_);
    return value_;
  }

private:
  mutable std::mutex mx_;
  int value_{0};
};
```

Design effect. The invariants are preserved by offering the right atomic compound operations.

17.4.10 Rule 7: Prefer Message Passing for Complex Shared Invariants

If your object has many fields with complex invariants and high contention, internal locking may become a bottleneck and a complexity trap. An actor-style API is often more scalable and easier to reason about.

```
#include <functional>
#include <queue>
#include <mutex>
#include <optional>
#include <utility>

template<class Msg>
class Mailbox {
public:
  void push(Msg m) {
    std::lock_guard<std::mutex> g(mx_);
    q_.push(std::move(m));
  }

  std::optional<Msg> try_pop() {
```

```
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    Msg m = std::move(q_.front());
    q_.pop();
    return m;
  }

private:
  std::mutex mx_;
  std::queue<Msg> q_;
};

class Actor {
public:
  using Msg = std::function<void()>;

  void post(Msg m) { mb_.push(std::move(m)); }

  void poll() { // called by the owning thread
    while (auto m = mb_.try_pop()) {
      (*m)(); // state changes occur only on owner thread
    }
  }

private:
  Mailbox<Msg> mb_;
};
```

Design effect. You move complexity from shared state to message ordering, which is often easier to control and test.

17.4.11 Rule 8: Establish Lock Ordering and Keep Lock Scope Small

If your API uses multiple locks internally:

- define a strict lock ordering and follow it everywhere,
- keep lock scope minimal,
- avoid nested locks unless proven necessary,
- never hold locks across blocking I/O.

A simple practice is to use one mutex per invariant group and avoid “lock graphs” unless absolutely required.

17.4.12 Testing and Verification Are Part of the API Design

Even with strong design, concurrency requires verification. However, verification validates a design; it does not replace it.
A complete engineering approach includes:

- stress tests with high contention,
- deterministic tests for ordering (where possible),
- race detection in CI for concurrent subsystems,
- code review checklists focused on ownership, lock scope, and callback policies.

17.4.13 A Practical Thread-Safe API Checklist

1. Declare the thread-safety level (safe/compatible/confined).
2. Make blocking behavior explicit; provide non-blocking variants where needed.

3. Encapsulate shared mutable state; never leak references/iterators to internals.
4. Never call user code under locks; copy state/callbacks and invoke outside.
5. Define atomicity: specify which operations are atomic and provide compound operations when needed.
6. Use `shared_ptr` for lifetime only; synchronize mutation separately or publish immutable snapshots.
7. Prefer ownership transfer or message passing for complex invariants.
8. Keep lock scope small; define lock ordering if multiple locks exist.

17.4.14 Summary

Designing thread-safe APIs is primarily about contract and structure:

- define the concurrency model,
- encode ownership and invariants behind clear boundaries,
- make atomicity and blocking explicit,
- prevent misuse by construction rather than by convention.

When an API is designed with these principles, thread-safety becomes predictable: users know what is allowed, the implementation can enforce it, and the system remains maintainable over time.

Modern C++ Concurrency Models

18.1 Message Passing vs Shared State

18.1.1 The Architectural Choice That Determines Everything

Most concurrency architectures reduce to one fundamental choice:

- **Shared state:** threads communicate by reading and writing the same memory.
- **Message passing:** threads communicate by exchanging messages; state is owned, not shared.

This is not a stylistic preference. It determines:

- how correctness is achieved (locks/atomics vs ownership/ordering),
- how failures occur (data races/deadlocks vs backpressure/queue overflow),
- how systems scale (contention vs queueing and locality),
- how APIs are designed (mutable references vs commands/events),
- how testing is performed (interleavings vs message traces).

The highest-leverage design decision in a concurrent system is choosing where you allow shared mutable state.

18.1.2 Definitions

Shared-State Concurrency

Threads access and mutate shared objects. Correctness requires synchronization: mutexes, atomics, condition variables, fences, and a disciplined ownership and lock-ordering model.

Message-Passing Concurrency

Each component (often an "actor") owns its state and processes incoming messages. Communication occurs via queues/channels; shared state is minimized and often limited to the mailbox itself.

18.1.3 The Cost Model: What You Pay For

Shared State: Cost Paid in Reasoning and Synchronization

Shared-state designs require:

- identifying all shared mutable variables,
- defining a single access protocol for each,
- preventing data races (undefined behavior on non-atomics),
- avoiding deadlocks and lock-order inversions,
- ensuring safe publication and memory visibility.

Performance costs include:

- lock contention,
- cache line bouncing and coherence traffic,

- false sharing,
- priority inversion and convoy effects.

Message Passing: Cost Paid in Queueing and Backpressure

Message passing shifts costs:

- queue management and memory allocation strategy,
- message copy/move overhead,
- latency introduced by queueing,
- backpressure and overload handling,
- ordering and delivery semantics.

In return, it reduces shared-mutable-state complexity.

18.1.4 Shared State in Practice: A Thread-Safe Shared Cache

A shared cache is a classic shared-state use case: multiple threads want fast access to common data.

```
#include <unordered_map>
#include <string>
#include <optional>
#include <mutex>

class SharedCache {
public:
  std::optional<int> get(const std::string& key) const {
    std::lock_guard<std::mutex> g(mx_);
```

```
    auto it = data_.find(key);
    if (it == data_.end()) return std::nullopt;
    return it->second;
  }

  void put(std::string key, int value) {
    std::lock_guard<std::mutex> g(mx_);
    data_[std::move(key)] = value;
  }

private:
  mutable std::mutex mx_;
  std::unordered_map<std::string, int> data_;
};
```

Properties.

- Simple interface; callers use it like a normal object.
- Correctness depends on a strict invariant: all accesses hold `mx_`.
- Scaling depends on contention (hot keys, frequent writes).

Typical failure modes.

- returning references/iterators to internal state (escaping lock),
- calling user callbacks while holding the lock (deadlocks),
- lock contention under load.

18.1.5 Message Passing in Practice: A Confined Cache Actor

Instead of sharing the cache, confine it to one thread and access it via messages.

```
#include <queue>
#include <mutex>
#include <optional>
#include <functional>
#include <unordered_map>
#include <string>
#include <utility>

template<class Msg>
class Mailbox {
public:
  void push(Msg m) {
    std::lock_guard<std::mutex> g(mx_);
    q_.push(std::move(m));
  }

  std::optional<Msg> try_pop() {
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    Msg m = std::move(q_.front());
    q_.pop();
    return m;
  }

private:
  std::mutex mx_;
  std::queue<Msg> q_;
};

class CacheActor {
public:
  using Msg = std::function<void(CacheActor&)>;
```

```
  void post(Msg m) { mb_.push(std::move(m)); }

  void poll() { // called on the actor's owning thread
    while (auto m = mb_.try_pop()) {
      (*m)(*this); // all state mutation occurs here
    }
  }

  // confined state operations (not thread-safe by themselves)
  std::optional<int> get_local(const std::string& key) const {
    auto it = data_.find(key);
    if (it == data_.end()) return std::nullopt;
    return it->second;
  }

  void put_local(std::string key, int value) {
    data_[std::move(key)] = value;
  }

private:
  Mailbox<Msg> mb_;
  std::unordered_map<std::string, int> data_;
};
```

How it is used. A client thread posts a message describing what to do, often capturing a promise, callback, or shared result holder. The actor thread polls and applies messages in order.

Properties.

- No data races on the cache state: it is confined to one thread.
- Correctness moves to message ordering and queue behavior.
- Scaling can be improved by sharding: multiple actors each own a partition.

Typical failure modes.

- unbounded queues causing memory growth,
- latency under load (queueing delays),
- slow actor thread becoming a bottleneck,
- complex request/reply wiring if not designed carefully.

18.1.6 Design Trade-offs: Correctness vs Complexity

Correctness Reasoning

Shared state reasoning is about:

- interleavings,
- lock coverage,
- atomicity boundaries,
- memory visibility.

Message passing reasoning is about:

- message ordering,
- ownership and confinement,
- queue capacity and backpressure,
- delivery guarantees.

Message passing often makes correctness more local: each actor can be reasoned about as single-threaded code.

Performance and Scalability

Shared state scales poorly under contention because:

- locks serialize access,
- writes trigger cache coherence traffic,
- critical sections grow over time.

Message passing can scale well if:

- messages are cheap to move/copy,
- actors are partitioned (sharded) to avoid single bottlenecks,
- backpressure is applied.

However, message passing adds latency due to queuing and scheduling.

18.1.7 Choosing the Model: A Practical Decision Framework

Prefer Shared State When:

- the shared data is small and low-contention,
- operations must be synchronous and low-latency at the call site,
- the invariant is simple and can be protected with a small critical section,
- you can enforce strict encapsulation (no leaking references).

Prefer Message Passing When:

- invariants are complex and multi-field (hard to protect correctly with fine-grained locks),
- you want to confine mutation to one thread for safety and simplicity,
- the system is naturally event-driven (servers, pipelines, UI, embedded controllers),
- you need clear ownership boundaries and explicit ordering,
- sharding across cores is feasible (many independent actors).

18.1.8 Hybrid Architectures (Common in Production)

Most real systems are hybrids:

- message passing between major components,
- shared state within a component for a small set of counters/flags,
- immutable snapshots for configuration,
- a thread pool for parallel execution of CPU-heavy tasks.

A healthy hybrid rule:

> Use message passing to structure ownership and isolate complex state. Use shared state only in small, well-encapsulated, low-contention places.

18.1.9 Backpressure: The Missing Design Requirement in Message Passing

Shared-state designs often fail by races and deadlocks. Message-passing designs often fail by **unbounded input**.
Therefore, a message-passing API must include backpressure design:

- bounded queues,
- drop policies (drop newest, drop oldest, coalesce),
- load shedding (reject requests),
- prioritization (control messages vs data messages),
- timeouts and cancellation.

A Bounded Queue Sketch

```
#include <queue>
#include <mutex>
#include <optional>

template<class T>
class BoundedQueue {
public:
  explicit BoundedQueue(std::size_t cap) : cap_(cap) {}

  bool try_push(T x) {
    std::lock_guard<std::mutex> g(mx_);
    if (q_.size() >= cap_) return false;
    q_.push(std::move(x));
    return true;
  }

  std::optional<T> try_pop() {
    std::lock_guard<std::mutex> g(mx_);
    if (q_.empty()) return std::nullopt;
    T v = std::move(q_.front());
    q_.pop();
    return v;
```

```
  }

private:
  std::mutex mx_;
  std::queue<T> q_;
  std::size_t cap_;
};
```

Design effect. Bounded queues force the system to define overload behavior explicitly.

18.1.10 API Design Rules for Both Models

Shared-State API Rules

- Encapsulate the mutex with the data; do not expose it.
- Do not return references/iterators/pointers to internals.
- Do not call unknown code (callbacks) while holding locks.
- Provide atomic compound operations (check+update) when invariants require it.
- Keep critical sections small and measurable.

Message-Passing API Rules

- Make ordering guarantees explicit (FIFO per sender? per actor?).
- Make capacity/backpressure explicit (bounded queue and failure semantics).
- Separate command messages (mutating) from query messages (read-only) when useful.
- Prefer move-only payloads for ownership transfer and to avoid sharing.
- Provide cancellation/timeouts for request/reply interactions.

18.1.11 Summary

Shared state and message passing are two fundamentally different concurrency models:

- **Shared state** is direct and synchronous, but correctness relies on disciplined synchronization.
- **Message passing** isolates state and simplifies race reasoning, but requires explicit queueing and backpressure design.

In long-lived C++ systems, the best architecture is usually hybrid: use message passing to isolate complex state and enforce ownership, and use shared state sparingly for low-contention, well-encapsulated data where it is truly justified.

18.2 Futures, Tasks, and Executors

18.2.1 Why These Three Concepts Form a Concurrency Model

Modern C++ concurrency in production systems is rarely about "raw threads." Instead, robust systems express concurrency using:

- **tasks**: units of work,
- **futures**: typed result channels for asynchronous completion,
- **executors**: the mechanism deciding *where and how* tasks run.

Architecturally, these concepts separate:

- **what** work is (task semantics),
- **how** completion is observed (future semantics),

- **where** it runs (execution policy and resources).

This separation is the difference between a maintainable concurrency architecture and a system where every feature invents its own thread management.

18.2.2 Futures and Promises: The Typed Completion Channel

A **future** represents a value that will become available later. A **promise** is the producer side that fulfills that value.
The design value:

- explicit ownership of completion,
- one-time delivery semantics,
- structured handoff between threads,
- exception propagation from producer to consumer.

Basic Producer/Consumer With `std::promise` and `std::future`

```
#include <future>
#include <thread>
#include <string>
#include <utility>

std::future<std::string> compute_async() {
  std::promise<std::string> p;
  auto f = p.get_future();

  std::thread([pp = std::move(p)]() mutable {
    try {
      std::string result = "done";
```

```
      pp.set_value(std::move(result));
    } catch (...) {
      pp.set_exception(std::current_exception());
    }
  }).detach();

  return f;
}

int main() {
  auto f = compute_async();
  auto s = f.get(); // blocks until ready; propagates exception if set_exception() was used
  (void)s;
}
```

Design contract.

- `get()` is a synchronization point and may block.
- Completion is delivered once; future is typically single-consumer.
- Exceptions can cross thread boundaries safely via the future.

18.2.3 `std::async`: Convenient Tasks With a Hidden Executor

`std::async` creates a task and returns a future. However, the execution policy is subtle: it may run:

- **asynchronously** (on another thread), or
- **deferred** (runs only when `get()` or `wait()` is called).

Because the scheduling is not always explicit, `std::async` is best treated as a convenience tool, not a full concurrency architecture.

Forcing Asynchronous Execution

```
#include <future>

int heavy(int x) {
  return x * x;
}

int main() {
  auto f = std::async(std::launch::async, heavy, 12); // request async execution
  int r = f.get();
  (void)r;
}
```

Architectural note. If a system needs predictable scheduling, bounded concurrency, or thread pools, it should not rely on `std::async` alone.

18.2.4 Tasks: Defining Work Units Correctly

A task is not simply a function. A well-designed task has:

- explicit inputs and outputs,
- explicit ownership/lifetime rules for captured resources,
- defined cancellation and timeout behavior (if needed),
- predictable blocking behavior (does it block? does it call I/O?).

Task Captures and Lifetime Safety

A frequent design failure is capturing references into asynchronous tasks that outlive the referents.

```
#include <future>
#include <string>

std::future<std::size_t> safe_task(std::string s) {
  return std::async(std::launch::async, [x = std::move(s)] {
    return x.size(); // owns its input
  });
}
```

Design effect. Move ownership into the task. Avoid reference captures unless lifetime is proven and enforced by design.

18.2.5 Blocking Semantics: `get()`, `wait()`, and Timeouts

Futures provide a clean design boundary, but they may block. A production system must decide where blocking is allowed:

- never in an event loop,
- never under locks,
- never on latency-critical threads,
- allowed only in worker threads or dedicated waiting contexts.

Timeout-Based Waiting

```
#include <future>
#include <chrono>

int main() {
  auto f = std::async(std::launch::async, [] { return 7; });
```

```
  if (f.wait_for(std::chrono::milliseconds{50}) == std::future_status::ready) {
    int v = f.get();
    (void)v;
  } else {
    // timeout path
  }
}
```

Design effect. Timeouts turn blocking into an explicit decision with a controlled fallback path.

18.2.6 Continuations: The Missing Piece in `std::future`

Many modern concurrency models rely on continuations: *when this future completes, schedule the next step*. Classical `std::future` is limited in this area. Architecturally, this leads to either:

- blocking composition (wait/get chains), or
- custom continuation frameworks.

The design lesson:

> If your architecture is naturally continuation-based, you need an executor-aware continuation mechanism.

18.2.7 Executors: Making "Where It Runs" a First-Class Policy

An **executor** is a component that accepts tasks and runs them under a chosen policy:

- on a dedicated thread,
- on a thread pool,
- inline (synchronous executor),

- on a CPU-affined set of threads,
- on an I/O loop thread,
- or on a priority-based scheduler.

In architecture, executors are essential because they:

- centralize scheduling decisions,
- enforce bounded concurrency,
- support predictable resource usage,
- separate domain code from threading details.

A Minimal Executor Interface (Architectural Sketch)

```
#include <functional>

class Executor {
public:
  virtual ~Executor() = default;
  virtual void execute(std::function<void()> task) = 0;
};
```

This interface is intentionally small: a single capability — schedule a task.

18.2.8 A Simple Thread-Pool Executor

Thread pools are a common production executor: bounded threads, queued work, predictable lifecycle.

```
#include <vector>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <utility>

class ThreadPoolExecutor {
public:
  explicit ThreadPoolExecutor(std::size_t n) {
    for (std::size_t i = 0; i < n; ++i) {
      workers_.emplace_back([this] { worker(); });
    }
  }

  ~ThreadPoolExecutor() {
    {
      std::lock_guard<std::mutex> g(mx_);
      stop_ = true;
    }
    cv_.notify_all();
    for (auto& t : workers_) t.join();
  }

  void execute(std::function<void()> task) {
    {
      std::lock_guard<std::mutex> g(mx_);
      q_.push(std::move(task));
    }
    cv_.notify_one();
  }
```

```
private:
  void worker() {
    for (;;) {
      std::function<void()> job;
      {
        std::unique_lock<std::mutex> lk(mx_);
        cv_.wait(lk, [&]{ return stop_ || !q_.empty(); });
        if (stop_ && q_.empty()) return;
        job = std::move(q_.front());
        q_.pop();
      }
      job();
    }
  }

  std::mutex mx_;
  std::condition_variable cv_;
  std::queue<std::function<void()>> q_;
  std::vector<std::thread> workers_;
  bool stop_{false};
};
```

Design properties.

- bounded worker threads,
- tasks are queued (backpressure requires additional design: bounded queues or rejection policies),
- a clear lifecycle: construction creates resources, destruction joins and releases.

18.2.9 Tasks + Futures on an Executor: Building a Result-Producing API

Executors schedule tasks; futures deliver results. A common pattern is to provide a `submit` API that returns a future.

```
#include <future>
#include <functional>
#include <type_traits>
#include <utility>

class Submitter {
public:
  explicit Submitter(ThreadPoolExecutor& ex) : ex_(ex) {}

  template<class F, class... Args>
  auto submit(F&& f, Args&&... args)
    -> std::future<std::invoke_result_t<F, Args...>>
  {
    using R = std::invoke_result_t<F, Args...>;

    std::packaged_task<R()> task(
      [ff = std::forward<F>(f), ... aa = std::forward<Args>(args)]() mutable -> R {
        return std::invoke(std::move(ff), std::move(aa)...);
      }
    );

    auto fut = task.get_future();
    ex_.execute([t = std::move(task)]() mutable { t(); });
    return fut;
  }

private:
  ThreadPoolExecutor& ex_;
};
```

Design effect. Domain code can submit work and receive a typed future. Threading becomes a centralized policy (the executor), not scattered `std::thread` creation.

18.2.10 Cancellation and Cooperative Stop

A robust task model often needs cancellation. In modern C++, cancellation is typically cooperative:

- a task periodically checks a stop request,
- cleanup is performed deterministically,
- cancellation becomes part of the task contract.

A Stop-Flag Pattern

```
#include <atomic>

class StopToken {
public:
  void request_stop() noexcept { stop_.store(true, std::memory_order_release); }
  bool stop_requested() const noexcept { return stop_.load(std::memory_order_acquire); }

private:
  std::atomic<bool> stop_{false};
};
```

Design note. Tasks should define whether they support cancellation and what cleanup guarantees they provide.

18.2.11 Design Rules for Futures, Tasks, and Executors

Rule 1: Do Not Mix Scheduling Concerns Into Domain Code

Domain logic should not decide thread counts, create threads, or manage lifetimes. It should submit tasks to an execution policy.

Rule 2: Make Blocking an Explicit Architectural Choice

- Never call `get()` under a lock.
- Avoid blocking in event loops.
- Use timeouts and cancellation where appropriate.

Rule 3: Control Backpressure

If tasks can be submitted faster than they complete:

- bound the queue,
- reject submissions,
- shed load,
- coalesce tasks,
- or scale out with sharding.

Rule 4: Prefer Value Ownership in Tasks

Avoid capturing references into async tasks unless the lifetime is guaranteed by design.

Rule 5: Decide Where Exceptions Go

Futures can transport exceptions, but the design must decide:

- are exceptions allowed across thread boundaries?
- do you translate exceptions into error codes/results at the boundary?
- do you log and terminate on unexpected exceptions inside worker threads?

18.2.12 Summary

Futures, tasks, and executors form a modern concurrency model by separating:

- **tasks** (what work is),
- **futures** (how completion and errors are observed),
- **executors** (where the work runs and how resources are managed).

A long-lived C++ architecture treats executors as scheduling infrastructure, treats futures as explicit completion channels, and treats tasks as well-specified units of work with clear lifetime, blocking, and cancellation contracts.

18.3 Lock-Free Design Principles

18.3.1 Lock-Free Is a Performance Tool, Not a Default

Lock-free programming is often presented as the “advanced” path, but in long-lived systems it should be treated as a specialized performance tool.
The architectural priority order is:

1. **Correctness** (race freedom, invariants, well-defined lifetime).

2. **Simplicity** (maintainable, reviewable, testable).

3. **Performance** (measured and justified).

Lock-free designs can improve latency and throughput under contention, but they can also amplify complexity, create subtle correctness hazards, and reduce portability if the design assumes specific atomic capabilities.
The correct design posture is:

> Use lock-free structures only when measurement shows that locks are the bottleneck and you can state the correctness argument precisely.

18.3.2 Progress Guarantees: Blocking, Lock-Free, Wait-Free

Concurrency algorithms are classified by their progress guarantees:

- **Blocking:** threads may block indefinitely (mutexes, condition variables).
- **Lock-free:** system-wide progress is guaranteed: at least one thread makes progress in a finite number of steps.
- **Wait-free:** per-thread progress is guaranteed: each thread completes its operation in a finite number of steps.

Lock-free does *not* mean "no waiting." A lock-free algorithm may still spin and retry; it only guarantees that the system does not stop progressing entirely.

18.3.3 The Memory Model Foundation You Must Respect

Lock-free code is built on the C++ atomic model. The essential design constraints:

- A data race on a non-atomic object is undefined behavior.

- Atomics provide well-defined inter-thread communication.
- Ordering must be explicit (relaxed, acquire/release, sequentially consistent) and justified.

A lock-free design is fundamentally a design of:

- **atomicity** (which variables are atomic),
- **ordering** (how updates become visible),
- **invariants** (what always holds despite interleavings),
- **lifetime** (who frees memory and when).

18.3.4 Principle 1: Keep the Shared State Minimal and Atomic

Lock-free algorithms succeed when shared state is small and the invariant is simple. The moment you need to update multiple variables consistently, you either:

- introduce complex multi-word protocols (hard),
- or you redesign (often best),
- or you use a lock (often best).

Example: Lock-Free Counter (Atomic, Simple Invariant)

```
#include <atomic>

class Counter {
public:
  void inc() noexcept {
    value_.fetch_add(1, std::memory_order_relaxed);
  }
```

```
  int get() const noexcept {
    return value_.load(std::memory_order_relaxed);
  }

private:
  std::atomic<int> value_{0};
};
```

Design note. For statistics counters, relaxed ordering is often sufficient: you want atomicity, not ordering of other data.

18.3.5 Principle 2: Prefer Acquire/Release to Express Publication

A common lock-free pattern is **publication**: one thread prepares data, then publishes it so other threads can safely read it.
Acquire/release ordering is designed for this.

Example: Publish a Pointer Safely

```
#include <atomic>
#include <memory>

struct Data { int x; };

class Publisher {
public:
  void publish(std::shared_ptr<const Data> d) {
    std::atomic_store_explicit(&p_, std::move(d), std::memory_order_release);
  }

  std::shared_ptr<const Data> consume() const {
```

```
    return std::atomic_load_explicit(&p_, std::memory_order_acquire);
  }

private:
  std::shared_ptr<const Data> p_;
};
```

Design effect. The acquire load ensures the consumer observes a fully initialized object as published by the release store.

18.3.6 Principle 3: Design for Contention and Backoff

Lock-free algorithms often use loops around `compare_exchange`. Under contention, repeated failures can waste CPU.
A robust design includes:

- minimizing contention (sharding, per-thread batching),
- backoff strategies (yield, pause, exponential backoff),
- bounding retries where possible.

CAS Loop Skeleton With Cooperative Backoff

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

class SpinFlag {
public:
  bool try_set() noexcept {
    bool expected = false;
    return flag_.compare_exchange_strong(expected, true,
                                         std::memory_order_acq_rel,
```

```
                                        std::memory_order_relaxed);
  }

  void set_with_backoff() {
    int spins = 0;
    while (!try_set()) {
      if (++spins % 64 == 0) std::this_thread::yield();
    }
  }

  void clear() noexcept {
    flag_.store(false, std::memory_order_release);
  }

private:
  std::atomic<bool> flag_{false};
};
```

Design note. This is not a mutex replacement; it illustrates that lock-free code must still manage contention.

18.3.7 Principle 4: Define Linearization Points

Lock-free correctness requires a precise statement of:

- what each operation appears to do atomically from the outside,
- and **where** that atomic effect occurs (the linearization point).

For CAS-based operations, the successful CAS is typically the linearization point. If you cannot state this clearly, the design is not ready.

18.3.8 Principle 5: Memory Reclamation Is Part of the Algorithm

In lock-based structures, a thread holding the lock can safely delete nodes because no other thread can access the structure concurrently.
In lock-free structures, other threads may still hold references to nodes. Therefore, freeing memory becomes a first-class problem.
Common reclamation strategies include:

- reference counting (often too expensive under contention),
- hazard pointers (threads publish which nodes they might access),
- epoch-based reclamation (defer frees until all threads pass a quiescent state),
- specialized RCU-like patterns for read-mostly structures.

If you ignore reclamation, you either leak memory or risk use-after-free.

A Minimal Read-Mostly Pattern Using Immutable Snapshots

This is a practical alternative to lock-free mutation:

- make data immutable once published,
- publish a new snapshot when updates occur,
- let readers keep old snapshots alive safely.

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

struct Table {
  std::vector<int> v;
```

```
};

class SnapshotStore {
public:
  void publish(std::shared_ptr<const Table> t) {
    std::atomic_store_explicit(&cur_, std::move(t), std::memory_order_release);
  }

  std::shared_ptr<const Table> current() const {
    return std::atomic_load_explicit(&cur_, std::memory_order_acquire);
  }

private:
  std::shared_ptr<const Table> cur_;
};
```

Design effect. This avoids complex reclamation while providing lock-free reads.

18.3.9 Principle 6: Prefer Compositional Designs Over "One Big Lock-Free Structure"

Lock-free structures are hard to compose because:

- each structure has its own invariants and reclamation strategy,
- combined operations across structures often need multi-object atomicity,
- memory ordering becomes difficult to reason about across modules.

Architectural guidance:

- isolate lock-free components behind narrow interfaces,
- keep them local to performance-critical hot paths,

- avoid exposing internal atomics and node lifetimes to the rest of the system.

18.3.10 Principle 7: Do Not Use Lock-Free to Avoid Thinking About Locks

Locks are not evil; unstructured locking is. The decision should be based on:

- measured contention,
- latency requirements,
- platform constraints,
- verification cost and maintenance risk.

A well-designed lock-based system can outperform a poorly designed lock-free system, and it is almost always easier to verify.

18.3.11 A Concrete Lock-Free Example: Treiber Stack (Core Idea)

The Treiber stack demonstrates the typical CAS-based lock-free pattern. This example focuses on structure; production use requires a reclamation strategy.

```
#include <atomic>
#include <utility>

template<class T>
class TreiberStack {
  struct Node {
    T value;
    Node* next;
  };

public:
```

```
  void push(T v) {
    Node* n = new Node{std::move(v), nullptr};
    Node* old = head_.load(std::memory_order_relaxed);
    do {
      n->next = old;
    } while (!head_.compare_exchange_weak(old, n,
                                          std::memory_order_release,
                                          std::memory_order_relaxed));
  }

  bool pop(T& out) {
    Node* old = head_.load(std::memory_order_acquire);
    while (old) {
      Node* next = old->next;
      if (head_.compare_exchange_weak(old, next,
                                      std::memory_order_acq_rel,
                                      std::memory_order_acquire)) {
        out = std::move(old->value);
        // WARNING: deleting old here is unsafe without reclamation.
        // delete old;
        return true;
      }
    }
    return false;
  }

private:
  std::atomic<Node*> head_{nullptr};
};
```

Design warning. The algorithm illustrates atomic head updates and linearization at successful CAS. But safe memory reclamation must be designed before this can be production-safe.

18.3.12 Practical Lock-Free Design Checklist

1. Identify the exact performance problem and measure lock contention first.
2. Keep shared mutable state minimal; prefer simple invariants.
3. Define linearization points for each operation.
4. Choose the weakest correct memory ordering; justify acquire/release boundaries.
5. Design contention strategy: sharding, batching, backoff.
6. Make memory reclamation part of the design, not an afterthought.
7. Encapsulate lock-free components behind narrow interfaces.
8. Prove correctness at the algorithm level; test to validate, not to discover logic.

18.3.13 Summary

Lock-free programming is the art of achieving concurrency progress guarantees without mutual exclusion, using atomics and carefully designed invariants.
Its benefits are real:

- reduced worst-case latency due to avoiding lock convoying,
- improved throughput for certain contention patterns,
- scalable read-mostly designs with snapshot publication.

Its costs are also real:

- complex correctness arguments,
- subtle memory ordering requirements,

- difficult memory reclamation,
- higher maintenance and review burden.

In long-lived modern C++ systems, lock-free is justified when it is measured, encapsulated, and engineered with explicit progress, ordering, and lifetime guarantees.

18.4 Structured Concurrency

18.4.1 What "Structured" Means in Concurrency

Structured concurrency is the design principle that concurrent work must be organized so that its lifetime is **bounded by lexical scope** and its relationships are explicit.
In unstructured concurrency, code can spawn background work that outlives the scope that created it:

- detached threads,
- fire-and-forget tasks,
- callbacks stored in global queues,
- implicit background operations with unclear ownership.

This creates the classic long-lived-system failures:

- leaked work (tasks never joined or cancelled),
- use-after-free (work captures references to objects that died),
- shutdown hangs (threads still running at program exit),
- unbounded fan-out (spawning without limits),

- error handling that disappears into background execution.

Structured concurrency treats concurrency as a resource that must be managed like memory:

> If you start concurrent work in a scope, you must complete it, cancel it, or account for it before leaving that scope.

18.4.2 The Architectural Objectives

Structured concurrency aims to achieve:

- **Bounded lifetime:** tasks cannot outlive their owner scope.
- **Explicit ownership:** a scope owns its child tasks.
- **Exception and error propagation:** failures are observed at the parent boundary.
- **Cancellation as a design feature:** parent cancellation propagates to children.
- **Deterministic shutdown:** leaving a scope implies a clean join or cancellation.
- **Concurrency limits:** fan-out becomes controllable and auditable.

In architecture terms, this is a move from “spawn threads” to “build a concurrency tree”.

18.4.3 The Canonical Anti-Pattern: Detached Work

```
#include <thread>
#include <string>

void fire_and_forget(std::string msg) {
  std::thread([m = std::move(msg)] {
    // work continues after caller returns
    // error handling is unclear
```

```
    // lifetime of captured resources may be unsafe
    (void)m;
  }).detach();
}
```

This pattern is dangerous because:

- no join exists, so shutdown is not deterministic,
- there is no structured error boundary,
- cancellation is absent,
- resource ownership is unclear.

Structured concurrency replaces detach with scoped ownership.

18.4.4 A Practical C++ Foundation: Scope-Bound Joining

Even without a full standardized structured concurrency framework, C++ can enforce basic structure via RAII:

- create threads/tasks,
- ensure join happens when the scope ends,
- propagate errors via futures or explicit aggregation.

A Minimal `joining_thread` (RAII Join)

```
#include <thread>
#include <utility>

class joining_thread {
```

```
public:
  joining_thread() = default;

  template<class F, class... Args>
  explicit joining_thread(F&& f, Args&&... args)
    : t_(std::forward<F>(f), std::forward<Args>(args)...) {}

  ~joining_thread() {
    if (t_.joinable()) t_.join();
  }

  joining_thread(joining_thread&&) noexcept = default;
  joining_thread& operator=(joining_thread&& other) noexcept {
    if (this != &other) {
      if (t_.joinable()) t_.join();
      t_ = std::move(other.t_);
    }
    return *this;
  }

  joining_thread(const joining_thread&) = delete;
  joining_thread& operator=(const joining_thread&) = delete;

  std::thread& get() noexcept { return t_; }

private:
  std::thread t_;
};
```

Design effect. Threads become scope-bound. Leaving scope means tasks are completed (joined), eliminating forgotten join paths.

18.4.5 Structured Concurrency Needs Cancellation

Joining alone is insufficient for production systems: a parent scope often must stop children early when:

- a sibling fails,
- a timeout occurs,
- the client cancels a request,
- shutdown begins.

Cancellation should be:

- **cooperative** (tasks check a token),
- **propagating** (parent cancels children),
- **composable** (nested scopes work predictably),
- **scoped** (cancellation has a lifetime).

A Minimal Stop Token Pattern

```
#include <atomic>

class StopToken {
public:
  void request_stop() noexcept { stop_.store(true, std::memory_order_release); }
  bool stop_requested() const noexcept { return stop_.load(std::memory_order_acquire); }

private:
  std::atomic<bool> stop_{false};
};
```

Cooperative Task

```
#include <cstddef>

void work_loop(const StopToken& st) {
  for (std::size_t i = 0; i < 1000000; ++i) {
    if (st.stop_requested()) return;
    // do work chunk...
  }
}
```

Design effect. Tasks become cancellable without unsafe thread termination.

18.4.6 A Structured Task Scope (Join + Cancellation)

A task scope owns child threads and ensures they finish before scope exit. On failure, it can request stop and then join.

A Minimal `task_scope` Sketch

```
#include <vector>
#include <exception>
#include <mutex>
#include <thread>
#include <utility>

class task_scope {
public:
  task_scope() = default;

  ~task_scope() {
    request_stop();
    join();
```

```
  }

  void request_stop() noexcept { stop_.request_stop(); }

  template<class F>
  void spawn(F&& f) {
    threads_.emplace_back([this, ff = std::forward<F>(f)]() mutable {
      try {
        ff(stop_);
      } catch (...) {
        record_exception(std::current_exception());
        stop_.request_stop(); // cancel siblings
      }
    });
  }

  void join() {
    for (auto& t : threads_) {
      if (t.joinable()) t.join();
    }
    threads_.clear();
    if (first_exception_) std::rethrow_exception(first_exception_);
  }

private:
  void record_exception(std::exception_ptr ep) {
    std::lock_guard<std::mutex> g(mx_);
    if (!first_exception_) first_exception_ = ep;
  }

  StopToken stop_;
  std::mutex mx_;
  std::exception_ptr first_exception_{};
```

```
  std::vector<std::thread> threads_;
};
```

Design effect.

- Child lifetime is bounded by `task_scope`.
- Exceptions do not disappear; they propagate at `join()` or scope exit.
- A failing child triggers cancellation of siblings.

18.4.7 Structured Concurrency as a Tree

The deeper architectural value is that structured concurrency forms a hierarchy:

- each scope owns child tasks,
- cancellation and errors flow from parent to children and back,
- the system has a well-defined shutdown order.

This is exactly what long-lived systems need: predictable resource cleanup, bounded work, and explicit failure propagation.

18.4.8 The Role of Executors

Structured concurrency becomes robust when tasks run on an executor:

- a thread pool provides bounded parallelism,
- a single-thread executor provides confinement,
- priority executors support QoS boundaries.

Architecturally:

> Structure defines ownership and lifetime; executors define where tasks run.

Even if the executor is a pool, the scope remains responsible for:

- joining/waiting,
- cancellation,
- error aggregation.

18.4.9 Time Bounds: Deadlines and Timeouts

A structured scope should support time bounds:

- tasks can check remaining time,
- cancellation is triggered when deadline expires,
- parent observes timeout as an explicit error path.

In C++ designs, deadlines are typically represented with `std::chrono` time points. The core architectural rule:

> Timeouts belong at boundaries (request handlers, user actions, subsystem calls), not scattered randomly.

18.4.10 Avoiding Concurrency Leaks in API Design

Structured concurrency must be reflected in APIs:

- Avoid returning raw `std::thread` from libraries.
- Avoid fire-and-forget APIs without ownership handles.

- Prefer APIs that return a future or a scoped handle whose destruction joins/cancels.
- Make cancellation explicit: accept a stop token or a cancellation source.
- Make blocking explicit: document which operations wait for children.

18.4.11 Testing Advantages

Structured concurrency improves testing because:

- scopes end, so tests do not leave background work behind,
- errors propagate deterministically to join points,
- concurrency boundaries are explicit, enabling targeted stress tests.

A concurrency architecture that cannot be tested deterministically is not a long-lived design.

18.4.12 Engineering Checklist

1. Eliminate detached threads in production code paths.
2. Ensure every spawned task has an owning scope and a join/cancel policy.
3. Provide cooperative cancellation (stop token) and propagate it down the task tree.
4. Define failure aggregation rules (first failure cancels siblings; collect all failures; etc.).
5. Bound concurrency using executors and queue limits.
6. Keep blocking joins out of event loops and latency-critical threads.
7. Make timeouts explicit at subsystem boundaries.

18.4.13 Summary

Structured concurrency is the discipline of making concurrency obey scope:

- tasks have owners,
- lifetimes are bounded,
- cancellation is cooperative and propagating,
- errors are observed and handled at structured join points,
- shutdown becomes deterministic.

In long-lived modern C++ systems, structured concurrency is not optional. It is the design foundation that prevents background-task leaks, unsafe lifetimes, and unbounded concurrency from becoming permanent architectural debt.

Performance-Driven Design

19.1 Latency vs Throughput

19.1.1 Why This Trade-off Is Architectural

Performance is not a single number. In concurrent systems, the two most important performance objectives are:

- **Latency:** how long a single operation takes from request to completion.
- **Throughput:** how many operations complete per unit of time.

A design that maximizes throughput can easily destroy tail latency. A design that minimizes latency can cap throughput and waste resources. Long-lived systems must treat the latency/throughput decision as a first-class architectural contract, not an implementation detail.

19.1.2 Precise Definitions

Latency

Latency is the end-to-end time for one unit of work. It includes:

- queueing time (waiting to be scheduled),

- service time (actual computation),
- synchronization time (locks, atomics, barriers),
- I/O time (disk, network),
- context switch and wakeup time.

Latency is typically described with percentiles:

- **p50** (median),
- **p90/p95**,
- **p99/p99.9** (tail latency).

Tail latency matters because users experience slow outliers, not averages.

Throughput

Throughput is the system-wide completion rate. It is typically constrained by:

- CPU capacity,
- memory bandwidth,
- cache and NUMA locality,
- lock contention and coherence traffic,
- I/O bandwidth and concurrency limits,
- scheduler and thread-pool behavior.

19.1.3 The Queueing Reality: Why Throughput Can Harm Latency

A fundamental reality of systems is that as utilization approaches 100%, queueing time grows rapidly. In practice:

> If you run a system near saturation to maximize throughput, latency becomes dominated by waiting rather than doing work.

This is why throughput-optimized architectures often show:

- acceptable average latency,
- catastrophic p99 latency under load,
- instability when a small workload spike occurs.

The architectural implication:

> Reserve capacity is not waste; it is latency insurance.

19.1.4 Concurrency and the Latency/Throughput Dial

Concurrency can increase throughput by keeping resources busy, but it also increases latency through:

- queueing in executors,
- contention on shared state,
- context switching overhead,
- cache thrashing and coherence costs,
- head-of-line blocking in queues.

Therefore, concurrency design must choose:

- how many tasks may run in parallel,
- how much work is buffered,
- how requests are prioritized.

19.1.5 Work Granularity: Too Small vs Too Large

Granularity is a key tuning dimension:

- **Too small:** overhead dominates (scheduling, synchronization, allocation).
- **Too large:** parallelism is underutilized; long tasks block others and harm tail latency.

A stable architecture picks a granularity that:

- amortizes overhead,
- supports load balancing,
- avoids long critical sections.

19.1.6 The Core Design Patterns

Pattern 1: Batching (Throughput up, Latency up)

Batching reduces per-item overhead by processing many items together:

- fewer lock acquisitions,
- fewer syscalls,
- better cache locality,

- higher throughput.

But batching adds waiting: items sit in a batch until it is full or a timeout triggers.

```
#include <vector>
#include <chrono>

class Batcher {
public:
  explicit Batcher(std::size_t max_batch) : max_(max_batch) {}

  void push(int x) {
    buf_.push_back(x);
    if (buf_.size() >= max_) flush();
  }

  void flush() {
    // process buf_ as one unit (placeholder)
    buf_.clear();
  }

private:
  std::size_t max_;
  std::vector<int> buf_;
};
```

Architectural rule. Batching must be bounded by size and by time. Otherwise it destroys latency.

Pattern 2: Prioritization (Latency down for important work)

When not all work is equal, you can protect latency for critical requests by prioritization:

- control-plane messages over data-plane messages,

- interactive requests over background maintenance,
- small requests over large requests (size-aware scheduling).

```
#include <queue>
#include <optional>

template<class T>
class PriorityQueue2 {
public:
  void push_high(T x) { hi_.push(std::move(x)); }
  void push_low(T x)  { lo_.push(std::move(x)); }

  std::optional<T> try_pop() {
    if (!hi_.empty()) { T v = std::move(hi_.front()); hi_.pop(); return v; }
    if (!lo_.empty()) { T v = std::move(lo_.front()); lo_.pop(); return v; }
    return std::nullopt;
  }

private:
  std::queue<T> hi_;
  std::queue<T> lo_;
};
```

Architectural rule. Prioritization must be paired with fairness rules to avoid starving low-priority work.

Pattern 3: Bounding Concurrency (Tail latency protection)

Unlimited concurrency typically increases queueing and contention. A bounded thread pool or executor can protect latency by limiting in-flight work.

```
#include <semaphore>
```

```
class InFlightLimiter {
public:
  explicit InFlightLimiter(std::ptrdiff_t limit) : sem_(limit) {}

  template<class F>
  void run(F&& f) {
    sem_.acquire();
    try {
      f();
    } catch (...) {
      sem_.release();
      throw;
    }
    sem_.release();
  }

private:
  std::counting_semaphore<> sem_;
};
```

Design effect. Bounding in-flight work reduces overload amplification and helps stabilize tail latency.

Pattern 4: Backpressure (Throughput stability, latency predictability)

When producers can outpace consumers, you must define behavior:

- block producer,
- drop requests,
- shed load,

- coalesce updates,
- degrade features gracefully.

A bounded queue is a backpressure mechanism. Without it, latency becomes unbounded.

19.1.7 Synchronization Costs: When a Lock Becomes a Latency Source

Mutex contention is a throughput and latency killer:

- it serializes critical sections,
- it can cause convoy effects,
- it increases tail latency under load.

Design strategies:

- reduce shared mutable state,
- shard state (per-core/per-thread),
- use read-mostly snapshots for configuration,
- keep critical sections small and free of I/O,
- avoid calling unknown code under locks.

19.1.8 CPU Caches and False Sharing

Throughput can drop and latency can spike when multiple threads write to the same cache line.
False sharing is a design failure of memory layout.
Common causes:

- per-thread counters in contiguous arrays,

- queue head/tail indices adjacent in memory,
- frequently updated flags near hot data.

Design strategies:

- partition write-heavy data per thread,
- pad or align hot fields when justified,
- prefer message passing or batching to reduce write frequency.

19.1.9 Measurement That Matches the Goal

A performance-driven design must measure the right thing:

- If you optimize for **throughput**, measure sustained operations/sec under realistic load.
- If you optimize for **latency**, measure percentiles (p50/p95/p99) under controlled load.
- Always measure under **different utilization levels** (50%, 70%, 90%).

A system that looks good only at 95–100% utilization is often fragile in production.

A Minimal Latency Measurement Skeleton

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

template<class F>
auto measure_many(std::size_t n, F&& f) {
  using clock = std::chrono::steady_clock;
  std::vector<long long> us;
```

```
  us.reserve(n);

  for (std::size_t i = 0; i < n; ++i) {
    auto t0 = clock::now();
    f();
    auto t1 = clock::now();
    auto dt = std::chrono::duration_cast<std::chrono::microseconds>(t1 - t0).count();
    us.push_back(dt);
  }

  std::sort(us.begin(), us.end());
  auto p = [&](double q) -> long long {
    std::size_t idx = static_cast<std::size_t>(q * (us.size() - 1));
    return us[idx];
  };

  return std::tuple{p(0.50), p(0.95), p(0.99)};
}
```

Design note. Real systems require careful benchmarking methodology, but the core idea remains: percentiles matter more than averages.

19.1.10 A Decision Framework for Architects

When Latency Dominates

Optimize for latency when:

- the system is interactive (UI, RPC, trading, real-time control),
- tail latency is a product requirement,
- long pauses are unacceptable (stop-the-world events, lock convoys),

- the workload is bursty and must remain responsive.

Typical architectural choices:

- bounded queues and backpressure,
- prioritization and fairness,
- avoiding global locks,
- structured concurrency and cancellation,
- reserving headroom (avoid saturation).

When Throughput Dominates

Optimize for throughput when:

- the system is batch-oriented (ETL, offline processing),
- average completion time matters more than worst-case,
- work can be batched and parallelized,
- resource efficiency is important.

Typical architectural choices:

- batching and vectorization,
- maximizing CPU utilization,
- large pipeline buffers (but still bounded),
- throughput-oriented schedulers.

19.1.11 Engineering Checklist

1. Define the performance contract: latency percentiles, throughput targets, and acceptable headroom.
2. Select a concurrency model that matches the contract (bounded pools, message passing, snapshots).
3. Bound concurrency and buffering to prevent overload-driven latency collapse.
4. Use batching carefully: bound by size and by time.
5. Reduce shared mutable state to control contention.
6. Measure percentiles under varying load; do not optimize only for the average.

19.1.12 Summary

Latency and throughput are competing objectives that must be balanced explicitly.

- **Latency** is dominated by queueing, contention, and outliers.
- **Throughput** is dominated by utilization, partitioning, and overhead amortization.

A performance-driven C++ architecture:

- defines targets in terms of both throughput and latency percentiles,
- enforces bounded concurrency and backpressure,
- avoids saturation to protect tail latency,
- applies batching and parallelism only with measured justification.

19.2 Cache-Aware Design

19.2.1 Why Caches Are a Design Constraint

On modern CPUs, the dominant performance factor is rarely arithmetic. It is **data movement**:

- from memory to cache,
- between cache levels,
- and between cores through cache-coherence protocols.

Cache-aware design is therefore not micro-optimization. It is a design discipline that shapes:

- data structures (layout and locality),
- algorithms (access patterns),
- concurrency (ownership, false sharing, coherence traffic),
- batching and pipelining (amortizing latency).

In performance-driven systems, you do not just ask:

> How many operations does this do?

You ask:

> How many cache lines does this touch, and how often does it force coherence traffic?

19.2.2 The Practical Cache Model for Designers

While cache hierarchies differ, the design-relevant properties are stable:

- **Cache lines:** memory is transferred in fixed-size blocks (cache lines).

- **Spatial locality:** contiguous access is fast because whole lines are fetched.
- **Temporal locality:** reusing recently accessed data is fast because it remains cached.
- **Coherence:** writes by one core invalidate or update cache lines on other cores.
- **Bandwidth vs latency:** caches provide both lower latency and higher bandwidth than DRAM.

The key design implication:

> If your working set fits in cache and your access is sequential, your system can be dramatically faster. If it misses cache frequently or bounces cache lines between cores, no amount of threading saves you.

19.2.3 The Two Golden Rules

Rule 1: Prefer Contiguous Storage for Hot Data

Contiguous layouts exploit spatial locality and reduce pointer chasing. In many designs, using a contiguous container for the hot path yields the largest speedup.

Rule 2: Minimize Cross-Core Writes to the Same Cache Line

Cross-core writes create coherence traffic. Even when each thread writes to "its own variable", if those variables share a cache line, performance collapses due to **false sharing**.

19.2.4 Data Layout: AoS vs SoA

Array of Structs (AoS)

AoS stores records contiguously as full objects. It is good when you typically use most fields of each record together.

Struct of Arrays (SoA)

SoA stores each field as a separate contiguous array. It is good when you process one or a few fields across many elements, enabling tight loops and better vectorization.

Example: AoS and SoA Side by Side

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

struct ParticleAoS {
  float x, y, z;
  float vx, vy, vz;
};

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

float sum_x_aos(const std::vector<ParticleAoS>& p) {
  float s = 0.0f;
  for (const auto& e : p) s += e.x; // pulls full struct lines
  return s;
}

float sum_x_soa(const ParticlesSoA& p) {
  float s = 0.0f;
  for (std::size_t i = 0; i < p.x.size(); ++i) s += p.x[i]; // tight contiguous stream
  return s;
}
```

Design implication. If your hot loop touches only `x`, SoA reduces wasted cache bandwidth and improves prefetching.

19.2.5 Pointer Chasing vs Streaming Access

Linked structures (lists, trees with pointers) often cause cache misses because:

- nodes are allocated across the heap,
- each pointer jump is a potential cache miss,
- hardware prefetchers cannot predict irregular pointer patterns.

Streaming arrays enable:

- automatic prefetching,
- fewer misses,
- better SIMD and pipeline utilization.

Design Practice

- Prefer `std::vector` for hot collections.
- Use node-based containers when you need stable references/iterators or cheap splicing, and keep them off the hot path if possible.
- Consider contiguous “node pools” when you need pointer semantics but want locality.

19.2.6 Working Set Size and Hot/Cold Splitting

A design often fails because hot and cold fields are stored together. If cold fields are rarely used but large, they poison cache lines.
Technique: **hot/cold split**

- store frequently used fields in a compact "hot" struct,
- store infrequently used fields in a separate "cold" structure accessed only when needed.

Example: Hot/Cold Split by Indirection

```
#include <string>
#include <memory>

struct UserCold {
  std::string biography;
  std::string notes;
};

struct UserHot {
  int id;
  int age;
  std::shared_ptr<UserCold> cold; // rarely touched on hot path
};
```

Design effect. Hot loops over many users touch only compact hot data, improving cache residency.

19.2.7 False Sharing: The Silent Throughput Killer

False sharing occurs when independent variables updated by different threads occupy the same cache line. Writes cause the entire line to bounce between cores.

Symptoms:

- scaling collapses as threads increase,
- CPU utilization is high but work done per second is low,
- performance improves when threads are pinned or reduced.

A Classic False Sharing Pattern

```
#include <vector>
#include <thread>
#include <atomic>
#include <cstddef>

struct CountersBad {
  std::atomic<long long> c0{0};
  std::atomic<long long> c1{0};
  std::atomic<long long> c2{0};
  std::atomic<long long> c3{0};
};

void bump(std::atomic<long long>& c) {
  for (int i = 0; i < 1000000; ++i)
    c.fetch_add(1, std::memory_order_relaxed);
}
```

Even though each thread updates a distinct atomic, they may share cache lines and cause coherence storms.

Mitigation: Per-Thread Storage and Reduction

The most robust mitigation is architectural: avoid shared writes in the hot path.

```cpp
#include <vector>
#include <thread>
#include <numeric>
#include <cstddef>

long long sum_counts_parallel(std::size_t threads) {
  std::vector<long long> local(threads, 0);

  std::vector<std::thread> ts;
  for (std::size_t i = 0; i < threads; ++i) {
    ts.emplace_back([&, i]{
      long long x = 0;
      for (int k = 0; k < 1000000; ++k) ++x;
      local[i] = x; // one write at end
    });
  }
  for (auto& t : ts) t.join();

  return std::accumulate(local.begin(), local.end(), 0LL);
}
```

Design effect. Local counting is cache-friendly; reduction is performed once, minimizing coherence traffic.

19.2.8 Cache Coherence and Shared Mutable State

Cache-aware concurrency design prefers:

- **read sharing** (multiple readers of immutable data),
- **write confinement** (one owner thread for mutable data),
- **sharding** (partition state so each thread/core writes mostly to its own shard),

- **immutable snapshots** (publish a new version; readers never contend).

A shared global lock protects correctness but may still harm performance: it serializes and also increases cache line bouncing due to shared writes.

19.2.9 Prefetching and Access Regularity

Hardware prefetchers are effective on predictable access patterns:

- sequential loops,
- fixed strides,
- stable iteration orders.

They are ineffective on:

- pointer chasing,
- random access,
- unpredictable branches.

Design implication:

> Choose algorithms and data structures that make access predictable on the hot path.

19.2.10 Allocation Strategy and Locality

Dynamic allocation can fragment memory and reduce locality. Cache-aware designs often:

- reduce allocations in hot loops,
- use arenas/pools for hot node allocations,

- reuse buffers (object pools) where ownership is clear,
- keep lifetimes grouped (allocate and free in batches).

In concurrency, allocator contention can also become a bottleneck; per-thread allocators or arenas can reduce shared allocator traffic.

19.2.11 NUMA Awareness (When It Matters)

On NUMA systems, “memory locality” extends beyond caches:

- local memory accesses are faster than remote node memory,
- cross-socket coherence traffic is expensive,
- thread placement and memory placement matter.

Design strategies:

- shard data per NUMA node,
- avoid frequent cross-node writes,
- keep worker threads close to their data,
- reduce global shared structures.

19.2.12 A Practical Cache-Aware Design Checklist

1. Identify hot paths and hot data (the small fraction executed frequently).
2. Make hot data compact and contiguous; split cold fields away.
3. Prefer streaming access patterns; minimize pointer chasing in hot loops.

4. Avoid shared writes; use per-thread/per-core shards and reduce later.
5. Watch for false sharing in per-thread counters, queues, and flags.
6. Bound synchronization: short critical sections, no I/O under locks.
7. Reduce allocation and fragmentation on the hot path; consider arenas for hot objects.
8. Measure: cache misses, branch misses, and scaling under thread count.

19.2.13 Summary

Cache-aware design recognizes that performance is dominated by locality and coherence:

- contiguous layouts and predictable access exploit caches and prefetching,
- shared mutable state creates coherence storms and destroys scaling,
- false sharing can collapse throughput even when logic is correct,
- hot/cold splitting and sharding are architectural optimizations, not micro tricks.

In performance-driven modern C++ systems, the best concurrency design is often the most cache-friendly: own your data, keep it local, minimize shared writes, and structure work to stream through memory efficiently.

19.3 Memory Layout as Architecture

19.3.1 The Architectural Claim

In performance-driven systems, **memory layout is architecture**. It determines:

- which invariants can be enforced efficiently,

- which concurrency model scales,
- how data moves through caches and between cores,
- how often the allocator is invoked,
- how easy it is to reason about ownership and lifetime.

If architecture defines the system's components and responsibilities, memory layout defines the system's *physical reality*: how information is stored, accessed, mutated, and shared.
Ignoring layout leads to predictable long-lived-system outcomes:

- slow hot paths due to cache misses and pointer chasing,
- scaling collapse due to coherence traffic and false sharing,
- fragile lifetime management due to scattered allocations,
- excessive memory usage and fragmentation,
- unpredictable latency due to allocation and page faults.

Therefore, memory layout must be designed deliberately, not inherited accidentally.

19.3.2 The Design Surface of Memory Layout

A layout design answers the following architectural questions:

1. **What is hot and what is cold?**
2. **What is read-mostly and what is write-heavy?**
3. **What is shared across threads and what is confined?**
4. **What must be contiguous for fast iteration?**

5. **What must have stable addresses?**

6. **What is the lifetime grouping of objects?**

The answers determine the appropriate data structures, allocation strategy, and ownership model.

19.3.3 Hot/Cold Splitting as an Architectural Pattern

When hot and cold fields share cache lines, cold data pollutes caches. A stable architecture splits them.

Example: Hot/Cold Split

```
#include <string>
#include <memory>

struct UserCold {
  std::string biography;
  std::string notes;
  std::string address;
};

struct UserHot {
  int id;
  int age;
  int flags;
  std::shared_ptr<UserCold> cold; // accessed only on cold path
};
```

Architectural effect. Hot loops over many users touch only `id`, `age`, and `flags`, improving cache residency and throughput.

19.3.4 Value Layout vs Reference Layout

A core architectural choice is whether your system is built from:

- **values** stored contiguously (tight, cache-friendly, copy/move semantics),
- **references** to heap nodes (flexible topology, stable addresses, pointer chasing).

Value-Oriented Layout (Contiguous)

```
#include <vector>

struct Order {
  int id;
  double price;
};

std::vector<Order> orders; // tight layout, fast iteration
```

Reference-Oriented Layout (Node-Based)

```
#include <memory>

struct Node {
  int id;
  double price;
  std::unique_ptr<Node> next;
};

std::unique_ptr<Node> head; // pointer chasing, stable nodes
```

Architectural rule. Use reference-oriented layouts only when you need:

- stable addresses across insert/erase,

- splicing and node-level operations,
- graph-like topology.

Otherwise, prefer contiguous value layouts for hot paths.

19.3.5 AoS vs SoA Is an Architectural API Decision

AoS (array of structs) and SoA (struct of arrays) differ in:

- cache utilization,
- vectorization opportunities,
- API shape and abstraction boundaries.

AoS and SoA

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

struct ParticleAoS { float x,y,z; float vx,vy,vz; };

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

float integrate_aos(std::vector<ParticleAoS>& p) {
  float s = 0;
  for (auto& e : p) { e.x += e.vx; s += e.x; }
  return s;
}
```

```
float integrate_soa(ParticlesSoA& p) {
  float s = 0;
  for (std::size_t i = 0; i < p.x.size(); ++i) { p.x[i] += p.vx[i]; s += p.x[i]; }
  return s;
}
```

Architectural effect. SoA makes it natural to process one field across many elements and can reduce wasted cache bandwidth.

19.3.6 Alignment, Padding, and the Cost of "Accidental Coupling"

Memory layout is shaped by:

- alignment requirements,
- padding between fields,
- object size and cache-line occupancy,
- placement of frequently written fields.

A common architectural failure is accidental coupling: two fields unrelated in semantics share cache lines and cause performance interference, especially across threads.

Write-Hot Fields Should Be Isolated

```
#include <atomic>

struct Metrics {
  std::atomic<long long> requests{0}; // write-hot
  std::atomic<long long> errors{0};   // write-hot
  // if many threads update these, they may contend on cache lines
};
```

The design mitigation is not just padding. The architectural mitigation is:

- per-thread counters,
- sharded counters (per core),
- periodic aggregation.

19.3.7 False Sharing as an Architectural Bug

False sharing occurs when independent thread-local variables share cache lines. It is an architectural bug because it emerges from layout decisions, not algorithm decisions.

Mitigation: Shard, Then Reduce

```
#include <vector>
#include <thread>
#include <numeric>
#include <cstddef>

long long count_requests(std::size_t threads) {
  std::vector<long long> local(threads, 0);

  std::vector<std::thread> ts;
  for (std::size_t i = 0; i < threads; ++i) {
    ts.emplace_back([&, i]{
      long long x = 0;
      for (int k = 0; k < 1000000; ++k) ++x;
      local[i] = x; // one store
    });
  }
  for (auto& t : ts) t.join();
```

```
  return std::accumulate(local.begin(), local.end(), 0LL);
}
```

Architectural effect. By removing shared writes in the hot loop, coherence traffic is minimized.

19.3.8 Lifetime Grouping: Allocation Strategy as Architecture

Layout is also about **allocation topology**. Objects allocated together tend to be close in memory. Objects freed together should be allocated together.
Architectural patterns:

- **arena allocation:** allocate many related objects from one arena and free them together,
- **pool allocation:** fixed-size node pools for stable nodes and improved locality,
- **buffer reuse:** reusing memory to avoid allocator overhead on hot paths.

Arena Sketch (Conceptual)

```
#include <vector>
#include <cstddef>
#include <new>
#include <utility>

class Arena {
public:
  explicit Arena(std::size_t cap) : buf_(cap), pos_(0) {}

  void* allocate(std::size_t n, std::size_t align) {
    std::size_t p = pos_;
    std::size_t aligned = (p + (align - 1)) & ~(align - 1);
    if (aligned + n > buf_.size()) throw std::bad_alloc{};
```

```
    pos_ = aligned + n;
    return buf_.data() + aligned;
  }

  void reset() noexcept { pos_ = 0; } // free all at once

private:
  std::vector<std::byte> buf_;
  std::size_t pos_;
};
```

Architectural effect. Arena allocation turns "many small lifetimes" into "one grouped lifetime", which improves locality and reduces allocator contention.

19.3.9 APIs That Respect Layout: Views and Spans

A layout-aware architecture often uses non-owning views:

- `std::span` for contiguous memory,
- string views for immutable text,
- iterator ranges for algorithms.

The design goal is to:

- avoid copying large buffers,
- preserve contiguity,
- keep ownership separate from access.

Example: Accepting `std::span`

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

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

Architectural effect. The API encourages contiguous layouts without forcing a specific container type.

19.3.10 Concurrency Model Follows Layout

Memory layout choices constrain concurrency:

- Contiguous, partitionable data supports data-parallel processing and sharding.
- Shared mutable node graphs create synchronization and coherence challenges.
- Immutable snapshots support lock-free reads with stable layout.
- Confinement plus message passing keeps mutable layout local to one thread.

A useful architectural rule:

> Choose a memory layout that makes the intended concurrency model natural.

19.3.11 Design Checklist: Layout-Driven Architecture

1. Identify hot data and hot loops; make them compact and contiguous.

2. Separate cold fields; avoid cache pollution.
3. Prefer value layouts for hot paths; use node layouts only when topology requires it.
4. Choose AoS vs SoA based on access patterns and vectorization needs.
5. Remove shared writes from hot loops; shard and reduce to avoid false sharing.
6. Group lifetimes with arenas/pools when allocation dominates or locality is poor.
7. Design APIs that preserve contiguity using spans/views and avoid unnecessary copies.
8. Verify by measurement: cache misses, branch misses, and scaling with thread count.

19.3.12 Summary

Memory layout is architecture because it determines the physical constraints under which the software runs:

- cache locality and prefetching,
- coherence traffic and scalability,
- allocation overhead and fragmentation,
- lifetime clarity and ownership enforcement.

A performance-driven modern C++ system designs layout explicitly: hot/cold separation, contiguous storage for hot paths, sharding to reduce shared writes, and allocation strategies that reflect lifetime grouping. When layout matches the concurrency model, the system becomes both faster and more maintainable over time.

19.4 Measurement-Driven Decisions

19.4.1 Performance Without Measurement Is Fiction

Performance-driven design is not “writing fast code.” It is the discipline of making architectural decisions using evidence.
In long-lived systems, the most dangerous performance failures are:

- optimizing the wrong bottleneck,
- improving average performance while destroying tail latency,
- scaling thread count while increasing contention and cache misses,
- making micro-optimizations that are irrelevant at system level,
- relying on synthetic benchmarks that do not resemble production traffic.

The fundamental rule:

> If you cannot measure it reliably, you cannot claim you improved it.

19.4.2 Define the Performance Contract First

Before measuring, define what “good” means. A production-ready contract typically includes:

- **throughput target:** operations per second under a specified workload,
- **latency targets:** p50/p95/p99 under specified utilization,
- **resource budgets:** CPU, memory, and I/O limits,
- **scaling requirements:** behavior as cores increase (1, 2, 4, 8, 16, ...),

- **stability requirements:** performance under bursts, failures, and partial degradation.

A contract without percentiles is incomplete. A contract without workload definition is meaningless.

19.4.3 Measure What the User Experiences

The system is not measured by:

- the fastest path,
- the best-case benchmark,
- the micro-benchmark loop in isolation.

It is measured by:

- the end-to-end request,
- the tail latency under realistic load,
- the throughput at the point where the system remains stable.

Therefore, measurement must cover:

- end-to-end timing,
- queueing delays,
- time spent waiting on locks,
- CPU time vs wall time,
- cache and memory behavior,
- thread scheduling behavior.

19.4.4 Latency: Percentiles, Not Averages

Average latency hides outliers. A system can have a good mean and still be unusable if p99 is poor.

A practical measurement set:

- p50 (typical),
- p95 (high load behavior),
- p99 (tail),
- p99.9 (critical services).

A Minimal Percentile Collector

```
#include <vector>
#include <algorithm>
#include <chrono>
#include <cstddef>

struct Percentiles {
  long long p50;
  long long p95;
  long long p99;
};

template<class F>
Percentiles measure_latency(std::size_t n, F&& f) {
  using clock = std::chrono::steady_clock;
  std::vector<long long> us;
  us.reserve(n);

  for (std::size_t i = 0; i < n; ++i) {
```

```
    auto t0 = clock::now();
    f();
    auto t1 = clock::now();
    auto dt = std::chrono::duration_cast<std::chrono::microseconds>(t1 - t0).count();
    us.push_back(dt);
  }

  std::sort(us.begin(), us.end());

  auto at = [&](double q) -> long long {
    std::size_t idx = static_cast<std::size_t>(q * (us.size() - 1));
    return us[idx];
  };

  return Percentiles{at(0.50), at(0.95), at(0.99)};
}
```

Engineering note. This illustrates the idea. Real measurement must include warmup, stable clocks, and careful control of noise sources.

19.4.5 Throughput: Sustain Under Load, Not Burst

Throughput must be measured as **sustained** completion rate under a stable workload:

- measure long enough to smooth noise,
- ensure the system is not just consuming a queue built up earlier,
- ensure steady-state resource usage.

A critical principle:

> If throughput improves only by letting queues grow, you did not improve throughput; you postponed latency.

19.4.6 The Overload Curve: Measure Across Utilization Levels

Performance must be evaluated across multiple load levels:

- 30–50% utilization: baseline behavior,
- 60–80%: typical production range,
- 85–95%: stress range where queueing dominates,
- near saturation: often unstable and misleading.

Many architectures look fine at 50% and collapse at 85%. If you never measure the high-utilization region, you will discover failure in production.

19.4.7 Design Decisions That Require Measurement

Some decisions cannot be made reliably by intuition:

1) Thread Count and Pool Size

More threads can:

- increase throughput (better utilization),
- or decrease throughput (context switching, contention),
- and increase latency (queueing, lock convoying).

Measure scaling across thread counts and choose the stable region.

2) Locking Strategy

Mutex vs sharding vs atomics vs message passing is workload-dependent. Measure:

- contention rate,
- time blocked on locks,
- tail latency at high load.

3) Memory Layout Choices

AoS vs SoA, hot/cold split, node vs contiguous containers must be validated by:

- cache miss rates,
- memory bandwidth consumption,
- throughput scaling.

4) Batching and Buffering

Batching improves throughput but increases latency. The right batch size is never a guess: it is chosen by measuring the latency/throughput curve.

19.4.8 Benchmarking Methodology: Avoid Self-Deception

Measurement must be designed to reduce bias and noise.

Warmup and Steady State

Modern systems have:

- CPU frequency scaling,

- cache warmup effects,
- allocator and page cache behavior,
- branch predictor training.

Therefore:

- run warmup iterations before recording,
- measure long enough to reach steady state,
- separate initialization time from steady-state performance.

Control Input and Randomness

Use reproducible inputs:

- fixed seeds for random generation,
- representative distributions (not just uniform),
- realistic payload sizes and request mixes.

Avoid Measuring Under Debug or Instrumented Builds Only

Debug builds distort performance drastically. A correct methodology includes:

- optimized builds for performance numbers,
- debug builds for correctness validation,
- consistent compiler flags and CPU affinity when required.

19.4.9 Instrumentation: Knowing Where Time Went

End-to-end metrics tell you *that* you are slow. Instrumentation tells you *why*.
Measure:

- queue wait time (executor queue, message queue),
- service time (actual work),
- lock wait time (time blocked),
- allocation time (malloc/new hotspots),
- cache misses and branch mispredicts (hardware counters),
- context switches and CPU migrations.

A performance-driven architecture typically builds:

- lightweight timers around major stages,
- counters for queue depth and contention,
- structured logs for rare slow requests.

19.4.10 A Practical Profiling Strategy (Architecture-Level)

A stable approach in long-lived systems:

1. **Measure end-to-end:** confirm the symptom (latency percentiles, throughput).
2. **Locate the bottleneck:** CPU, I/O, locks, allocator, cache misses, scheduler.
3. **Form a hypothesis:** a specific change with a predicted impact.

4. **Change one thing:** avoid confounded experiments.
5. **Re-measure:** confirm improvement and check regressions elsewhere.
6. **Validate under multiple loads:** ensure you did not just move the failure point.

19.4.11 A/B and Regression Testing for Performance

Long-lived systems require performance regression protection. This is architectural hygiene:

- maintain baseline benchmarks,
- run them regularly,
- compare percentiles and throughput,
- fail builds on significant regressions,
- track performance trends over time.

19.4.12 From Measurements to Decisions: A Decision Table

If p99 Latency Is Bad Under Load

Likely causes:

- queueing and overload,
- lock contention and convoys,
- GC-like pauses (in non-C++ subsystems) or allocator stalls,
- cache thrashing and cross-core migrations,
- I/O stalls.

Design responses:

- add backpressure and bound queues,
- reduce shared mutable state; shard or confine,
- avoid blocking under locks; shorten critical sections,
- isolate hot data and reduce cache misses,
- apply timeouts and cancellation to prevent request pileups.

If Throughput Does Not Scale With Cores

Likely causes:

- global locks or shared write-hot cache lines,
- false sharing,
- allocator contention,
- memory bandwidth saturation,
- excessive synchronization and context switching.

Design responses:

- shard state and reduce shared writes,
- use per-thread arenas or reduce allocations on hot paths,
- improve locality and reduce pointer chasing,
- tune thread pools and avoid oversubscription.

19.4.13 Engineering Checklist

1. Define a performance contract: workload + throughput + latency percentiles + resource budgets.
2. Measure end-to-end first; then instrument stages to locate bottlenecks.
3. Measure across utilization levels; do not trust single-load results.
4. Use warmup and steady-state methodology; control inputs and randomness.
5. Compare changes against baselines; protect against regressions.
6. Prefer changes that improve tail latency and stability, not just averages.

19.4.14 Summary

Measurement-driven design is the discipline that prevents performance work from becoming folklore.

- Define the contract.
- Measure end-to-end and by percentile.
- Identify the bottleneck with instrumentation.
- Change one variable, then re-measure under multiple loads.
- Institutionalize regression testing.

This approach turns performance from a subjective debate into an engineering process, and it is the only reliable path for performance-driven decisions in long-lived modern C++ systems.

Part VII

Architectural Styles in C++

Layered and Modular Architectures

20.1 Traditional Layering Revisited

20.1.1 Why Layering Still Matters (When Done Correctly)

Layered architecture is one of the oldest and most successful structural patterns in software design:

- it limits dependency chaos,
- it localizes change,
- it supports replaceability,
- it enables different skill sets to work independently,
- it creates a vocabulary for discussing boundaries.

Yet many long-lived C++ systems eventually "outgrow" their original layered design. Not because layering is wrong, but because it is applied naively:

- layers become thick and leaky,
- the dependency direction gets violated,

- convenience shortcuts accumulate into permanent coupling,
- performance work punches holes through boundaries,
- concurrency and asynchronous control flow spread across layers without an explicit model.

Revisiting traditional layering in modern C++ means keeping the benefits while fixing the failure modes, using contemporary language and tooling capabilities.

20.1.2 What a Layer Is (Architecturally)

A layer is not a folder. A layer is a **dependency constraint**.
A valid layer definition specifies:

- **responsibility:** what this layer owns and what it must never own,
- **public surface:** the API exposed upward,
- **dependency direction:** what it may depend on,
- **policy vs mechanism:** what is decided here vs delegated,
- **error model:** how failures are represented and propagated,
- **concurrency model:** thread-safety assumptions and ownership.

If these are not explicit, layering becomes a naming convention rather than architecture.

20.1.3 Classical Layering (and Why It Breaks in C++)

The classic stack:

- Presentation (UI/API)
- Application (use cases)

- Domain (business rules)
- Infrastructure (DB, network, OS)

Common failure modes in large C++ systems:

1. **Upward dependency leaks:** lower layers include headers from upper layers.
2. **Domain polluted by infrastructure types:** database, serialization, networking types invade core logic.
3. **Header coupling explosion:** transitive includes force full rebuilds and fragile integration.
4. **Exception model mismatch:** some layers throw, others return error codes, others log-and-continue.
5. **Concurrency ambiguity:** thread-safety and ownership assumptions differ per layer.
6. **Performance tunnel vision:** micro-optimizations break boundaries instead of redesigning interfaces.

Modern layering is about preventing these breakdowns systematically.

20.1.4 Layering in Modern C++: The Updated Rules

Rule 1: Dependencies Must Be Enforced, Not Hoped For

A layered architecture is only real if the build system enforces it:

- each layer builds as an independent target,
- layers link only to allowed lower layers,
- include paths are controlled so forbidden headers cannot be included.

Rule 2: Headers Are Part of the Architecture

In C++, headers define compile-time coupling. Therefore:

- public headers must be minimal and stable,
- internal headers must not leak upward,
- avoid including heavy dependencies in public interfaces,
- prefer forward declarations when possible,
- use the PImpl idiom when ABI stability or dependency isolation is needed.

Rule 3: The Domain Layer Must Be Pure

The domain layer is the most valuable part of a long-lived system. It should:

- avoid framework types,
- avoid OS types and handles,
- avoid serialization formats,
- expose intent-focused types and invariants,
- remain testable without infrastructure.

Rule 4: Invert Dependencies With Interfaces

When the domain needs something from infrastructure, invert the dependency:

- domain defines an abstract interface,
- infrastructure implements it,
- composition happens at the boundary (application wiring).

20.1.5 A Layered Example With Explicit Boundaries

Domain Defines the Interface

```
#pragma once
#include <string>
#include <optional>

struct UserId {
  int value;
};

struct User {
  UserId id;
  std::string name;
};

class IUserRepository {
public:
  virtual ~IUserRepository() = default;
  virtual std::optional<User> find(UserId id) = 0;
  virtual void save(const User& u) = 0;
};
```

Design effect. The domain owns the abstraction. No database types appear here.

Application Layer Defines Use Cases

```
#pragma once
#include <optional>

class GetUserUseCase {
public:
```

```
  explicit GetUserUseCase(IUserRepository& repo) : repo_(repo) {}

  std::optional<User> execute(UserId id) {
    return repo_.find(id);
  }

private:
  IUserRepository& repo_;
};
```

Design effect. Use cases depend on domain abstractions, not on infrastructure.

Infrastructure Implements the Interface

```
#pragma once
#include <unordered_map>

class InMemoryUserRepository final : public IUserRepository {
public:
  std::optional<User> find(UserId id) override {
    auto it = db_.find(id.value);
    if (it == db_.end()) return std::nullopt;
    return it->second;
  }

  void save(const User& u) override {
    db_[u.id.value] = u;
  }

private:
  std::unordered_map<int, User> db_;
};
```

Design effect. Infrastructure depends on domain, not the other way around.

20.1.6 Layering vs Performance: The Modern Resolution

Traditional layering often fails under performance pressure: "we cannot afford abstraction here." In modern systems, the correct resolution is:

- keep the boundary, but change the boundary *shape*,
- design interfaces that allow efficient execution,
- avoid needless allocation and virtual dispatch in hot paths,
- provide bulk APIs, spans, and views for data movement,
- move performance-critical policy downward only when it truly belongs there.

Bulk APIs Reduce Boundary Overhead

```
#include <span>
#include <vector>

class IUserRepositoryBulk {
public:
  virtual ~IUserRepositoryBulk() = default;
  virtual void load_many(std::span<const UserId> ids,
                         std::vector<User>& out) = 0;
};
```

Design effect. The interface remains layered, but becomes cache- and allocation-aware.

20.1.7 Layering vs Concurrency: Make the Model Explicit

Traditional layering assumes synchronous call stacks. Modern systems are often asynchronous:

- thread pools,
- event loops,
- futures and tasks,
- message passing.

A layered architecture must specify:

- which layer owns scheduling decisions (often application/infrastructure boundary),
- which APIs are thread-safe vs thread-compatible vs thread-confined,
- where blocking is allowed (never under locks, never in event loops),
- how cancellation and timeouts propagate across layers.

20.1.8 Revisiting "Strict" Layering: Practical Exceptions

Strict layering says a layer may only call the one below it. In practice, long-lived systems often adopt a **relaxed** but controlled variant:

- layers may call multiple lower layers, but not upward,
- cross-layer calls must go through explicit interfaces,
- shortcuts must be formalized (facades) rather than ad-hoc includes,
- shared utilities must not become a dependency dumping ground.

The key is that any exception must be **designed**, not accidental.

20.1.9 Modern Layering Tools in C++

A modern C++ layered architecture uses language and build mechanisms to reduce coupling:

- **Namespaces** and consistent naming to make boundaries obvious.
- **PImpl** to isolate heavy includes and stabilize ABI where needed.
- **Forward declarations** to prevent include explosions.
- **Non-owning views** such as `std::span` to avoid copying.
- **Modules** (where adopted) to reduce header coupling and speed builds.
- **Separate targets** per layer in the build system to enforce dependencies.

20.1.10 Layer Health Metrics (What to Watch Over Time)

Layering is not a one-time design; it must be defended.
Practical signals of layer decay:

- cyclic dependencies between targets,
- increasing include graph depth,
- domain types containing infrastructure handles,
- frequent rebuild cascades when small changes occur,
- "utility" layers growing without clear ownership,
- performance patches that bypass boundaries instead of refining them.

20.1.11 Engineering Checklist

1. Define each layer as a dependency rule, not a directory.
2. Keep the domain layer pure: no infrastructure types, no framework leakage.
3. Invert dependencies via interfaces owned by higher layers (domain/application).
4. Treat headers as architecture: keep public headers minimal and stable.
5. Enforce layering in the build system: separate targets, controlled include paths.
6. Make concurrency and blocking rules explicit at boundaries.
7. When performance demands it, redesign boundary APIs (bulk operations, views) instead of bypassing layers.

20.1.12 Summary

Traditional layering remains one of the strongest tools for long-lived systems, but it must be revisited through the realities of C++:

- compile-time coupling through headers,
- performance demands requiring efficient boundary design,
- modern concurrency requiring explicit scheduling and thread-safety contracts.

Modern C++ layering succeeds when:

- dependency direction is enforced,
- the domain remains pure and stable,
- interfaces invert infrastructure dependencies,
- boundaries are designed for performance and concurrency rather than bypassed.

20.2 Dependency Inversion in Practice

20.2.1 The Architectural Problem Dependency Inversion Solves

In a layered system, the highest-value code is typically:

- domain rules,
- application use cases,
- long-lived business invariants.

The highest-volatility code is typically:

- databases,
- serialization formats,
- network protocols,
- operating-system integration,
- third-party frameworks.

Without discipline, the volatile code "infects" the stable code: domain types begin carrying database identifiers, framework handles, JSON structures, and OS resources. Over time, this destroys testability, replaceability, and long-term maintainability.

Dependency Inversion reverses that coupling:

> High-level policy does not depend on low-level details. Both depend on abstractions.

In practical C++ architecture, this means:

- stable layers own the interfaces,
- volatile layers implement them,
- composition occurs at a well-defined boundary.

20.2.2 What Dependency Inversion Is (and Is Not)

It Is:

- an architectural rule about *dependency direction*,
- a way to isolate volatility,
- a method to keep your domain testable without infrastructure,
- a technique to stop header and build coupling from spreading.

It Is Not:

- “use virtual everywhere”,
- “introduce interfaces for every class”,
- “a framework requirement”,
- “an excuse to build an over-engineered abstraction layer”.

Dependency inversion should be applied exactly where volatility enters the system.

20.2.3 The Most Important Practical Principle: Own the Interface in the Stable Layer

If the database layer defines the repository interface, then the domain depends on the database. That is not inversion; it is dependency capture.
The stable layer must define:

- the types of the operations it needs,
- the error model it expects,

- the concurrency model it assumes,
- the data shapes it consumes and produces.

The infrastructure layer must adapt to that contract.

20.2.4 A Complete Example (Domain/Application Own the Contract)

Domain Types

```
#pragma once
#include <string>

struct UserId {
  int value{};
};

struct User {
  UserId id{};
  std::string name{};
};
```

A Stable Error Model

A practical design needs an explicit error model. One common approach is to separate:

- **domain failures** (invalid input, rule violations),
- **infrastructure failures** (I/O, connectivity, timeouts).

```
#pragma once
#include <variant>

enum class InfraError {
```

```
  timeout,
  unavailable,
  corrupted_data
};

template<class T>
using Result = std::variant<T, InfraError>;
```

Domain-Owned Port (Interface)

```
#pragma once
#include <optional>

class IUserRepository {
public:
  virtual ~IUserRepository() = default;

  virtual Result<std::optional<User>> find(UserId id) = 0;
  virtual Result<void> save(const User& u) = 0;
};
```

Architectural effect. The domain/application controls:

- the API shape,
- the data types,
- the error vocabulary.

No database types appear here.

Application Use Case Depends Only on the Port

```
#pragma once
```

```
class RenameUser {
public:
  explicit RenameUser(IUserRepository& repo) : repo_(repo) {}

  Result<void> execute(UserId id, std::string new_name) {
    auto r = repo_.find(id);
    if (auto* err = std::get_if<InfraError>(&r)) return *err;

    auto opt = std::get<std::optional<User>>(std::move(r));
    if (!opt) return InfraError::corrupted_data; // placeholder for "not found" policy

    opt->name = std::move(new_name);
    return repo_.save(*opt);
  }

private:
  IUserRepository& repo_;
};
```

Architectural effect. The use case remains testable and independent of storage technology.

Infrastructure Implements the Port

```
#pragma once
#include <unordered_map>

class InMemoryUserRepository final : public IUserRepository {
public:
  Result<std::optional<User>> find(UserId id) override {
    auto it = db_.find(id.value);
    if (it == db_.end()) return std::optional<User>{};
    return it->second;
```

```
  }

  Result<void> save(const User& u) override {
    db_[u.id.value] = u;
    return {};
  }

private:
  std::unordered_map<int, User> db_;
};
```

Architectural effect. Infrastructure is swappable. The stable code does not need recompilation when swapping implementations, beyond the composition boundary.

20.2.5 Composition Root: Where the System Is Wired Together

Dependency inversion requires a well-defined **composition root**: the location where concrete implementations are created and connected to stable code.
This is typically:

- the executable entry point,
- an application bootstrap module,
- a service host initialization unit.

```
#include <iostream>

int main() {
  InMemoryUserRepository repo;
  RenameUser use_case(repo);

  auto r = use_case.execute(UserId{1}, "Ayman");
```

```
  if (auto* err = std::get_if<InfraError>(&r)) {
    std::cout << "infra error\n";
    return 1;
  }
  std::cout << "ok\n";
}
```

Design rule. Only the composition root is allowed to mention both:

- high-level policy (use cases),
- and low-level details (repositories, transports, frameworks).

Everywhere else, the dependency direction must remain clean.

20.2.6 Dependency Inversion and Header/Build Hygiene

In C++, dependency inversion is not only runtime coupling. It is also compile-time coupling. Practical rules:

- Public interfaces must avoid including heavy framework headers.
- Prefer forward declarations in interfaces when possible.
- Keep ports (interfaces) in stable targets with minimal include paths.
- Put implementations in separate targets that depend on the ports.

This reduces:

- rebuild cascades,
- ABI churn,
- accidental transitive dependency leakage.

20.2.7 When Virtual Interfaces Are Not Desired

Sometimes you want dependency inversion without virtual dispatch:

- performance-critical hot paths,
- header-only libraries,
- compile-time customization.

Two common options:

- **type erasure** (runtime polymorphism without exposing templates),
- **concept-based ports** (compile-time polymorphism).

Concept-Based Port (Compile-Time Inversion)

```
#include <concepts>
#include <optional>

template<class Repo>
concept UserRepository =
  requires(Repo r, UserId id, const User& u) {
    { r.find(id) } -> std::same_as<Result<std::optional<User>>>;
    { r.save(u) }  -> std::same_as<Result<void>>;
  };

template<UserRepository Repo>
class RenameUserT {
public:
  explicit RenameUserT(Repo& repo) : repo_(repo) {}

  Result<void> execute(UserId id, std::string new_name) {
```

```
    auto r = repo_.find(id);
    if (auto* err = std::get_if<InfraError>(&r)) return *err;

    auto opt = std::get<std::optional<User>>(std::move(r));
    if (!opt) return InfraError::corrupted_data;

    opt->name = std::move(new_name);
    return repo_.save(*opt);
  }

private:
  Repo& repo_;
};
```

Architectural effect. The stable layer still defines the contract (the concept), and infrastructure still provides an implementation, but polymorphism is resolved at compile time.

20.2.8 Concurrency and Dependency Inversion

Ports should include concurrency expectations. For example:

- Is the repository thread-safe?
- Is it thread-confined (must be called on a specific thread)?
- Does it block? Can it be used in an event loop?

A practical technique is to separate:

- synchronous ports (blocking operations),
- asynchronous ports (task/future-based operations),
- message-based ports (command/event interfaces).

Asynchronous Port Shape

```cpp
#include <future>
#include <optional>

class IUserRepositoryAsync {
public:
  virtual ~IUserRepositoryAsync() = default;

  virtual std::future<Result<std::optional<User>>> find(UserId id) = 0;
  virtual std::future<Result<void>> save(User u) = 0;
};
```

Design effect. The stable layer controls the async contract. Infrastructure implements it, potentially using executors, thread pools, or I/O loops internally.

20.2.9 Common Failure Modes (and How to Avoid Them)

Failure 1: Interface Owned by Infrastructure

If infrastructure defines the interface, stable layers inherit infrastructure concerns. Fix: move the interface into the stable layer.

Failure 2: Leaky Abstractions

Ports returning database rows, JSON objects, or framework types leak details upward. Fix: ports use domain types or boundary DTOs owned by the stable layer.

Failure 3: "Over-Abstracting" Everything

Abstracting every class creates complexity without isolating volatility. Fix: invert only where volatility enters (I/O, persistence, platform, third-party frameworks).

Failure 4: Inconsistent Error Models

If some implementations throw and others return errors, the use case becomes fragile. Fix: define the error model in the stable layer and enforce it uniformly.

Failure 5: Concurrency Model Not Specified

Thread-safety assumptions differ and cause production races. Fix: define thread-safety and blocking semantics in the port contract.

20.2.10 Engineering Checklist

1. Identify volatility boundaries (DB, network, OS, frameworks).
2. Define ports in the stable layer using stable domain types and a clear error model.
3. Keep port headers small; avoid leaking heavy dependencies upward.
4. Implement ports in infrastructure targets; link and wire only in the composition root.
5. Decide and document concurrency and blocking semantics for each port.
6. Choose virtual dispatch, type erasure, or concepts based on performance and build constraints.

20.2.11 Summary

Dependency inversion is the practical mechanism that keeps long-lived C++ systems stable:

- stable policy layers own the abstractions,
- volatile detail layers implement them,
- composition is centralized and explicit,

- headers and build targets enforce the dependency direction,
- error and concurrency models remain consistent across implementations.

When applied precisely at volatility boundaries, dependency inversion prevents architectural decay, enables testing without infrastructure, and makes large systems resilient to inevitable technology change.

20.3 C++ Modules as Architectural Boundaries

20.3.1 Why Modules Change Architecture (Not Just Build Speed)

C++ Modules are often introduced as a compilation-speed feature, but their deeper impact is architectural. A module is a **language-level boundary** that replaces the accidental, textual inclusion model of headers with a **semantic import model**.
In traditional header-based systems:

- `#include` is textual substitution,
- macros and transitive includes silently shape APIs,
- build dependencies are hard to reason about,
- private implementation details frequently leak through headers,
- layering rules can be violated by a single include.

Modules change the default:

- names are **exported intentionally**,
- transitive visibility is controlled,

- macro leakage is minimized by design,
- the build system must respect `import` dependency ordering.

The architectural claim is:

> A module boundary is an enforceable boundary: it makes dependency direction, API surfaces, and layering intent explicit.

20.3.2 The Mental Model: From Files to "Packages"

A useful architectural mapping:

- **module** ≈ package boundary (the unit of distribution and ownership),
- **exported declarations** ≈ public API,
- **non-exported declarations** ≈ internal implementation details,
- **partitions** ≈ internal submodules for structure without exposing them.

This aligns with layered architecture:

- domain modules export domain vocabulary,
- application modules export use-case APIs,
- infrastructure modules export integration surfaces,
- composition roots import and wire modules.

20.3.3 Core Building Blocks You Must Know

Interface Units and Implementation Units

A named module typically has:

- an **interface unit** that declares and exports the public surface,
- one or more **implementation units** that define the exported entities and additional internal entities.

Architectural guideline:

> Place architectural boundaries in interface units; keep implementation units free to change.

Partitions

Partitions structure a module internally without necessarily making each partition part of the public surface.
Two common patterns:

- **internal partitions:** factor complex implementation; not imported directly by clients,
- **API partitions:** split large APIs into stable subsets, re-exported by a facade interface.

Global Module Fragment

The global module fragment (the `module;` line) exists to allow including legacy headers *inside* a module unit before the module declaration. This is primarily a migration mechanism.
Architectural guideline:

> Avoid using the global module fragment as an excuse to keep legacy include-graphs alive indefinitely.

Header Units

Header units are a bridge:

- they allow importing a header as a module-like unit,
- they improve build performance and reduce repeated parsing,
- they can help adoption when full named modules are not yet practical.

However, header units do not automatically fix all architectural problems: the underlying header may still carry macros and transitive coupling.

20.3.4 A Minimal Named Module as a Boundary

Module Interface Unit (Public API)

```
// math.core.cppm
export module math.core;

export namespace math {
  export int add(int a, int b) noexcept;
  export int mul(int a, int b) noexcept;
}
```

Module Implementation Unit (Internal Details Allowed)

```
// math.core.impl.cpp
module math.core;

namespace math {
  int add(int a, int b) noexcept { return a + b; }
  int mul(int a, int b) noexcept { return a * b; }
}
```

Client Code Imports the Boundary

```
// main.cpp
import math.core;

int main() {
  return math::add(2, 3) == 5 ? 0 : 1;
}
```

Architectural effect. Clients see only what is exported. They do not inherit transitive includes and private names.

20.3.5 Facade Modules: A Practical Architectural Pattern

Large systems rarely want clients to import dozens of fine-grained modules directly. A common pattern is a facade module that re-exports a curated surface.

Facade Interface

```
// mylib.cppm
export module mylib;

export import mylib.domain;
export import mylib.app;
```

Architectural effect. `mylib` becomes the stable boundary; internal module graph remains flexible.

20.3.6 Partitions as Layering Inside a Module

Partitions are particularly useful when you want:

- a single public boundary,
- but multiple internal subsystems,
- without exposing those subsystems as first-class import targets.

Interface Unit Re-exports Partitions

```
// mylib.domain.cppm
export module mylib.domain;

export import :types;
export import :rules;
```

Partition: Types

```
// mylib.domain-types.cppm
export module mylib.domain:types;

export namespace domain {
  struct UserId { int value{}; };
  struct User { UserId id{}; };
}
```

Partition: Rules

```
// mylib.domain-rules.cppm
module;

#include <string> // legacy header usable in global module fragment if needed

export module mylib.domain:rules;

import :types;
```

```
export namespace domain {
  inline bool valid_name(const std::string& s) {
    return !s.empty();
  }
}
```

Architectural effect. You can structure domain vocabulary internally while presenting a single stable import surface.

20.3.7 How Modules Enforce Architectural Hygiene

1) Reduced Transitive Visibility

Header inclusion often exposes names unintentionally. Modules require explicit export, which encourages:

- smaller public surfaces,
- fewer “utility” leaks,
- a clearer separation between API and implementation.

2) Macro Containment

Macros are a notorious boundary breaker in header-based designs. With modules:

- imported module interfaces do not behave like textual includes,
- macros generally do not propagate across module boundaries the way headers do.

Architectural consequence:

> Modules push you toward stable, typed configuration instead of macro-based configuration.

3) Explicit Dependency Graph

The build must understand module dependencies and compile in the correct order. This discourages the “include anything anywhere” culture and promotes explicit layering.

20.3.8 Build System Reality: Modules Require Dependency Scanning

Unlike headers, modules require build systems to:

- scan sources for `import` relationships,
- produce and consume compiled module artifacts (BMIs/IFCs),
- ensure correct compile ordering and rebuild triggers.

For architecture, this is a benefit:

- cyclic dependencies become visible quickly,
- accidental dependency direction violations are harder to hide,
- the module graph becomes a first-class artifact of the design.

CMake Named Modules (Modern Tooling Shape)

```
cmake_minimum_required(VERSION 3.28)
project(MyLib LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_library(mylib)
target_sources(mylib
  PUBLIC
```

```
    FILE_SET cxx_modules TYPE CXX_MODULES FILES
      mylib.cppm
      mylib.domain.cppm
      mylib.domain-types.cppm
      mylib.domain-rules.cppm
)
```

Architectural effect. Modules become explicit build targets and file sets, aligning build boundaries with architectural boundaries.

20.3.9 Common Architectural Pitfalls (And How to Avoid Them)

Pitfall 1: Treating Modules as "Headers With New Syntax"

If you export everything and keep transitive coupling, you get:

- a big surface with unclear ownership,
- brittle rebuild behavior,
- no real architectural improvement.

Fix: export only stable API. Keep implementation and policy details unexported.

Pitfall 2: Cyclic Module Dependencies

Cycles often appear when boundaries are wrong:

- domain imports infrastructure for persistence,
- application imports UI types,
- shared "utils" become a dumping ground.

Fix: apply dependency inversion:

- stable layer defines ports (interfaces/concepts),
- volatile layer implements them,
- composition root wires them.

Pitfall 3: Exporting Heavy Dependencies

Exporting standard containers is normal, but exporting types that drag massive dependencies can:

- enlarge rebuild scope,
- expose ABI-sensitive implementation details,
- lock you into third-party types across boundaries.

Fix: use boundary DTOs or stable abstractions. Prefer `std::span` and views for data movement when appropriate.

Pitfall 4: ABI and Distribution Expectations

Modules improve compilation model, not ABI by themselves. If you ship binaries across toolchains or versions, you still need:

- stable ABI discipline,
- versioning rules,
- controlled exposure of inline/template-heavy APIs.

Fix: treat exported entities as a published contract; prefer stable, minimal exports.

Pitfall 5: Mixing Concurrency Semantics Across Boundaries

A module boundary should also specify:

- thread-safety level (thread-safe vs thread-compatible vs thread-confined),
- blocking behavior (may block vs must not block),
- cancellation and timeout expectations (if asynchronous).

Fix: encode concurrency expectations in exported types and documented contracts, and keep scheduling decisions near the composition boundary.

20.3.10 Migration Strategy for Long-Lived Codebases

A practical migration path that minimizes risk:

1. **Stabilize layering rules** in targets and include paths.
2. **Start with leaf libraries** (low fan-in) to reduce graph complexity.
3. **Adopt header hygiene** (forward declarations, minimal public headers).
4. **Introduce header units** where they provide immediate value.
5. **Create named modules** for the most stable APIs first (domain vocabulary, core utilities).
6. **Use facade modules** to control how clients import your system.
7. **Refactor cycles** using dependency inversion and ports.

The architectural goal is not "everything is a module". The goal is:

> The module graph should reflect the architecture you want to defend for years.

20.3.11 Engineering Checklist

1. Define module boundaries along stable architectural ownership lines (domain, application, infrastructure).
2. Export only the stable public surface; keep implementation details unexported.
3. Use partitions to structure internals while keeping a single public facade.
4. Prevent cyclic imports by applying dependency inversion at volatility boundaries.
5. Avoid exporting third-party framework types across core boundaries.
6. Align build targets and module boundaries; make dependency direction enforceable.
7. Specify concurrency and blocking contracts as part of exported API design.

20.3.12 Summary

C++ Modules enable architectural boundaries that are stronger than header boundaries:

- explicit exports reduce accidental coupling,
- imports form a visible dependency graph,
- macro leakage is reduced,
- build tooling must respect dependency ordering,
- facade modules and partitions support clean API design at scale.

When used deliberately, modules are not just a compilation feature. They are a mechanism for enforcing layered and modular architecture in C++ systems that must remain healthy for the long term.

20.4 Preventing Architecture Erosion

20.4.1 Architecture Erosion: The Default Outcome if You Do Nothing

In long-lived C++ systems, architecture erosion is not a surprise event. It is the natural result of continuous change under time pressure.
Typical erosion symptoms include:

- **layer violations:** upper layers include or link to lower-level details directly, bypassing defined boundaries,
- **dependency cycles:** once cycles exist, refactoring becomes expensive and teams avoid touching risky areas,
- **utility dumping grounds:** “common” modules that grow without ownership and become dependency magnets,
- **interface bloat:** public APIs expose internals, forcing widespread rebuilds and increasing coupling,
- **hidden concurrency assumptions:** thread-safety differs across components without explicit contracts,
- **build-time collapse:** small changes trigger large rebuilds because headers, includes, and targets are not isolated,
- **performance shortcuts:** hot-path optimizations punch holes through boundaries instead of improving boundary design.

The fundamental insight is:

> Architecture is a set of constraints; without enforcement, constraints decay into suggestions.

20.4.2 Designing for Non-Erosion: The Three Defense Layers

A durable architecture is defended by three complementary mechanisms:

1. **Social defense:** conventions, code review expectations, architectural ownership.
2. **Build defense:** targets, include-path control, dependency direction enforced by the build.
3. **Automated defense:** CI checks, static analysis, layering tests, dependency rules as executable policies.

If any one layer is missing, erosion accelerates.

20.4.3 Principle 1: Make Boundaries Explicit and Small

Ambiguous boundaries are unenforceable. Each module/layer must have a clear contract:

- **Responsibility:** what it owns and what it must never own.
- **Public surface:** exported API (headers/modules) meant for other layers.
- **Dependency rules:** what it may depend on (compile-time and link-time).
- **Threading model:** thread-safe vs thread-compatible vs thread-confined; blocking vs non-blocking.
- **Error model:** exceptions vs error return types; logging policy boundaries.

A boundary that mixes policy and mechanism becomes porous. A boundary that is too large becomes unreviewable.

20.4.4 Principle 2: Enforce Dependency Direction in the Build System

In C++, architecture is primarily enforced through **build targets** and **include paths**. Directories do not enforce rules; targets do.
A practical layering approach:

- Each layer is its own library target.
- Upper layers link only to the layer(s) below.
- Public includes are restricted to the layer's published headers/modules.
- Private includes are not visible outside the layer target.

CMake Pattern: Public vs Private Include Boundaries

```
add_library(domain)
target_include_directories(domain
  PUBLIC  ${CMAKE_CURRENT_SOURCE_DIR}/include
  PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/src
)
# domain must not depend on infra

add_library(app)
target_include_directories(app
  PUBLIC  ${CMAKE_CURRENT_SOURCE_DIR}/include
  PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/src
)
target_link_libraries(app PUBLIC domain)

add_library(infra)
target_include_directories(infra
  PUBLIC  ${CMAKE_CURRENT_SOURCE_DIR}/include
  PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/src
```

```
)
target_link_libraries(infra PUBLIC domain) # infra may depend on domain ports

add_executable(service main.cpp)
target_link_libraries(service PRIVATE app infra)
```

Why this prevents erosion. When include directories and link dependencies are constrained per-target, forbidden dependencies become build errors rather than architectural arguments.

20.4.5 Principle 3: Treat Headers as Architecture (Because They Are)

Even with correct library dependencies, headers can silently break architecture by:

- leaking heavy dependencies through public includes,
- exposing internal types unintentionally,
- creating transitive include graphs that couple unrelated layers.

Practical header hygiene rules:

- Public headers include only what they must.
- Prefer forward declarations where valid.
- Avoid inline implementation details in public headers unless they are truly part of the contract.
- Separate stable API headers from internal headers; do not expose internal headers in public include paths.

A Boundary-Friendly Header Pattern

```
#pragma once

// Forward declare instead of including heavy headers.
namespace net { class Connection; }

namespace app {

class Session {
public:
  explicit Session(net::Connection& c) noexcept;
  void handle_request();
private:
  net::Connection* conn_; // stable dependency on an abstraction or forward declared type
};

} // namespace app
```

20.4.6 Principle 4: Dependency Inversion Is the "Erosion Firewall"

Architecture erodes most rapidly when stable policy depends on volatile details. Dependency inversion creates a firewall:

- stable layer defines a port (interface or concept),
- volatile layer implements it,
- composition happens at the application boundary.

This prevents common erosion vectors:

- database types entering domain entities,

- framework headers creeping into application use cases,
- platform handles leaking into core logic.

20.4.7 Principle 5: Make Concurrency Contracts Part of the Architecture

Concurrency-related erosion is particularly expensive because it becomes correctness debt.
Minimum concurrency contract per component:

- **thread-safety level:** thread-safe / thread-compatible / thread-confined,
- **blocking behavior:** may block / must not block,
- **ownership model:** which thread owns mutation, which operations require external synchronization,
- **cancellation/timeouts:** where applicable for asynchronous flows.

Without this, architecture "works" until production load introduces races, deadlocks, and tail-latency collapse.

20.4.8 Principle 6: Institutionalize Architectural Decisions

Erosion often begins when teams cannot explain why a boundary exists. Decisions must remain discoverable.
Practical artifacts that reduce drift:

- a short architecture overview document (one page per major subsystem),
- an explicit dependency diagram (targets/modules),
- a policy for adding dependencies (approval rules),
- a small set of architectural invariants stated as rules.

The goal is not bureaucracy. The goal is preserving intent over years of change.

20.4.9 Automated Defense: Executable Architectural Rules

A long-lived architecture requires automation that makes violations expensive immediately.

1) Dependency Rule Checks

A common tactic is a CI job that validates:

- forbidden includes (e.g., domain must not include infra headers),
- forbidden link edges (e.g., domain must not link to infra),
- absence of cycles in the target/module graph.

Example: A Simple Include-Rule Checker (Conceptual)

```
# check_includes.py (conceptual)
# Enforce: domain must not include infra headers.

import sys
from pathlib import Path

forbidden_prefixes = ["infra/", "thirdparty/"]
domain_root = Path("domain/include")

bad = []
for p in domain_root.rglob("*.h"):
  text = p.read_text(encoding="utf-8", errors="ignore")
  for line in text.splitlines():
    line = line.strip()
    if line.startswith("#include"):
      for pref in forbidden_prefixes:
        if f"<{pref}" in line or f"\"{pref}" in line:
          bad.append((p.as_posix(), line))
```

```
if bad:
  for f, line in bad:
    print(f"{f}: forbidden include: {line}")
  sys.exit(1)
```

Design effect. The rule becomes executable: violations fail the build and cannot accumulate silently.

2) Static Analysis and Linting

Static analysis prevents erosive patterns such as:

- unsafe lifetime and ownership leaks,
- missing virtual destructors on polymorphic bases,
- exception-safety hazards at boundaries,
- concurrency anti-patterns (locking discipline violations, risky atomics usage).

The architectural point:

> Linting is not about style; it is about preventing repeated patterns that degrade structure and correctness.

3) Architectural Tests

Beyond unit tests, architecture needs tests of *structure*:

- "domain builds without infrastructure",
- "application can run with a fake repository",

- “no forbidden dependencies appear in exported interfaces”,
- “public API surface remains stable” (for published libraries).

A powerful minimal test:

- build and run domain + app with *only* test doubles (no infra linked),
- ensuring dependency inversion is actually real.

20.4.10 Human Defense: Code Review With Structural Questions

Code review is where architecture either survives or dies. A review checklist that prevents erosion focuses on questions like:

1. Does this change introduce a new dependency edge? Is it permitted?
2. Is this include in a public header necessary, or is it accidental coupling?
3. Does this change leak infrastructure types into stable layers?
4. Does this boundary now require more knowledge than before (leakiness)?
5. Are concurrency assumptions explicit, or did we add hidden thread-safety requirements?
6. Did we add a “utility” in the wrong place because it was convenient?

A review process that checks only correctness and formatting will not stop erosion.

20.4.11 The Three Classic Erosion Traps (and Their Repairs)

Trap 1: The “Common” Module

Symptom: a growing common/util module imported by everything.
Why it is erosive:

- it becomes an uncontrolled dependency hub,
- it hides ownership,
- it encourages dumping unrelated code.

Repair:

- split by ownership and responsibility,
- define multiple small utilities with clear domains,
- keep utilities leaf-like (few dependencies, stable surface).

Trap 2: "Just Include It" in Public Headers

Symptom: public headers include large frameworks and internal headers.
Repair:

- push heavy includes into implementation files,
- use forward declarations and PImpl where appropriate,
- define narrow boundary types (DTOs, spans, views) owned by stable layers.

Trap 3: Performance "Punch-Through"

Symptom: hot-path changes bypass layers for speed.
Repair:

- redesign boundary APIs for efficiency (bulk operations, views, amortization),
- introduce dedicated performance-oriented boundary objects,
- isolate hot paths behind explicit interfaces instead of scattered shortcuts,
- measure and validate improvements to avoid permanent unjustified coupling.

20.4.12 Modules and Targets as Long-Term Boundary Reinforcement

When available, C++ Modules can strengthen boundaries by:

- exporting only intentional APIs,
- reducing accidental transitive visibility,
- making the dependency graph explicit via imports,
- reducing macro-driven boundary breakage.

But modules are not sufficient alone: the build graph and ownership rules must still be enforced.

20.4.13 A Practical Anti-Erosion Program

A sustainable program for long-lived C++ systems:

1. Define layers/modules as targets; enforce allowed dependencies.
2. Minimize and stabilize public surfaces; treat headers as contracts.
3. Apply dependency inversion at every volatility boundary.
4. Establish concurrency and error-model contracts at boundaries.
5. Automate structural checks in CI (include rules, forbidden edges, cycle detection).
6. Run static analysis continuously and enforce key rules.
7. Review changes with explicit architectural questions.
8. Track erosion metrics: dependency edges, include depth, rebuild scope, cycle count.

20.4.14 Summary

Preventing architecture erosion is the practice of turning architectural intent into enforceable constraints:

- build targets and include paths enforce dependency direction,
- dependency inversion prevents volatile details from infecting stable policy,
- headers are treated as architectural contracts,
- concurrency and error models become explicit boundary rules,
- CI and static analysis make structural violations fail immediately,
- code review defends boundaries using structural questions, not intuition.

In long-lived Modern C++ systems, architecture remains healthy only when it is defended continuously, and when the defense is automated enough that erosion becomes expensive at the moment it is introduced.

Component-Based and Plugin Architectures

21.1 Runtime Extensibility

21.1.1 What Runtime Extensibility Means

Runtime extensibility is the architectural capability to add, replace, enable, or disable behavior **without rebuilding or redeploying the core executable**. It is commonly implemented through:

- plugin modules loaded dynamically (shared libraries),
- component registries and factories,
- scripting or configuration-driven policy selection,
- service-provider mechanisms where implementations are discovered at runtime.

In long-lived systems, runtime extensibility is not a gimmick. It is a way to manage change safely:

- different customers or deployments need different features,
- features evolve faster than the core,

- third parties must integrate without touching internal code,
- you need controlled experimentation and staged rollouts,
- some functionality is optional or licensed.

The architectural objective is:

> Make extension a supported path, not a source-code fork.

21.1.2 The Three Axes of Extensibility

A robust design distinguishes three axes:

1) Discovery

How does the system find extensions?

- directory scanning,
- configuration listing,
- explicit loading of named modules,
- platform-specific registries.

2) Binding

How does the system connect extension code to the core?

- abstract interfaces with factories,
- type-erased services,
- message-based interfaces,
- callback registration.

3) Compatibility and Safety

How does the system keep extensions from breaking the host?

- versioned APIs/ABIs,
- strict lifecycle and ownership rules,
- validation and sandboxing where needed,
- crash isolation policies and failure handling.

Many “plugin systems” fail because they focus only on discovery and ignore compatibility.

21.1.3 Runtime Extensibility vs Compile-Time Extensibility

C++ supports deep compile-time extensibility through templates and configuration. Runtime extensibility serves different goals:

- **Compile-time extensibility:** fast, optimized, type-rich; requires rebuild.
- **Runtime extensibility:** late binding, replaceable; must manage ABI and safety.

A long-lived system typically uses both:

- compile-time mechanisms inside the core for performance-critical composition,
- runtime plugins at boundaries where change is expected and third-party integration matters.

21.1.4 The Plugin Boundary: Define a Minimal Stable Contract

The plugin boundary must be **smaller** and **more stable** than internal APIs. Practical requirements for the contract:

- minimal surface (only what extensions need),
- stable data types (avoid templates and complex inline-heavy types across the boundary),
- explicit ownership and lifetime rules,
- explicit error handling contract,
- explicit thread-safety contract,
- versioning and compatibility strategy.

In C++, the most reliable boundary is a C-compatible ABI or a carefully controlled C++ ABI used only with strict toolchain/version constraints.

21.1.5 ABI Reality: Why C Interfaces Are Common for Plugins

C++ ABIs can vary across:

- compiler families,
- compiler versions,
- standard library implementations,
- compilation flags affecting layout and calling conventions.

Therefore, many robust plugin systems expose a **C ABI** as the boundary:

- stable calling convention,

- predictable symbol names,
- explicit data layout choices.

Internally, both host and plugin can still be written in modern C++.

21.1.6 A Practical Minimal Plugin ABI

The typical shape:

- the host loads a shared library,
- finds a known exported symbol (entry point),
- asks the plugin for metadata and factory functions,
- creates an implementation object behind an abstract interface,
- destroys it with an API-controlled function.

Host-Side Stable Types (C ABI Header)

```
#pragma once
#include <cstdint>
#include <cstddef>

extern "C" {

// Increment when breaking ABI, not when adding backward-compatible features.
static constexpr std::uint32_t PLUGIN_ABI_VERSION = 1;

struct PluginInfo {
  std::uint32_t abi_version;
  const char*   name;
```

```
  const char*   vendor;
  const char*   version;
};

struct HostApi {
  // Host-provided services (logging, allocation, etc.)
  void (*log)(int level, const char* msg) noexcept;
  void* (*alloc)(std::size_t n) noexcept;
  void  (*dealloc)(void* p) noexcept;
};

// Opaque handle for a plugin-created service instance.
struct ServiceHandle;

using CreateServiceFn  = ServiceHandle* (*)(const HostApi* host) noexcept;
using DestroyServiceFn = void (*)(ServiceHandle*) noexcept;

struct PluginExports {
  PluginInfo info;
  CreateServiceFn create;
  DestroyServiceFn destroy;
};

// Entry point symbol: must be present in every plugin.
bool plugin_get_exports(std::uint32_t host_abi_version,
                        PluginExports* out) noexcept;

} // extern "C"
```

Design effect. The boundary:

- avoids C++ name mangling,
- encodes ABI versioning explicitly,

- centralizes ownership control via `create/destroy`,
- allows host services to be accessed without linking to host internals.

21.1.7 Plugin Implementation: C ABI Outside, C++ Inside

Plugin Internal C++ Service Type

```
#include <cstring>
#include <new>

struct ServiceHandle {
  virtual ~ServiceHandle() = default;
  virtual const char* process(const char* input) noexcept = 0;
};

class ExampleService final : public ServiceHandle {
public:
  explicit ExampleService(const HostApi* host) : host_(host) {
    host_->log(1, "ExampleService constructed");
  }

  const char* process(const char* input) noexcept override {
    host_->log(2, input ? input : "(null)");
    return "ok";
  }

private:
  const HostApi* host_;
};
```

Plugin Entry Point and Factory Functions

```
extern "C" {
```

```
static ServiceHandle* create_service(const HostApi* host) noexcept {
  try {
    return new ExampleService(host);
  } catch (...) {
    if (host && host->log) host->log(0, "create_service failed");
    return nullptr;
  }
}

static void destroy_service(ServiceHandle* p) noexcept {
  delete p;
}

bool plugin_get_exports(std::uint32_t host_abi_version,
                        PluginExports* out) noexcept {
  if (!out) return false;
  if (host_abi_version != PLUGIN_ABI_VERSION) return false;

  out->info = PluginInfo{
    PLUGIN_ABI_VERSION,
    "example.plugin",
    "ExampleVendor",
    "1.0.0"
  };
  out->create  = &create_service;
  out->destroy = &destroy_service;
  return true;
}

} // extern "C"
```

Architectural effect. The plugin can be written in modern C++ with classes and RAII, but the boundary remains stable and explicit.

21.1.8 Host Loader: Discovery, Loading, Validation

Runtime extensibility requires a loader that:

- loads libraries from trusted locations,
- resolves the entry point symbol,
- validates ABI version and plugin metadata,
- creates instances,
- ensures deterministic cleanup.

Portable Loader Design (Abstracted)

Platform details differ (`dlopen/dlsym` vs `LoadLibrary/GetProcAddress`), but the architectural structure is similar:

- isolate platform code in a small `SharedLibrary` wrapper,
- keep policy (validation, selection, lifecycle) in the core.

```
#include <string>
#include <stdexcept>

class SharedLibrary {
public:
  explicit SharedLibrary(const std::string& path);
  ~SharedLibrary();
```

```
  SharedLibrary(const SharedLibrary&) = delete;
  SharedLibrary& operator=(const SharedLibrary&) = delete;

  SharedLibrary(SharedLibrary&&) noexcept;
  SharedLibrary& operator=(SharedLibrary&&) noexcept;

  void* symbol(const char* name);

private:
  void* handle_{nullptr}; // platform handle
};
```

Design effect. The host architecture separates:

- **mechanism** (load symbols) from
- **policy** (what to load, when, and how to validate).

21.1.9 Versioning Strategy: Compatibility Is a Design Feature

A plugin architecture without versioning is fragile.
A practical compatibility model:

- **ABI version:** increment only when breaking binary compatibility.
- **Feature flags:** allow optional capabilities without breaking ABI.
- **Negotiation:** host checks ABI, then checks supported feature set.

Feature Flags Example

```
#include <cstdint>
```

```
enum PluginFeature : std::uint64_t {
  FEATURE_PROCESS = 1ull << 0,
  FEATURE_STREAM  = 1ull << 1,
  FEATURE_ADMIN   = 1ull << 2
};

struct PluginInfo2 {
  std::uint32_t abi_version;
  std::uint64_t features;
  const char*   name;
};
```

Architectural effect. The host can load older plugins and use only the features it understands, or reject plugins that do not meet required capabilities.

21.1.10 Lifecycle: Ownership and Deterministic Unload

Erosion and crashes often come from lifecycle mistakes:

- unloading a library while objects from it still exist,
- host freeing plugin memory (or vice versa) with mismatched allocators,
- passing STL containers across the ABI boundary.

Lifecycle rules that prevent these failures:

- The plugin must destroy what it creates (use plugin-provided `destroy`).
- The host must not unload the library until all plugin objects are destroyed.
- Cross-boundary memory allocation must be owned by one side explicitly.
- Avoid passing STL types across a C ABI boundary; use plain buffers and explicit lengths.

21.1.11 Threading and Concurrency Contracts for Plugins

Plugins can destroy host stability if concurrency rules are unclear. The plugin contract must specify:

- whether plugin services are thread-safe,
- whether calls will be serialized by the host,
- whether callbacks into the host are allowed and from which threads,
- whether plugin code may block.

A robust strategy:

- default to **thread-confined** plugin instances unless explicitly designed otherwise,
- use message-passing to interact with host services,
- isolate blocking operations to dedicated worker threads controlled by the host.

21.1.12 Security and Trust: Plugins Are Code Injection by Default

A plugin mechanism is effectively a controlled form of code injection. Therefore, a production design must address:

- search path safety (avoid loading from untrusted directories),
- signature validation where required,
- least-privilege execution model,
- configuration for allowed/blocked plugins,
- failure containment policy (disable plugin on repeated faults).

At minimum:

> Only load plugins from locations owned and controlled by the product, not from user-writable directories by default.

21.1.13 Designing Extension Points: Stable Core, Flexible Edges

The best plugin architectures keep the core small:

- a service registry,
- a minimal plugin contract,
- a well-defined host API surface,
- stable domain vocabulary.

Extensions should integrate through **explicit extension points**:

- codecs, protocols, drivers,
- import/export formats,
- analytics and rules engines,
- UI widgets (when applicable),
- domain-specific strategies.

Avoid extension points that expose internal data structures directly. Instead, expose:

- boundary DTOs,
- immutable views,
- message/event interfaces.

21.1.14 Common Failure Modes (and Their Architectural Fixes)

Failure 1: Large Plugin API Surface

Symptom: plugins depend on many host headers and internal types.
Fix: shrink the contract, introduce a stable host API struct, and move all wiring to the composition boundary.

Failure 2: ABI Fragility

Symptom: plugins break after toolchain updates.
Fix: C ABI boundary, strict version negotiation, avoid STL types in the ABI.

Failure 3: Memory Ownership Bugs

Symptom: crashes on destruction or random heap corruption.
Fix: single-owner allocation rules; explicit `create/destroy`; allocator callbacks if needed.

Failure 4: Unbounded Plugin Impact

Symptom: a slow plugin destroys tail latency.
Fix: isolate plugins behind queues, timeouts, and circuit breakers; limit concurrency; measure plugin contribution.

Failure 5: Unsafe Load Paths

Symptom: untrusted code loaded accidentally.
Fix: controlled search paths, explicit configuration, signature verification policies.

21.1.15 Engineering Checklist

1. Define a minimal, stable plugin contract with explicit versioning.

2. Prefer a C ABI boundary; keep modern C++ internal to each side.
3. Do not pass STL containers across the ABI boundary; use explicit buffers and lengths.
4. Make ownership explicit: plugin destroys plugin objects; host does not unload too early.
5. Specify thread-safety and blocking contracts for plugin calls and callbacks.
6. Implement safe discovery: controlled directories, explicit allow/deny rules.
7. Measure plugin impact on latency and throughput; enforce resource limits.

21.1.16 Summary

Runtime extensibility allows long-lived C++ systems to evolve without forking the core. It succeeds only when treated as architecture:

- a minimal stable boundary,
- explicit lifecycle and ownership rules,
- ABI and version negotiation,
- concurrency contracts,
- security-aware discovery and loading,
- measurable impact and enforceable limits.

A plugin system is not merely dynamic loading. It is an engineered boundary that turns “change” into a supported, controlled, and testable capability.

21.2 ABI-Safe Plugin Design

21.2.1 Why ABI Safety Is the Core Problem

A plugin architecture is not primarily about dynamic loading. It is about **binary compatibility** across independently built artifacts.
In a long-lived system, plugins and the host may differ by:

- compiler family and compiler version,
- standard library implementation and version,
- build flags (exceptions, RTTI, LTO, struct packing, debug/release),
- CPU features and calling conventions,
- operating system loader behavior,
- security hardening settings.

The ABI boundary must therefore be engineered as a stable contract, not assumed from C++ types.

21.2.2 The Two Golden Strategies

Strategy A: C ABI Boundary (Most Robust)

Expose a C ABI:

- stable calling convention and symbol naming via `extern "C"`,
- explicit ownership rules,
- explicit buffers and sizes rather than STL types,

- simple POD structs with versioning.

Internally, both host and plugin can still be modern C++.

Strategy B: Toolchain-Locked C++ ABI (Sometimes Acceptable)

A pure C++ boundary can work only when you can enforce:

- same compiler family and version,
- same standard library,
- consistent build options across host and plugins,
- controlled distribution environment.

This is common in closed ecosystems, but fragile for third-party plugin distribution.

21.2.3 ABI Stability Principles

Principle 1: Never Let Exceptions Cross the Boundary

Across a binary boundary, exception propagation can break due to:

- incompatible runtime libraries,
- differing unwind mechanisms or metadata,
- mismatched exception types and RTTI.

Rule:

> Catch all exceptions inside the plugin/host boundary and translate them into explicit error codes.

Principle 2: Do Not Pass STL Types Across the Boundary

Avoid passing across the ABI boundary:

- `std::string`, `std::vector`, `std::function`,
- iterators, allocators, `std::optional`, `std::variant`,
- any template-heavy or inline-heavy types.

Even if it works today, it becomes a compatibility trap over time.
Use instead:

- plain pointers + length (`char* + size`, `void* + size`),
- simple POD structs with fixed layout,
- opaque handles.

Principle 3: Use Opaque Handles for Objects

Never expose C++ class layout. Use:

- forward-declared opaque struct types,
- create/destroy functions owned by the side that allocates.

Principle 4: One Side Owns Allocation

Mixing allocators between host and plugin is a common cause of heap corruption, especially on platforms where CRTs differ.
Rule:

> The side that allocates must provide the function that frees, or the boundary must provide explicit allocator callbacks.

Principle 5: Version Everything Explicitly

A stable plugin system requires version negotiation:

- ABI version (break when binary layout or function signatures change),
- API feature flags (add capabilities without breaking),
- struct size/version fields for extensible data structures.

21.2.4 A Reference ABI Contract (C ABI, Versioned, Extensible)

Header: ABI-Safe Types and Entry Point

```
#pragma once
#include <stdint.h>
#include <stddef.h>

#if defined(_WIN32)
  #define PLUGIN_EXPORT __declspec(dllexport)
  #define PLUGIN_IMPORT __declspec(dllimport)
#else
  #define PLUGIN_EXPORT __attribute__((visibility("default")))
  #define PLUGIN_IMPORT
#endif

#ifdef __cplusplus
extern "C" {
#endif

// ABI version changes ONLY on breaking binary changes.
enum { PLUGIN_ABI_VERSION = 1 };

// Generic status codes: stable and easy to bind from other languages.
```

```
typedef enum PluginStatus {
  PLUGIN_OK = 0,
  PLUGIN_E_INVALID_ARGUMENT = 1,
  PLUGIN_E_UNSUPPORTED_ABI  = 2,
  PLUGIN_E_OUT_OF_MEMORY    = 3,
  PLUGIN_E_INTERNAL         = 4,
  PLUGIN_E_NOT_IMPLEMENTED  = 5
} PluginStatus;

// Opaque handle type: never reveal its layout publicly.
typedef struct PluginService PluginService;

// A versioned, size-tagged struct pattern for ABI extensibility.
typedef struct PluginInfoV1 {
  uint32_t struct_size;     // must be set by plugin to sizeof(PluginInfoV1)
  uint32_t abi_version;     // must match PLUGIN_ABI_VERSION
  const char* name;         // UTF-8, null-terminated, owned by plugin (static storage
  ↪  recommended)
  const char* vendor;       // same
  const char* version;      // same
  uint64_t features;        // bitmask
} PluginInfoV1;

// Host callbacks: avoid passing C++ objects across boundary.
typedef struct HostApiV1 {
  uint32_t struct_size; // set by host to sizeof(HostApiV1)

  // Logging: host owns policy; plugin calls it.
  void (*log)(int level, const char* msg) noexcept;

  // Optional allocator hooks for cross-boundary buffers.
  void* (*alloc)(size_t n) noexcept;
  void  (*dealloc)(void* p) noexcept;
```

```
} HostApiV1;

// Function table: stable function pointers avoid C++ vtable ABI issues.
typedef struct PluginExportsV1 {
  uint32_t struct_size; // set by plugin to sizeof(PluginExportsV1)
  PluginInfoV1 info;

  // Lifecycle:
  PluginStatus (*create)(const HostApiV1* host, PluginService** out_service) noexcept;
  void         (*destroy)(PluginService* service) noexcept;

  // Example service operations:
  PluginStatus (*process)(PluginService* service,
                          const uint8_t* in_bytes, size_t in_size,
                          uint8_t** out_bytes, size_t* out_size) noexcept;

  // Free buffers allocated by plugin for the host (if plugin allocates output).
  void (*free_buffer)(uint8_t* p) noexcept;

} PluginExportsV1;

// Single required symbol: returns exports if compatible.
// Host passes its ABI version; plugin validates and fills out exports.
PLUGIN_EXPORT PluginStatus
plugin_get_exports(uint32_t host_abi_version, PluginExportsV1* out_exports) noexcept;

#ifdef __cplusplus
} // extern "C"
#endif
```

Why this contract is ABI-safe.

- **No C++ types** cross the boundary.

- **No exceptions** cross the boundary (all `noexcept`).
- **Opaque handles** prevent layout coupling.
- **Function table** avoids vtable ABI dependence.
- **struct_size** enables forward-compatible extension.
- **Explicit buffer ownership** avoids allocator mismatch.

21.2.5 Plugin-Side Implementation (C ABI Outside, Modern C++ Inside)

Plugin Service Internal Type

```
#include <new>
#include <cstring>

struct PluginService {
  const HostApiV1* host;
  int state;
};

static void safe_log(const HostApiV1* host, int level, const char* msg) noexcept {
  if (host && host->log) host->log(level, msg);
}
```

Lifecycle and Operations

```
static PluginStatus create_service(const HostApiV1* host, PluginService** out) noexcept {
  if (!out) return PLUGIN_E_INVALID_ARGUMENT;
  *out = nullptr;

  try {
    PluginService* s = new (std::nothrow) PluginService{};
```

```
    if (!s) return PLUGIN_E_OUT_OF_MEMORY;
    s->host = host;
    s->state = 0;
    *out = s;
    safe_log(host, 1, "plugin: service created");
    return PLUGIN_OK;
  } catch (...) {
    safe_log(host, 0, "plugin: create_service exception");
    return PLUGIN_E_INTERNAL;
  }
}

static void destroy_service(PluginService* s) noexcept {
  delete s;
}

static PluginStatus free_buffer(uint8_t* p) noexcept {
  // If plugin allocates outputs with new[] or malloc, it must free them consistently.
  delete[] p;
  return PLUGIN_OK;
}

static PluginStatus process_bytes(PluginService* s,
                                  const uint8_t* in, size_t in_size,
                                  uint8_t** out, size_t* out_size) noexcept {
  if (!s || (!in && in_size != 0) || !out || !out_size) return PLUGIN_E_INVALID_ARGUMENT;
  *out = nullptr;
  *out_size = 0;

  try {
    // Example: echo input with a suffix.
    const char suffix[] = " processed";
    const size_t suffix_n = sizeof(suffix) - 1;
```

```
    size_t total = in_size + suffix_n;
    uint8_t* buf = new (std::nothrow) uint8_t[total];
    if (!buf) return PLUGIN_E_OUT_OF_MEMORY;

    if (in_size) std::memcpy(buf, in, in_size);
    std::memcpy(buf + in_size, suffix, suffix_n);

    *out = buf;
    *out_size = total;
    return PLUGIN_OK;
  } catch (...) {
    safe_log(s->host, 0, "plugin: process exception");
    return PLUGIN_E_INTERNAL;
  }
}
```

Export Table and Entry Point

```
extern "C" PLUGIN_EXPORT PluginStatus
plugin_get_exports(uint32_t host_abi_version, PluginExportsV1* out) noexcept {
  if (!out) return PLUGIN_E_INVALID_ARGUMENT;
  if (host_abi_version != PLUGIN_ABI_VERSION) return PLUGIN_E_UNSUPPORTED_ABI;

  // Fill size-tagged struct fields.
  out->struct_size = (uint32_t)sizeof(PluginExportsV1);

  out->info.struct_size = (uint32_t)sizeof(PluginInfoV1);
  out->info.abi_version = PLUGIN_ABI_VERSION;
  out->info.name    = "example.plugin";
  out->info.vendor  = "ExampleVendor";
  out->info.version = "1.0.0";
  out->info.features = 1ull; // FEATURE bit 0 for demonstration
```

```
  out->create = &create_service;
  out->destroy = &destroy_service;
  out->process = &process_bytes;
  out->free_buffer = (void(*)(uint8_t*) noexcept)&free_buffer;

  return PLUGIN_OK;
}
```

21.2.6 Host-Side Safety Rules

Rule 1: Validate Struct Sizes Before Use

Because structs may grow in future versions, the host must:

- check `struct_size` is at least the size it expects,
- only access fields within the known prefix.

Rule 2: Never Unload While Objects Exist

The host must ensure:

- all `PluginService` instances are destroyed before unloading the shared library,
- no function pointers from the table are used after unload.

Rule 3: Define Buffer Ownership Precisely

Two safe patterns:

- **Plugin-allocates output:** plugin returns buffer pointer; host frees via `free_buffer`.
- **Host-provides output buffer:** host allocates and passes buffer; plugin fills it.

The contract must pick one or support both explicitly; ambiguity causes corruption.

21.2.7 Cross-Platform ABI Pitfalls and Defenses

Symbol Visibility

On ELF platforms, default visibility can export unintended symbols and increase collision risk. On Windows, exports must be explicit for stable loading.
Defense:

- export only the single entry point symbol by default,
- hide internal symbols (visibility hidden) and expose only ABI surface.

Calling Convention

On Windows in particular, calling convention mismatches can crash immediately. Defense:

- keep boundary functions in C with consistent compiler defaults,
- if you specify calling conventions, specify them consistently in the public header for both sides.

Struct Packing and Alignment

If packing pragmas differ, struct layout differs. Defense:

- do not use `#pragma pack` in public ABI headers,
- keep ABI structs simple and naturally aligned,
- include `struct_size` and validate it.

Endianness and Data Representation

If plugins may run in heterogeneous environments, treat binary blobs as protocols:

- define endianness explicitly,
- define integer widths explicitly (`uint32_t`, `uint64_t`),
- avoid passing raw structs as serialized data.

21.2.8 Designing the ABI Surface: Minimal and Stable

A stable ABI surface avoids:

- templates,
- inline-heavy headers,
- complex ownership graphs,
- callbacks that capture C++ objects.

It prefers:

- handle-based APIs (opaque pointers),
- function tables,
- explicit buffers and sizes,
- explicit error codes,
- explicit feature negotiation.

21.2.9 Feature Negotiation Without ABI Breaks

Avoid adding new required functions directly to the exports struct without planning. Instead, use one of these:

- include a `struct_size` field and allow the struct to grow,
- expose an extension query function: `get_interface("name", version)`,
- use feature bits and optional function pointers that may be null.

Optional Function Pointer Pattern

```
typedef PluginStatus (*OptionalFn)(PluginService*) noexcept;

typedef struct PluginExportsV2 {
  uint32_t struct_size;
  PluginInfoV1 info;
  PluginStatus (*create)(const HostApiV1*, PluginService**) noexcept;
  void (*destroy)(PluginService*) noexcept;

  // Optional: may be NULL if not supported.
  OptionalFn reset;
} PluginExportsV2;
```

Host rule:

- check `struct_size` and function pointer before calling.

21.2.10 Threading and Reentrancy Contracts

ABI safety is not only binary layout; it is also behavioral contract.
The plugin contract must specify:

- whether service instances are thread-safe or thread-confined,
- whether callbacks into host may occur and on which threads,
- whether functions may block,
- whether functions are reentrant.

A conservative, safe default:

- each `PluginService` instance is thread-confined unless stated otherwise,
- host serializes calls or uses separate instances per thread,
- plugin must not call host functions from unknown threads unless explicitly allowed.

21.2.11 Testing ABI Safety

A robust plugin ecosystem requires tests beyond unit tests:

- **Compatibility tests:** load an old plugin with a new host and verify behavior.
- **Negative tests:** wrong ABI version must be rejected cleanly.
- **Fuzz tests at the boundary:** invalid inputs must not crash the host.
- **Lifecycle tests:** create/destroy cycles, unload safety, repeated loads.
- **Stress tests:** concurrency and memory pressure at boundary.

21.2.12 Engineering Checklist

1. Prefer a C ABI boundary with `extern "C"` and a single entry point.
2. Use opaque handles and function tables; never expose C++ object layout.
3. Forbid exceptions across boundary; use `noexcept` and explicit status codes.
4. Do not pass STL types across boundary; use pointer+size buffers and POD structs.
5. Enforce one-side allocation: provide `free_buffer` or allocator callbacks.
6. Version aggressively: ABI version + struct sizes + feature negotiation.
7. Specify threading and blocking semantics explicitly as part of the contract.
8. Export only the ABI surface; hide internal symbols; validate everything on load.

21.2.13 Summary

ABI-safe plugin design is the discipline of treating the plugin boundary as a stable protocol:

- minimal surface, explicit ownership, explicit versioning,
- C ABI with handle-based objects and function tables,
- no exceptions or STL across the boundary,
- validated struct sizes and feature negotiation,
- clear lifecycle, concurrency, and security rules.

When these constraints are enforced, runtime extensibility becomes a reliable architectural capability instead of a source of fragile, production-only failures.

21.3 Versioning and Compatibility

21.3.1 Compatibility Is a First-Class Architectural Feature

In a component-based or plugin architecture, versioning is not documentation. It is **runtime safety**.
Without a compatibility strategy, the system eventually becomes one of:

- a brittle monolith that cannot accept external plugins safely,
- a fragmented ecosystem where every host build requires matching plugin rebuilds,
- a system that silently loads incompatible components and fails unpredictably.

A long-lived system must support two independent rates of change:

- the host evolves continuously,
- extensions evolve independently (and sometimes slower).

Therefore, compatibility must be engineered as a contract:

> A component boundary is a protocol, and protocols must be versioned deliberately.

21.3.2 Terminology: Distinguish the Four Kinds of Versioning

1) Product Version

The human-facing version of the application. Useful for support, but insufficient for binary compatibility.

2) API Version

The logical interface contract:

- which functions exist,
- what they mean,
- what errors they return.

API compatibility is about behavior and semantics.

3) ABI Version

The binary contract:

- function signatures at the binary level,
- calling conventions,
- struct layout and alignment,
- symbol presence and name resolution.

ABI compatibility is about linking and runtime calling safety.

4) Data/Protocol Version

The version of persisted or exchanged data:

- serialized payload formats,
- on-disk storage schema,
- network messages.

A robust component architecture defines *all four* and ties them together.

21.3.3 The Compatibility Promise: Define What You Guarantee

Before choosing a version scheme, define the promise:

- **Backward compatibility:** new host runs old plugins.
- **Forward compatibility:** old host runs new plugins (rare and limited).
- **Side-by-side compatibility:** multiple plugin versions can coexist.
- **Graceful degradation:** unknown features are ignored or treated as optional.

Most ecosystems aim for:

- backward compatibility for a defined window,
- forward compatibility only through feature negotiation and optional interfaces.

21.3.4 The Practical Rule: ABI Versions Change Only on Breaking Changes

The single most important rule:

> Only increment the ABI version when binary compatibility is broken.

Examples of ABI-breaking changes:

- changing function signatures,
- changing struct layout (reordering fields, changing types, packing),
- removing or renaming exported symbols,
- changing calling convention assumptions,
- changing ownership rules in a way that changes who frees memory.

Examples of ABI-preserving changes (if designed correctly):

- adding new optional functions behind feature checks,
- extending size-tagged structs (host reads only known prefix),
- adding new feature flags while keeping existing ones unchanged.

21.3.5 The Foundation: Negotiation on Load

A compatible host must **negotiate** with the plugin at load time:

1. validate ABI version,
2. validate the size/version of exported tables,
3. validate required capabilities,
4. select an interface version (if multiple are supported),
5. reject incompatible plugins deterministically.

This is a security and stability requirement, not a convenience.

21.3.6 A Robust Pattern: Size-Tagged Structs and Function Tables

Two patterns dominate reliable plugin compatibility:

1) Size-Tagged Structs (Extensible Layout)

Each exported struct begins with:

- `struct_size` field (set by producer),
- `abi_version` field (or separate).

The consumer rule:

- treat any missing fields beyond `struct_size` as absent,
- never read beyond the minimum of `struct_size` and expected size.

2) Function Tables

Instead of relying on C++ vtables or symbol sets, the plugin exports a stable table:

- a single entry point returns the table,
- the host calls through function pointers,
- new functions can be added to the end while preserving the prefix.

Reference Example: Versioned Exports

```
#include <cstdint>
#include <cstddef>

extern "C" {

enum : std::uint32_t { ABI_V1 = 1 };

enum Status : std::uint32_t {
  OK = 0,
  E_UNSUPPORTED_ABI = 1,
  E_INVALID = 2
};

struct InfoV1 {
  std::uint32_t struct_size;
  std::uint32_t abi_version;
```

```
  const char*   name;
  const char*   version;    // semantic product version string
  std::uint64_t features;   // feature bits
};

struct Service; // opaque

using CreateFn  = Status (*)(Service** out) noexcept;
using DestroyFn = void   (*)(Service*) noexcept;

struct ExportsV1 {
  std::uint32_t struct_size;
  InfoV1 info;

  CreateFn  create;
  DestroyFn destroy;

  // V1 functions...
};

Status plugin_get_exports(std::uint32_t host_abi, ExportsV1* out) noexcept;

} // extern "C"
```

Compatibility mechanism.

- ABI mismatch: reject immediately.
- Struct grows: host reads prefix only.
- New optional capabilities: represented by feature bits or extra function pointers.

21.3.7 Semantic Versioning: Useful, but Not Sufficient

Semantic Versioning (major.minor.patch) is useful for communicating API changes, but it cannot replace ABI versioning.
In plugin systems:

- **SemVer major** often correlates with API breaking changes.
- **ABI version** must correlate with binary breaking changes.

These may diverge:

- you can introduce API changes that preserve ABI (e.g., new optional functions),
- you can break ABI without changing high-level semantics (e.g., layout changes).

Therefore:

> Use semantic versions for humans and API policies; use ABI versions for the loader.

21.3.8 Feature Negotiation: Compatibility Without Rebuilds

Forward compatibility is typically achieved through **feature negotiation**.
Mechanisms:

- feature bitmasks,
- optional function pointers that may be null,
- interface query functions (get interface by name/version),
- capability descriptors (structured metadata).

Feature Bits and Optional Functions

```
enum : std::uint64_t {
  FEAT_PROCESS   = 1ull << 0,
  FEAT_STREAMING = 1ull << 1,
  FEAT_ADMIN     = 1ull << 2
};

using ResetFn = Status (*)(Service*) noexcept;

struct ExportsV2 {
  std::uint32_t struct_size;
  InfoV1 info;

  CreateFn  create;
  DestroyFn destroy;

  // Optional; only valid if info.features & FEAT_ADMIN
  ResetFn reset;
};
```

Host rule:

- check `struct_size`,
- check feature bit,
- check function pointer not null.

21.3.9 Multi-Version Interfaces: Supporting Old and New Simultaneously

A robust host often supports multiple ABI versions simultaneously:

- it can load V1 plugins and wrap them into a modern internal interface,

- it can load V2 plugins and use newer features directly.

This is a powerful anti-erosion technique because it prevents:

- forcing ecosystem-wide rebuilds,
- freezing the host at an old version just to support old plugins.

Adapter Pattern: Old Plugin Wrapped into New Internal API

```
struct IProcessor {
  virtual ~IProcessor() = default;
  virtual Status process_bytes(const std::uint8_t* in, std::size_t n) = 0;
};

// Adapter around a V1 plugin.
class ProcessorFromV1 final : public IProcessor {
public:
  explicit ProcessorFromV1(ExportsV1 ex) : ex_(ex), svc_(nullptr) {
    ex_.create(&svc_);
  }
  ~ProcessorFromV1() override {
    if (svc_) ex_.destroy(svc_);
  }
  Status process_bytes(const std::uint8_t* in, std::size_t n) override {
    (void)in; (void)n;
    return OK; // placeholder: call V1 functions
  }

private:
  ExportsV1 ex_;
  Service* svc_;
};
```

Design effect. The host stays modern internally while supporting legacy plugins externally.

21.3.10 Compatibility of Data Formats: Versioned Serialization

Even if ABI is compatible, plugins often exchange data. Data compatibility requires:

- explicit schema versions in payloads,
- backward-compatible readers (new code reads old data),
- strict validation to prevent malformed input from crashing the host.

A practical rule:

> Readers should be liberal in what they accept only within strict validation rules.
> Writers should be conservative and produce canonical forms.

In plugin systems, payload versioning should be separate from ABI versioning, because data formats may evolve independently.

21.3.11 Deprecation and Removal: Managing the Compatibility Window

Long-lived systems must define a compatibility window:

- how many ABI versions are supported,
- how long an old plugin remains loadable,
- how deprecation is communicated and enforced.

A practical deprecation policy includes:

- warnings on load for deprecated versions,
- telemetry or reporting of loaded plugin versions (when available),

- a published schedule for removing support.

Architectural principle:

> Compatibility without a removal policy becomes permanent legacy.

21.3.12 Failure Handling: Compatibility Must Fail Safe

If compatibility checks fail, the failure must be:

- deterministic,
- early (on load),
- explicit (clear reason codes),
- safe (no partial initialization left behind).

Never allow:

- partial function tables,
- null function calls without checks,
- "best effort" ABI calls that may corrupt memory.

21.3.13 Practical Compatibility Checklist for C++ Plugin Ecosystems

1. Separate product version, API version, ABI version, and data/protocol version.
2. Negotiate at load time: ABI check, struct size check, required capability check.
3. Prefer size-tagged structs and function tables; keep a stable prefix forever.

4. Use feature negotiation (bits, optional functions, interface queries) to extend without breaking.
5. Never pass exceptions or STL across the boundary; keep boundary C-like.
6. Support multiple ABI versions in the host when ecosystem stability matters.
7. Version serialized data formats explicitly; validate input strictly.
8. Define a deprecation window and removal policy; warn and measure usage before removal.
9. Fail safe: reject incompatible plugins before any unsafe call occurs.

21.3.14 Summary

Versioning and compatibility in plugin architectures is an engineering system, not a number scheme:

- ABI versions protect binary safety,
- API and semantic versions communicate logical evolution,
- feature negotiation enables forward growth without rebuild pressure,
- multi-version host support stabilizes the ecosystem,
- versioned data formats preserve long-term interoperability,
- deprecation policies prevent permanent legacy accumulation.

When treated as architecture, compatibility becomes predictable and safe, allowing independent evolution of host and plugins across years of change.

21.4 Dependency Isolation

21.4.1 What Dependency Isolation Means in Plugin Systems

In a component-based or plugin architecture, **dependency isolation** is the discipline of ensuring that:

- a plugin does not accidentally depend on the host's internal headers or libraries,
- plugins do not conflict with each other through shared third-party dependencies,
- the host can evolve without forcing plugin rebuilds,
- loading a plugin cannot silently change the host's runtime behavior through global side effects.

Without isolation, a plugin ecosystem becomes fragile:

- a plugin update breaks unrelated plugins,
- dependency upgrades become impossible without ecosystem-wide rebuilds,
- the host becomes locked to old library versions,
- runtime loader behavior becomes unpredictable.

Architectural goal:

> Plugins should be *replaceable units* with controlled integration surfaces, not hidden compile- and link-time entanglements.

21.4.2 The Four Dependency Planes You Must Isolate

Dependency isolation is multi-dimensional. A robust design addresses these four planes:

1) Compile-Time Isolation

Prevent plugins from including host internals:

- expose a small SDK (headers) that defines the plugin ABI/API,
- keep host private headers inaccessible,
- avoid transitive includes in the SDK,
- make layering rules enforceable by the build system.

2) Link-Time Isolation

Prevent plugins from linking against host implementation libraries:

- plugin links only against the stable SDK boundary,
- host exposes functionality through function tables / callback APIs,
- avoid exporting internal symbols from the host.

3) Load-Time Isolation (Dynamic Loader Behavior)

Prevent dependency collisions and unintended resolution:

- control search paths,
- avoid accidental symbol interposition,
- prevent a plugin from overriding host symbols (or vice versa),
- ensure deterministic resolution of the plugin's third-party libraries.

4) Runtime Isolation (Global State and Side Effects)

Prevent runtime cross-talk:

- avoid global singletons shared implicitly across modules,
- avoid process-wide mutable state changes (locale, signal handlers, environment variables),
- control thread creation and background work,
- sandbox plugin resource usage when necessary.

Most “dependency problems” are not only compile-time problems; they are often load-time and runtime problems.

21.4.3 The Stable SDK: A Hard Boundary Between Host and Plugins

The foundation of isolation is the **SDK boundary**:

- a small set of headers and/or module interfaces,
- a C ABI or carefully controlled interface,
- fixed ownership, error, and concurrency contracts.

Rules:

- the SDK is the **only** header surface a plugin may include from the host,
- the SDK must avoid including host internal headers,
- the SDK should be stable and versioned.

SDK Layout Pattern

```
sdk/
  include/
    plugin_api.h         (C ABI boundary)
    plugin_host_api.h    (host callback surface)
  lib/                   (optional import libs)
host/
  src/                   (private)
  include_internal/      (private)
plugins/
  myplugin/
```

21.4.4 Preventing Compile-Time Leakage

Compile-time leakage is the most common first erosion step: a developer includes an internal host header because it is convenient.
Defense mechanisms:

- do not ship internal headers,
- do not add host internal include paths to the plugin build,
- keep SDK headers minimal and free of heavy transitive dependencies,
- provide mocks and test harnesses in the SDK so plugin authors are not tempted to include internals.

A Minimal Host API Surface via Function Table

```c
#pragma once
#include <stdint.h>
#include <stddef.h>
```

```c
#ifdef __cplusplus
extern "C" {
#endif

typedef struct HostApiV1 {
  uint32_t struct_size;

  void (*log)(int level, const char* msg) noexcept;

  // Controlled services: avoid exposing internal types.
  uint64_t (*get_config_u64)(const char* key, uint64_t def) noexcept;

  // Optional allocator hooks for explicit ownership.
  void* (*alloc)(size_t n) noexcept;
  void  (*dealloc)(void* p) noexcept;
} HostApiV1;

#ifdef __cplusplus
} // extern "C"
#endif
```

Isolation effect. The plugin obtains required host services without including or linking to host internals.

21.4.5 Link-Time Isolation: Avoid Linking Plugins to Host Implementation

If a plugin links against host internal libraries, you have lost isolation:

- the plugin now requires exact symbol and ABI matches,
- host refactoring breaks plugins,
- the plugin may call host internals in unexpected orders.

Preferred approach:

- host exposes services through the SDK function table,
- plugin calls through function pointers,
- host retains control of implementation and evolution.

21.4.6 Load-Time Isolation: Preventing Dependency Collisions

Dynamic linking introduces a unique class of failures:

- **diamond dependency conflicts:** host uses libX v1, plugin uses libX v2,
- **symbol interposition:** one module's symbols override another's,
- **search path injection:** loader finds dependencies in unintended locations,
- **ODR-like collisions across shared objects:** duplicated globals with same names.

The architecture must define a loader policy that minimizes these risks.

Policy 1: Control the Plugin Search Path

Rules:

- load plugins only from controlled directories,
- do not search user-writable paths by default,
- do not rely on process-wide environment variables for plugin discovery.

Policy 2: Prefer “Local” Symbol Binding for Plugins

A strong isolation posture aims to:

- keep plugin symbols private unless explicitly exported,
- avoid exposing host internal symbols globally,
- prevent plugins from overriding host symbols inadvertently.

This is achieved by:

- exporting only the plugin entry point and required ABI surface,
- using hidden/default visibility strategies on ELF,
- avoiding global symbol exposure of host internals.

Policy 3: Vendor or Bundle Plugin Dependencies When Needed

If plugin dependencies must not collide with host dependencies:

- bundle plugin-specific dependencies next to the plugin,
- load them from a plugin-private directory,
- avoid relying on global system resolution where collisions are likely.

Architectural trade-off:

- bundling improves isolation,
- but increases size and update complexity.

21.4.7 Runtime Isolation: Preventing Global State Contamination

Even with clean compilation and linking, plugins can still interfere through global state:

- changing locale or time zone settings,
- registering signal handlers,
- setting environment variables,
- installing global hooks,
- spawning uncontrolled background threads,
- writing to global log sinks in unexpected formats.

Practical constraints:

- plugins must treat the process as a shared environment,
- the SDK should provide controlled facilities (logging, config, scheduling),
- the host should own global policy.

Provide Host-Controlled Services Instead of Globals

```
typedef struct HostSchedulerV1 {
  uint32_t struct_size;

  // Schedule work on host-managed execution context.
  void (*post)(void (*fn)(void*), void* user) noexcept;

  // Optional: create a host-managed timer.
  void (*set_timer_ms)(uint32_t ms, void (*fn)(void*), void* user) noexcept;
} HostSchedulerV1;
```

Isolation effect. Plugins avoid creating their own uncontrolled thread pools and timers, reducing global interference and simplifying shutdown.

21.4.8 The “No Shared CRT” and “No Mixed Allocator” Rules

A common failure mode in plugin systems is memory ownership mismatch:

- host allocates, plugin frees (or the reverse),
- host and plugin use different runtime libraries,
- debug/release CRT mismatch on some platforms.

Isolation-friendly rules:

- the boundary defines who allocates and who frees,
- if outputs cross the boundary, provide `free_buffer` callbacks,
- for complex allocation patterns, provide allocator functions in `HostApi`.

Explicit Free Callback Pattern

```
typedef struct PluginExportsV1 {
  uint32_t struct_size;

  // Plugin returns buffer allocated by plugin.
  int (*process)(const uint8_t* in, size_t n,
                 uint8_t** out, size_t* out_n) noexcept;

  // Host frees via plugin, never directly.
  void (*free_buffer)(uint8_t* p) noexcept;
} PluginExportsV1;
```

21.4.9 Avoiding ODR-Style Collisions Across Components

Even though the One Definition Rule is a language rule, similar collisions appear at runtime:

- two shared libraries may each carry a copy of a header-only library with globals,
- two different versions of an inline-heavy library may both be present,
- static locals and singletons may diverge across boundaries.

Mitigation strategies:

- keep header-only libraries out of the ABI boundary,
- avoid global state in shared libraries used as plugins,
- prefer handle-based state stored inside plugin instances,
- keep exported symbols minimal and explicitly versioned.

21.4.10 Isolation Between Plugins: Preventing Plugin-to-Plugin Coupling

Plugins should not depend on each other unless your architecture explicitly supports that. Otherwise:

- upgrades become dependency puzzles,
- load order becomes a hidden contract,
- failure in one plugin cascades to others.

Preferred model:

- plugins depend only on the host SDK,
- the host offers shared services and mediation,
- plugins communicate through host-defined messages/events, not direct linking.

Host-Mediated Messaging Sketch

```
typedef struct HostBusV1 {
  uint32_t struct_size;

  // Publish an event; host routes to interested plugins.
  void (*publish)(const char* topic,
                  const uint8_t* payload, size_t n) noexcept;

  // Subscribe callback registered by plugin.
  int (*subscribe)(const char* topic,
                   void (*cb)(const uint8_t*, size_t, void*), void* user) noexcept;
} HostBusV1;
```

Isolation effect. Plugins remain decoupled; host controls routing, threading, and backpressure.

21.4.11 Deployment Isolation: Packaging and Reproducibility

Isolation is reinforced at deployment time:

- ship each plugin with explicit metadata (name, version, ABI version, required features),
- place plugins and their dependencies in a controlled directory layout,
- keep plugin loading deterministic and configuration-driven.

A robust deployment model supports:

- side-by-side plugin versions,
- safe rollback,
- compatibility validation before enabling plugins.

21.4.12 Testing Dependency Isolation

Dependency isolation must be tested as architecture, not assumed.
Critical tests:

- load plugin in a clean process with minimal environment,
- verify plugin does not require host internal libraries,
- run with two plugins that use different versions of a third-party library (if allowed) and validate behavior,
- unload safety: repeated load/unload cycles without leaks or crashes,
- concurrency stress: plugin calls under load without global interference.

21.4.13 Engineering Checklist

1. Provide a small, stable SDK; plugins must not include host internals.
2. Enforce compile-time isolation via build targets and controlled include paths.
3. Enforce link-time isolation: plugins link only to the SDK boundary.
4. Control load-time resolution: safe search paths, minimal exports, avoid symbol interposition.
5. Define explicit ownership rules for memory; never mix allocators across the boundary.
6. Prevent runtime global-state contamination by providing host-controlled services (logging, scheduling, messaging).
7. Keep plugins independent; use host-mediated messaging rather than plugin-to-plugin linking.

8. Test isolation explicitly: clean environment loads, collision tests, lifecycle stress tests.

21.4.14 Summary

Dependency isolation is what turns “dynamic loading” into a sustainable ecosystem:

- the SDK forms a hard boundary,
- build and link rules prevent accidental coupling,
- loader policy prevents dependency collisions,
- runtime services prevent global-state interference,
- explicit memory ownership prevents allocator mismatches,
- host-mediated communication prevents plugin-to-plugin entanglement.

When these constraints are enforced, plugins become truly replaceable units, and the host can evolve without locking the entire ecosystem to one dependency graph.

Data-Oriented Design

22.1 Object-Oriented vs Data-Oriented Thinking

22.1.1 The Real Difference: What You Optimize For

Object-Oriented Design (OOD) and Data-Oriented Design (DOD) are often presented as competing ideologies. In practice, they are **two optimization strategies** applied at different levels of a system.

- **OOD optimizes for:** modeling, encapsulation, substitutability, change isolation, and reasoning about behavior.
- **DOD optimizes for:** throughput, cache efficiency, predictable performance, and reasoning about data movement.

Neither is universally superior. The correct question is:

> Which constraints dominate this subsystem: change complexity or performance and data movement?

22.1.2 Object-Oriented Thinking: The Classical Mental Model

Object-oriented thinking begins with **entities** and **responsibilities**. We ask:

- What are the conceptual objects in the domain?
- What behavior belongs with each object's state?
- What invariants must each object enforce?
- How do objects collaborate through interfaces?

The primary unit of design is the **object**:

- data and behavior are packaged together,
- encapsulation hides representation details,
- polymorphism supports substitutability and late binding,
- composition builds larger systems from smaller units.

OOD Strengths in Long-Lived C++ Systems

OOD tends to work well when:

- business rules evolve often,
- invariants and correctness dominate,
- integration boundaries need stable interfaces,
- testability and separation of responsibilities matter more than raw throughput,
- object counts are moderate and locality is not the bottleneck.

Typical OOD Costs (Especially in Performance-Critical Domains)

In C++, a naive OOD implementation often introduces:

- pointer chasing (heap-allocated object graphs),
- poor locality (scattered allocations),
- virtual dispatch overhead in hot loops,
- fragmented memory causing cache and TLB misses,
- complex ownership and lifetime management for interconnected objects.

These costs are not theoretical; they appear under profiling and production load.

22.1.3 Data-Oriented Thinking: The Performance Mental Model

Data-oriented thinking begins with **work** and **data flow**. We ask:

- What does the CPU do in the hot path?
- What data is touched, in what order, and how frequently?
- How can we arrange data so the CPU reads it sequentially?
- How can we reduce branches and cache misses?

The primary unit of design becomes **data layout** and **processing pipelines**:

- group data by how it is accessed, not by conceptual ownership,
- separate stable data from rarely used data,
- process in batches to improve locality and amortize overhead,
- minimize indirection and unpredictable memory access.

DOD treats the CPU cache hierarchy as a first-class architectural constraint.

DOD Strengths

DOD is typically effective when:

- processing large collections (thousands to millions of elements),
- loops dominate runtime,
- predictable latency and throughput are required,
- parallelism and vectorization matter,
- memory bandwidth is the bottleneck.

DOD Costs

DOD can increase:

- conceptual distance from domain models,
- complexity of invariants and correctness reasoning (because behavior is separated),
- upfront design effort (profiling-driven layout decisions),
- coupling to performance assumptions (access patterns).

Therefore, DOD must be applied where it pays off: the hot paths.

22.1.4 The Core Tension: Encapsulation vs Locality

Encapsulation in OOD often implies:

- each object decides its representation,
- objects are individually allocated,

- object graphs represent relationships.

Locality in DOD often implies:

- representation is chosen by access patterns,
- collections are stored contiguously,
- relationships are represented by indices or handles rather than pointers.

This leads to a practical rule:

> Use OOD at subsystem boundaries and policy layers; use DOD within performance-critical cores.

22.1.5 A Concrete Illustration in Modern C++

Object-Oriented Style: Polymorphic Entities

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

struct Shape {
  virtual ~Shape() = default;
  virtual double area() const noexcept = 0;
};

struct Circle final : Shape {
  double r;
  explicit Circle(double radius) : r(radius) {}
  double area() const noexcept override { return 3.141592653589793 * r * r; }
};

struct Rect final : Shape {
```

```
  double w, h;
  Rect(double w_, double h_) : w(w_), h(h_) {}
  double area() const noexcept override { return w * h; }
};

double total_area(const std::vector<std::unique_ptr<Shape>>& shapes) {
  double sum = 0.0;
  for (auto& p : shapes) {
    sum += p->area(); // virtual dispatch + pointer chasing
  }
  return sum;
}
```

This design is expressive, extensible, and easy to evolve. But in a hot loop over millions of shapes, it can be slow due to:

- non-contiguous allocations,
- branch misprediction in virtual dispatch,
- cache misses when following pointers.

Data-Oriented Style: Separate Arrays by Shape Type

```
#include <vector>

struct Circles {
  std::vector<double> r;
};

struct Rects {
  std::vector<double> w;
  std::vector<double> h;
};
```

```
double total_area_dod(const Circles& c, const Rects& r) {
  double sum = 0.0;

  for (double radius : c.r) {
    sum += 3.141592653589793 * radius * radius; // tight loop, contiguous reads
  }
  for (size_t i = 0; i < r.w.size(); ++i) {
    sum += r.w[i] * r.h[i]; // contiguous reads, predictable branch behavior
  }
  return sum;
}
```

Here, we sacrifice some OOD extensibility for:

- fewer allocations,
- predictable memory access,
- reduced indirection,
- better cache use.

22.1.6 A Key DOD Technique: Structure of Arrays (SoA) vs Array of Structures (AoS)

DOD often replaces AoS with SoA.

AoS: Easy Modeling, Mixed Access

```
struct Particle {
  float x, y, z;
  float vx, vy, vz;
```

```
  float mass;
};

std::vector<Particle> particles;
```

SoA: Better Locality for Hot Fields

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

If the hot loop updates only positions and velocities, SoA avoids pulling `mass` into cache unnecessarily, reducing memory bandwidth waste.

22.1.7 Identity and References: Pointers vs Handles

OOD often uses pointers/references to represent identity and relationships. DOD often uses indices/handles:

- stable integer IDs,
- generational indices (to detect stale references),
- indirection tables to keep storage relocatable.

This enables:

- compact storage relocation and defragmentation,
- stable external identity while internal layout changes,
- better cache behavior by avoiding pointer graphs.

A Minimal Handle Pattern

```
#include <cstdint>

struct Handle {
  std::uint32_t index;
  std::uint32_t generation;
};
```

The underlying storage can validate `generation` to reject stale handles.

22.1.8 How to Decide: A Practical Decision Framework

Use OOD When

- requirements change frequently and unpredictably,
- correctness, invariants, and domain reasoning dominate,
- object counts are not extreme,
- the subsystem is mostly orchestration and coordination,
- you need plugin-like extensibility and stable interfaces.

Use DOD When

- data sets are large and processed frequently,
- profiling shows cache misses and memory bandwidth dominate,
- you have tight latency budgets or high throughput requirements,
- the same operations apply to many elements (batch processing),
- you want to exploit SIMD and parallel loops.

The Long-Lived Systems Rule

In long-lived systems, it is common to:

- keep a clean OOD boundary model at the edges (APIs, integration, domain),
- implement a DOD core for the performance-sensitive pipeline,
- connect them through explicit translation layers and stable DTOs.

22.1.9 The Hybrid Reality: OOD as Policy, DOD as Engine

A robust architecture often follows this separation:

- **Policy layer (OOD):** defines what should happen, orchestrates workflows, enforces invariants.
- **Engine layer (DOD):** executes bulk computation efficiently over well-laid-out data.

This reduces the risk of extremes:

- pure OOD everywhere leads to hidden performance cliffs,
- pure DOD everywhere leads to systems that are hard to evolve and reason about.

22.1.10 Engineering Checklist

1. Start with OOD for clarity and invariants; measure before rewriting.
2. Profile real workloads to identify hot loops and cache miss bottlenecks.
3. Apply DOD in the hot paths: contiguous storage, batching, SoA where beneficial.
4. Replace pointer graphs with handles/indices when locality matters.

5. Keep OOD boundaries stable; isolate DOD cores behind clear APIs.
6. Validate invariants at boundaries; keep correctness explicit when behavior is separated from data.
7. Re-evaluate regularly: access patterns change as features evolve.

22.1.11 Summary

Object-oriented thinking organizes software around **responsible entities** and stable interfaces. Data-oriented thinking organizes software around **data flow**, **memory layout**, and CPU execution efficiency.
For modern C++ long-lived systems:

- use OOD to express policy, boundaries, invariants, and evolution,
- use DOD to engineer high-throughput, low-latency engines,
- connect them deliberately through explicit interfaces and translation layers.

The correct architecture is not ideological. It is the disciplined combination that matches the system's dominant constraints.

22.2 Designing for Hardware

22.2.1 The Premise: Hardware Is the Ground Truth

Data-Oriented Design (DOD) starts from a practical premise:

> The performance characteristics of software are dominated by how it uses hardware resources: caches, memory bandwidth, branch predictors, vector units, and concurrency.

Modern CPUs are extremely fast at computation but comparatively slow at fetching data. Therefore, many real-world workloads are not compute-bound. They are **latency- and bandwidth-bound**.
Designing for hardware means:

- treating memory hierarchy as an architectural constraint,
- shaping data layout to match access patterns,
- reducing unpredictability (branches, pointer chasing, contention),
- structuring work to exploit parallelism and SIMD,
- measuring continuously because intuition is often wrong.

22.2.2 The Hardware Constraints That Matter Most

1) Cache Hierarchy and Locality

The CPU reads memory in **cache lines**. If the program touches data scattered across memory, it triggers many cache misses.
Two forms of locality dominate:

- **Spatial locality:** accessing nearby addresses (contiguous arrays, SoA).
- **Temporal locality:** reusing the same data soon (hot working sets).

Design implications:

- prefer contiguous storage for hot data,
- avoid pointer graphs in hot loops,
- keep frequently accessed fields together; separate cold fields.

2) Memory Bandwidth and Working Set Size

Even with caching, large working sets saturate memory bandwidth. Performance then scales with:

- bytes moved per useful operation,
- cache miss rate and prefetch effectiveness,
- avoided reads of cold fields.

Design implications:

- reduce the bytes processed per element,
- batch operations,
- compress or pack data where it improves bandwidth efficiency.

3) Branch Prediction and Control-Flow Unpredictability

Branches that are difficult to predict cause pipeline flushes. Unpredictable branches appear in:

- virtual calls and polymorphic dispatch in hot loops,
- irregular conditional logic per element,
- switch-heavy interpreters without batching or specialization.

Design implications:

- reduce branches in the hot path,
- restructure work to group similar cases together (sorting by type),
- consider table-driven logic or specialization for frequent cases.

4) SIMD and Vector Units

Modern CPUs can apply one operation to multiple data elements per instruction. But this depends on:

- contiguous data,
- regular access patterns,
- low aliasing ambiguity,
- minimal control-flow divergence.

Design implications:

- adopt SoA layouts for vectorizable loops,
- keep loops simple and predictable,
- separate scalar edge cases from bulk processing.

5) Multicore and Contention

Concurrency issues can destroy scaling:

- lock contention,
- false sharing (threads modify data on the same cache line),
- atomic hot spots,
- memory bandwidth saturation.

Design implications:

- partition data per thread (sharding),

- avoid shared mutable state in hot paths,
- align and pad to avoid false sharing where necessary,
- batch synchronization rather than synchronizing per element.

22.2.3 The Central Practice: Layout Data by Access Pattern

Hardware-aware design starts by answering:

> What is the hottest loop, and what exact bytes does it touch?

Then apply two transformations:

Hot/Cold Splitting

Separate frequently accessed data (hot) from rarely accessed data (cold).

```
#include <cstdint>
#include <string>
#include <vector>

struct EntityHot {
  float x, y, z;
  float vx, vy, vz;
  std::uint32_t flags;
};

struct EntityCold {
  std::string debug_name;
  std::uint64_t last_modified_ns;
  // other rarely used fields
};
```

```
struct Entities {
  std::vector<EntityHot> hot;
  std::vector<EntityCold> cold; // parallel arrays by index
};
```

Hardware effect. Hot loops over `hot` no longer pull cold fields into cache lines.

AoS to SoA (When Appropriate)

```
#include <vector>

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

void integrate(ParticlesSoA& p, float dt) {
  const size_t n = p.x.size();
  for (size_t i = 0; i < n; ++i) {
    p.x[i] += p.vx[i] * dt;
    p.y[i] += p.vy[i] * dt;
    p.z[i] += p.vz[i] * dt;
  }
}
```

Hardware effect. This loop is friendly to caches and vectorization because each array is contiguous and accessed linearly.

22.2.4 Reducing Pointer Chasing: Indices and Handles

Pointer-rich object graphs are expensive in hot paths due to cache misses. DOD commonly uses:

- indices into arrays,

- handles with generation counters,
- indirection tables to allow relocation without invalidating identity.

```
#include <cstdint>

struct Handle {
  std::uint32_t index;
  std::uint32_t generation;
};

struct Slot {
  std::uint32_t generation;
  std::uint32_t dense_index;
};

struct HandleTable {
  // sparse slots map handles to dense storage positions
  std::vector<Slot> slots;
};
```

Hardware effect. Dense storage remains contiguous and movable; external references remain stable and validated.

22.2.5 Avoiding False Sharing: Aligning and Sharding

False sharing occurs when multiple threads update different variables that happen to share the same cache line.
A typical defense is to shard per-thread counters or state, then reduce:

- each thread writes to its own cache line,
- aggregation is performed occasionally.

```
#include <atomic>
#include <cstddef>

struct alignas(64) PaddedCounter {
  std::atomic<std::uint64_t> value{0};
};

void increment(PaddedCounter& c) noexcept {
  c.value.fetch_add(1, std::memory_order_relaxed);
}
```

Design note. Padding is not a universal solution; it is a targeted defense for known contention points. Always validate with profiling because it increases memory footprint.

22.2.6 Batching: Amortize Overhead and Improve Prefetch

Batching is a hardware-friendly tactic:

- fewer branches per element,
- improved prefetch effectiveness,
- better use of instruction cache (same loop body repeated),
- fewer synchronization events.

Batch Processing Example

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

struct WorkItem { int id; float value; };
```

```cpp
float process_batch(const std::vector<WorkItem>& items) {
  float sum = 0.0f;
  for (const auto& w : items) {
    // hot loop: keep logic simple
    sum += w.value * 1.25f;
  }
  return sum;
}
```

The architectural point is not the arithmetic. It is the **shape**: a tight loop over contiguous data.

22.2.7 Cache-Friendly Algorithms: Sort, Group, and Stream

Algorithms that group similar work improve hardware efficiency:

- sort entities by type to avoid branch-heavy polymorphic dispatch,
- group by spatial locality to improve cache reuse,
- stream through arrays rather than jumping between structures.

This often changes architecture:

- from “each object updates itself”,
- to “a system processes a batch of components” (systems-based architecture).

22.2.8 I-Cache and Code Layout: Keep Hot Code Small

Modern CPUs can also be limited by instruction cache misses:

- large template instantiations,
- excessive inlining across large call graphs,

- big virtual dispatch tables and cold code mixed into hot paths.

Hardware-aware design includes:

- isolating hot loops into small translation units,
- controlling inlining (not maximizing it blindly),
- keeping debug/logging out of hot code,
- separating cold features into separate code paths or modules.

22.2.9 Measurement: Hardware-Aware Design Must Be Evidence-Based

Hardware is complex; performance intuition is often wrong. Therefore, DOD requires measurement at three levels:

1) Correctness and Invariants First

Never sacrifice correctness for micro-optimizations. Validate invariants at boundaries and in debug builds.

2) Profiling Real Workloads

Identify:

- hot functions and hot loops,
- cache miss patterns,
- allocation hot spots,
- contention points.

3) Microbenchmarks for Targeted Hypotheses

Use microbenchmarks to validate specific claims:

- AoS vs SoA layout effect,
- batching size impact,
- alignment and padding changes,
- branch reduction strategies.

A Minimal Benchmark Harness (Stable and Practical)

```
#include <chrono>
#include <iostream>

template<class F>
std::chrono::nanoseconds time_it(F&& f, int iters) {
  using clock = std::chrono::steady_clock;
  auto start = clock::now();
  for (int i = 0; i < iters; ++i) f();
  auto end = clock::now();
  return std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
}

int main() {
  auto ns = time_it([]{
    volatile int x = 0;
    for (int i = 0; i < 1000; ++i) x += i;
  }, 100000);

  std::cout << "ns: " << ns.count() << "\n";
}
```

Design note. This harness is intentionally simple. In real benchmarking, control for:

- compiler optimizations,
- warm-up effects,
- CPU frequency scaling,
- measurement noise.

22.2.10 A Practical Hardware-Driven Design Workflow

1. Identify the dominant workload and hot paths.
2. Make access patterns explicit: what data is touched per operation.
3. Choose data layouts: contiguous storage, SoA for hot fields, hot/cold splitting.
4. Reduce indirection: replace pointer graphs with indices/handles in hot paths.
5. Reduce branches: group similar cases, batch processing, avoid polymorphism in hot loops.
6. Scale across cores: shard data, reduce synchronization frequency, avoid false sharing.
7. Validate with profiling and benchmarks; keep changes only if they measurably improve.

22.2.11 Engineering Checklist

1. Treat cache and memory bandwidth as architectural constraints.
2. Lay out data by access patterns: contiguous, hot/cold split, SoA where beneficial.
3. Avoid pointer chasing in hot paths; prefer dense storage with handles/indices.
4. Reduce unpredictable control flow; batch and group similar work.

5. Design for multicore: shard state, avoid false sharing, minimize synchronization.
6. Keep hot code small; isolate cold features away from hot loops.
7. Measure continuously with profiling and targeted benchmarks.

22.2.12 Summary

Designing for hardware in Modern C++ means shaping architecture around the realities of execution:

- data movement dominates compute,
- locality and bandwidth define throughput,
- branches and indirection define predictability,
- vectorization and multicore scaling require regular access patterns,
- evidence (profiling) must guide changes.

A data-oriented architecture is therefore not a style preference. It is the disciplined practice of aligning data layout and processing pipelines with the CPU and memory hierarchy, so long-lived systems remain fast, predictable, and scalable as workloads grow.

22.3 ECS and Modern C++

22.3.1 Why ECS Exists: Data-Oriented Architecture at Scale

Entity–Component–System (ECS) is an architectural response to a recurring performance reality:

- games, simulations, real-time analytics, and robotics often process **large numbers of similar objects**,

- object-oriented graphs and polymorphic hierarchies often degrade due to **cache misses**, **pointer chasing**, and **branch unpredictability**,
- the dominant cost is frequently **data movement**, not arithmetic.

ECS reorganizes the program around:

- **entities:** stable identities (IDs/handles),
- **components:** pure data attached to entities (often stored in dense arrays),
- **systems:** functions that operate on batches of components (tight loops).

The key idea is architectural:

> Do not design around objects that *contain* behavior. Design around data that is processed by pipelines.

22.3.2 The ECS Mental Model (In Hardware Terms)

ECS improves hardware efficiency by:

- storing components contiguously (spatial locality),
- processing in tight loops (instruction locality),
- grouping similar work (branch predictability),
- enabling vectorization (regular access patterns),
- enabling parallel execution across disjoint component sets.

This is not only a game-development pattern. It is a general technique for **large-scale state + repeated bulk computation**.

22.3.3 ECS as an Architectural Decomposition

ECS decomposes responsibilities differently than classical OOP:

Entities

Entities are **identity only**. An entity is typically:

- an integer ID,
- optionally with a generation counter to detect stale references.

Components

Components are **data only**. A component type:

- stores the minimal fields required by systems,
- avoids owning heavy resources by default in hot components,
- is designed for compactness and locality.

Systems

Systems are **behavior and policies**. A system:

- queries a set of entities with specific component combinations,
- runs bulk processing over contiguous arrays,
- is the unit of scheduling (single-threaded, parallel, vectorized).

22.3.4 Modern C++ Enables Practical ECS Designs

Modern C++ (C++20/23) is particularly suitable for ECS because it provides:

- strong typing and zero-cost abstractions,
- templates and compile-time composition for component queries,
- `constexpr` utilities to precompute metadata,
- ranges and views for expressive iteration (when not harming hot loops),
- `std::span` for safe non-owning views into dense storage,
- improved move semantics and RAII for resource components,
- atomics and concurrency primitives for safe scheduling and parallel systems.

The crucial design rule remains:

> Use abstraction to reduce complexity, but do not hide data layout and hot-loop behavior.

22.3.5 A Minimal ECS Core (Educational, Not a Full Engine)

The following example illustrates the essential ECS mechanics:

- entity IDs with generation,
- a dense component store with a sparse index,
- a system iterating over the dense storage efficiently.

Entity Identity: ID + Generation

```
#include <cstdint>

struct Entity {
  std::uint32_t index{};
  std::uint32_t generation{};
};

constexpr bool operator==(Entity a, Entity b) noexcept {
  return a.index == b.index && a.generation == b.generation;
}
```

A Dense Component Store with Sparse Mapping

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

template<class T>
class DenseStore {
public:
  bool has(Entity e) const noexcept {
    return e.index < sparse_.size() && sparse_[e.index] != npos_;
  }

  T* get(Entity e) noexcept {
    if (!has(e)) return nullptr;
    return &dense_[sparse_[e.index]];
  }

  const T* get(Entity e) const noexcept {
    if (!has(e)) return nullptr;
    return &dense_[sparse_[e.index]];
```

```
}

// Insert or replace.
T& emplace(Entity e, T value) {
  ensure_sparse(e.index);
  if (has(e)) {
    dense_[sparse_[e.index]] = std::move(value);
    return dense_[sparse_[e.index]];
  }
  const std::size_t pos = dense_.size();
  dense_.push_back(std::move(value));
  entities_.push_back(e);
  sparse_[e.index] = pos;
  return dense_.back();
}

// Remove using swap-with-back to keep dense packing.
void erase(Entity e) noexcept {
  if (!has(e)) return;

  const std::size_t pos = sparse_[e.index];
  const std::size_t last = dense_.size() - 1;

  if (pos != last) {
    dense_[pos] = std::move(dense_[last]);
    entities_[pos] = entities_[last];
    sparse_[entities_[pos].index] = pos;
  }

  dense_.pop_back();
  entities_.pop_back();
  sparse_[e.index] = npos_;
}
```

```
  std::span<T> dense_view() noexcept { return dense_; }
  std::span<const T> dense_view() const noexcept { return dense_; }

  std::span<const Entity> entities_view() const noexcept { return entities_; }

private:
  static constexpr std::size_t npos_ = static_cast<std::size_t>(-1);

  void ensure_sparse(std::uint32_t idx) {
    if (idx >= sparse_.size()) sparse_.resize(static_cast<std::size_t>(idx) + 1, npos_);
  }

  std::vector<T> dense_;
  std::vector<Entity> entities_;          // parallel with dense_
  std::vector<std::size_t> sparse_;       // entity.index -> dense position
};
```

Data-oriented property.

- Dense arrays are contiguous (cache friendly).
- Sparse array enables O(1) lookup by entity ID.
- Erase preserves density via swap-with-back.

Component Types

```
struct Position { float x{}, y{}, z{}; };
struct Velocity { float x{}, y{}, z{}; };
```

A System: Integrate Positions from Velocities

A system ideally iterates over the smallest hot set and touches minimal bytes.

```
#include <algorithm>

void integrate(DenseStore<Position>& pos, DenseStore<Velocity>& vel, float dt) {
  // Simple strategy: iterate over the smaller set.
  auto& a = (pos.dense_view().size() <= vel.dense_view().size()) ? pos : vel;

  // If iterating positions: for each entity with Position, check Velocity.
  if (&a == &pos) {
    auto entities = pos.entities_view();
    auto pview = pos.dense_view();
    for (std::size_t i = 0; i < pview.size(); ++i) {
      Entity e = entities[i];
      if (auto* v = vel.get(e)) {
        pview[i].x += v->x * dt;
        pview[i].y += v->y * dt;
        pview[i].z += v->z * dt;
      }
    }
  } else {
    // Iterating velocities: update position if present.
    auto entities = vel.entities_view();
    auto vview = vel.dense_view();
    for (std::size_t i = 0; i < vview.size(); ++i) {
      Entity e = entities[i];
      if (auto* p = pos.get(e)) {
        p->x += vview[i].x * dt;
        p->y += vview[i].y * dt;
        p->z += vview[i].z * dt;
      }
    }
  }
}
```

Why this looks different from OOP. There is no `Entity::update()` method. Behavior is not distributed across objects; it is centralized in systems operating on dense data.

22.3.6 Archetypes vs Sparse Sets: Two Major ECS Storage Families

Modern ECS implementations generally fall into two storage families.

Sparse Set Storage

- Each component type has its own dense store (like the example above).
- Queries combine stores at runtime (intersections, lookups).
- Very flexible for adding/removing components.

Trade-offs:

- querying multiple components may require repeated lookups,
- best when systems touch a small number of component types or when sets are sparse.

Archetype (Chunk) Storage

- Entities are grouped by their *component composition* (archetype).
- Each archetype stores components in contiguous "chunks" for iteration.
- Systems iterate chunk-by-chunk over entities that match required component sets.

Trade-offs:

- extremely fast iteration for common queries,
- moving entities between archetypes when components are added/removed.

Architectural observation:

> Archetypes optimize iteration; sparse sets optimize mutation flexibility.

22.3.7 ECS and Strong Typing: Safer Queries with Modern C++

In large ECS codebases, the biggest maintenance risk is incorrect assumptions about which components exist during a system run. Modern C++ helps by expressing requirements explicitly.

Compile-Time "System Signature"

```
template<class... Components>
struct SystemSignature { };

using IntegrateSig = SystemSignature<Position, Velocity>;
```

In a real ECS, signatures can map to:

- compile-time IDs,
- cached query plans,
- prevalidated archetype matches.

22.3.8 Scheduling and Parallelism: Systems as Units of Work

ECS typically encourages:

- clear read/write sets (which components a system reads or writes),
- scheduling based on data hazards,
- parallel execution of non-conflicting systems.

A modern C++ architecture can encode this:

- read-only component views passed as `std::span<const T>`,
- write views as `std::span<T>`,
- explicit system declarations of access mode.

Read/Write Views Sketch

```
#include <span>

template<class T>
using ReadView = std::span<const T>;

template<class T>
using WriteView = std::span<T>;
```

This turns a hidden concurrency risk into a declared API contract.

22.3.9 ECS and Resource Management in C++

Components are often plain data, but real systems also need resources:

- textures, meshes, file handles, sockets,
- OS handles and GPU resources,
- large buffers and caches.

A practical approach:

- keep hot components small and POD-like,
- store heavy resources indirectly (IDs/handles),
- manage resources in dedicated subsystems with RAII,
- keep ownership centralized; avoid shared ownership in hot paths.

Resource Handle Component

```
#include <cstdint>

struct TextureId { std::uint32_t value{}; };
struct Sprite { TextureId tex; float u0{}, v0{}, u1{}, v1{}; };
```

Design effect. The ECS remains data-oriented; resources are handled by a separate manager.

22.3.10 Where ECS Fits in Long-Lived Systems

ECS shines when:

- the system processes many similar entities frequently,
- performance is dominated by hot loops,
- you need scalable parallel updates,
- you want a uniform way to add features through new components and systems.

ECS may be a poor fit when:

- object counts are small and dominated by irregular logic,
- most complexity is in business rules rather than bulk computation,
- the team needs rapid evolution without adopting a new mental model,
- the overhead of an ECS framework outweighs benefits.

A practical long-lived architecture often uses a hybrid:

- ECS for simulation and high-throughput state updates,
- classical OOD for orchestration, UI, integration boundaries, and domain policy,
- translation layers between DOD core and OOD edges.

22.3.11 Engineering Checklist

1. Keep entities as identity only (ID + generation); never embed behavior.
2. Keep hot components small and contiguous; split hot/cold data.
3. Prefer dense storage and batch iteration; avoid pointer chasing in hot systems.
4. Choose a storage model (sparse sets vs archetypes) based on mutation vs iteration needs.
5. Declare system read/write access to enable safe scheduling and parallelism.
6. Use handles/IDs for heavy resources; manage resources separately with RAII.
7. Measure: ECS is justified by profiling and scaling behavior, not ideology.

22.3.12 Summary

ECS is a data-oriented architecture that aligns naturally with Modern C++:

- strong typing and zero-cost abstractions help express component and system contracts,
- contiguous storage and batching exploit cache locality and SIMD opportunities,
- explicit access patterns enable parallel scheduling,
- RAII and resource managers integrate safely with handle-based components.

When applied to the right domains, ECS turns performance constraints into a maintainable structure: data stays compact and movable, and behavior becomes scalable pipelines over that data.

22.4 When Data Beats Objects

22.4.1 The Practical Claim

“Data beats objects” does not mean object-oriented design is wrong. It means that for certain workloads, the dominant constraints are:

- cache locality,
- memory bandwidth,
- branch predictability,
- vectorization and batching,
- multicore scaling.

In those workloads, organizing software around **data layout and bulk processing** outperforms organizing software around **individually allocated objects with distributed behavior**.
The correct engineering question is:

> Where is the time spent: in computation, or in moving and touching memory?

If the answer is “moving memory”, data-oriented techniques usually win.

22.4.2 A Decision Framework: Identify the DOD Sweet Spot

Data-oriented design is most effective when these conditions hold:

1) High Element Counts

You process thousands to millions of elements per frame/iteration:

- particles,

- physics bodies,
- network packets,
- market data ticks,
- telemetry events,
- pixels, samples, and signals,
- graph edges in large traversals.

2) Repeated, Similar Operations

The same operations are applied repeatedly:

- integrate, filter, transform, classify,
- update positions, apply constraints, compute aggregates,
- validate, compress, hash, encode/decode.

3) The Hot Path Touches a Small Subset of Fields

You only need some fields frequently. Objects often bundle hot and cold fields together, wasting bandwidth.

4) Access Patterns Are Regular Enough to Exploit Locality

Contiguous scanning, batched processing, and grouped work are possible.

5) The Bottleneck Is Memory, Not Arithmetic

Profiling shows:

- high cache miss rates,
- high branch misprediction,
- stalls on memory loads,
- limited scaling due to bandwidth or contention.

When these conditions are present, data layout becomes architecture.

22.4.3 Why Objects Lose in These Cases

Objects often lose in hot, large-scale workloads for mechanical reasons:

1) Pointer Chasing

Heap-allocated object graphs force the CPU to load scattered memory. Caches and prefetchers cannot help much when addresses are unpredictable.

2) Poor Spatial Locality

A `std::vector<std::unique_ptr<T»` is contiguous only in pointers. The actual objects are scattered.

3) Mixed Hot/Cold Fields

Objects often include rarely used metadata alongside hot fields. This increases bytes moved per useful operation.

4) Virtual Dispatch and Branch Unpredictability

Polymorphism can introduce unpredictable branches in tight loops.

5) Fragmentation and Allocation Overhead

Frequent allocation and deallocation:

- consumes CPU time,
- increases fragmentation,
- destroys locality.

22.4.4 A Concrete Demonstration: Same Problem, Two Shapes

Object-Centric Design

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

struct Particle {
  virtual ~Particle() = default;
  virtual void integrate(float dt) noexcept = 0;
};

struct SimpleParticle final : Particle {
  float x, y, z;
  float vx, vy, vz;

  void integrate(float dt) noexcept override {
    x += vx * dt; y += vy * dt; z += vz * dt;
  }
```

```
};

void step_objects(std::vector<std::unique_ptr<Particle>>& ps, float dt) {
  for (auto& p : ps) {
    p->integrate(dt); // virtual dispatch + pointer chasing
  }
}
```

This design is expressive and extensible. But it is often poor in a hot loop over large populations.

Data-Centric Design (SoA + Tight Loop)

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

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

void step_data(Particles& p, float dt) {
  const std::size_t n = p.x.size();
  for (std::size_t i = 0; i < n; ++i) {
    p.x[i] += p.vx[i] * dt;
    p.y[i] += p.vy[i] * dt;
    p.z[i] += p.vz[i] * dt;
  }
}
```

This design:

- reads contiguous arrays,
- has predictable control flow,

- is friendly to SIMD and prefetching,
- minimizes per-element overhead.

22.4.5 Canonical Situations Where Data Beats Objects

1) Stream Processing and Pipelines

When you process a stream of records:

- decode → validate → transform → aggregate,
- parse → normalize → index,
- filter → map → reduce.

The pipeline wants:

- contiguous buffers,
- batch sizes tuned for cache,
- predictable loops.

Objects introduce overhead without adding value.

2) SIMD-Friendly Numeric Work

Vectorization thrives on:

- SoA layouts,
- minimal aliasing,
- straight-line loops,
- avoidance of virtual dispatch.

Objects tend to break these properties.

3) Indexing, Searching, and Analytics Over Large Data

Large-scale operations (search, indexing, scoring, analytics) usually depend on:

- minimizing bytes touched,
- scanning contiguous segments,
- using cache-friendly structures (dense arrays, bitsets, compact tables).

4) Physics, Simulation, and ECS-Style Updates

These are archetypal DOD domains:

- many entities,
- repeated steps,
- frequent iteration over subsets,
- heavy emphasis on locality and batching.

5) Networking and Packet Processing

Packets are naturally:

- homogeneous,
- processed in batches,
- better handled as buffers and views rather than objects.

6) Real-Time Systems With Tight Latency Budgets

In real-time and low-latency systems:

- predictable timing matters,
- allocations and polymorphic dispatch can create jitter,
- data-oriented loops reduce tail latency.

22.4.6 A More Subtle Point: DOD Can Improve Maintainability Too

DOD is often mischaracterized as "only for speed". In the right domain, DOD can also improve maintainability by making:

- data flow explicit,
- processing stages explicit,
- invariants and preconditions explicit at stage boundaries,
- performance behavior more predictable.

This is particularly valuable in long-lived systems where performance regressions are frequent.

22.4.7 The Hybrid Architecture Rule

In long-lived C++ systems, the best structure is often hybrid:

OOD at the Edges

- domain policy, orchestration, API boundaries,
- resource ownership and lifecycle (RAII),
- integration with external systems.

DOD in the Core

- bulk processing engines,
- high-throughput pipelines,
- simulation kernels,
- analytics and scoring loops.

The architectural principle:

> Encapsulate DOD as an engine behind stable interfaces, and keep OOD as the coordination language around it.

22.4.8 Practical Refactoring Steps: From Objects to Data (Safely)

A safe transition rarely means "rewrite everything". Instead, evolve subsystem-by-subsystem:

1. **Profile first:** identify the actual hot loops and memory stalls.
2. **Flatten object graphs:** move from scattered objects to contiguous arrays of IDs and data.
3. **Split hot/cold:** separate rarely used fields into cold storage.
4. **Batch operations:** replace per-object calls with system-level loops.
5. **Replace polymorphism in hot paths:** group by type or use data tags.
6. **Introduce handles:** replace raw pointers with indices/generational handles when relocation matters.
7. **Measure after each change:** keep only changes with measurable benefit.

22.4.9 Engineering Checklist

1. Apply DOD where element counts are large and work is repeated.
2. Optimize for locality: contiguous storage, SoA where beneficial, hot/cold splitting.
3. Reduce indirection: avoid pointer-chasing object graphs in hot loops.
4. Reduce unpredictability: minimize branches and virtual calls in hot paths.
5. Batch to amortize overhead and improve prefetch and instruction locality.
6. Prefer handles/indices over pointers in performance cores.
7. Keep OOD for policy and boundaries; keep DOD for engines and kernels.
8. Validate with profiling and targeted benchmarks.

22.4.10 Summary

Data beats objects when hardware realities dominate:

- memory access patterns and bytes moved per operation define throughput,
- scattered objects amplify cache misses and branch unpredictability,
- contiguous data and batch processing enable SIMD and multicore scaling,
- hot/cold separation prevents wasted bandwidth,
- explicit data pipelines reduce performance cliffs and improve predictability.

The goal is not to abandon objects. The goal is to place objects where they help (policy, invariants, boundaries) and place data-oriented engines where they win (hot loops, large populations, streaming workloads).

Part VIII

Large-Scale System Design

Designing for Scale and Longevity

23.1 Code That Lives 20+ Years

Long-lived systems are not written once; they are *operated* and *evolved* for decades. The defining skill is not cleverness, but the ability to keep change safe, cheap, and observable while the system grows in users, data, teams, platforms, and external dependencies.
This section presents concrete engineering practices that keep C++ codebases healthy for 20+ years: stable boundaries, disciplined evolution, operational feedback loops, and a culture that treats compatibility as a feature.

23.1.1 Longevity as an Explicit Design Goal

A system that must live for decades needs a different set of defaults:

- **Change is constant:** the code is rewritten continuously in small pieces; architecture must support incremental change.
- **Compatibility is a product feature:** old clients, data, and integrations outlive teams and vendors.
- **Operations matter as much as implementation:** failures will happen; you need fast diagnosis, graceful degradation, and safe recovery.

- **Constraints accumulate:** performance budgets, security policies, compliance, and legacy protocols increase over time.

The most durable architectures treat *interfaces, data formats, and operational behavior* as the stable core, while implementations remain replaceable.

23.1.2 The Three Stability Layers

Long-lived C++ systems succeed when they stabilize the right things:

Stable External Contracts

- API behavior, error semantics, and performance envelopes
- Data formats and schema evolution rules
- Security and authentication expectations

Stable Internal Boundaries

- Clear module ownership and dependency direction
- Replaceable components with explicit seams
- Build-time isolation (headers, macros, compile options)

Stable Operational Guarantees

- Observability (metrics, logs, traces) and SLO thinking
- Backward-compatible rollouts and safe rollbacks
- Disaster recovery plans, migrations, and incident processes

In practice: stabilize *interfaces and invariants*, not the internal implementation.

23.1.3 Designing Boundaries That Survive Decades

Prefer Narrow, Explicit Interfaces

A boundary should expose only what callers truly need. Keep interfaces small, stable, and intention-revealing.

- Avoid “kitchen sink” classes and wide headers.
- Separate *commands* (do something) from *queries* (observe state).
- Make invariants explicit: valid ranges, ordering rules, thread-safety guarantees.

Avoid Leaking Representation

If representation leaks (containers, allocation strategy, layout, third-party types), you freeze implementation details into the API.
A practical longevity pattern in C++ is to publish *value-oriented* handles and keep heavy representation behind a stable façade (PImpl, type erasure, or pure C ABI at boundaries).

PImpl for ABI and Compile-Time Isolation (When Needed)

```
#pragma once
#include <memory>
#include <string>
#include <vector>

namespace app {

class SearchIndex final {
public:
  SearchIndex();
  ~SearchIndex();
```

```
  SearchIndex(SearchIndex&&) noexcept;
  SearchIndex& operator=(SearchIndex&&) noexcept;

  SearchIndex(const SearchIndex&) = delete;
  SearchIndex& operator=(const SearchIndex&) = delete;

  void add_document(std::string id, std::string text);
  std::vector<std::string> query(std::string_view q, std::size_t top_k) const;

private:
  struct Impl;                 // representation is private and can evolve
  std::unique_ptr<Impl> impl_; // stable ABI surface (in many build setups)
};

} // namespace app
```

Use this when you must reduce rebuild cascades, hide third-party dependencies, and control ABI surface. Do not use it blindly: it adds indirection, allocation, and complexity.

Dependency Direction: Keep the Core Clean

A durable design has a stable core with minimal dependencies, surrounded by adapters and integrations.

- Core domain types should not depend on databases, networking, UI, or frameworks.
- Integrations depend on core, not the reverse.
- Enforce with build rules (targets), include policies, and CI checks.

23.1.4 API Evolution Without Breaking the World

Longevity is mostly about *evolution strategy*. The key is to make change compatible by default.

Semantic Versioning for Behavior, Not Just Headers

Versioning only matters if it reflects:

- Behavioral changes
- Error semantics changes
- Performance regressions beyond a stated budget

Deprecation as a First-Class Process

A long-lived system needs an explicit lifecycle:

- Mark deprecated
- Provide a migration path
- Measure usage
- Remove only after a defined window and communication process

Additive Change Over Mutating Change

Prefer:

- Adding new functions/types over changing existing meaning
- Introducing new fields in data formats (with defaults) over reinterpreting old ones
- Extending behavior with feature flags over changing defaults abruptly

A Practical "New API, Old API" Wrapper

```
enum class Mode { legacy, modern };

struct Config {
  Mode mode = Mode::legacy;
  int timeout_ms = 2000;
};

class Service {
public:
  // Old stable entry point:
  void run_legacy();

  // New entry point:
  void run(const Config& cfg) {
    if (cfg.mode == Mode::legacy) {
      run_legacy();
      return;
    }
    run_modern(cfg);
  }

private:
  void run_modern(const Config& cfg);
};
```

This pattern allows gradual adoption while preserving legacy behavior.

Versioned Namespaces for Coexisting APIs

When you truly must ship parallel APIs for years, use explicit versioning.

```
namespace api::v1 {
```

```
  struct Request { /* ... */ };
  struct Response { /* ... */ };
  Response handle(const Request&);
}

namespace api::v2 {
  struct Request { /* revised ... */ };
  struct Response { /* revised ... */ };
  Response handle(const Request&);
}
```

Avoid ambiguous overload sets across versions; keep call sites explicit to reduce accidental mixing.

23.1.5 Data Longevity: Formats, Schema, and Migrations

Most "20-year bugs" are data bugs. Code can be rewritten; data often cannot.

Design Data with Evolution Rules

- Every stored object has a version.
- New fields are optional with defaults.
- Unknown fields are preserved (round-tripping) when acting as a proxy.
- Deletions are staged: stop writing, then stop reading, then remove.

A Minimal Versioned Serialization Pattern

```
#include <cstdint>
#include <string>
#include <vector>
```

```
struct UserV1 {
  std::string name;
};

struct UserV2 {
  std::string name;
  std::string email; // added later, optional in older data
};

struct Blob {
  std::uint32_t version = 0;
  std::vector<std::byte> bytes;
};

Blob serialize(const UserV2& u);
UserV2 deserialize_user(const Blob& b);

UserV2 deserialize_user(const Blob& b) {
  switch (b.version) {
    case 1: {
      UserV1 old = /* parse v1 */;
      UserV2 now;
      now.name = std::move(old.name);
      now.email = ""; // default
      return now;
    }
    case 2: {
      UserV2 now = /* parse v2 */;
      return now;
    }
    default:
      throw std::runtime_error("Unsupported user blob version");
```

```
  }
}
```

The central idea is not the parsing technology; it is the policy: explicit versions, strict handling rules, and predictable defaults.

Migrations Must Be Observable and Reversible

- Measure migration progress and error rates.
- Keep old readers long enough to roll back.
- Prefer expand/contract migrations:
  - *Expand:* add new structures while keeping old.
  - *Backfill:* write both or transform lazily.
  - *Contract:* remove old only after confidence.

23.1.6 Performance Longevity: Budgets, Not Optimizations

Over decades, performance expectations rise and hardware changes. Sustainable performance comes from budgets and measurement.

Define Performance Contracts

- Latency targets per operation class (P50/P95/P99)
- Throughput and memory ceilings
- Startup time and steady-state CPU budgets

Guardrails in CI

- Microbenchmarks for critical primitives (stable harness)
- Regression detection with tolerance bands
- Profiling playbooks for incidents and releases

A Simple "Budget Check" Pattern

```
#include <chrono>
#include <stdexcept>

template <class F>
auto run_with_budget(F&& f, std::chrono::milliseconds budget) {
  const auto start = std::chrono::steady_clock::now();
  auto out = f();
  const auto end = std::chrono::steady_clock::now();
  if (end - start > budget) {
    throw std::runtime_error("Performance budget exceeded");
  }
  return out;
}
```

Use this conceptually in tests/benchmarks (not necessarily in production hot paths).

23.1.7 Correctness Longevity: Make Failure Predictable

Undefined behavior, data races, and inconsistent error handling destroy long-term maintainability. Durable code makes failure modes explicit.

Make Invalid States Unrepresentable (Where Practical)

- Strong types for IDs, sizes, and units

- Validated construction
- Explicit “optional” and “expected” outcomes

Value Types for Safety at Scale

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

class UserId {
public:
  static UserId parse(std::string_view s) {
    if (s.empty()) throw std::invalid_argument("empty id");
    return UserId(std::string{s});
  }
  const std::string& str() const noexcept { return v_; }

private:
  explicit UserId(std::string v) : v_(std::move(v)) {}
  std::string v_;
};
```

Standardize Error Semantics Across the System

Pick a model and apply it consistently:

- Exceptions for exceptional failures (with clear boundaries)
- Error codes / `expected`-style returns for routine failures
- Never mix ad-hoc conventions across modules without adapters

A Small Expected-Style Result Type (Illustrative)

```
#include <variant>
#include <string>

struct Error {
  int code = 0;
  std::string message;
};

template <class T>
class Result {
public:
  static Result ok(T v) { return Result(std::move(v)); }
  static Result fail(Error e) { return Result(std::move(e)); }

  bool has_value() const noexcept { return std::holds_alternative<T>(v_); }
  const T& value() const { return std::get<T>(v_); }
  const Error& error() const { return std::get<Error>(v_); }

private:
  explicit Result(T v) : v_(std::move(v)) {}
  explicit Result(Error e) : v_(std::move(e)) {}
  std::variant<T, Error> v_;
};
```

The durability comes from uniformity: you can build tooling, logging, and policies around one approach.

23.1.8 Concurrency Longevity: Rules, Not Hope

As systems scale, concurrency becomes the default. Long-lived systems survive by establishing and enforcing thread-safety rules.

Define Thread-Safety at the Type Level

- Is this type thread-safe for concurrent reads?
- Does it require external synchronization?
- What are the lifetime and ownership rules?

Prefer Message Passing for Scaling Boundaries

Shared mutable state becomes a long-term liability. For cross-module scaling:

- Use queues, commands, immutable snapshots
- Make data ownership explicit at boundaries

Command Queue Skeleton

```
#include <mutex>
#include <queue>
#include <functional>
#include <condition_variable>

class CommandQueue {
public:
  void push(std::function<void()> cmd) {
    {
      std::lock_guard<std::mutex> lk(m_);
      q_.push(std::move(cmd));
    }
    cv_.notify_one();
  }

  std::function<void()> pop() {
```

```
    std::unique_lock<std::mutex> lk(m_);
    cv_.wait(lk, [&]{ return !q_.empty(); });
    auto cmd = std::move(q_.front());
    q_.pop();
    return cmd;
  }

private:
    std::mutex m_;
    std::condition_variable cv_;
    std::queue<std::function<void()>> q_;
};
```

This pattern keeps mutation localized and reviewable.

23.1.9 Operational Longevity: Observability and Diagnose-Ability

The best long-lived systems are *debuggable in production*. This requires consistent signals, not heroic engineers.

Design for Observability

- **Structured logs** with stable event names and fields
- **Metrics** for rates, errors, latency, saturation, and queue depth
- **Tracing** for request flow across components

Make Diagnostics Part of Interfaces

A longevity trick: every important subsystem exposes a status snapshot (health, counters, config).

```
#include <cstdint>
#include <string>
```

```
struct Health {
  bool ok = true;
  std::string summary;
  std::uint64_t requests = 0;
  std::uint64_t failures = 0;
};

class Component {
public:
  virtual ~Component() = default;
  virtual Health health() const = 0;
};
```

When incidents happen years later, this interface pays for itself.

23.1.10 Build and Tooling Longevity

C++ longevity is tightly coupled to build hygiene. Code that lives decades must keep builds deterministic and upgrades manageable.

Deterministic Builds and Reproducibility

- Lock toolchains per release line (compiler, standard library, linker)
- Record build flags and configuration in artifacts
- Avoid environment-dependent behavior in builds and tests

Header Discipline and Compilation Firebreaks

- Keep public headers minimal and stable
- Avoid including heavy dependencies in widely-shared headers

- Use forward declarations and opaque pointers when it improves isolation

Upgrade Strategy as Architecture

- Support multiple compiler versions for a window (as policy)
- Use feature detection and encapsulated portability layers
- Centralize warnings, sanitizers, and static analysis in shared build logic

23.1.11 Security Longevity: Assume the Threat Model Evolves

Over 20+ years, vulnerabilities change, cryptography ages, and policies tighten.

- Treat parsing and deserialization as hostile input zones.
- Reduce attack surface: minimal privileges, minimal exposed endpoints.
- Apply defense-in-depth: validation, bounds checks, fuzzing, sandboxing.
- Maintain a patch pipeline: quick triage, backports, and coordinated releases.

In C++, longevity security also means:

- Eliminating UB patterns and data races
- Using safer abstractions for ownership and lifetimes
- Hardening builds (where applicable): sanitizers in CI, hardened flags in release

23.1.12 Human Longevity: Teams Change, Code Must Stay Legible

Over decades, most engineers who wrote the system will leave. Your system must teach newcomers how to change it safely.

Make Architecture Discoverable

- A living architecture overview (what depends on what, why)
- Module READMEs that explain invariants and failure modes
- Examples of "correct usage" in tests and sample code

Prefer Boring Code Over Clever Code

Boring code:

- is easier to review and refactor
- has fewer "hidden contracts"
- invites consistent patterns across teams

Institutionalize Quality

Longevity requires that correctness does not depend on individual heroics:

- mandatory code review with clear checklists
- consistent style and guidelines
- testing pyramid: unit, integration, end-to-end, plus fuzzing where input parsing exists
- incident postmortems that feed back into design rules

23.1.13 A Practical Longevity Checklist for Large C++ Systems

Architecture and Boundaries

- Stable contracts at boundaries; minimize leakage of representation

- Clean dependency direction; enforce with build system rules
- Replaceable components via seams (adapters, facades)

Evolution

- Additive changes preferred; deprecation policy defined and measurable
- Versioning strategy for APIs and data formats
- Backward-compatible rollout and rollback paths

Correctness and Safety

- Consistent error model (exceptions vs result types) with adapters at boundaries
- Thread-safety rules documented per type/module
- UB and race reduction via policies, reviews, and tooling in CI

Operations

- Observability is built-in: logs/metrics/traces with stable naming
- Health/status surfaces exist for critical subsystems
- Incident response and postmortem loop feeds back into design

Build and Tooling

- Reproducible builds; toolchain strategy for upgrades and backports
- Header discipline; compilation firebreaks where beneficial
- Automated quality gates: tests, analysis, and performance regression checks

23.1.14 Closing Principle: Build for Change, Not for Today

Code that lives 20+ years is built on the assumption that:

- requirements will change,
- teams will change,
- platforms will change,
- the system will fail in unexpected ways,

and therefore the architecture must make change *safe, observable, and incremental*.
Durability in Modern C++ comes from disciplined boundaries, explicit contracts, stable data evolution rules, consistent error and concurrency models, and an operational culture that treats maintainability and compatibility as first-class engineering outputs.

23.2 Managing Technical Debt

Technical debt is the accumulation of design and implementation compromises that make future change slower, riskier, or more expensive. In long-lived C++ systems, debt is not a moral failure; it is an economic reality. The engineering objective is to *make debt explicit, measurable, and governable* so that the system can scale in features, teams, and lifetime without collapsing under its own complexity.
This section provides a practical playbook for managing technical debt in Modern C++ at large scale: how to classify and measure debt, how to create feedback loops that keep it under control, how to prioritize repayment, and which concrete refactoring strategies produce the best long-term return.

23.2.1 What Technical Debt Really Means in Large C++ Systems

Debt vs. Complexity vs. Legacy

- **Essential complexity:** inherent to the problem domain (cannot be removed without changing requirements).
- **Accidental complexity:** created by design choices, layering violations, and tooling/build friction (can be reduced).
- **Legacy:** old but stable and valuable behavior (not necessarily debt).
- **Debt:** any condition that increases the *cost of change* beyond what is reasonable for the business goals.

A 20-year-old component can be low-debt if its boundaries are stable, tests are strong, observability is present, and change remains predictable. Conversely, brand-new code can be high-debt if it hides invariants, couples layers, and produces brittle builds.

The Interest Model

Debt becomes dangerous when it charges interest:

- every feature takes longer than expected,
- outages become harder to diagnose,
- refactors require risky “big bang” rewrites,
- onboarding new engineers becomes slow and error-prone.

Your job is not to eliminate debt. Your job is to prevent *runaway interest*.

23.2.2 A Practical Taxonomy of Technical Debt

Managing debt starts by classifying it with enough precision to act.

Design and Architecture Debt

- unclear boundaries and ownership
- dependency direction violations (UI depends on DB, core depends on framework)
- god objects, service locators, hidden global state
- cross-cutting concerns (logging, auth, caching) implemented ad-hoc everywhere

Code Debt

- duplicated logic, copy-paste variants
- unclear invariants and preconditions
- inconsistent error handling models
- manual memory/lifetime hazards and implicit ownership rules

Concurrency and Performance Debt

- shared mutable state without clear synchronization rules
- data races avoided only by convention
- unbounded queues, unbounded caches, unmeasured memory growth
- performance-sensitive code with no benchmarks or budgets

Test Debt

- missing tests for critical business behavior
- tests that are slow, flaky, or too coupled to internals
- lack of integration tests around boundaries (APIs, storage, IPC)
- no regression harness for previously fixed incidents

Build and Tooling Debt

- fragile builds, slow builds, and non-reproducible environments
- excessive header coupling and rebuild cascades
- inconsistent compiler flags, warnings, sanitizers, and CI gates
- hidden platform-specific hacks and unowned scripts

Operational Debt

- poor observability (logs without structure, missing metrics)
- unknown failure modes and no runbooks
- unsafe deployment practices, no rollback plan
- lack of incident learning loop

23.2.3 Debt That Is *Acceptable* vs. Debt That Is *Fatal*

A mature organization allows some debt, but draws hard lines around debt that threatens longevity.

Acceptable Debt (Managed)

- a temporary workaround with a written expiration date,
- a tactical shortcut for validated business urgency,
- debt isolated behind a stable boundary with tests and monitoring,
- consciously chosen performance trade-offs with budgets and measurement.

Fatal Debt (Must Be Addressed)

- memory safety hazards and undefined behavior in critical paths,
- concurrency hazards (data races, deadlocks) with no reliable reproduction,
- unclear ownership of key modules and no maintainers,
- architecture that prevents incremental change (forces big bang rewrites),
- lack of operational visibility for production-critical behavior.

23.2.4 Make Debt Visible: Measurement and Signals

If you cannot see debt, you cannot manage it. Large systems need objective signals.

Engineering Signals That Predict High Interest

- **Lead time increasing:** more time from change request to safe release.
- **Defect escape rate:** more production bugs per unit of change.
- **Hotspot churn:** the same files are changed repeatedly and still break.
- **Build pain:** CI times increase; developers avoid running tests locally.

- **Operational fragility:** incidents are frequent or hard to diagnose.

Static Metrics: Use Carefully

Static metrics can guide attention but should not drive decisions blindly:

- cyclomatic complexity and nesting depth
- dependency graph health (cycles, forbidden edges)
- header include fan-out and rebuild impact
- duplication density

Use these metrics to identify *hotspots*, then confirm with engineering judgment and production data.

The Hotspot Rule

Prioritize debt where:

- the code changes frequently,
- failures are expensive,
- the module is central to multiple teams.

A rarely touched ugly module is often low-interest debt. A frequently touched medium-ugly module is high-interest debt.

23.2.5 Governance: Turning Debt into a Managed Portfolio

Debt Register (Lightweight, Not Bureaucratic)

Maintain a small, living inventory for major debts:

- what is the debt?
- where is it located (module/boundary)?
- what is the impact (time, risk, incidents)?
- what is the plan and success criteria?
- who owns it?

Explicit Policies

Long-lived systems require policies that make debt management routine:

- **Definition of Done includes:** tests for behavior, minimal observability, documentation of invariants.
- **No new debt without a ticket:** temporary hacks require an expiration plan.
- **Boy Scout rule:** leave code better than you found it, within reason.
- **Architectural guardrails:** automated checks that prevent forbidden dependencies.

Capacity Allocation

A stable approach is to reserve recurring capacity for debt work:

- continuous refactoring (small, always),
- scheduled investments (larger, quarterly),
- incident-driven fixes (add regression tests, remove root causes).

23.2.6 Prioritization: Choosing the Right Debt to Pay

A technical debt item is worth paying when the return is clear and the risk is controlled.

The "Interest vs. Principal" Heuristic

- If the debt increases failure probability or blocks work, it has high interest.
- If the debt is merely ugly but stable and isolated, it may have low interest.

Decision Matrix

- **Impact:** time lost, incident frequency, security exposure, blocked features.
- **Reach:** number of teams/features affected.
- **Confidence:** do we understand the root cause and path to fix?
- **Risk:** will refactoring destabilize production?

Pay Debt at Natural Change Points

Debt repayment is cheapest when aligned with:

- a planned feature in that module,
- an infrastructure upgrade (toolchain, OS),
- a known migration window,
- an incident where root cause is clear.

23.2.7 Refactoring Strategies That Work at Scale in C++

Rule 1: Make Changes Incremental

Avoid big bang rewrites. They fail for the same reasons: incomplete behavioral equivalence, delayed feedback, and unbounded risk.
Prefer *strangler patterns* and parallel runs:

- introduce a new component behind an adapter,
- route a small percentage of traffic,
- compare results, then expand gradually.

Extract Seams Before You Refactor

If code is tightly coupled, first create a seam.

Introduce an Interface and Adapter to Create a Seam

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

struct IUserStore {
  virtual ~IUserStore() = default;
  virtual std::optional<std::string> get_email(std::string_view user_id) const = 0;
};

class LegacyUserStore final : public IUserStore {
public:
  std::optional<std::string> get_email(std::string_view user_id) const override {
    // Old code path preserved for now.
    return legacy_get_email(user_id);
  }

private:
  std::optional<std::string> legacy_get_email(std::string_view user_id) const;
};
```

Once you have a seam, you can replace the implementation with lower risk and better testing.

The Strangler Façade

Wrap legacy behavior behind a stable boundary, then replace internals piece by piece.

```
#include <memory>

class Engine {
public:
  virtual ~Engine() = default;
  virtual int compute(int x) = 0;
};

class LegacyEngine : public Engine {
public:
  int compute(int x) override { return legacy_compute(x); }
private:
  int legacy_compute(int);
};

class ModernEngine : public Engine {
public:
  int compute(int x) override { return modern_compute(x); }
private:
  int modern_compute(int);
};

class EngineFacade final {
public:
  explicit EngineFacade(std::unique_ptr<Engine> impl) : impl_(std::move(impl)) {}
  int compute(int x) { return impl_->compute(x); }

private:
  std::unique_ptr<Engine> impl_;
};
```

Now switching engines becomes a configuration and rollout decision, not a rewrite.

Stabilize Semantics with Characterization Tests

Before refactoring legacy logic, capture what it does today, including quirks. These tests are not "ideal" tests; they are *behavioral documentation* that prevents accidental regressions.

Characterization Test Shape

```
#include <cassert>

// A tiny example illustrating the idea.
// In real systems, use your existing test framework.
int legacy_rule(int input);

void characterize_legacy() {
  assert(legacy_rule(0) == 17);   // captures existing odd behavior
  assert(legacy_rule(1) == 17);
  assert(legacy_rule(2) == 21);
}
```

After refactoring, the new implementation must satisfy these tests until you intentionally change behavior with a controlled migration.

Reduce Header Coupling (A Major C++ Debt Lever)

Header coupling is one of the highest-interest debts in large C++:

- it slows build iteration,
- it forces unnecessary recompilation,
- it makes dependencies implicit and hard to reason about.

Tactics:

- forward declare in headers where possible,
- move heavy includes to `.cpp`,
- isolate third-party headers behind adapters,
- use PImpl selectively for large interfaces.

Forward Declaration Example

```
#pragma once
#include <memory>

class BigDependency; // forward declare

class Widget {
public:
  Widget();
  ~Widget();
  void run();

private:
  std::unique_ptr<BigDependency> dep_; // include in .cpp
};
```

Standardize Error Handling and Ownership

Inconsistent error handling creates combinatorial complexity. Standardization reduces friction.

- Choose boundaries where exceptions are allowed and where they are not.
- Convert at boundaries using adapters.

- Use RAII types and clear ownership rules (value types, `unique_ptr`, shared ownership only when necessary).

Boundary Adapter Example (Illustrative)

```
#include <string>

struct Error { int code; std::string msg; };

template <class T>
struct Expected {
  bool ok;
  T value;
  Error err;
};

int core_compute(int x); // may throw

Expected<int> api_compute(int x) noexcept {
  try {
    return {true, core_compute(x), {}};
  } catch (const std::exception& e) {
    return {false, 0, {1, e.what()}};
  }
}
```

This makes external behavior stable even as internals evolve.

23.2.8 Testing Debt: The Highest ROI Category

If you can only repay one category of debt, repay *test and observability debt*. It unlocks safe refactoring across all other debt types.

Focus on Boundary Tests

High-value tests:

- validate business behavior at service/module boundaries,
- verify data format and schema evolution,
- cover concurrency invariants (where possible),
- reproduce past incidents (regression tests).

Control Flakiness

Flaky tests create "CI debt":

- developers stop trusting the pipeline,
- teams merge risky changes to avoid re-running,
- release confidence collapses.

Treat flaky tests as production bugs: triage, fix root cause, and prevent recurrence.

23.2.9 Operational Debt: Paying for Faster Diagnosis

Long-lived systems must be diagnosable in production years later.

Structured Logging Contract

Define stable event names and fields.

```
#include <string>
#include <map>
#include <iostream>
```

```
struct LogEvent {
  std::string name;
  std::map<std::string, std::string> fields;
};

void log_info(const LogEvent& e) {
  std::cout << e.name;
  for (auto& [k, v] : e.fields) std::cout << " " << k << "=" << v;
  std::cout << "\n";
}

void example() {
  log_info({"user.login", {{"user_id","42"}, {"result","ok"}}});
}
```

The durability comes from consistency: tools and dashboards can rely on stable keys.

Runbooks and "Known Failure Modes"

Write down:

- the top failure modes,
- how to detect them (metrics/logs),
- how to mitigate safely (rollbacks, toggles, restarts),
- how to prevent recurrence (design changes and tests).

23.2.10 When Debt Repayment Becomes a Project

Sometimes debt is too large for "continuous refactoring" and requires an explicit project.

Define a Narrow, Measurable Outcome

Examples of measurable outcomes:

- reduce build time from 40 minutes to 15,
- reduce incident rate for subsystem X by 50%,
- enable safe migration to a new storage format,
- eliminate a class of UB/race in critical paths.

Use Parallelism with Safety

- keep old path running while new path is validated,
- compare outputs in shadow mode,
- ramp traffic gradually,
- preserve rollback capability.

23.2.11 Anti-Patterns in Technical Debt Management

The "Debt Sprint" Without Follow-Through

A one-time cleanup without governance usually fails. Debt returns because incentives and guardrails did not change.

Refactoring Without Tests

Refactoring without a safety net is not debt repayment; it is debt gambling.

Big Bang Rewrites

Big bang rewrites delay feedback and multiply unknowns. They are sometimes unavoidable, but in long-lived systems they should be treated as last resort, and still executed incrementally where possible.

23.2.12 A Large-Scale Technical Debt Playbook (C++ Focused)

Weekly (Continuous)

- apply the Boy Scout rule in touched areas,
- add characterization tests around modified legacy code,
- pay small build debt (remove unused includes, reduce header exposure),
- improve observability for new features by default.

Monthly (Governance)

- review debt register: top hotspots and interest signals,
- ensure module ownership is current,
- address top CI/build pain points,
- kill or revise obsolete deprecations and migrations.

Quarterly (Investment)

- tackle one or two major debt items with clear outcomes,
- reduce architectural coupling and dependency cycles,

- standardize error handling and concurrency rules in key subsystems,
- perform toolchain modernization with compatibility windows.

23.2.13 Closing Principle: Debt Is Managed by Feedback Loops

In a 20+ year C++ codebase, technical debt management is not a one-time cleanup. It is a system of feedback loops:

- **Visibility:** debt is detectable through metrics, hotspots, and operational signals.
- **Governance:** ownership, policies, and capacity exist to act on debt continuously.
- **Safety nets:** tests and observability make refactoring low risk.
- **Incremental change:** adapters, seams, and gradual rollouts keep evolution predictable.

When these loops exist, technical debt becomes a controlled portfolio that supports scale and longevity rather than a hidden force that slowly destroys the cost of change.

23.3 Design for Team Growth

Systems that scale in users, data, and features often fail to scale in the most critical dimension: the number of engineers who can change them safely. In long-lived C++ codebases, team growth amplifies every weakness in boundaries, build discipline, testing strategy, ownership, and operational clarity. Designing for team growth means designing so that *more people can work in parallel with less coordination cost, lower defect rates, and predictable delivery*.
This section provides concrete architectural, process, and code-level practices that enable team growth without collapsing into merge conflicts, brittle releases, and architectural drift.

23.3.1 The Core Problem: Coordination Cost Dominates

As a team grows, the dominant cost becomes coordination, not coding. A scalable architecture reduces:

- **Cognitive load:** how much a developer must understand before making a safe change.
- **Coupling:** how many other components must be touched for one change.
- **Synchronization:** how often developers must coordinate timing, releases, and migrations.
- **Integration friction:** how hard it is to build, test, and ship changes reliably.

Design for team growth is therefore the design of *parallelism*: parallel development, parallel testing, parallel deployment, and parallel ownership.

23.3.2 Organize Architecture Around Change and Ownership

Modules as Units of Ownership

A module is not only a technical boundary; it is an ownership boundary. For team growth, each module should have:

- a clear purpose and public API,
- explicit invariants and thread-safety rules,
- a build target with declared dependencies,
- tests and a small set of maintainers.

When ownership is unclear, changes become political and slow, and quality becomes inconsistent.

Choose Boundaries to Minimize Cross-Team Changes

A boundary is successful when most changes stay within a single module. If a feature routinely requires touching many modules owned by different teams, boundaries are misaligned with the actual change patterns.
Practical guidance:

- align modules with business capabilities or stable technical responsibilities,
- avoid splitting by "layer" only (UI team, DB team) when domain behavior spans layers,
- keep integration points narrow and stable.

A Stable Core with Replaceable Adapters

A durable structure for team growth is:

- **Core:** domain logic and invariants, minimal dependencies,
- **Ports:** interfaces that the core requires,
- **Adapters:** implementations for databases, messaging, HTTP, files, etc.

This structure allows different teams to work on adapters without destabilizing the core.

Port/Adapter Skeleton

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

struct IUserRepository {
  virtual ~IUserRepository() = default;
  virtual std::optional<std::string> email_for(std::string_view user_id) const = 0;
};
```

```
class UserService {
public:
  explicit UserService(const IUserRepository& repo) : repo_(repo) {}

  bool can_send_email(std::string_view user_id) const {
    auto email = repo_.email_for(user_id);
    return email.has_value() && !email->empty();
  }

private:
  const IUserRepository& repo_;
};
```

Teams can replace repository implementations independently while preserving service behavior.

23.3.3 Design APIs for Many Callers and Many Authors

APIs Must Be Hard to Misuse

As the number of developers grows, accidental misuse becomes a dominant failure mode. A scalable API:

- encodes invariants in types,
- makes invalid states unrepresentable where practical,
- documents preconditions and ownership rules in the interface,
- provides safe defaults and explicit “dangerous” escape hatches.

Strong Types Reduce Coordination

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

class UserId {
public:
  static UserId from_string(std::string_view s) {
    if (s.empty()) throw std::invalid_argument("UserId: empty");
    return UserId(std::string{s});
  }
  std::string_view view() const noexcept { return v_; }

private:
  explicit UserId(std::string v) : v_(std::move(v)) {}
  std::string v_;
};

struct EmailAddress {
  static EmailAddress from_string(std::string_view s) {
    if (s.find('@') == std::string_view::npos) throw
    ↪  std::invalid_argument("Email: invalid");
    return EmailAddress(std::string{s});
  }
  std::string_view view() const noexcept { return v_; }

private:
  explicit EmailAddress(std::string v) : v_(std::move(v)) {}
  std::string v_;
};
```

Strong types prevent classes of bugs without requiring social coordination.

Stabilize Error Semantics

In multi-team systems, inconsistent error handling becomes integration debt.

Recommendations:

- choose one dominant error model per layer (exceptions or result types),
- convert at boundaries with explicit adapters,
- standardize error categories (validation, not found, conflict, transient, internal).

Boundary Conversion Adapter (Illustrative)

```
#include <string>
#include <variant>

struct Error { int code; std::string msg; };

template <class T>
using Result = std::variant<T, Error>;

int core_op(int x); // may throw

Result<int> api_op(int x) noexcept {
  try {
    return core_op(x);
  } catch (const std::exception& e) {
    return Error{1, e.what()};
  }
}
```

Compatibility Windows Enable Independent Evolution

As teams grow, independent release cycles become necessary. Support compatibility windows:

- keep old APIs for a defined time,
- deprecate with clear migration paths,

- allow clients to upgrade gradually.

23.3.4 Reduce Developer Friction: Build, Test, and Tooling

In C++, build friction is a team-scaling killer. The fastest way to slow a growing organization is to let build and CI times grow without control.

Build as a Product

Treat build and CI like production systems:

- stable, reproducible toolchains,
- deterministic builds,
- fast iteration loops (incremental builds, caching),
- consistent compiler flags and warning policies.

Header Hygiene to Reduce Rebuild Cascades

Actions that scale well:

- keep public headers minimal and stable,
- forward declare to reduce include fan-out,
- move heavy includes to `.cpp`,
- isolate third-party dependencies behind adapters.

Header Firebreak Example

```
#pragma once
#include <memory>

class ThirdPartyClient; // forward declaration

class PaymentGateway {
public:
  PaymentGateway();
  ~PaymentGateway();
  bool charge(int cents);

private:
  std::unique_ptr<ThirdPartyClient> client_; // include third-party header in .cpp
};
```

Testing Strategy That Supports Parallel Change

A scalable test strategy:

- fast unit tests for pure logic,
- integration tests around module boundaries,
- contract tests for inter-module APIs,
- regression tests for incidents and bugs.

Key team-growth principle: *tests should make changes safer without making changes slow*. Prefer tests that fail deterministically and run quickly in CI.

23.3.5 Make Architecture Discoverable

A growing team requires the system to teach itself.

Living Documentation in the Repository

Focus on documentation that answers:

- What does this module do?
- What invariants must hold?
- What are the failure modes?
- How do you run tests and debug production issues?
- What is the safe way to extend the module?

Examples as Contract

In C++, small compilable examples are often better than prose. Provide usage examples that encode rules about ownership, threading, and error handling.

Usage Example as Documentation

```
void example_usage(UserService& svc, UserId id) {
  // Rule: service methods are thread-safe for concurrent reads.
  // Rule: returned values are owned by the caller (value types).
  if (svc.can_send_email(id.view())) {
    // ...
  }
}
```

23.3.6 Prevent Architectural Drift

As more teams and developers contribute, architecture drifts unless you enforce constraints.

Guardrails Over Guidelines

Guidelines are necessary but insufficient. Guardrails are enforceable:

- build system rules for allowed dependencies,
- linting and formatting gates,
- CI checks that prevent new cycles or forbidden includes,
- review checklists focused on invariants and boundaries.

Dependency Rules as Code

Even a simple rule set helps prevent gradual decay:

- core cannot depend on adapters,
- domain types cannot include framework headers,
- platform code cannot leak into cross-platform modules.

23.3.7 Design for Safe Parallel Development

Feature Flags and Incremental Rollouts

Feature flags allow teams to ship incomplete work safely:

- merge code early and often,
- enable gradually,
- rollback without reverting commits.

Feature Flag Skeleton

```
#include <string_view>

struct Flags {
  bool new_ranking = false;
};

int rank_legacy(std::string_view q);
int rank_modern(std::string_view q);

int rank(std::string_view q, const Flags& f) {
  return f.new_ranking ? rank_modern(q) : rank_legacy(q);
}
```

Branch by Abstraction

Instead of long-lived branches in version control, branch in code with abstractions:

- introduce an interface,
- ship both implementations,
- migrate callers,
- delete the old implementation.

This reduces merge pain and makes progress visible.

23.3.8 Concurrency Rules That Scale Across Teams

Concurrency is a coordination problem disguised as a technical problem. Make it explicit.

Thread-Safety Annotations as Policy

Every shared type should declare:

- thread-safe for reads only,
- thread-safe for reads/writes,
- not thread-safe (requires external synchronization).

Prefer Ownership Transfer and Message Passing

When multiple teams build features on the same runtime, shared mutable state becomes a bottleneck. Prefer:

- immutable data passed between subsystems,
- explicit ownership transfer,
- queues and command processing.

Message Passing Skeleton

```
#include <mutex>
#include <queue>
#include <condition_variable>
#include <functional>

class Mailbox {
public:
  void send(std::function<void()> msg) {
    {
      std::lock_guard<std::mutex> lk(m_);
      q_.push(std::move(msg));
```

```
    }
    cv_.notify_one();
  }

  std::function<void()> receive() {
    std::unique_lock<std::mutex> lk(m_);
    cv_.wait(lk, [&]{ return !q_.empty(); });
    auto msg = std::move(q_.front());
    q_.pop();
    return msg;
  }

private:
  std::mutex m_;
  std::condition_variable cv_;
  std::queue<std::function<void()>> q_;
};
```

This confines mutation to known points and reduces cross-team concurrency bugs.

23.3.9 Operational Design for Many Teams

Multiple teams operating the same system must share a common operational language.

Standard Observability Contracts

Standardize:

- metric naming and dimensions,
- structured log fields,
- trace propagation and correlation identifiers.

This allows teams to debug cross-component behavior without bespoke knowledge.

Runbooks and On-Call Readiness

For team growth, runbooks are not optional:

- how to detect major failure modes,
- safe mitigation steps,
- rollback procedures,
- escalation paths and ownership.

23.3.10 Leadership Practices Encoded into the System

A scalable system encodes good engineering practices into defaults.

Make the Right Thing the Easy Thing

Examples:

- new module templates include tests and logging scaffolding,
- CI runs fast and is trusted,
- common patterns are provided as libraries (error model, config, metrics),
- safe primitives are readily available (strong types, RAII wrappers).

Review Culture Focused on Invariants

In large C++ systems, reviews should emphasize:

- boundary integrity and dependency direction,
- ownership and lifetime rules,

- concurrency safety,
- error and failure semantics,
- observability and operability.

23.3.11 A Checklist for Team-Scalable Design

Architecture

- modules are small enough to understand, large enough to own,
- dependency direction is enforced,
- core is clean; adapters are replaceable,
- boundaries align with change patterns.

APIs

- hard to misuse, invariants encoded in types,
- consistent error semantics and conversions at boundaries,
- compatibility windows and deprecation policy exist.

Build and Tests

- fast local builds and fast CI,
- header hygiene reduces rebuild cascades,
- tests are reliable and focus on boundaries and regression.

Operations

- standardized observability across components,
- runbooks exist for top failure modes,
- safe rollout and rollback practices are routine.

23.3.12 Closing Principle: Team Growth Is a First-Class Scalability Requirement

Scaling a C++ system for team growth requires the same seriousness as scaling for performance:

- invest in boundaries that enable parallel work,
- reduce friction in build and testing,
- enforce architecture with guardrails,
- encode safety and operability into defaults.

When done correctly, adding engineers increases throughput rather than increasing chaos, and the system remains evolvable for decades.

23.4 Designing for Maintenance

Maintenance is the dominant lifecycle activity of long-lived systems. The cost of a system is rarely its initial implementation; it is the continuous work of understanding, changing, testing, deploying, observing, and operating it over many years. Designing for maintenance means designing so that *routine change stays routine*: changes remain local, behavior remains predictable, defects remain diagnosable, and improvements can be delivered incrementally with controlled risk.

In Modern C++, maintainability is strongly influenced by how we manage boundaries, dependencies, ownership and lifetime rules, error semantics, concurrency rules, tests, and build discipline. This section provides concrete principles and patterns that keep maintenance cheap and reliable at scale.

23.4.1 Maintenance as an Engineering Constraint

A maintainable system optimizes for:

- **Local reasoning:** developers can understand the impact of a change from nearby code and explicit contracts.
- **Predictable behavior:** explicit invariants, consistent error semantics, and stable boundaries.
- **Low-risk change:** fast feedback (tests, CI), safe deployment, and rollback paths.
- **Diagnosability:** strong observability and reproducible failure modes.

If maintenance becomes expensive, the system stops evolving safely, and eventually the organization is forced into risky rewrites or lives with stagnation.

23.4.2 The Maintainability Model: Reduce the Cost of Change

You can think of maintainability as a function of the *cost of change*:

- **Understanding cost:** time to understand what to change and why.
- **Modification cost:** time to implement the change correctly.
- **Verification cost:** time to prove the change is safe (tests, reviews, CI).
- **Integration cost:** time to merge, deploy, monitor, and roll back if needed.

Good design reduces each of these costs systematically.

23.4.3 Boundary Discipline: Maintenance Happens at Interfaces

Prefer Narrow, Stable Interfaces

Maintenance improves when interfaces are:

- small (few operations),
- explicit about invariants and ownership,
- stable over time,
- hard to misuse.

Wide interfaces create accidental coupling, increase the blast radius of changes, and encourage callers to depend on incidental behavior.

Hide Volatility Behind Stable Façades

Parts of the system that change frequently (libraries, protocols, storage engines) should be hidden behind stable facades or adapters.

Adapter Pattern to Isolate a Volatile Dependency

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

struct IKeyValueStore {
  virtual ~IKeyValueStore() = default;
  virtual std::optional<std::string> get(std::string_view key) const = 0;
  virtual void put(std::string key, std::string value) = 0;
};

class Settings {
```

```
public:
  explicit Settings(const IKeyValueStore& store) : store_(store) {}

  std::optional<std::string> read(std::string_view k) const { return store_.get(k); }

private:
  const IKeyValueStore& store_;
};
```

A stable interface allows internal replacement without forcing system-wide edits.

Avoid Leaking Representation

Do not expose:

- internal containers and allocators,
- third-party types,
- threading primitives,
- layout-sensitive structures,

unless you intentionally commit to them as part of the public contract.

23.4.4 Dependency Management: The Most Common Maintenance Failure

Enforce Dependency Direction

A maintainable architecture has a clear direction of dependencies:

- core logic is independent of frameworks and I/O,
- adapters depend on the core,

- orchestration ties components together.

When dependencies flow backward (core depends on DB/UI), maintenance becomes expensive because changes to infrastructure ripple through the system.

Break Cycles Early

Dependency cycles create:

- fragile initialization order,
- hidden global interactions,
- build-time and runtime entanglement,
- refactoring paralysis.

Break cycles with:

- interface extraction,
- event/message passing,
- moving shared concepts into a small common module,
- removing hidden global state.

Header Hygiene for Long-Term Change

In large C++ systems, header sprawl is a maintenance killer.
Maintainability improves when:

- public headers are minimal and stable,
- heavy includes move to `.cpp`,

- forward declarations are used where appropriate,
- third-party headers are isolated behind adapters.

Forward Declaration to Reduce Coupling

```
#pragma once
#include <memory>

class HeavyType; // forward declare

class Processor {
public:
  Processor();
  ~Processor();
  void run();

private:
  std::unique_ptr<HeavyType> impl_; // include heavy headers in .cpp
};
```

23.4.5 Consistency: The Maintenance Multiplier

Every inconsistency multiplies mental load. Consistency is an engineering choice.

Consistent Error Semantics

Pick a clear policy:

- exceptions within trusted boundaries,
- result-based errors across module boundaries,
- conversions only at defined seams.

Conversion at Boundaries (Illustrative)

```
#include <string>
#include <variant>

struct Error { int code; std::string msg; };
template <class T>
using Result = std::variant<T, Error>;

int core_compute(int x); // may throw

Result<int> api_compute(int x) noexcept {
  try {
    return core_compute(x);
  } catch (const std::exception& e) {
    return Error{1, e.what()};
  }
}
```

Maintenance improves because the external contract remains stable even if internal implementations evolve.

Consistent Ownership and Lifetime Rules

In long-lived C++ systems, hidden ownership rules are a common root cause of bugs and fear-driven development.
Maintenance-friendly rules:

- value semantics by default,
- unique ownership with `std::unique_ptr`,
- shared ownership only when a clear shared-lifetime requirement exists,
- no raw owning pointers; document any non-owning pointer/view.

Express Ownership Explicitly

```
#include <memory>
#include <string_view>

class Node;

class Graph {
public:
  Node& add_node(); // owned by Graph
  Node* find(std::string_view id) noexcept; // non-owning

private:
  std::vector<std::unique_ptr<Node>> nodes_;
};
```

Consistent Concurrency Rules

Declare and enforce thread-safety expectations:

- thread-safe for reads only,
- thread-safe for reads/writes,
- not thread-safe (external synchronization required).

Ambiguity here produces years of defensive, slow, bug-prone maintenance.

23.4.6 Testing as the Maintenance Safety Net

Maintenance work is mostly change work. Without tests, change is unsafe.

Prefer Boundary-Focused Tests

High ROI tests:

- unit tests for pure logic,
- integration tests around module boundaries (storage, network, IPC),
- contract tests for APIs shared by teams,
- regression tests for incidents.

Characterization Tests for Legacy Code

When refactoring old code, capture current behavior first, including quirks. This prevents accidental changes.

```
#include <cassert>

int legacy_rule(int);

void characterize() {
  assert(legacy_rule(0) == 17); // documents existing behavior
  assert(legacy_rule(1) == 17);
  assert(legacy_rule(2) == 21);
}
```

Make Tests Fast and Reliable

A maintainable pipeline is trusted. Flaky tests are maintenance debt that grows rapidly:

- developers rerun pipelines until “green”,
- failures are ignored,

- confidence collapses.

Treat flaky tests as production defects: triage and fix root causes.

23.4.7 Observability: Maintenance Requires Diagnose-Ability

If a system cannot be observed, maintenance becomes guesswork.

Design for Debugging

Every critical component should expose:

- structured logs for significant events,
- metrics for throughput, latency, and error rates,
- health/status snapshots.

Health Snapshot Interface

```
#include <string>
#include <cstdint>

struct Health {
  bool ok = true;
  std::string summary;
  std::uint64_t requests = 0;
  std::uint64_t failures = 0;
};

struct IHealthProvider {
  virtual ~IHealthProvider() = default;
  virtual Health health() const = 0;
};
```

Make Failures Reproducible

Maintenance accelerates when failures can be reproduced:

- deterministic tests and stable seeds for randomized tests,
- minimal reproducible cases stored as fixtures,
- recorded inputs for parsing and protocol failures.

23.4.8 Refactoring Strategy: Maintainability Without Rewrites

Prefer Incremental Refactoring

Large rewrites are high risk and delay feedback. Maintainability improves via:

- small steps merged frequently,
- seams (interfaces) added before large changes,
- feature flags and gradual rollouts when behavior changes.

Branch by Abstraction

Keep old and new implementations behind an interface and migrate gradually.

```
#include <memory>

struct Engine {
  virtual ~Engine() = default;
  virtual int compute(int x) = 0;
};

struct LegacyEngine : Engine {
  int compute(int x) override { return legacy(x); }
```

```
  int legacy(int);
};

struct NewEngine : Engine {
  int compute(int x) override { return modern(x); }
  int modern(int);
};

class EngineFacade {
public:
  explicit EngineFacade(std::unique_ptr<Engine> e) : e_(std::move(e)) {}
  int compute(int x) { return e_->compute(x); }

private:
  std::unique_ptr<Engine> e_;
};
```

This keeps maintenance safe during migrations and enables rollback.

23.4.9 Maintenance-Friendly Modern C++ Practices

Prefer Value Types and Immutable Data Where Possible

Immutable data reduces hidden coupling and concurrency hazards, and makes reasoning local.

Use RAII to Make Resource Management Boring

Boring is good for maintenance. RAII reduces leak risk and error paths. Prefer composition of small RAII wrappers that encode rules (file handles, sockets, locks, transactions).

RAII Wrapper Skeleton

```
#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("fopen failed");
  }
  ~File() { if (f_) std::fclose(f_); }

  File(const File&) = delete;
  File& operator=(const File&) = delete;

  File(File&& o) noexcept : f_(o.f_) { o.f_ = nullptr; }
  File& operator=(File&& o) noexcept {
    if (this != &o) {
      if (f_) std::fclose(f_);
      f_ = o.f_;
      o.f_ = nullptr;
    }
    return *this;
  }

  std::FILE* get() const noexcept { return f_; }

private:
  std::FILE* f_ = nullptr;
};
```

Make Invariants Explicit

Document invariants in code and enforce them through construction and validation.

23.4.10 Culture and Process Embedded in Design

Make the Right Thing the Easy Thing

A maintainable system provides:

- module templates that include tests and logging scaffolding,
- standard libraries for configuration, error reporting, metrics,
- a unified style and review checklist.

Ownership and On-Call Feedback Loop

Maintenance quality improves when:

- every module has owners,
- incidents feed regression tests,
- postmortems produce design improvements and guardrails.

23.4.11 A Practical Checklist for Designing for Maintenance

Architecture

- boundaries are clear and narrow,
- dependency direction is enforced,
- volatile details are isolated behind adapters,
- representation does not leak through APIs.

Code and Contracts

- invariants are explicit and enforced,
- ownership and lifetime are documented and encoded in types,
- error semantics are consistent across modules,
- concurrency rules are stated and testable where possible.

Tooling and Build

- builds are reproducible and fast enough for iteration,
- headers are minimal; rebuild cascades are controlled,
- CI is trusted and stable.

Testing and Operations

- boundary tests exist and run quickly,
- flakiness is treated as a defect,
- observability is standard and supports diagnosis,
- failures are reproducible and captured as regression tests.

23.4.12 Closing Principle: Maintenance Is Designed, Not Hoped For

A system becomes maintainable when it is designed to keep change local, behavior predictable, and verification fast. In Modern C++, this requires disciplined boundaries, consistent contracts (ownership, errors, thread-safety), strong testing and observability, and incremental refactoring strategies. With these in place, the codebase remains evolvable, safe, and economically sustainable for decades.

Testing as Design Feedback

24.1 Testability as a Design Property

Testability is not a testing technique; it is a *design property*. A system is testable when its behavior can be verified *quickly, deterministically, and locally* without requiring heroic setup, fragile mocks, or production-only observation. In large-scale C++ systems, testability is a leading indicator of long-term health: the more testable the design, the cheaper and safer change becomes.

This section treats testability as an architectural requirement, explains why testability is a strong proxy for good design, and provides concrete Modern C++ patterns that make systems intrinsically testable.

24.1.1 Why Testability Is a Design Property (Not an Afterthought)

Testability Measures the Quality of Boundaries

If a module is hard to test, it is often because:

- boundaries are unclear or too wide,
- responsibilities are mixed (I/O + policy + domain logic),
- dependencies are hidden (globals, singletons, environment),

- behavior depends on timing, randomness, or ordering without control.

A testable design makes dependencies explicit and isolates side effects. This is a design decision, not a test-suite decision.

Testability Forces You to Make Contracts Explicit

To test something, you must be able to state what "correct" means. That pressure produces:

- explicit invariants and preconditions,
- explicit error semantics,
- stable and narrow APIs,
- deterministic behavior under controlled inputs.

In Large Systems, Testability Is the Feedback Loop

Testing provides the fastest feedback on:

- whether a change preserved behavior,
- whether refactoring improved boundaries,
- whether new code is modular or entangled.

Without fast, reliable tests, teams compensate with slower processes: larger reviews, slower releases, conservative changes, and eventually stagnation.

24.1.2 A Practical Definition of Testability

A design is testable when it supports the following properties:

Isolation

The unit under test can be exercised without real databases, real networks, or real clocks.

Determinism

Given the same inputs, the outcome is the same (or at least controllably random).

Observability

The test can observe the effect of the unit under test through outputs, state, or emitted events without intrusive instrumentation.

Controllability

The test can control external influences: time, randomness, concurrency scheduling boundaries, and I/O outcomes.

Locality

A small change typically requires changing a small set of tests, and the test failure points to the relevant code.

24.1.3 The Main Enemies of Testability in C++ Systems

Hidden Dependencies

- global singletons and service locators,
- implicit environment reads (filesystem paths, env vars, current time),
- static initialization order and hidden registries.

Mixed Concerns

- business logic intertwined with I/O,
- parsing intertwined with state mutation,
- concurrency intertwined with domain rules.

Non-Determinism

- wall-clock time, real timers, thread races,
- random generators without fixed seeds,
- background threads without controllable shutdown.

Poor Interfaces

- wide classes with many responsibilities,
- methods with complex side effects and implicit state,
- APIs that return nothing and hide outcomes (“void everywhere”).

24.1.4 Design Moves That Create Testability

Separate Pure Policy from Side Effects

The strongest design move for testability is to keep domain rules as pure, value-based transformations, and push side effects to the edges.

Pure Function as Policy

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

enum class PasswordStrength { weak, medium, strong };

PasswordStrength evaluate_password(std::string_view s) {
  bool has_upper = false, has_lower = false, has_digit = false;
  for (char c : s) {
    if (c >= 'A' && c <= 'Z') has_upper = true;
    if (c >= 'a' && c <= 'z') has_lower = true;
    if (c >= '0' && c <= '9') has_digit = true;
  }
  if (s.size() >= 12 && has_upper && has_lower && has_digit) return
  ↪  PasswordStrength::strong;
  if (s.size() >= 8  && (has_upper || has_lower) && has_digit) return
  ↪  PasswordStrength::medium;
  return PasswordStrength::weak;
}
```

Pure policy is trivial to test: no mocks, no setup, no flakiness.

Make Dependencies Explicit (Constructor Injection)

Testability improves when dependencies are explicit parameters rather than hidden globals.

Dependency Injection via Interface

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

struct IUserRepository {
  virtual ~IUserRepository() = default;
```

```
  virtual std::optional<std::string> email_for(std::string_view user_id) const = 0;
};

class UserService {
public:
  explicit UserService(const IUserRepository& repo) : repo_(repo) {}

  bool can_send_email(std::string_view user_id) const {
    auto e = repo_.email_for(user_id);
    return e.has_value() && !e->empty();
  }

private:
  const IUserRepository& repo_;
};
```

Now the test supplies a fake repository deterministically.

Simple Fake for Tests

```
#include <unordered_map>

class FakeUserRepository final : public IUserRepository {
public:
  void add(std::string id, std::string email) { data_[std::move(id)] = std::move(email); }

  std::optional<std::string> email_for(std::string_view user_id) const override {
    auto it = data_.find(std::string(user_id));
    if (it == data_.end()) return std::nullopt;
    return it->second;
  }

private:
  std::unordered_map<std::string, std::string> data_;
```

```
};
```

Prefer Interfaces at Architectural Boundaries, Not Everywhere

A common mistake is “interface everything.” Testability comes from seams at boundaries:

- persistence, clocks, randomness, network clients,
- external services and process boundaries,
- expensive or nondeterministic dependencies.

Inside a module, value types and direct calls often produce simpler, more testable code than deep interface graphs.

24.1.5 Control Time: Clocks as Dependencies

Time is one of the biggest sources of flakiness. Treat time as a dependency.

Clock Interface

```
#include <chrono>

struct IClock {
  using time_point = std::chrono::steady_clock::time_point;
  virtual ~IClock() = default;
  virtual time_point now() const noexcept = 0;
};

class SystemClock final : public IClock {
public:
  time_point now() const noexcept override {
    return std::chrono::steady_clock::now();
  }
```

```
};

class FakeClock final : public IClock {
public:
  time_point now() const noexcept override { return t_; }
  void advance(std::chrono::milliseconds d) { t_ += d; }

private:
  time_point t_ = time_point{};
};
```

Using a Clock to Avoid Sleeping in Tests

```
#include <chrono>

class ExpiringToken {
public:
  ExpiringToken(const IClock& clk, std::chrono::milliseconds ttl)
    : clk_(clk), created_(clk.now()), ttl_(ttl) {}

  bool expired() const noexcept {
    return (clk_.now() - created_) > ttl_;
  }

private:
  const IClock& clk_;
  IClock::time_point created_;
  std::chrono::milliseconds ttl_;
};
```

Tests advance `FakeClock` rather than waiting in real time.

24.1.6 Control Randomness: Deterministic Generators

Randomness without control creates irreproducible failures. Always inject generators or seeds.

Inject a Generator

```
#include <cstdint>

struct IRng {
  virtual ~IRng() = default;
  virtual std::uint32_t next_u32() noexcept = 0;
};

class FixedRng final : public IRng {
public:
  explicit FixedRng(std::uint32_t v) : v_(v) {}
  std::uint32_t next_u32() noexcept override { return v_; }
private:
  std::uint32_t v_;
};

std::uint32_t choose_bucket(IRng& rng, std::uint32_t buckets) noexcept {
  return buckets == 0 ? 0u : (rng.next_u32() % buckets);
}
```

24.1.7 Design for Observability: Return Values, Events, and State Snapshots

Tests need something reliable to assert.

Prefer Returning Results Over “Void With Side Effects”

```
#include <string>
```

```
enum class SaveStatus { ok, conflict, invalid, transient_error };

struct SaveResult {
  SaveStatus status = SaveStatus::ok;
  std::string message;
};

class Saver {
public:
  SaveResult save(std::string payload);
};
```

This improves testability and production observability simultaneously.

Event Sinks as Testable Outputs

Instead of writing directly to global loggers or metrics systems, depend on an event sink.

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

struct Event { std::string name; };

struct IEventSink {
  virtual ~IEventSink() = default;
  virtual void emit(Event e) = 0;
};

class VectorSink final : public IEventSink {
public:
  void emit(Event e) override { events.push_back(std::move(e)); }
  std::vector<Event> events;
};
```

```
class Component {
public:
  explicit Component(IEventSink& sink) : sink_(sink) {}

  void do_work(bool ok) {
    sink_.emit({ok ? "work.ok" : "work.fail"});
  }

private:
  IEventSink& sink_;
};
```

Tests can assert on `VectorSink::events` with zero production dependencies.

24.1.8 Testability and Error Handling Design

Testability improves when failures are explicit and classifiable.

Errors as Values (Useful at Boundaries)

```
#include <string>
#include <variant>

struct Error {
  enum class Kind { validation, not_found, conflict, transient, internal };
  Kind kind = Kind::internal;
  std::string message;
};

template <class T>
using Result = std::variant<T, Error>;

Result<int> parse_positive_int(const std::string& s) {
```

```
  if (s.empty()) return Error{Error::Kind::validation, "empty"};
  int v = 0;
  for (char c : s) {
    if (c < '0' || c > '9') return Error{Error::Kind::validation, "non-digit"};
    v = v * 10 + (c - '0');
  }
  if (v <= 0) return Error{Error::Kind::validation, "non-positive"};
  return v;
}
```

Tests can cover all branches deterministically, and call sites can handle errors consistently.

Exceptions as Internal Enforcement (Useful Inside Modules)

Within a trusted module boundary, exceptions can express “this should not happen” states while keeping APIs clean. The key for testability is consistency:

- do not throw across boundaries that promise `noexcept`,
- convert exceptions to error values at API edges,
- keep exception types meaningful and stable.

24.1.9 Testability Under Concurrency

Concurrency makes determinism harder, but design can restore control.

Prefer Message Passing at Boundaries

Message passing confines mutation and simplifies tests.

```
#include <functional>
#include <queue>
```

```
class StepExecutor {
public:
  void post(std::function<void()> f) { q_.push(std::move(f)); }

  bool step_one() {
    if (q_.empty()) return false;
    auto f = std::move(q_.front());
    q_.pop();
    f();
    return true;
  }

private:
  std::queue<std::function<void()>> q_;
};
```

A test can drive concurrency-like logic deterministically by calling `step_one()` in a controlled sequence.

Design for Controlled Shutdown

Background threads without clean shutdown protocols create flaky tests and production leaks. Provide:

- explicit `start()` / `stop()` ownership,
- idempotent stop,
- bounded queues and timeouts,
- join semantics (or futures) that tests can wait on deterministically.

24.1.10 Testability and Architecture: Seams, Not Scaffolding

Seams Are Architectural Assets

A seam is a deliberate boundary where behavior can be swapped for testing or evolution:

- storage seam (repository),
- time seam (clock),
- randomness seam (rng),
- external service seam (client),
- process seam (IPC boundary).

Avoid Overspecifying Internals in Tests

A maintainable test suite asserts:

- externally observable behavior,
- invariants and contracts,
- boundary interactions (contract tests),

not the internal call order or private data layouts. Otherwise refactoring becomes expensive.

24.1.11 Design Patterns That Increase Testability

Functional Core, Imperative Shell

- core: pure functions and value types (easy to test),
- shell: I/O and integration (tested with a smaller number of integration tests).

Ports and Adapters

Define interfaces for what the core needs and implement adapters outside the core. This supports both testing and long-term evolution.

Command/Query Separation

Separate “do” from “observe” to make tests easier:

- commands change state and return status,
- queries return state without mutation.

Explicit State Machines

State machines make behavior explicit and testable across all transitions.

```
enum class State { idle, running, failed, done };

struct Machine {
  State s = State::idle;

  bool start() noexcept {
    if (s != State::idle) return false;
    s = State::running;
    return true;
  }

  void succeed() noexcept { if (s == State::running) s = State::done; }
  void fail() noexcept    { if (s == State::running) s = State::failed; }
};
```

State transitions become straightforward to test exhaustively.

24.1.12 A Testability Checklist for Modern C++ Designs

Dependency and Side-Effect Control

- dependencies are explicit (constructor parameters),
- time and randomness are injectable,
- no hidden globals required for core logic.

Determinism

- no sleeping in tests; advance fake clocks,
- deterministic scheduling where possible (step executors),
- stable seeds and fixtures for randomized tests.

Observability

- functions return meaningful results or status,
- events are emitted via injectable sinks,
- state snapshots exist where helpful.

Contracts

- invariants documented and enforced,
- consistent error semantics and conversions at boundaries,
- APIs hard to misuse, with clear ownership and lifetime rules.

Maintainability of Tests

- tests assert behavior, not internal implementation details,
- fast unit tests dominate; slower integration tests are targeted,
- flakiness treated as a defect.

24.1.13 Closing Principle: Testability Predicts Longevity

In large-scale Modern C++ systems, testability is not merely a means to verify correctness; it is a lens that reveals design quality. A testable system has clear boundaries, explicit dependencies, controllable nondeterminism, and observable outcomes. These same properties make the system maintainable, refactorable, and evolvable for decades. When testability is designed in, testing becomes a low-friction feedback loop that continuously improves architecture rather than a costly after-the-fact activity.

24.2 Designing Mockable Systems

Mockable systems are systems whose dependencies can be replaced in tests with controlled substitutes: fakes, stubs, spies, and mocks. Mockability is valuable, but it is also easy to misuse. In large Modern C++ codebases, the goal is not to "mock everything"; the goal is to design *clear seams* where nondeterminism, side effects, latency, or external ownership live, and to keep the core logic as simple value-based code that needs little or no mocking.
This section presents a disciplined approach to mockability: where to introduce abstractions, which forms of test doubles to use, how to avoid fragile interaction-based tests, and how to design C++ interfaces and components that remain testable and evolvable over years.

24.2.1 Mockability: What It Is and What It Is Not

Mockability Is About Controllability and Observation

A component is mockable when tests can:

- **control** dependency outcomes (success, failures, timeouts),
- **observe** interactions at boundaries (what was called, with what data),
- **simulate** rare or expensive conditions deterministically.

Mockability Is Not the Same as Good Design

Mockability can be achieved through excessive indirection and interface explosion, which harms maintainability. Good design often yields mockability naturally:

- pure logic in the core (no mocks),
- explicit dependencies at edges (few mocks),
- stable seams at external boundaries (controlled mocking).

24.2.2 The Test Double Spectrum (Practical Taxonomy)

Use the right substitute for the right purpose:

Stub

Provides canned answers; does not record interactions.

- best for controlling return values and failure modes.

Fake

A lightweight implementation (often in-memory) that behaves like the real thing.

- best for repository/storage boundaries and complex stateful behavior.

Spy

Records calls for later assertions.

- best when you want to verify an effect occurred without enforcing strict call ordering.

Mock

Asserts expected interactions (often with strict expectations).

- best for narrow boundary protocols where interaction is the contract.
- dangerous when used to overspecify implementation details.

Rule of thumb for long-lived systems:

- prefer **fakes** and **stubs** for stable tests,
- use **spies** for minimal interaction verification,
- use strict **mocks** only when interaction itself is the contract.

24.2.3 Where to Put Seams: Mock at the Edges, Not the Core

Mockability should appear where the real world enters your system:

- persistence (databases, files),
- network and external services,

- clocks and timers,
- randomness,
- OS integration and process execution,
- message brokers and queues.

Avoid introducing mocks deep inside pure domain logic. If you feel the need to do so, the design is likely mixing policy and side effects.

24.2.4 Design Technique 1: Functional Core, Imperative Shell

The most effective way to reduce mocking is to keep core logic pure and push I/O to a thin shell.

Pure Core

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

struct User {
  std::string id;
  bool active = true;
};

std::vector<std::string> active_user_ids(const std::vector<User>& users) {
  std::vector<std::string> out;
  out.reserve(users.size());
  for (const auto& u : users)
    if (u.active) out.push_back(u.id);
  return out;
}
```

This requires no mocks.

Imperative Shell

The shell gathers inputs (DB/network), calls pure logic, and performs side effects. Only the shell needs mocking in tests.

24.2.5 Design Technique 2: Explicit Dependencies (Constructor Injection)

Hidden dependencies (globals, singletons) are the primary enemy of mockability. Make dependencies explicit.

Boundary Interface

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

struct IUserRepository {
  virtual ~IUserRepository() = default;
  virtual std::optional<std::string> email_for(std::string_view user_id) const = 0;
};
```

Service Uses Interface

```
class UserService {
public:
  explicit UserService(const IUserRepository& repo) : repo_(repo) {}

  bool can_send_email(std::string_view user_id) const {
    auto e = repo_.email_for(user_id);
    return e.has_value() && !e->empty();
  }

private:
  const IUserRepository& repo_;
};
```

24.2.6 Design Technique 3: Prefer Fakes for Stateful Boundaries

For repositories and caches, a fake is often more stable than a strict mock because it models behavior, not call sequences.

In-Memory Fake Repository

```
#include <unordered_map>

class FakeUserRepository final : public IUserRepository {
public:
  void add(std::string id, std::string email) { users_[std::move(id)] = std::move(email); }

  std::optional<std::string> email_for(std::string_view user_id) const override {
    auto it = users_.find(std::string(user_id));
    if (it == users_.end()) return std::nullopt;
    return it->second;
  }

private:
  std::unordered_map<std::string, std::string> users_;
};
```

This yields high signal tests:

- no coupling to internal call order,
- easy to set up,
- deterministic and fast.

24.2.7 Design Technique 4: Spies for Minimal Interaction Verification

When you need to ensure “we sent a notification” but do not want strict ordering constraints, use a spy.

Spy Event Sink

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

struct Event { std::string name; };

struct IEventSink {
  virtual ~IEventSink() = default;
  virtual void emit(Event e) = 0;
};

class SpySink final : public IEventSink {
public:
  void emit(Event e) override { events.push_back(std::move(e)); }
  std::vector<Event> events;
};
```

Use in Production Code

```
class Worker {
public:
  explicit Worker(IEventSink& sink) : sink_(sink) {}

  void do_work(bool ok) {
    sink_.emit({ok ? "work.ok" : "work.fail"});
  }

private:
  IEventSink& sink_;
};
```

Tests assert on recorded events without enforcing internal call structure beyond what matters.

24.2.8 Design Technique 5: Control Time and Randomness

Clocks and RNGs should be dependencies. This avoids sleeping and nondeterministic test failures.

Clock Seam

```
#include <chrono>

struct IClock {
  using time_point = std::chrono::steady_clock::time_point;
  virtual ~IClock() = default;
  virtual time_point now() const noexcept = 0;
};

class FakeClock final : public IClock {
public:
  time_point now() const noexcept override { return t_; }
  void advance(std::chrono::milliseconds d) { t_ += d; }
private:
  time_point t_ = time_point{};
};
```

RNG Seam

```
#include <cstdint>

struct IRng {
  virtual ~IRng() = default;
  virtual std::uint32_t next_u32() noexcept = 0;
};

class SequenceRng final : public IRng {
```

```
public:
  explicit SequenceRng(std::initializer_list<std::uint32_t> xs) : xs_(xs), i_(0) {}
  std::uint32_t next_u32() noexcept override {
    if (xs_.empty()) return 0;
    auto v = xs_[i_ % xs_.size()];
    ++i_;
    return v;
  }
private:
  std::vector<std::uint32_t> xs_;
  std::size_t i_;
};
```

24.2.9 Avoiding the Mocking Trap: Fragile Interaction Tests

The Problem: Overspecifying Implementation

A fragile test asserts:

- exact sequence of calls,
- exact number of calls,
- internal helper interactions,

even though these are not part of the external contract.
Such tests fail during refactoring even when behavior remains correct, making maintainability worse.

Prefer Behavioral Assertions

Good tests assert:

- returned values and state changes,

- emitted events,
- boundary-level interactions that represent true contracts.

24.2.10 Mockable Interfaces: C++ Design Guidelines

Keep Interfaces Small

A mockable interface should represent one role. Large interfaces force huge mocks and create brittle tests.

Prefer Stable Value Types at Boundaries

Avoid exposing volatile third-party types. Use value types and small structs to keep mocks simple and stable.

Document Thread-Safety and Lifetime Rules

Mocks often run in tests with different timing than production. Make rules explicit:

- can the interface be called concurrently?
- who owns returned memory?
- can callbacks outlive the call?

Avoid Complex Global State

If an interface depends on global configuration, hide it behind an injectable config provider.

Config Provider Seam

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

struct IConfig {
  virtual ~IConfig() = default;
  virtual std::optional<std::string> get(std::string_view key) const = 0;
};
```

24.2.11 Mockability and Resource Management

Mockable systems must also be safe systems. In C++, design mockable boundaries without sacrificing RAII.

Use RAII for Real Resources, Test Doubles for Behavior

- production code owns resources via RAII wrappers,
- tests swap behavior by swapping adapters, not by bypassing ownership rules.

RAII-Friendly Boundary

```cpp
#include <memory>

struct IConnection {
  virtual ~IConnection() = default;
  virtual bool send(std::string_view bytes) = 0;
};

struct IConnectionFactory {
  virtual ~IConnectionFactory() = default;
  virtual std::unique_ptr<IConnection> connect() = 0;
};
```

This allows tests to supply fake connections while production uses real socket-backed implementations.

24.2.12 Contract Tests: A Scalable Alternative to Extensive Mocking

When multiple teams build components behind an interface, contract tests keep them aligned:

- define behavioral requirements for the interface,
- run the same test suite against all implementations (real and fake),
- prevent subtle divergence over time.

Contract Test Shape (Conceptual Skeleton)

```
// Pseudocode shape in C++ style.
// A real codebase would integrate with an existing test framework.

template <class Repo>
void user_repo_contract(Repo& repo) {
  repo.add("u1", "a@b.com");
  auto e = repo.email_for("u1");
  // assert(e.has_value() && *e == "a@b.com");
}
```

Contract tests reduce the need for strict mocks by ensuring all implementations behave consistently.

24.2.13 A Mockability Checklist for Large C++ Systems

Seams

- seams exist at I/O and nondeterministic boundaries,
- core logic remains mostly mock-free.

Interfaces

- small interfaces representing one role,
- stable value types in signatures,
- explicit ownership and thread-safety rules.

Test Doubles Strategy

- fakes for stateful boundaries,
- stubs for simple outcomes,
- spies for minimal interaction verification,
- strict mocks only when interaction is the contract.

Determinism

- injectable clocks and RNGs,
- no sleeping in tests,
- controlled shutdown for background work.

Maintainability

- tests assert behavior, not internal call order,
- refactoring does not require rewriting large test suites.

24.2.14 Closing Principle: Mockability Is a Consequence of Good Boundaries

A system becomes mockable when it has:

- a functional core with minimal side effects,
- explicit dependencies and controlled nondeterminism,
- stable boundary interfaces designed around roles,
- test doubles used sparingly and deliberately.

When mockability is designed correctly, tests become fast, reliable design feedback. When mockability is pursued through indiscriminate abstraction, the system becomes harder to understand and maintain. The durable path in Modern C++ is to place seams at the edges, keep the core clean, and treat mocks as tools for boundary control rather than as a default testing strategy.

24.3 Property-Based Testing

Property-Based Testing (PBT) verifies *general laws* of behavior across many generated inputs, rather than verifying a small set of hand-picked examples. In large-scale Modern C++ systems, PBT is a powerful design feedback tool: it forces you to state invariants precisely, exposes edge cases that example-based tests miss, and produces minimal counterexamples that accelerate debugging. Used correctly, PBT strengthens both correctness and design clarity.
This section explains how to apply property-based testing in Modern C++ with practical patterns that scale: defining properties, controlling randomness, shrinking failures, combining PBT with unit and integration testing, and using PBT to validate refactors and long-lived invariants.

24.3.1 Why Property-Based Testing Matters in Large C++ Systems

Example-based tests answer: “Does this work for these cases?” Property-based tests answer: “Does this always satisfy this law?”
PBT is most valuable when:

- the input space is large and full of edge cases (parsers, protocols, formatting),
- correctness depends on invariants (sorting, hashing, state machines),
- bugs appear only in unusual combinations,
- refactoring requires high confidence in preserved semantics.

In long-lived systems, properties become *executable specifications*. They survive refactors better than brittle tests tied to internal structure.

24.3.2 Core Concepts

Properties

A property is a statement that should hold for many (often all) inputs. Common shapes:

- **Invariant:** “This condition is always true.”
- **Round-trip law:** “Encode then decode yields the original.”
- **Idempotence:** “Applying the operation twice equals once.”
- **Monotonicity:** “Increasing input cannot decrease output under this metric.”
- **Equivalence:** “Two implementations produce the same result.”

Generators

Generators produce random (or systematic) test inputs. Good generators:

- cover typical cases and edge cases,
- can bias toward tricky values (0, 1, max, empty, duplicates),
- are composable for structured data (trees, JSON-like structures).

Shrinking

When a property fails, the framework tries to reduce the failing input to a smaller counterexample. Shrinking is what makes PBT practical at scale: it turns "failed for input of size 10000" into "failed for empty string" or "failed for vector {0,0}".

Reproducibility

A property failure must be reproducible:

- record and print the seed,
- store the minimal counterexample as a regression test.

24.3.3 Design Feedback: What PBT Teaches You

Property-based testing is a strong signal about design quality:

- If a component is hard to property-test, it often has hidden dependencies (time, randomness, I/O).
- If it produces flaky results, nondeterminism is leaking into the core.
- If properties are hard to state, invariants are unclear or responsibilities are mixed.

Therefore PBT naturally pushes designs toward:

- pure functions and value types,
- explicit invariants,
- separable side effects (functional core, imperative shell),
- deterministic and controlled randomness.

24.3.4 Property Shapes That Scale Well

1) Sorting Properties

Sorting is the classic demonstration because it has simple, strong properties:

- output is ordered,
- output is a permutation of input,
- sorting twice does not change result (idempotence).

```
#include <algorithm>
#include <vector>
#include <unordered_map>
#include <cstddef>

static bool is_sorted_non_decreasing(const std::vector<int>& v) {
  for (std::size_t i = 1; i < v.size(); ++i)
    if (v[i-1] > v[i]) return false;
  return true;
}

static std::unordered_map<int, std::size_t> multiset_count(const std::vector<int>& v) {
  std::unordered_map<int, std::size_t> m;
```

```
  for (int x : v) ++m[x];
  return m;
}

static void property_sorting_laws(const std::vector<int>& input) {
  auto v = input;
  std::sort(v.begin(), v.end());

  // Property 1: output is sorted
  // (In a test framework, use REQUIRE/ASSERT.)
  if (!is_sorted_non_decreasing(v)) throw "not sorted";

  // Property 2: permutation preserved
  if (multiset_count(v) != multiset_count(input)) throw "not a permutation";

  // Property 3: idempotence
  auto v2 = v;
  std::sort(v2.begin(), v2.end());
  if (v2 != v) throw "not idempotent";
}
```

2) Round-Trip Properties for Serialization

If you own both sides of serialization:

- encode then decode yields original (lossless round-trip),
- decode then encode yields canonical representation (normalization).

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

struct Token {
  std::string text;
```

```
  int kind = 0;
};

std::string encode(const Token& t);
Token decode(std::string_view s);

static void property_round_trip(const Token& t) {
  auto s = encode(t);
  auto t2 = decode(s);
  if (t2.text != t.text || t2.kind != t.kind) throw "round-trip failed";
}
```

3) Algebraic Laws for Data Structures

For custom containers or APIs, identify algebraic laws:

- associativity: `(a op b) op c == a op (b op c)`
- identity element: `a op e == a`
- commutativity (when expected): `a op b == b op a`

4) State Machine Properties

For workflows (orders, sessions, protocol handling), model allowed transitions and assert invariants across generated sequences of actions.

```
#include <vector>
#include <cstdint>

enum class State { idle, running, failed, done };

struct Machine {
  State s = State::idle;
```

```
  bool start() noexcept { if (s != State::idle) return false; s = State::running; return
  ↪  true; }
  void succeed() noexcept { if (s == State::running) s = State::done; }
  void fail() noexcept { if (s == State::running) s = State::failed; }
  bool terminal() const noexcept { return s == State::done || s == State::failed; }
};

enum class Action : std::uint8_t { start, succeed, fail };

static void apply(Machine& m, Action a) {
  switch (a) {
    case Action::start:   (void)m.start(); break;
    case Action::succeed: m.succeed(); break;
    case Action::fail:    m.fail(); break;
  }
}

static void property_state_machine_never_leaves_terminal(const std::vector<Action>&
↪  actions) {
  Machine m{};
  bool was_terminal = false;
  for (auto a : actions) {
    apply(m, a);
    if (was_terminal && !m.terminal()) throw "left terminal state";
    was_terminal = m.terminal();
  }
}
```

This style catches surprising bugs in transition logic and invalid state combinations.

5) Differential Testing (Reference Implementation)

When you have two implementations (old/new, optimized/simple), assert equivalence across many inputs. This is excellent for safe refactors.

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

std::string normalize_slow(std::string_view s);
std::string normalize_fast(std::string_view s);

static void property_differential(std::string_view s) {
  auto a = normalize_slow(s);
  auto b = normalize_fast(s);
  if (a != b) throw "implementations differ";
}
```

24.3.5 Generators: Getting Input Quality Right

Edge Cases Are Not Optional

A generator for integers should frequently include:

- 0, 1, -1,
- minimum/maximum values,
- repeated values,
- values near boundaries (powers of two, lengths 255/256, etc.).

Structured Generators

For structured data (JSON-like, ASTs), build composable generators:

- base cases (empty, minimal),
- recursive composition with depth limits,
- bias toward smaller structures for shrinking.

Size Control and Performance Budgets

PBT can generate huge cases if unchecked. Long-lived test suites must be fast and predictable:

- cap sizes (string length, vector length, depth),
- run many small tests by default,
- run larger "stress properties" in nightly or pre-release pipelines.

24.3.6 Shrinking and Counterexample Management

Always Capture Reproducibility Data

When a failure occurs, capture:

- seed,
- minimized failing input,
- environment configuration relevant to the test.

Promote Counterexamples to Regression Tests

A discovered counterexample should become:

- a deterministic unit test with the minimized input,
- a documented bug scenario in the issue tracker,
- a safeguard for future refactors.

24.3.7 Property-Based Testing and Mocking

PBT and mocking can coexist, but the design goal is to avoid heavy mocks:

- Prefer pure properties over interaction properties.
- Use fakes for dependencies (in-memory store, fake clock).
- If a property depends on I/O, property-test the pure core and use a small number of integration tests for the shell.

24.3.8 Property-Based Testing in C++: Integration Patterns

1) Property Harness Wrapper

Even without a dedicated PBT framework, you can implement a basic harness that:

- uses a deterministic RNG with a seed,
- generates N cases,
- stops and prints the seed on failure.

```
#include <random>
#include <vector>
#include <iostream>
#include <stdexcept>

static std::vector<int> gen_vector(std::mt19937& rng) {
  std::uniform_int_distribution<int> len_dist(0, 200);
  std::uniform_int_distribution<int> val_dist(-1000, 1000);

  int n = len_dist(rng);
  std::vector<int> v;
```

```
  v.reserve(static_cast<std::size_t>(n));
  for (int i = 0; i < n; ++i) v.push_back(val_dist(rng));
  return v;
}

template <class Property>
static void run_property(Property&& prop, std::uint32_t seed, int cases) {
  std::mt19937 rng(seed);
  for (int i = 0; i < cases; ++i) {
    try {
      prop(rng);
    } catch (...) {
      std::cerr << "Property failed with seed=" << seed << " at case=" << i << "\n";
      throw;
    }
  }
}

static void property_sort_wrapper(std::mt19937& rng) {
  auto v = gen_vector(rng);
  property_sorting_laws(v);
}
```

In production codebases, a dedicated PBT framework will also provide shrinking and richer generators, but the underlying structure remains the same.

2) Split Properties by Cost

- **Fast properties:** run per commit (hundreds to thousands of cases, small inputs).
- **Stress properties:** run nightly (more cases, larger inputs, heavier checks).

24.3.9 PBT as Refactoring and Design Feedback

PBT is exceptionally effective for:

- refactoring parsers, normalizers, and formatters,
- optimizing algorithms (differential testing),
- validating state machines and workflows,
- asserting invariants on containers and caches.

A recommended workflow:

- define properties for current behavior,
- run PBT to discover counterexamples,
- fix bugs and store counterexamples as regression tests,
- refactor/optimize,
- rerun PBT to validate that invariants still hold.

24.3.10 Common Failure Modes and How to Avoid Them

Writing Weak Properties

A weak property does not constrain behavior enough and can pass while the code is wrong. Prefer multi-constraint properties:

- invariants + permutation + idempotence (sorting),
- round-trip + canonicalization (serialization),
- equivalence to reference implementation (differential).

Ignoring Flakiness

Flaky property tests typically indicate hidden nondeterminism:

- time dependence,
- data race,
- non-deterministic iteration ordering exposed in outputs.

Treat flakiness as a design defect, not a test defect.

Unbounded Generators

If generators produce huge structures, CI becomes slow and failures are hard to shrink. Always control size and complexity.

24.3.11 A Property-Based Testing Checklist

Properties

- properties express invariants and laws, not specific examples,
- properties constrain behavior strongly (multiple checks),
- properties are stable across refactors (behavior-level).

Input Generation

- generators include edge cases deliberately,
- size and depth are bounded,
- structured generators are composable.

Reproducibility

- seed is printed and recorded,
- failing inputs are minimized (shrinking),
- counterexamples become deterministic regression tests.

Performance

- fast properties run per commit,
- stress properties run less frequently,
- properties avoid expensive global setup.

24.3.12 Closing Principle: Properties Are Executable Invariants

Property-based testing elevates tests from "examples" to "laws." In large-scale Modern C++ systems, these laws become a durable feedback loop: they expose design flaws, enforce invariants across refactors, and systematically discover edge cases that would otherwise escape into production. The best long-lived systems treat core invariants as properties, make nondeterminism controllable, and promote every discovered counterexample into a permanent regression safeguard.

24.4 Using Tests to Validate Architecture

In large-scale systems, architecture is not what is written in diagrams; it is what the codebase *allows* and what it *prevents*. Over time, architectural intent erodes through incremental changes: convenience imports, "temporary" shortcuts, cross-layer coupling, and ad-hoc bypasses of

established boundaries. The result is architectural drift: the system still compiles, but it no longer behaves like the architecture you believe you have.
The most effective countermeasure is to treat architecture as a set of *enforceable constraints* and to validate those constraints with tests. Architectural tests are not primarily about functional correctness; they are about preserving structural integrity, dependency direction, layering rules, boundary contracts, and operational invariants. In Modern C++, architecture validation must also include build topology and header discipline, because compile-time coupling is often the earliest and most expensive form of architectural decay.
This section shows how to use tests as a design feedback loop for architecture: what to validate, how to express constraints in code, how to build fast and reliable enforcement in CI, and how to integrate architectural validation with refactoring and long-term evolution.

24.4.1 Architecture as Constraints, Not Documentation

Architecture becomes real when it is expressed as constraints such as:

- allowed dependencies (who may depend on whom),
- forbidden dependencies (who must not depend on whom),
- layering rules (core is independent of frameworks and I/O),
- stability rules (public APIs are stable and versioned),
- boundary contracts (interaction and error semantics),
- build structure rules (targets, includes, compile options).

Architectural tests enforce those constraints continuously, turning architecture into something that can fail fast in CI rather than fail slowly in production.

24.4.2 Why Architectural Validation Must Be Automated

Manual review cannot scale in large systems because:

- the dependency graph is too large for any person to hold,
- drift happens through many small changes,
- incentives favor shipping features over preserving structure,
- subtle coupling often looks harmless in a single diff.

Automated architectural tests provide:

- fast feedback,
- consistent enforcement,
- measurable architecture health,
- reduced organizational reliance on hero reviewers.

24.4.3 What to Validate with Architectural Tests

1) Dependency Direction and Layering

Typical rules:

- domain/core may not depend on infrastructure (DB, HTTP, OS),
- adapters depend on the core,
- UI depends on application services, not domain internals,
- platform-specific code is isolated behind interfaces.

2) Boundary Integrity and "No Bypass" Rules

- no direct DB calls from domain logic,
- no network calls from pure computation modules,
- no direct access to internal headers from outside the module.

3) Build Topology and Header Discipline (C++ Specific)

- public headers must not include heavy third-party headers,
- forbidden includes from lower layers,
- include fan-out limits for key public headers,
- avoid cycles and large rebuild cascades.

4) Architectural Contracts via "Contract Tests"

For stable module boundaries:

- define a behavioral contract for an interface,
- run the same test suite against multiple implementations,
- ensure implementations remain substitutable over time.

5) Operational Architecture Invariants

Architecture includes operability:

- every request path emits required metrics/events,
- error codes are consistent at boundaries,

- shutdown/startup protocols are deterministic and testable,
- resource usage remains within defined budgets (memory, latency).

24.4.4 Architectural Testing Layers

A mature architecture validation strategy uses multiple test types:

Static Structural Tests (Fast, Always-On)

- dependency graph rules,
- forbidden includes,
- layering constraints,
- build target rules.

Contract Tests (Behavioral at Boundaries)

- interface substitutability,
- error semantics,
- concurrency expectations (where applicable).

Integration Architecture Tests (Representative Paths)

- run a slice through multiple modules,
- validate that orchestration uses proper adapters,
- validate configuration and wiring rules.

24.4.5 A Practical C++ Pattern: Architectural "Ports" Verified by Tests

If architecture says the core uses ports and adapters, you can encode that into interfaces and test that the core does not bypass the ports.

Core Defines a Port

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

struct IUserRepository {
  virtual ~IUserRepository() = default;
  virtual std::optional<std::string> email_for(std::string_view user_id) const = 0;
};
```

Core Uses the Port Only

```
class UserService {
public:
  explicit UserService(const IUserRepository& repo) : repo_(repo) {}

  bool can_send_email(std::string_view user_id) const {
    auto e = repo_.email_for(user_id);
    return e.has_value() && !e->empty();
  }

private:
  const IUserRepository& repo_;
};
```

Now architecture can be validated:

- the core layer compiles without linking any DB/network libraries,

- tests provide fakes for the port,
- dependency rules forbid including DB headers in core targets.

24.4.6 Contract Tests: Architecture Through Substitutability

Contract tests encode what it means to be a valid implementation of a boundary interface.

Define the Contract Once

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

template <class Repo>
static void user_repo_contract(Repo& repo) {
  // The contract: storing then reading yields the same value.
  repo.put("u1", "a@b.com");
  auto e = repo.get("u1");
  if (!e.has_value() || *e != "a@b.com") throw "contract violation";

  // The contract: missing keys return empty.
  auto missing = repo.get("missing");
  if (missing.has_value()) throw "contract violation: missing should be empty";
}
```

Run Against Multiple Implementations

```
#include <unordered_map>

struct InMemoryRepo {
  void put(std::string k, std::string v) { m[std::move(k)] = std::move(v); }
  std::optional<std::string> get(std::string_view k) const {
    auto it = m.find(std::string(k));
```

```
    if (it == m.end()) return std::nullopt;
    return it->second;
  }
  std::unordered_map<std::string, std::string> m;
};

struct DbRepo {
  void put(std::string k, std::string v);                // real DB adapter
  std::optional<std::string> get(std::string_view k) const;
};

static void run_contracts() {
  InMemoryRepo mem;
  user_repo_contract(mem);

  // In CI, you may run DB contracts in a separate job.
  // DbRepo db{...};
  // user_repo_contract(db);
}
```

This validates an architectural principle: adapters remain replaceable, and the core depends on stable contracts.

24.4.7 Architectural Tests for Dependency Rules (Practical Strategies)

C++ does not enforce layering by default. You must encode it in build structure and validate it.

Strategy 1: Targets as Architecture

Define build targets that match architectural layers:

- `core` target contains only domain logic,
- `adapters` target contains DB/network implementations,

- app target wires dependencies,
- ui target depends on app.

Architectural validation then becomes testable:

- core must compile without adapter headers,
- forbidden includes break the build early.

Strategy 2: Forbidden Include Tests

A lightweight approach is to scan includes for forbidden patterns. This is not perfect, but it is effective and fast for preventing obvious drift.

```
#include <string_view>
#include <vector>

// Illustrative rule representation.
struct Rule {
  std::string_view from_path_prefix;  // e.g. "core/"
  std::string_view forbidden_include; // e.g. "<pqxx/>", "<curl/curl.h>", "adapters/"
};

static bool violates(std::string_view file_path, std::string_view included, const Rule& r) {
  // Simplified checks. Real code would parse includes more robustly.
  auto starts_with = [](std::string_view s, std::string_view p) {
    return s.size() >= p.size() && s.substr(0, p.size()) == p;
  };
  return starts_with(file_path, r.from_path_prefix) && (included.find(r.forbidden_include)
  ↪  != std::string_view::npos);
}
```

This style of test is easy to maintain and provides immediate feedback, even if it is not as semantically rich as a full dependency graph tool.

Strategy 3: Dependency Graph Validation

For larger systems, build a dependency graph from:

- target dependencies,
- include relationships (header-to-header),
- link dependencies.

Then assert:

- no cycles among architectural layers,
- no forbidden edges,
- no direct dependencies that bypass intended intermediates.

The exact implementation varies across toolchains and build systems, but the principle remains the same: architecture is validated by automated graph rules.

24.4.8 Using Tests to Validate Architectural Quality Attributes

Architecture is more than structure; it is also about systemic qualities.

Performance Budgets as Architectural Tests

If architecture promises scalability, enforce budgets:

- latency ceilings for key operations,
- memory ceilings for caches and pipelines,
- throughput targets under representative loads.

Budget Check Skeleton

```
#include <chrono>
#include <stdexcept>

template <class F>
static void assert_under(std::chrono::milliseconds budget, F&& f) {
  auto start = std::chrono::steady_clock::now();
  f();
  auto end = std::chrono::steady_clock::now();
  if (end - start > budget) throw std::runtime_error("budget exceeded");
}
```

Run such tests with stable harnesses and controlled environments (often as separate CI jobs), and treat regressions as architectural drift.

Resilience and Shutdown Protocols

If the architecture includes service lifecycle guarantees, test them:

- startup succeeds with minimal config,
- shutdown completes within a deadline,
- resources are released deterministically.

24.4.9 Making Architectural Tests Maintainable

Architectural tests can become bureaucracy if designed poorly. Keep them:

- **fast:** most rules should run in seconds,
- **actionable:** failures must explain what rule was violated and where,
- **stable:** rules should reflect intentional architecture, not personal preferences,

- **few and strong:** focus on the constraints that prevent expensive drift.

Prefer "Guardrails" Over Exhaustive Policing

Architectural tests should prevent the most damaging forms of drift:

- forbidden dependencies into the core,
- new cycles across high-level modules,
- leakage of third-party headers into public APIs,
- bypass of established ports and adapters.

Evolve Rules Intentionally

When architecture evolves, update rules deliberately:

- change the architecture document and the tests together,
- migrate in phases with temporary allowances,
- remove temporary rules before they become permanent decay.

24.4.10 A Scalable Workflow: Architecture Validation as Continuous Feedback

A mature workflow:

- developers run fast architectural checks locally,
- CI enforces full architecture rules on every merge,
- contract tests run per implementation (fast fakes per commit, real adapters in scheduled jobs),

- architecture failures trigger refactoring tasks like functional bugs do.

24.4.11 Closing Principle: Architectural Integrity Is a Testable Asset

If the architecture matters, it must be enforced. In large Modern C++ systems, tests are the mechanism that turns architectural intent into a durable, evolvable reality. Structural rules prevent slow drift, contract tests preserve substitutability, and quality-attribute tests enforce performance and operability promises. Together, these tests form a continuous design feedback loop that keeps the system scalable and maintainable for decades.

Evolution, Refactoring, and Survival

25.1 Designing for Change

Long-lived systems do not survive by being "finished." They survive by changing continuously while preserving correctness, performance budgets, security guarantees, and operational stability. In large Modern C++ codebases, the primary design goal is not to minimize change, but to make change *safe, local, observable, and incremental*. A system that cannot change cheaply accumulates technical debt, freezes, and eventually collapses into high-risk rewrites. Designing for change is therefore a set of architectural commitments:

- boundaries that limit blast radius,
- contracts that allow compatibility over time,
- feedback loops (tests, observability) that reduce risk,
- refactoring strategies that avoid big-bang transitions.

This section provides a practical, Modern C++ oriented playbook for designing systems that evolve safely for decades.

25.1.1 The Economics of Change: Cost of Change as the Primary Metric

A scalable architecture reduces the *cost of change*:

- **Understanding cost:** how much you must learn before editing.
- **Modification cost:** how many files/modules must change.
- **Verification cost:** how quickly you can prove safety (tests, CI).
- **Deployment cost:** how safely you can ship and roll back.

If these costs grow over time, the system becomes fragile. Designing for change means actively keeping those costs bounded as scale increases.

25.1.2 Principle 1: Make Change Local (Boundaries and Ownership)

Boundaries Are the Unit of Evolution

Change becomes expensive when it crosses boundaries frequently. A well-chosen boundary:

- concentrates related changes within a module,
- hides volatile details behind stable interfaces,
- aligns with ownership (who maintains it),
- enforces dependency direction.

Stable Core, Replaceable Adapters

A durable structure for evolution:

- **Core:** domain rules and invariants (stable, minimal dependencies).
- **Ports:** interfaces the core needs.
- **Adapters:** volatile implementations (DB, HTTP, OS, third-party APIs).

Port in Core

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

struct IUserStore {
  virtual ~IUserStore() = default;
  virtual std::optional<std::string> email_for(std::string_view user_id) const = 0;
};
```

Core Logic Depends Only on the Port

```
class UserRules {
public:
  explicit UserRules(const IUserStore& store) : store_(store) {}

  bool can_notify(std::string_view user_id) const {
    auto email = store_.email_for(user_id);
    return email.has_value() && !email->empty();
  }

private:
  const IUserStore& store_;
};
```

The core can evolve with high confidence because it is isolated from volatile infrastructure.

25.1.3 Principle 2: Make Contracts Explicit (So You Can Evolve Safely)

Define Behavioral Contracts

A contract includes:

- valid inputs and invariants,

- error semantics (what errors mean, and when they occur),
- performance budgets (when relevant),
- thread-safety and lifetime rules.

A design that hides contracts forces callers to discover them by accidents and bugs, making change dangerous.

Prefer APIs That Are Hard to Misuse

Encode invariants in types where practical:

- strong types for IDs and units,
- validated construction,
- explicit ownership and lifetime.

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

class UserId {
public:
  static UserId parse(std::string_view s) {
    if (s.empty()) throw std::invalid_argument("UserId: empty");
    return UserId(std::string{s});
  }
  std::string_view view() const noexcept { return v_; }

private:
  explicit UserId(std::string v) : v_(std::move(v)) {}
  std::string v_;
};
```

When invariants are encoded, refactoring is safer because fewer invalid states can exist.

Standardize Error Semantics at Boundaries

To evolve without chaos, large systems must be consistent about error handling at boundaries:

- choose exceptions or result values per layer,
- convert only at seams,
- classify errors (validation, not found, conflict, transient, internal).

Boundary Result Type (Illustrative)

```
#include <string>
#include <variant>

struct Error {
  enum class Kind { validation, not_found, conflict, transient, internal };
  Kind kind = Kind::internal;
  std::string message;
};

template <class T>
using Result = std::variant<T, Error>;
```

25.1.4 Principle 3: Make Change Safe (Feedback Loops)

Tests as Design Feedback

Change is safe when tests are:

- fast enough to run continuously,
- deterministic (no time/scheduling flakiness),
- focused on boundary behavior,

- backed by regression tests for incidents.

Observability as Change Safety

In production, observability is the other half of the feedback loop:

- structured logs with stable event names,
- metrics for rates, errors, latency, saturation,
- trace correlation across components (where applicable).

The more observable the system, the less risky evolution becomes.

25.1.5 Principle 4: Prefer Incremental Evolution (Avoid Big-Bang Rewrites)

The Strangler Pattern

Wrap legacy behavior with a stable facade and gradually replace parts behind it.

```
#include <memory>

struct Engine {
  virtual ~Engine() = default;
  virtual int compute(int x) = 0;
};

struct LegacyEngine : Engine {
  int compute(int x) override { return legacy(x); }
  int legacy(int);
};

struct NewEngine : Engine {
  int compute(int x) override { return modern(x); }
```

```
  int modern(int);
};

class EngineFacade {
public:
  explicit EngineFacade(std::unique_ptr<Engine> e) : e_(std::move(e)) {}
  int compute(int x) { return e_->compute(x); }
private:
  std::unique_ptr<Engine> e_;
};
```

This design enables:

- parallel implementation,
- controlled rollout,
- easy rollback.

Branch by Abstraction

Instead of long-lived VCS branches, create abstractions in code:

- introduce a seam (interface),
- provide both old and new implementations,
- migrate callers gradually,
- remove old code when stable.

Feature Flags for Safe Deployment

Feature flags allow shipping incomplete work safely and controlling rollout.

```
#include <string_view>

struct Flags { bool new_path = false; };

int handle_legacy(std::string_view req);
int handle_new(std::string_view req);

int handle(std::string_view req, const Flags& f) {
  return f.new_path ? handle_new(req) : handle_legacy(req);
}
```

25.1.6 Principle 5: Evolve Data with Explicit Rules

Data outlives code. Change becomes dangerous when data evolution is ad-hoc.

Versioned Data Formats

Adopt explicit rules:

- every stored object has a version,
- new fields are optional with defaults,
- unknown fields are preserved when acting as a proxy,
- deletions are staged (stop writing, stop reading, remove).

Versioned Decode Skeleton

```
#include <cstdint>
#include <vector>
#include <stdexcept>

struct Blob {
```

```
  std::uint32_t version = 0;
  std::vector<std::byte> bytes;
};

struct Model {
  int a = 0;
  int b = 0; // introduced later
};

Model decode(const Blob& b) {
  switch (b.version) {
    case 1: {
      Model m;
      // parse v1 -> fills m.a, default m.b
      m.b = 0;
      return m;
    }
    case 2: {
      Model m;
      // parse v2 -> fills m.a and m.b
      return m;
    }
    default:
      throw std::runtime_error("unsupported version");
  }
}
```

Expand/Contract Migrations

A safe evolution model:

- **Expand:** add new representation while keeping old.
- **Backfill:** populate new fields gradually.

- **Contract:** remove old representation after confidence.

This enables rollback and reduces risk.

25.1.7 Principle 6: Reduce Coupling at Compile Time (C++ Specific)

Compile-time coupling is a major barrier to change in C++ because it inflates build times and increases cross-team friction.

Header Discipline

- keep public headers minimal and stable,
- use forward declarations where possible,
- move heavy includes to `.cpp`,
- isolate third-party headers behind adapters.

Forward Declaration Firebreak

```
#pragma once
#include <memory>

class HeavyDependency;

class Service {
public:
  Service();
  ~Service();
  void run();

private:
  std::unique_ptr<HeavyDependency> dep_; // include heavy header in .cpp
};
```

Avoid Exposing Volatile Types in Public APIs

If your public API exposes:

- third-party types,
- template-heavy containers with volatile compile options,
- layout-sensitive structs,

you make evolution expensive. Prefer stable value types and opaque handles at boundaries.

25.1.8 Refactoring as Survival: A Disciplined Workflow

Step 1: Characterize Current Behavior

Before refactoring legacy code, capture behavior with characterization tests (including quirks).

```
#include <cassert>

int legacy_rule(int);

static void characterize_legacy() {
  assert(legacy_rule(0) == 17);
  assert(legacy_rule(2) == 21);
}
```

Step 2: Create Seams

Introduce interfaces or adapters to isolate the portion you intend to replace.

Step 3: Replace Incrementally

Migrate one path at a time under feature flags or controlled routing.

Step 4: Remove Dead Code

Dead code and old paths are hidden debt. Removal is part of evolution, not optional cleanup.

25.1.9 Designing for Team-Scale Change

As teams grow, change must be parallelizable:

- modules as units of ownership and release,
- architectural guardrails in CI (forbidden dependencies, cycles),
- compatibility windows and deprecation policies,
- contract tests for shared interfaces.

25.1.10 Common Anti-Patterns That Kill Evolvability

Big Ball of Mud Coupling

Everything depends on everything. Change becomes a multi-team negotiation.

Global State and Service Locators

Hidden dependencies prevent local reasoning and make tests fragile.

No Explicit Contracts

Callers rely on incidental behavior, making refactors risky.

No Migration Strategy

Schemas and APIs mutate without compatibility, forcing synchronized releases.

25.1.11 A Practical Checklist: Designing for Change

Boundaries and Dependencies

- stable interfaces hide volatile internals,
- dependency direction is enforced,
- compile-time coupling is controlled (headers and targets).

Contracts

- invariants and error semantics are explicit,
- ownership and thread-safety rules are documented and encoded,
- performance budgets exist for critical paths.

Safety Nets

- fast deterministic tests,
- regression tests for incidents,
- observability for production validation.

Evolution Mechanics

- feature flags and incremental rollout,
- strangler/branch-by-abstraction refactoring,
- versioned data and expand/contract migrations.

25.1.12 Closing Principle: Evolvability Is the Highest Form of Scalability

A system that scales in traffic but cannot evolve is not truly scalable. Designing for change in Modern C++ requires intentional boundaries, explicit contracts, controlled coupling, strong feedback loops, and incremental refactoring strategies. When these elements are designed in, evolution becomes routine, and the system can survive and improve for decades without catastrophic rewrites.

25.2 Safe Refactoring Strategies

Refactoring is the disciplined improvement of internal structure without intentionally changing externally observable behavior. In long-lived Modern C++ systems, refactoring is not an occasional cleanup; it is the mechanism by which the system continues to evolve without collapsing under complexity. The challenge is safety: C++ systems often combine performance-critical logic, concurrency, manual resource boundaries, and heavy build-time coupling. Unsafe refactoring can introduce subtle correctness regressions, data races, undefined behavior, ABI breaks, and operational failures.
Safe refactoring therefore requires a repeatable strategy: *tight feedback loops, controlled change scope, explicit seams, and reversible steps*. This section presents a practical toolbox of strategies that allow teams to refactor continuously while minimizing risk.

25.2.1 What Makes Refactoring Safe

A refactoring is safe when:

- behavior is specified by tests at the right level (contracts and boundaries),
- changes are incremental and reversible,
- risk is reduced before edits by creating seams and isolating effects,

- nondeterminism is controlled (time, randomness, concurrency),
- observability detects regressions early in production,
- build and ABI constraints are respected.

25.2.2 The Golden Rule: Never Refactor Without a Safety Net

1) Characterization Tests for Legacy Code

If behavior is unclear, capture it before changing anything. These tests document current behavior, including quirks, so refactoring does not accidentally change semantics.

```
#include <cassert>

int legacy_rule(int);

static void characterize_legacy() {
  // These assertions encode existing behavior discovered from production or usage.
  assert(legacy_rule(0) == 17);
  assert(legacy_rule(1) == 17);
  assert(legacy_rule(2) == 21);
}
```

2) Boundary Tests Over Internal Tests

Internal tests that assert private call order are brittle. Safe refactoring requires tests that assert:

- returned values and observable effects,
- invariants,
- contract-level interactions at module boundaries.

3) Make Flaky Tests a Stop Sign

If tests are flaky, they cannot protect refactoring. Treat flakiness as a defect and fix it before large structural changes.

25.2.3 Strategy 1: Refactor in Small, Reversible Steps

The highest-leverage practice in large systems is to avoid large diffs.

- Prefer a sequence of tiny, reviewable commits.
- Keep each step either purely mechanical or behavior-preserving.
- Ensure each step compiles and tests pass.

A safe sequence often looks like:

- rename for clarity,
- extract function,
- introduce seam (interface),
- move code behind seam,
- replace implementation,
- delete old path.

25.2.4 Strategy 2: Create Seams Before Changing Internals

If code is entangled, do not refactor it directly. First, create a seam: a boundary where behavior can be swapped.

Introduce an Interface at a Boundary

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

struct IUserStore {
  virtual ~IUserStore() = default;
  virtual std::optional<std::string> email_for(std::string_view user_id) const = 0;
};
```

Wrap Legacy Implementation Behind the Interface

```
class LegacyUserStore final : public IUserStore {
public:
  std::optional<std::string> email_for(std::string_view user_id) const override {
    return legacy_get_email(user_id);
  }
private:
  std::optional<std::string> legacy_get_email(std::string_view user_id) const;
};
```

Now core logic can depend on `IUserStore` and be tested with a fake, while the legacy store can be refactored independently.

25.2.5 Strategy 3: Branch by Abstraction (Replace Internals Gradually)

Branching in version control for long periods is risky. Instead, branch inside the code:

- define a stable abstraction,
- provide old and new implementations,
- route gradually,

- remove old implementation once stable.

```
#include <memory>

struct Engine {
  virtual ~Engine() = default;
  virtual int compute(int x) = 0;
};

struct LegacyEngine : Engine {
  int compute(int x) override { return legacy(x); }
  int legacy(int);
};

struct NewEngine : Engine {
  int compute(int x) override { return modern(x); }
  int modern(int);
};

class EngineFacade {
public:
  explicit EngineFacade(std::unique_ptr<Engine> e) : e_(std::move(e)) {}
  int compute(int x) { return e_->compute(x); }
private:
  std::unique_ptr<Engine> e_;
};
```

Benefits:

- both paths are runnable,
- tests can compare outputs (differential testing),
- rollback is easy (swap implementation).

25.2.6 Strategy 4: Strangler Façade (Wrap and Replace)

When you have a monolith or a legacy subsystem:

- build a facade around it,
- route requests through the facade,
- replace internal pieces behind the facade incrementally.

Facade With Explicit Routing

```
#include <string_view>

struct Flags { bool new_path = false; };

int handle_legacy(std::string_view req);
int handle_new(std::string_view req);

int handle(std::string_view req, const Flags& f) {
  return f.new_path ? handle_new(req) : handle_legacy(req);
}
```

This allows gradual rollout and rollback while keeping the external contract stable.

25.2.7 Strategy 5: Differential Testing (Reference vs. Optimized)

When refactoring for performance or clarity, validate new behavior against a reference implementation across many inputs.

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

std::string normalize_reference(std::string_view s); // slower but clear
```

```
std::string normalize_new(std::string_view s);       // faster/refactored

static void differential_test(std::string_view s) {
  auto a = normalize_reference(s);
  auto b = normalize_new(s);
  if (a != b) throw "mismatch";
}
```

This is one of the safest methods for refactoring algorithms: the old code becomes the specification until you decide to change semantics deliberately.

25.2.8 Strategy 6: Functional Core, Imperative Shell (Move Side Effects Out)

Side effects (I/O, time, concurrency) are refactoring hazards. Move policy into pure functions and keep side effects at the edges.

Extract Pure Logic

```
#include <vector>

static int sum_positive(const std::vector<int>& xs) {
  int s = 0;
  for (int x : xs) if (x > 0) s += x;
  return s;
}
```

Keep I/O Outside

```
#include <iostream>

static void shell() {
  std::vector<int> xs{1, -2, 3};
  int result = sum_positive(xs); // easy to test
```

```
  std::cout << result << "\n";   // isolated side effect
}
```

This makes refactoring cheap because most changes happen in pure code with deterministic tests.

25.2.9 Strategy 7: Control Nondeterminism Before Refactoring

Inject Clocks and Randomness

Refactoring code that depends on wall-clock time or randomness is dangerous if tests sleep or rely on chance.

```
#include <chrono>

struct IClock {
  using time_point = std::chrono::steady_clock::time_point;
  virtual ~IClock() = default;
  virtual time_point now() const noexcept = 0;
};

class FakeClock final : public IClock {
public:
  time_point now() const noexcept override { return t_; }
  void advance(std::chrono::milliseconds d) { t_ += d; }
private:
  time_point t_ = time_point{};
};
```

After injecting control points, refactoring becomes deterministic and testable.

Refactor Concurrency by Introducing a Scheduler Seam

For concurrency-heavy code, create a seam where work scheduling can be controlled in tests.

```cpp
#include <functional>
#include <queue>

class StepExecutor {
public:
  void post(std::function<void()> f) { q_.push(std::move(f)); }

  bool step_one() {
    if (q_.empty()) return false;
    auto f = std::move(q_.front());
    q_.pop();
    f();
    return true;
  }

private:
  std::queue<std::function<void()>> q_;
};
```

This allows deterministic testing and safer refactoring of asynchronous flows.

25.2.10 Strategy 8: Preserve ABI and Public API Stability

For long-lived systems, refactoring must respect compatibility constraints.

Do Not Leak Volatile Types

Avoid exposing:

- third-party types,
- heavy templates,
- layout-sensitive structs,

in public APIs intended to be stable over time.

Use PImpl Selectively for ABI Stability

PImpl can isolate implementation changes from clients at the cost of indirection.

```
#pragma once
#include <memory>

class Widget {
public:
  Widget();
  ~Widget();
  Widget(Widget&&) noexcept;
  Widget& operator=(Widget&&) noexcept;

  Widget(const Widget&) = delete;
  Widget& operator=(const Widget&) = delete;

  void run();

private:
  struct Impl;
  std::unique_ptr<Impl> p_;
};
```

This makes refactoring internals safer when binary compatibility matters.

25.2.11 Strategy 9: Refactor Data and Schemas Safely (Expand/Contract)

Data often outlives code. Schema refactoring must be staged:

- **Expand:** add new fields while preserving old behavior.

- **Backfill:** populate new representation.
- **Switch:** read from new field, keep writing both temporarily.
- **Contract:** stop writing old field, remove after confidence.

This staged approach preserves rollback capability and minimizes risk.

25.2.12 Strategy 10: Reduce Compile-Time Coupling Before Large Refactors

In C++, build friction can make refactoring economically impossible. Before deep refactors:

- reduce header include fan-out,
- forward declare and move heavy includes to `.cpp`,
- isolate third-party dependencies behind adapters,
- split monolithic targets into smaller ones.

Forward Declaration Firebreak

```
#pragma once
#include <memory>

class HeavyDependency;

class Service {
public:
  Service();
  ~Service();
  void run();

private:
  std::unique_ptr<HeavyDependency> dep_;
};
```

This reduces rebuild cascades and increases refactoring velocity.

25.2.13 Refactoring Playbooks (Repeatable Procedures)

Playbook A: Refactoring a Legacy Function

- Add characterization tests.
- Extract pure sub-functions.
- Replace complex conditionals with named predicates.
- Introduce strong types for ambiguous parameters.
- Keep behavior identical; rerun tests.
- Add new tests for intended behavior changes (if any).

Playbook B: Replacing a Dependency

- Introduce port/interface and adapter for the current dependency.
- Add contract tests for the port.
- Implement the new adapter.
- Run contract tests against both adapters.
- Switch via configuration/feature flag.
- Remove old adapter after stable rollout.

Playbook C: Refactoring a Stateful Subsystem

- Identify state machine and make states explicit.
- Add tests for allowed transitions and invariants.
- Isolate side effects behind seams (clock, I/O, scheduler).
- Refactor transition logic in small steps.
- Add regression tests for discovered edge cases.

25.2.14 Common Anti-Patterns (Unsafe Refactoring)

Big-Bang Refactors

Large changes without intermediate checkpoints reduce debuggability and increase risk. Prefer incremental steps.

Refactoring Without Tests

Without a safety net, refactoring becomes guessing.

Interaction-Only Mocking

Overly strict mocking makes refactoring painful because tests encode internal call order. Prefer behavioral assertions and contract tests.

Changing Semantics Accidentally

Accidental behavior changes are the most expensive kind. If you intend to change behavior, do it explicitly with migration plans and release notes.

25.2.15 Closing Principle: Safe Refactoring Is a System

Safe refactoring is not a single technique; it is a system of practices:

- strong tests and observability as feedback loops,
- incremental and reversible steps,
- seams before replacement,
- differential and contract testing for high confidence,
- controlled nondeterminism (time, randomness, scheduling),
- respect for API/ABI and data migration constraints.

When these practices are institutionalized, refactoring becomes routine, and the codebase remains evolvable and survivable for decades.

25.3 API Deprecation Policies

In long-lived systems, APIs are promises. Over time, requirements change, designs improve, vulnerabilities are discovered, and performance constraints evolve. Yet production systems cannot freeze; they must evolve without breaking downstream users, internal teams, build pipelines, or deployed binaries. This is the engineering purpose of an API deprecation policy: to enable change while preserving stability, trust, and delivery velocity.
In Modern C++, deprecation must consider:

- source compatibility (code still compiles),
- behavioral compatibility (same semantics),
- binary compatibility (ABI stability),

- deployment compatibility (mixed-version systems),
- ecosystem compatibility (tooling, wrappers, bindings).

This section provides a practical deprecation policy framework suitable for large-scale C++ systems: classifications of API changes, timelines, annotations, compatibility windows, migration patterns, and enforcement mechanisms.

25.3.1 Goals of a Deprecation Policy

A well-designed policy optimizes for:

- **Safety:** avoid sudden breaking changes.
- **Predictability:** users know what will be removed and when.
- **Actionability:** clear migration paths exist.
- **Velocity:** old APIs do not block progress indefinitely.
- **Trust:** consumers learn they can upgrade without fear.

25.3.2 Change Taxonomy: What Kind of Change Is This?

Classify changes explicitly, because the policy depends on the risk category.

Non-Breaking (Compatible) Changes

- adding new API alongside old,
- adding overloads without ambiguity,
- extending behavior in backward-compatible ways,
- adding optional parameters via new overloads.

Source-Breaking Changes

- removing or renaming functions,
- changing signatures (types, constness, overload sets),
- moving symbols across namespaces/modules without compatibility shims.

Behavior-Breaking Changes

- changing semantics while keeping the same signature,
- modifying defaults or edge-case behavior,
- changing threading guarantees or ordering.

Binary-Breaking Changes (ABI)

In C++ libraries distributed in binary form, ABI breaks include:

- changing class layout (adding/reordering data members),
- changing virtual function order or adding virtuals,
- changing inline function bodies in headers used by clients (ODR/ABI implications),
- changing exception specifications in exported interfaces,
- changing visibility/export of symbols.

A deprecation policy must explicitly state which stability levels are promised: source-only, ABI-stable, or both.

25.3.3 The Deprecation Lifecycle (A Practical Model)

A robust lifecycle typically has stages:

Stage 0: Introduce Replacement

- new API exists and is documented,
- new API is better (safer, clearer, faster),
- migration guide exists before deprecation begins.

Stage 1: Deprecate Old API

- mark old API as deprecated,
- provide clear message and replacement hint,
- keep behavior stable and supported.

Stage 2: Enforce Migration (Warnings to Errors Internally)

- internal builds treat deprecated usage as errors,
- external consumers still see warnings by default,
- automated refactoring tools or scripts are provided where possible.

Stage 3: Remove After a Compatibility Window

- removal occurs on a major version boundary or a scheduled “breaking change” release,
- last supported version with the old API is clearly identified,
- long-term support strategy is clarified.

25.3.4 Compatibility Windows and Support Policies

A policy must define how long deprecated APIs remain:

- **time-based window:** e.g., 12–24 months,
- **release-based window:** e.g., 3–6 minor releases,
- **usage-based window:** removal after adoption crosses a threshold.

In large organizations, a practical approach is:

- minimum window for internal migrations (e.g., N releases),
- longer window for external or partner ecosystems,
- explicit exceptions for security-critical removals (accelerated process).

25.3.5 Communicating Deprecation: Make It Impossible to Miss

Deprecation Messages Must Be Actionable

The deprecation message should include:

- what to use instead,
- why (safety/performance/correctness),
- how to migrate (short hint; full guide elsewhere).

Use Compiler-Supported Deprecation

Modern C++ supports standardized deprecation attributes.

Using [[deprecated]] With a Message

```
#include <string_view>

// New API
int parse_int(std::string_view s);

// Old API
[[deprecated("Use parse_int(std::string_view). Old parseInt has ambiguous error behavior.")]]
int parseInt(const char* s);
```

This provides:

- immediate feedback at compile time,
- visibility across all call sites,
- low operational overhead.

Deprecate at the Right Level (API, Not Just Implementation)

Deprecating an internal helper does not help if a public interface still requires it. Deprecation must match the user-visible contract.

25.3.6 Migration Patterns That Preserve Stability

Pattern 1: Wrapper That Preserves Old Behavior

Keep old API but implement it in terms of the new API to avoid divergence.

```
#include <string_view>

int parse_int(std::string_view s);
```

```
[[deprecated("Use parse_int(std::string_view).")]]
int parseInt(const char* s) {
  return parse_int(s ? std::string_view{s} : std::string_view{});
}
```

This reduces maintenance burden:

- one code path to fix bugs,
- consistent behavior,
- safer evolution.

Pattern 2: Overload Bridge

Introduce safer overloads first, then deprecate unsafe overloads.

```
#include <string_view>
#include <span>

int write_bytes(std::span<const std::byte> bytes);

[[deprecated("Use write_bytes(std::span<const std::byte>) to avoid length mismatch.")]]
int write_bytes(const void* p, std::size_t n);
```

Pattern 3: Type-Safe Replacement (Strong Types)

If the old API is ambiguous, replace it with strong types.

```
#include <chrono>

struct Timeout {
  std::chrono::milliseconds value;
};
```

```
void connect_with_timeout(Timeout t);

[[deprecated("Use connect_with_timeout(Timeout{...}) to make units explicit.")]]
void connect_with_timeout_ms(int ms);
```

Pattern 4: Feature Flags for Behavioral Deprecations

If semantics must change, use a staged rollout:

- introduce a new API with new semantics,
- optionally provide a compatibility mode,
- migrate consumers,
- remove compatibility mode later.

Pattern 5: Expand/Contract for Data and Protocol Evolution

For APIs tied to data schemas or wire protocols:

- **expand:** support both formats,
- **migrate:** write new format while still reading old,
- **contract:** stop supporting old after window.

25.3.7 Deprecation Under ABI Constraints (C++ Library Reality)

When ABI stability matters, removal may be delayed or require special techniques.

Avoid ABI Breaks by Adding, Not Changing

Within an ABI-stable major line:

- avoid changing class layouts,
- avoid reordering virtual functions,
- add new APIs via free functions or non-virtual member functions when possible,
- prefer pImpl for types with evolving internals.

pImpl for ABI-Stable Evolution

```
#pragma once
#include <memory>

class Client {
public:
  Client();
  ~Client();

  Client(Client&&) noexcept;
  Client& operator=(Client&&) noexcept;

  Client(const Client&) = delete;
  Client& operator=(const Client&) = delete;

  void send_request(int code);


  ↪  [[deprecated("Use send_request(int). send(int) will be removed in the next major release.")]]
  void send(int code);
```

```
private:
  struct Impl;
  std::unique_ptr<Impl> p_;
};
```

Symbol Versioning and Compatibility Shims (When Applicable)

Some platforms support symbol versioning and compatibility layers; others do not. The policy must specify supported targets and what is guaranteed.

25.3.8 Enforcement: Make Deprecation Real

A deprecation policy fails if it is not enforced.

Treat Deprecation as a Build Signal

Common enforcement steps:

- warnings enabled by default,
- CI gates new deprecated usage (“no new debt”),
- allow a temporary suppression mechanism with tracking.

No-New-Uses Rule

An effective rule:

- existing deprecated uses may remain temporarily,
- new uses are forbidden by CI.

This prevents deprecation from expanding while the migration proceeds.

Local Suppression (Use Sparingly and Track)

```
#if defined(__clang__)
  #define DIAG_PUSH _Pragma("clang diagnostic push")
  #define DIAG_POP  _Pragma("clang diagnostic pop")
  #define DIAG_IGNORE_DEPRECATED _Pragma("clang diagnostic ignored
  ↪ \"-Wdeprecated-declarations\"")
#else
  #define DIAG_PUSH
  #define DIAG_POP
  #define DIAG_IGNORE_DEPRECATED
#endif

DIAG_PUSH
DIAG_IGNORE_DEPRECATED
void call_legacy(); // temporary, tracked usage
DIAG_POP
```

A policy should require that suppressions:

- are localized (small scopes),
- have an issue reference in commit/PR metadata (outside code),
- have an expiry plan.

25.3.9 Documentation Requirements (Without External Links Here)

Even without embedding links, the policy should require internal artifacts:

- a deprecation registry (API name, version deprecated, removal target),
- migration guide snippets and examples,
- compatibility guarantees (source/ABI/behavior),
- known exceptions (security, legal, platform removals).

25.3.10 Special Cases and Accelerated Deprecation

Security Vulnerabilities

If an API is insecure by design (e.g., unsafe buffer handling), the policy may allow:

- shortened window,
- automatic hardening (runtime checks),
- removal in a patch release if risk is severe.

Operational and Cost Reasons

APIs tied to expensive dependencies or fragile behavior may also require faster retirement. The policy should define who approves acceleration and how consumers are notified.

25.3.11 A Practical Deprecation Policy Template

A usable policy statement can be summarized as:

- **Replacement first:** no deprecation without a supported replacement.
- **Announce with a plan:** deprecated in release X, removed in release Y.
- **Compatibility window:** at least N releases (or M months), except for security.
- **Actionable diagnostics:** `[[deprecated("Use ...")]]`.
- **One behavior source:** old API forwards to new API when possible.
- **Enforcement:** no new uses; CI gates; tracked suppressions.
- **ABI rules:** explicitly define ABI promises and approved techniques (pImpl, shims).
- **Removal discipline:** remove dead code after migration; update tests and contracts.

25.3.12 Closing Principle: Deprecation Is a Contract With Your Future

In long-lived Modern C++ systems, deprecation is not a warning label; it is a structured evolution mechanism. A strong deprecation policy preserves trust, enables parallel development, and prevents architectural stagnation. When replacements are introduced first, compatibility windows are explicit, diagnostics are actionable, and enforcement is real, the system can evolve continuously without forcing rewrites or breaking its users.

25.4 Knowing When to Redesign

A long-lived system survives by evolving, not by being rebuilt repeatedly. Yet there are times when incremental refactoring is no longer enough: the current structure prevents essential change, and every new feature increases fragility. The engineering challenge is to distinguish between:

- **systems that need disciplined refactoring**, and
- **systems that need redesign of some boundaries, subsystems, or core assumptions**.

Redesign does not necessarily mean a full rewrite. In large-scale Modern C++ systems, the most successful redesigns are usually *bounded*: they redesign a critical subsystem, an architectural boundary, a data model, or a deployment topology while keeping the rest stable. The key is to make redesign a deliberate decision based on measurable signals, risk analysis, and a migration strategy.

This section provides practical criteria, diagnostics, and redesign strategies suitable for long-lived C++ codebases.

25.4.1 Definitions: Refactor vs. Redesign vs. Rewrite

Refactor

- improves internal structure without changing the externally observable contract,
- typically incremental and reversible,
- uses existing architecture as the container.

Redesign

- changes architectural structure or core abstractions,
- may change interfaces, data models, or deployment assumptions,
- must preserve continuity for users via compatibility and migration.

Rewrite

- rebuilds a system largely from scratch,
- highest risk, often loses production knowledge encoded in edge cases,
- should be the last option unless constraints make it unavoidable.

The goal is to choose the minimal level of disruption that achieves the needed evolution.

25.4.2 The Core Question: Is Incremental Change Still Economically Viable?

A practical test is the **cost-of-change curve**. If you can keep the cost of change bounded by refactoring and boundary adjustments, you should. You consider redesign when the cost-of-change grows faster than your ability to reduce it.
Indicators include:

- **lead time** from idea to production grows continuously,
- **defect rate** rises with each release,
- **on-call load** increases even when feature velocity decreases,
- **build and CI time** grows without control,
- **coordination cost** between teams dominates engineering time.

25.4.3 Hard Signals That Suggest Redesign Is Needed

1) Architectural Drift Has Become Structural Debt

When the intended architecture is no longer enforceable:

- layering rules are routinely bypassed,
- dependency cycles prevent modular changes,
- core logic depends on infrastructure,
- volatile third-party details leak into stable APIs.

If guardrails and refactoring cannot restore boundaries without major disruption, redesign a boundary.

2) The System Cannot Support New Requirements Without Contortions

If new requirements consistently require:

- cross-cutting edits across many modules,
- global conditionals and feature flags everywhere,

- duplicating logic because reuse is unsafe,
- unsafe performance hacks,

then the current abstractions do not match reality. Redesign means creating abstractions aligned with the true change patterns.

3) Invariants Are Implicit and Unstable

When invariants cannot be stated clearly, tested reliably, or enforced in types, the system becomes unmaintainable. A redesign may be needed to:

- clarify domain boundaries,
- introduce explicit state machines,
- define stable contracts and error semantics.

4) Data Model and Evolution Are Blocking Delivery

Data outlives code. Redesign is often required when:

- schema changes require synchronized deployments,
- migrations are unsafe or take too long,
- data ownership boundaries are unclear,
- the data model no longer matches the business reality.

5) Operational Failure Modes Are Inherent to the Design

If reliability problems arise from architectural structure:

- single points of failure,

- inability to isolate faults (failure cascades),
- unbounded resource usage (memory growth, queue explosions),
- shutdown/startup sequences that are fragile,

then refactoring code inside the same structure may not solve it. Redesign might require different boundaries, backpressure mechanisms, or isolation strategies.

6) Performance Bottlenecks Are Architectural, Not Local

When performance problems come from core structure:

- excessive copying due to API design,
- global locks due to shared mutable state,
- poor data layout due to object graph fragmentation,
- tight coupling that prevents parallelization,

then incremental micro-optimizations will not fix the root cause. Redesign may require data-oriented structures, ownership changes, or concurrency model changes.

25.4.4 Soft Signals (Warning Signs) That Often Precede Redesign

The System "Fights Back" Against Clean Code

- developers avoid touching certain modules,
- changes require experts and tribal knowledge,
- the safest strategy becomes "copy and modify",
- refactors are postponed because tests are weak.

Testing and Debugging Become Disproportionately Expensive

- tests require full system setup for small changes,
- flaky tests increase over time,
- bugs require production reproduction to diagnose,
- observability is inconsistent and unstructured.

This often signals that dependencies and side effects are not controlled by design.

Coordination Dominates Work

If every feature requires coordination across multiple teams, boundaries are not aligned with ownership. Redesign may mean reorganizing modules around capabilities and stable seams.

25.4.5 Decision Framework: Redesign or Refactor?

Step 1: Identify the Constraint That Blocks Evolution

Name the constraint precisely:

- “core depends on DB types”,
- “API contract cannot represent new states safely”,
- “schema migration requires downtime”,
- “global lock prevents scaling”,
- “build graph prevents modular compilation”.

Step 2: Attempt Minimal Intervention

Before redesign:

- enforce architectural rules in CI,
- introduce seams (ports/adapters),
- apply branch-by-abstraction and strangler patterns,
- reduce compile-time coupling.

If these fail to remove the constraint at reasonable cost, redesign is justified.

Step 3: Bound the Redesign

Prefer redesigning a subsystem or boundary rather than the whole system.

- redesign the module boundary,
- redesign a data model slice,
- redesign the concurrency model for one pipeline,
- redesign the API surface while preserving compatibility.

Step 4: Define a Migration Plan Before Implementing

A redesign without a migration plan is a rewrite in disguise.
Migration must include:

- compatibility window,
- incremental rollout path,
- rollback strategy,

- test plan (contract + regression),
- operational validation plan (metrics + logs).

25.4.6 Redesign Patterns That Avoid Full Rewrites

Pattern 1: Strangler Façade

Wrap legacy behind a stable facade and replace functionality behind it incrementally.

```
#include <string_view>

struct Flags { bool new_path = false; };

int legacy_handle(std::string_view req);
int redesigned_handle(std::string_view req);

int handle(std::string_view req, const Flags& f) {
  return f.new_path ? redesigned_handle(req) : legacy_handle(req);
}
```

Pattern 2: Branch by Abstraction

Create an interface, provide both implementations, migrate callers gradually, then remove the old one.

```
#include <memory>

struct Engine {
  virtual ~Engine() = default;
  virtual int compute(int x) = 0;
};
```

```
struct LegacyEngine : Engine { int compute(int x) override { return legacy(x); } int
↪  legacy(int); };
struct NewEngine    : Engine { int compute(int x) override { return modern(x); } int
↪  modern(int); };

class EngineFacade {
public:
  explicit EngineFacade(std::unique_ptr<Engine> e) : e_(std::move(e)) {}
  int compute(int x) { return e_->compute(x); }
private:
  std::unique_ptr<Engine> e_;
};
```

Pattern 3: Expand/Contract for Data Model Redesign

Safe data redesign:

- expand schema and support both representations,
- backfill,
- switch reads,
- contract old representation later.

Pattern 4: Contract Tests as Guardrails for Redesign

When redesigning behind an interface, define contract tests and run them against both implementations to ensure substitutability.

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

```
template <class Repo>
static void repo_contract(Repo& r) {
  r.put("k", "v");
  auto x = r.get("k");
  if (!x.has_value() || *x != "v") throw "contract violation";
}
```

25.4.7 C++-Specific Redesign Triggers

Compile-Time Coupling Has Become a Delivery Blocker

If build times and rebuild cascades dominate engineering time, redesign may be needed at the build/module level:

- split monolithic targets,
- move heavy includes out of public headers,
- introduce pImpl where ABI matters,
- isolate third-party headers behind adapters.

Ownership and Lifetime Model Is Incoherent

If the system contains pervasive shared ownership or ambiguous lifetimes leading to bugs and fear-driven code, redesign might focus on:

- clearer ownership boundaries,
- value semantics where possible,
- message passing instead of shared mutable state.

Concurrency Model Requires Redesign

Symptoms:

- global locks and contention,
- data races in production,
- shutdown issues due to background threads,
- inability to scale CPU utilization.

A redesign may adopt:

- actor/message passing boundaries,
- explicit task scheduling and backpressure,
- immutable data flow for pipelines.

25.4.8 Avoiding the Rewrite Trap

Redesign decisions fail when:

- they are motivated by aesthetics rather than constraints,
- they attempt to replace too much at once,
- they do not preserve compatibility and migration,
- they underestimate the knowledge embedded in legacy behavior.

A safe principle:

- treat existing behavior as a specification (capture with tests),
- redesign behind stable seams,
- migrate incrementally with measured rollout.

25.4.9 A Practical Redesign Readiness Checklist

Redesign is justified when most of these are true:

- critical business requirements cannot be met with incremental refactoring at reasonable cost,
- architecture guardrails cannot prevent ongoing drift,
- operational failure modes are structural,
- data evolution requires downtime or synchronized deploys,
- performance bottlenecks are architectural,
- a bounded redesign scope is possible,
- a migration plan with rollback exists,
- contract and regression tests can protect equivalence during migration.

25.4.10 Closing Principle: Redesign Is a Controlled Evolution, Not a Reset

Knowing when to redesign is about recognizing structural constraints that block safe evolution. In long-lived Modern C++ systems, the best redesigns are bounded, contract-driven, test-protected, and migration-first. They preserve the investment in existing behavior while changing the architecture where it matters most. This approach avoids the rewrite trap and enables the system to survive and improve for decades.

Part IX

Case Studies

Designing a Core Library

26.1 API Design Walkthrough

A core library is the most durable layer in a large-scale system. It becomes the foundation upon which many teams, services, and products build. Once adopted, its API surface evolves slowly, and its design decisions propagate into every dependent codebase. Designing a core library in Modern C++ therefore requires more than clean code: it requires explicit contracts, stability strategy, performance predictability, portability, and long-term maintainability.
This section is a practical walkthrough for designing a core library API. It does not focus on a specific domain; instead, it models a realistic "foundation" library that provides:

- error/result types,
- non-owning views and value types,
- a small logging/event interface,
- configuration access,
- extension points for adapters.

The goal is to show how to make design choices that scale across teams and survive many years of evolution.

26.1.1 Step 0: Define the Core Library's Mission and Non-Goals

Before writing code, define what the library is *for* and what it is *not* for.

Mission

A core library should:

- provide stable primitives used everywhere,
- encode consistent engineering policies (errors, contracts, ownership),
- reduce duplication and accidental coupling,
- be cheap to adopt and hard to misuse.

Non-Goals

A core library should avoid:

- application-specific workflows,
- heavy dependencies and wide transitive includes,
- complex dynamic configuration systems,
- feature creep ("it is convenient, so add it").

26.1.2 Step 1: Decide Stability Guarantees

Explicitly state what compatibility you promise:

- source compatibility within a major version,
- ABI stability (if distributed as binaries),

- behavior stability (semantics do not change silently),
- supported compilers and language levels.

For a core library, the default assumption should be: breaking changes are rare and must be justified.

26.1.3 Step 2: Establish API Design Principles

Principle A: Make Contracts Visible

Every public function should communicate:

- what inputs are valid,
- what errors can occur and how they are represented,
- ownership and lifetime rules,
- thread-safety expectations.

Principle B: Prefer Value Types and Non-Owning Views

Prefer:

- small value types for domain-neutral concepts,
- views (`std::string_view`, `std::span`) to avoid copies,
- explicit ownership types for lifetime-sensitive resources.

Principle C: Keep the Surface Small

The best API is often the smallest API that cannot be misused easily.

Principle D: Separate Policy From Mechanism

Core libraries should provide mechanisms and stable contracts, not application policy.

26.1.4 Step 3: Design Namespace and Header Structure

A core library must minimize compile-time coupling.

- prefer a small set of focused headers,
- avoid including heavy headers in public headers,
- isolate platform-specific code behind internal `.cpp` files or adapters.

Illustrative Header Layout

```
// core/result.hpp    - Result<T>, Error
// core/contract.hpp  - preconditions, assertions (policy-free)
// core/view.hpp      - aliases and small helpers for spans/views
// core/event.hpp     - IEventSink, Event
// core/config.hpp    - IConfig
```

26.1.5 Step 4: Design the Error and Result Model (Foundation of Contracts)

In large systems, inconsistent error handling is a major source of accidental complexity. A core library often starts by defining:

- an error taxonomy,
- a result type for boundary-safe propagation,
- conversion points between exceptions and result errors (if exceptions are used internally).

Error Type: Stable and Classifiable

```
#pragma once
#include <string>
#include <string_view>

namespace core {

struct Error {
  enum class Kind {
    validation,
    not_found,
    conflict,
    permission,
    transient,
    timeout,
    internal
  };

  Kind kind = Kind::internal;
  int code = 0;                 // stable numeric code for logs/telemetry
  std::string message;          // human-readable summary
  std::string detail;           // optional extended context

  static Error make(Kind k, int c, std::string msg) {
    Error e;
    e.kind = k;
    e.code = c;
    e.message = std::move(msg);
    return e;
  }
};
```

```
inline std::string_view kind_name(Error::Kind k) noexcept {
  switch (k) {
    case Error::Kind::validation: return "validation";
    case Error::Kind::not_found:  return "not_found";
    case Error::Kind::conflict:   return "conflict";
    case Error::Kind::permission: return "permission";
    case Error::Kind::transient:  return "transient";
    case Error::Kind::timeout:    return "timeout";
    default:                      return "internal";
  }
}

} // namespace core
```

Result Type: Use at Module Boundaries

```
#pragma once
#include <variant>
#include "core/result_error.hpp" // assumes Error in a separate header

namespace core {

template <class T>
class Result {
public:
  Result(T v) : v_(std::move(v)) {}
  Result(Error e) : v_(std::move(e)) {}

  bool ok() const noexcept { return std::holds_alternative<T>(v_); }

  const T& value() const { return std::get<T>(v_); }
  T& value() { return std::get<T>(v_); }
```

```
  const Error& error() const { return std::get<Error>(v_); }

private:
  std::variant<T, Error> v_;
};

} // namespace core
```

Design notes:

- Keep it small and stable.
- Avoid surprising implicit conversions.
- Provide `ok()`, `value()`, `error()` to make intent explicit.

26.1.6 Step 5: Define Non-Owning Views and Safe Parameter Conventions

Core libraries should prefer non-owning views in interfaces to reduce copies and allocation pressure.

String and Byte Views

```
#pragma once
#include <string_view>
#include <span>
#include <cstddef>

namespace core {

using str_view = std::string_view;
using bytes_view = std::span<const std::byte>;
using mut_bytes_view = std::span<std::byte>;
```

```
} // namespace core
```

API Convention Example

```
#include "core/view.hpp"
#include "core/result.hpp"

namespace core {

// Parses a positive integer from text.
// Contract: accepts ASCII digits only; rejects empty and overflow.
// Returns validation error on invalid input.
Result<int> parse_positive_int(str_view s) noexcept;

} // namespace core
```

The convention:

- use `str_view` and `bytes_view` for inputs,
- return `Result<T>` for recoverable boundary errors,
- keep exceptions for internal invariants only (if used at all).

26.1.7 Step 6: Design Extension Points (Events/Logging Without Coupling)

Core libraries should not embed a full logging framework, but they can define an event contract.

Event Sink Interface

```
#pragma once
#include <string>
#include <cstdint>
```

```
namespace core {

struct Event {
  std::string name;         // stable event name
  std::string message;      // summary
  std::uint64_t timestamp;  // epoch millis or monotonic mapping (policy defined elsewhere)
};

struct IEventSink {
  virtual ~IEventSink() = default;
  virtual void emit(Event e) = 0;
};

} // namespace core
```

Why This Is a Good Core API

- It is stable and minimal.
- It avoids transitive dependencies.
- It supports testing (tests can inject a spy sink).
- It supports multiple logging/metrics implementations outside the core.

26.1.8 Step 7: Configuration as a Stable Read-Only Interface

A core library can define configuration access as a port.

```
#pragma once
#include <string>
#include <optional>
#include <string_view>
```

```
namespace core {

struct IConfig {
  virtual ~IConfig() = default;
  virtual std::optional<std::string> get(std::string_view key) const = 0;
};

} // namespace core
```

This keeps policy outside the core (where config comes from, how it reloads, etc.) while enabling consistent consumption.

26.1.9 Step 8: Assemble a Case Study API: A Small Core "Client"

Now we combine the primitives into a realistic consumer API that illustrates design choices.

Public API Definition

```
#pragma once
#include "core/result.hpp"
#include "core/view.hpp"
#include "core/event.hpp"
#include "core/config.hpp"

namespace core {

struct ClientOptions {
  str_view endpoint;           // non-owning; must outlive call to create
  int connect_timeout_ms = 5000;
};

class Client {
```

```
public:
  // Factory: returns Result to avoid throwing across boundaries.
  static Result<Client> create(const IConfig& cfg, IEventSink& sink, ClientOptions opt)
  ↪  noexcept;

  // Core action: explicit input view, explicit output Result.
  Result<int> send_request(str_view payload) noexcept;

private:
  // Keep constructor private to enforce invariant creation.
  Client(const IConfig& cfg, IEventSink& sink, std::string endpoint, int timeout_ms)
    : cfg_(&cfg), sink_(&sink), endpoint_(std::move(endpoint)), timeout_ms_(timeout_ms) {}

  const IConfig* cfg_;
  IEventSink* sink_;
  std::string endpoint_; // stored owning copy to avoid dangling view
  int timeout_ms_ = 0;
};

} // namespace core
```

Design Rationale

- **Factory with `Result`:** creation can fail (config missing, invalid endpoint).
- **Ownership clarity:** input `endpoint` is a view; the object stores an owning copy.
- **Seams:** config and event sink are interfaces to keep the core library decoupled.
- **No hidden globals:** dependencies are explicit and testable.

26.1.10 Step 9: Provide a Testability Walkthrough (Design Feedback)

A good API is testable by construction.

Spy Sink

```
#include <vector>

namespace core {

class SpySink final : public IEventSink {
public:
  void emit(Event e) override { events.push_back(std::move(e)); }
  std::vector<Event> events;
};

} // namespace core
```

Fake Config

```
#include <unordered_map>

namespace core {

class FakeConfig final : public IConfig {
public:
  void set(std::string k, std::string v) { m_[std::move(k)] = std::move(v); }

  std::optional<std::string> get(std::string_view key) const override {
    auto it = m_.find(std::string(key));
    if (it == m_.end()) return std::nullopt;
    return it->second;
  }

private:
  std::unordered_map<std::string, std::string> m_;
};
```

```
} // namespace core
```

With these, tests can validate behavior without real environment setup, which is a primary success criterion for core libraries.

26.1.11 Step 10: Deprecation and Evolution Built In

Core libraries must evolve carefully. Design the API to allow:

- new overloads instead of signature changes,
- additive evolution of structs (with defaults),
- deprecation annotations with clear migration paths,
- stable contract tests for ports/interfaces.

Deprecation Example

```
namespace core {

[[deprecated("Use ClientOptions.connect_timeout_ms. This overload will be removed in the next major
Result<Client> create_client_legacy(const IConfig& cfg, IEventSink& sink, str_view
↪  endpoint) noexcept;

} // namespace core
```

26.1.12 Step 11: Performance and Allocation Considerations (Core Library Reality)

Core libraries often sit on hot paths. The API should:

- avoid implicit allocations in common calls,

- use views for inputs, owning copies only when required,
- document complexity expectations (e.g., O(n) in payload size),
- avoid exposing template-heavy types unnecessarily in public headers.

Avoiding Allocation Surprises

A common policy:

- accept `std::string_view` for input,
- return results by value when small and cheap,
- provide output parameters or spans only when necessary and clearly documented.

26.1.13 Step 12: Final Core Library API Checklist

Before shipping a core API, verify:

Contracts

- preconditions and invariants are explicit,
- error semantics are consistent and classifiable,
- thread-safety and lifetime rules are documented.

Surface and Coupling

- surface area is small and coherent,
- public headers are lightweight,
- third-party and platform specifics are hidden.

Testability

- interfaces exist at seams (config, events, external calls),
- fakes/spies can validate behavior without environment setup,
- properties and contract tests can protect refactoring.

Evolution

- additive changes are supported,
- deprecation path is defined,
- ABI strategy is clear where relevant.

26.1.14 Closing Principle: Core APIs Are Long-Term Infrastructure

A core library API is not just code; it is organizational infrastructure. Its most important qualities are stability, clarity of contract, and low coupling. By designing small, explicit, testable primitives (Result/Error, views, ports for events and config) and assembling them into coherent public APIs, you create a foundation that can be trusted and reused across many teams for many years—and that can evolve safely without breaking the systems built upon it.

26.2 Ownership Decisions

Ownership is the most fundamental design decision in a C++ core library. It determines who controls lifetime, who is responsible for cleanup, what can be shared safely, what can be moved cheaply, and what invariants clients are allowed to assume. In long-lived systems, ownership mistakes become systemic: they amplify complexity, cause memory leaks and use-after-free defects, force pervasive defensive programming, and make refactoring risky.

A core library must therefore establish an ownership model that is:

- **explicit** (no hidden lifetime assumptions),
- **mechanically enforceable** by types and RAII,
- **scalable** across teams and modules,
- **compatible** with performance and ABI constraints,
- **testable** and friendly to change.

This section is a practical walkthrough of ownership decisions you must make when designing a core library API, with Modern C++ patterns that scale.

26.2.1 1. The Ownership Spectrum in C++

Every API parameter and return value should sit clearly on one point of the spectrum:

Non-owning views (borrowed)

The caller owns; the callee borrows temporarily.

- examples: `std::string_view`, `std::span<T>`, raw pointers used as *observers*.
- best for: avoiding copies, accepting data without taking lifetime responsibility.
- risk: dangling if stored beyond the call.

Exclusive owning (unique ownership)

Exactly one owner exists; lifetime is unambiguous.

- examples: `std::unique_ptr<T>`, move-only resource wrappers.

- best for: resources with strict ownership (handles, connections, buffers).
- benefit: simple reasoning; easy refactoring.

Shared owning (shared ownership)

Multiple owners keep the object alive; destruction is delayed until the last owner releases.

- examples: `std::shared_ptr<T>`.
- best for: rare cases where lifetime is truly shared and cannot be expressed by topology.
- risk: cycles, hidden retention, unclear ownership boundaries, increased overhead.

Intrusive/arena ownership (region-based)

Objects are owned by an arena; individual lifetimes are bounded by the region.

- examples: arena allocators, monotonic buffers, pools.
- best for: high-throughput pipelines and batch lifetimes.
- risk: dangling references when objects escape the region.

A core library should define when each category is allowed and how it must be expressed.

26.2.2 2. Ownership Rules for Public APIs (Core Library Policy)

A durable, team-scalable policy often looks like:

Rule A: Prefer Value Semantics by Default

If a type can be copied/moved cheaply and has clear identity as a value, return and accept it by value.

- easy reasoning,
- strong exception safety,
- fewer lifetime bugs.

Rule B: Use Views for Inputs, Owning Types for Storage

Accept data by view, but store it as an owning copy if the object must outlive the call.

Correct: View in API, Own Internally

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

class Endpoint {
public:
  explicit Endpoint(std::string_view s) : value_(s) {} // owning copy
  std::string_view view() const noexcept { return value_; }

private:
  std::string value_;
};
```

This avoids dangling references while preserving low-cost call sites.

Rule C: Prefer `unique_ptr` for Transfer of Ownership

If an API transfers ownership, make it explicit.

Ownership Transfer In

```
#include <memory>

struct Task { virtual ~Task() = default; virtual void run() = 0; };

class Scheduler {
public:
  void submit(std::unique_ptr<Task> t) { queue_.push_back(std::move(t)); }

private:
  std::vector<std::unique_ptr<Task>> queue_;
};
```

Ownership Transfer Out

```
#include <memory>

class Factory {
public:
  std::unique_ptr<Task> make_task();
};
```

Rule D: Shared Ownership Is an Architectural Decision, Not a Convenience

A core library should restrict `shared_ptr` to well-justified cases:

- cross-thread asynchronous lifetimes where ownership is genuinely shared,
- plugin or callback graphs where the lifetime topology is not hierarchical,
- caches requiring shared access with independent lifetimes.

If `shared_ptr` is used, the API must clearly define:

- who is expected to hold ownership,
- whether cycles are possible and how they are prevented,
- whether `weak_ptr` is required for back-references.

26.2.3 3. Borrowing: Designing Safe Non-Owning Interfaces

Prefer References for Required Parameters

A reference communicates non-null and required dependence.

```
struct Logger {
  void info(std::string_view msg);
};

class Service {
public:
  explicit Service(Logger& log) : log_(log) {}

private:
  Logger& log_; // non-owning, required, must outlive Service
};
```

This is explicit but requires documenting lifetime: the logger must outlive `Service`.

Use Raw Pointers for Optional Observers

If a parameter is optional, a pointer communicates “may be null”.

```
struct Metrics {
  void count(std::string_view name);
};
```

```
class Worker {
public:
  explicit Worker(Metrics* m) : metrics_(m) {}

  void run() {
    if (metrics_) metrics_->count("worker.run");
  }

private:
  Metrics* metrics_ = nullptr; // optional observer
};
```

This is a clear, minimal convention that scales.

Never Store a Borrowed View Without an Ownership Guarantee

If you store a `string_view` or `span`, you are storing a pointer to memory you do not own. If the core library cannot guarantee lifetime, it must store an owning copy instead.

26.2.4 4. Ownership in Return Values

Return value design is a primary place where ownership bugs originate.

Return Values (Copy/Move) for Small Types

For small value types, returning by value is ideal.

```
struct Version { int major, minor, patch; };
Version current_version() noexcept { return {1, 2, 0}; }
```

Return `std::unique_ptr` for Polymorphic Ownership

For interface-based designs, return owning pointers explicitly.

```
#include <memory>

struct IConnection { virtual ~IConnection() = default; virtual bool send(std::string_view)
↪  = 0; };

std::unique_ptr<IConnection> connect();
```

Avoid Returning Borrowed Pointers Without Context

Returning raw pointers or references is safe only when the lifetime is tied to an owning object and documented clearly.

Borrowed Return With Explicit Anchor

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

class UserDb {
public:
  const std::string* find_email(std::string_view id) const {
    auto it = emails_.find(std::string(id));
    if (it == emails_.end()) return nullptr;
    return &it->second; // pointer valid as long as *this is alive and not mutated in ways
    ↪  that rehash
  }

private:
  std::unordered_map<std::string, std::string> emails_;
};
```

In core libraries, this pattern should be used cautiously because container mutation can invalidate pointers. Prefer returning owning copies or stable handles when required.

26.2.5 5. Lifetime, Concurrency, and Ownership

Ownership becomes harder under concurrency because tasks outlive call stacks.

Asynchronous Work Requires an Explicit Lifetime Strategy

If you start background work, you must define:

- who owns the worker thread/task,
- how shutdown is requested,
- how completion is observed,
- which objects must remain alive until completion.

Prefer Hierarchical Ownership for Concurrency

A robust pattern is: the parent object owns the executor and resources, and provides a clear `stop()/join()` protocol.

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

class Runner {
public:
  void start() {
    stop_.store(false);
    th_ = std::thread([this]{ loop(); });
  }

  void stop() {
    stop_.store(true);
    if (th_.joinable()) th_.join();
```

```
  }

  ~Runner() { stop(); }

private:
  void loop() {
    while (!stop_.load()) {
      // do work
    }
  }

  std::atomic<bool> stop_{false};
  std::thread th_;
};
```

This is a core-library friendly ownership model: RAII shutdown is explicit and deterministic.

Use `shared_ptr` Only When Topology Requires It

For detached asynchronous callbacks, `shared_ptr` can be justified, but it must be constrained.

Constrained Shared Ownership Example

```
#include <memory>
#include <functional>

class Session : public std::enable_shared_from_this<Session> {
public:
  using Ptr = std::shared_ptr<Session>;

  void async_op(std::function<void(int)> cb) {
    auto self = shared_from_this(); // keep alive during async op
    start_async([self, cb](int rc) {
```

```
      cb(rc);
    });
  }

private:
  template <class F>
  void start_async(F&& f); // calls f later
};
```

Even here, the policy must address:

- potential cycles,
- bounded lifetime and cancellation,
- memory retention risks.

26.2.6 6. Ownership and ABI: Designing Stable Public Types

If ABI stability matters, exposing complex ownership structures in public types can be dangerous.

Prefer Opaque Types for Evolving Internals

pImpl is a common technique for ABI-stable evolution of complex objects.

```
#pragma once
#include <memory>
#include <string_view>

class CoreClient {
public:
  CoreClient();
  ~CoreClient();
```

```
  CoreClient(CoreClient&&) noexcept;
  CoreClient& operator=(CoreClient&&) noexcept;

  CoreClient(const CoreClient&) = delete;
  CoreClient& operator=(const CoreClient&) = delete;

  bool send(std::string_view payload);

private:
  struct Impl;
  std::unique_ptr<Impl> p_;
};
```

The ownership decision:

- clients own a stable handle (`CoreClient`),
- internals can change without changing layout,
- refactoring does not break binary consumers.

26.2.7 7. Ownership and Error Handling

Ownership and error handling interact strongly.

Prefer Strong Exception Safety Where Practical

If an operation fails, objects should remain valid and invariants preserved. RAII and value semantics help.

Result Types at Boundaries Avoid Partial Construction Hazards

Factories returning `Result<T>` (or `std::optional<T>` when failure has no diagnostic) prevent partially initialized objects from escaping.

```
#include <variant>
#include <string>

struct Error { std::string message; };

template <class T>
using Result = std::variant<T, Error>;

class Resource {
public:
  static Result<Resource> create(std::string_view path) noexcept;
private:
  explicit Resource(int fd) : fd_(fd) {}
  int fd_ = -1;
};
```

26.2.8 8. Ownership Policy for Extensions and Plug-ins

Core libraries often host user-defined extensions. Ownership must be explicit at the boundary.

Option A: Library Owns Plugin (Unique Ownership)

```
#include <memory>

struct IPlugin { virtual ~IPlugin() = default; virtual void on_event(int) = 0; };

class Host {
public:
```

```
  void set_plugin(std::unique_ptr<IPlugin> p) { plugin_ = std::move(p); }

private:
  std::unique_ptr<IPlugin> plugin_;
};
```

Option B: User Owns Plugin (Borrowed)

```
class Host2 {
public:
  void set_plugin(IPlugin& p) { plugin_ = &p; }
private:
  IPlugin* plugin_ = nullptr; // must outlive Host2
};
```

Both can be valid, but the policy must choose intentionally based on lifecycle and loading model.

26.2.9 9. A Repeatable Ownership Decision Procedure

When designing any API element, answer these questions:

Q1: Who Creates the Object?

If the library creates it, it may be natural to return an owning type (`unique_ptr` or value).

Q2: Who Destroys It?

Destruction responsibility must be visible in the type system.

Q3: Must It Outlive the Call?

If yes, do not store borrowed views; store ownership or store a stable handle.

Q4: Can It Be Shared?

If true sharing is required, decide:

- `shared_ptr` with clear cycle prevention,
- or redesign to hierarchical ownership and message passing.

Q5: Is ABI Stability Required?

If yes, prefer pImpl and avoid exposing volatile layout decisions.

26.2.10 10. Ownership Checklist for a Core Library

- Non-owning parameters are expressed as references (required) or pointers (optional).
- Views (`string_view`/`span`) are accepted but not stored unless lifetime is guaranteed.
- Ownership transfers use `unique_ptr` (in/out).
- Shared ownership is rare, justified, and documented (with cycle strategy).
- Concurrency lifetimes are explicit and RAII-safe (`stop()`/`join()` or equivalent).
- Public API types avoid leaking volatile ownership details (especially for ABI stability).
- Error handling preserves invariants and prevents partially-formed objects from escaping.

26.2.11 Closing Principle: Ownership Is Architecture

In a core library, ownership decisions define the architecture of lifetime. Good ownership design produces code that is safe, testable, and refactorable. Poor ownership design produces invisible coupling through lifetime assumptions that no diagram can capture. The durable approach in Modern C++ is to prefer value semantics, make ownership transfers explicit with RAII and

`unique_ptr`, treat shared ownership as a constrained architectural decision, and enforce non-owning usage through clear conventions and stable contracts.

26.3 Error and Contract Strategy

A core library sets the behavioral tone of an entire codebase. If its error handling and contracts are inconsistent, every dependent module inherits ambiguity: callers do not know what can fail, how failures are reported, what is safe to recover from, and what is a programming bug. In long-lived systems, this ambiguity becomes systemic technical debt.
An effective core-library strategy must answer, precisely and consistently:

- What is a *contract* (precondition, postcondition, invariant)?
- What happens when a contract is violated?
- What is an *error* (recoverable failure) vs. a *bug* (programmer misuse)?
- How do errors propagate across module boundaries?
- How do we preserve diagnosability, performance, and ABI stability?

This section provides a modern, large-scale, C++-oriented approach that is stable under evolution: a taxonomy of failures, a unified error model, consistent contracts, and patterns for composing APIs safely.

26.3.1 1) Failure Taxonomy: Bug vs. Error vs. Fault

Before choosing mechanisms, define categories.

Programmer Bugs (Contract Violations)

Examples:

- passing null where non-null is required,
- violating preconditions (negative sizes, invalid enum values),
- violating invariants (object used after move, double-stop),
- internal logic errors (impossible states).

These are not expected in correct programs. The strategy is to fail fast in debug builds and keep invariants explicit.

Recoverable Errors (Expected Failures)

Examples:

- input validation failure,
- resource exhaustion (temporary),
- network timeout,
- missing configuration,
- not-found conditions.

These are expected in real systems and must be representable and composable.

Operational Faults (External/Environmental Failures)

Examples:

- transient infrastructure failures,
- partial outages,
- inconsistent remote states,
- degraded service behavior.

The core library typically should not embed retry policy, but it must classify these failures so higher layers can apply policies consistently.

26.3.2 2) Contracts: What the API Promises and Requires

A contract contains:

Preconditions

What must be true when calling.

- e.g., buffer size is non-zero, pointer is non-null, value is in range.

Postconditions

What is guaranteed on success.

- e.g., returned result is valid, output is normalized, resource is opened.

Invariants

What remains true during the lifetime of an object.

- e.g., `fd_` is valid or equals -1; state machines cannot leave terminal states.

A core library must make these contracts explicit and consistent across all APIs.

26.3.3 3) Strategy: Contracts Fail Fast; Errors Propagate

A scalable strategy:

- **Contract violations** are programmer bugs: detect early (assert/terminate) rather than encode as recoverable values.
- **Recoverable failures** are returned explicitly via a stable error propagation type.
- **Exceptions** (if allowed) are limited to enforcing invariants inside a module, and are not the default cross-boundary error channel.

This creates a clear mental model:

- If you violate the contract, the program is wrong.
- If the environment fails, you get a structured error you can handle.

26.3.4 4) A Unified Error Model for a Core Library

A core library should define a stable, minimal `Error` and `Result<T>` for boundary propagation.

Error Taxonomy: Stable and Actionable

```
#pragma once
#include <string>
#include <string_view>
#include <cstdint>

namespace core {

struct Error {
  enum class Kind : std::uint8_t {
    validation,   // caller input invalid
    not_found,    // missing resource
    conflict,     // state conflict (version, lock, uniqueness)
    permission,   // authz/authn failure (policy decided above core)
    timeout,      // time budget exceeded
    transient,    // retry may succeed
    internal      // unexpected failure (not contract violation)
  };

  Kind kind = Kind::internal;
  int code = 0;               // stable numeric code (for telemetry)
  std::string message;        // short human-readable summary
  std::string detail;         // optional context (avoid sensitive data)

  static Error make(Kind k, int c, std::string msg) {
    Error e;
    e.kind = k;
    e.code = c;
    e.message = std::move(msg);
    return e;
  }
};
```

```
inline std::string_view kind_name(Error::Kind k) noexcept {
  switch (k) {
    case Error::Kind::validation:  return "validation";
    case Error::Kind::not_found:   return "not_found";
    case Error::Kind::conflict:    return "conflict";
    case Error::Kind::permission:  return "permission";
    case Error::Kind::timeout:     return "timeout";
    case Error::Kind::transient:   return "transient";
    default:                       return "internal";
  }
}

} // namespace core
```

Design points:

- **Kind** is small and stable: it drives policy at higher layers.
- **code** enables metrics dashboards and alerts without parsing text.
- **message/detail** support diagnosability but must be curated to avoid leaking secrets.

Result Type: Small, Explicit, Composable

```
#pragma once
#include <variant>
#include <utility>
#include "core/error.hpp"

namespace core {

template <class T>
class Result {
```

```
public:
  Result(T v) : v_(std::move(v)) {}
  Result(Error e) : v_(std::move(e)) {}

  bool ok() const noexcept { return std::holds_alternative<T>(v_); }

  const T& value() const { return std::get<T>(v_); }
  T& value() { return std::get<T>(v_); }

  const Error& error() const { return std::get<Error>(v_); }

private:
  std::variant<T, Error> v_;
};

template <>
class Result<void> {
public:
  Result() : ok_(true) {}
  Result(Error e) : ok_(false), e_(std::move(e)) {}

  bool ok() const noexcept { return ok_; }
  const Error& error() const { return e_; }

private:
  bool ok_ = true;
  Error e_;
};

} // namespace core
```

26.3.5 5) Contract Enforcement: A Core `contract.hpp`

Contracts must be enforced consistently without tying the core library to heavyweight frameworks.

Minimal Contract Macros (Debug vs Release)

```
#pragma once
#include <cstdlib>
#include <cstdio>

namespace core::contract {

[[noreturn]] inline void fail(const char* expr, const char* file, int line) noexcept {
  std::fprintf(stderr, "Contract violation: %s (%s:%d)\n", expr, file, line);
  std::fflush(stderr);
  std::abort();
}

} // namespace core::contract

#ifndef CORE_EXPECT
  #if !defined(NDEBUG)
    #define CORE_EXPECT(expr) ((expr) ? (void)0 : ::core::contract::fail(#expr, __FILE__,
    ↪  __LINE__))
  #else
    // In release builds, you can still fail-fast for contract violations.
    // Alternatively, you can compile this out; policy must be explicit.
    #define CORE_EXPECT(expr) ((expr) ? (void)0 : ::core::contract::fail(#expr, __FILE__,
    ↪  __LINE__))
  #endif
#endif
```

Policy choice:

- Many core libraries keep `CORE_EXPECT` active in release for truly impossible states.
- For preconditions that depend on caller behavior, projects often use debug-only asserts plus defensive checks at boundaries.

Using Contracts in Code

```
#include "core/contract.hpp"
#include "core/result.hpp"
#include <string_view>
#include <limits>

namespace core {

Result<int> parse_positive_int(std::string_view s) noexcept {
  // Boundary validation: recoverable error.
  if (s.empty()) return Error::make(Error::Kind::validation, 1001, "empty input");

  int value = 0;
  for (char c : s) {
    if (c < '0' || c > '9')
      return Error::make(Error::Kind::validation, 1002, "non-digit");
    int digit = c - '0';

    // Defensive overflow check -> validation error.
    if (value > (std::numeric_limits<int>::max() - digit) / 10)
      return Error::make(Error::Kind::validation, 1003, "overflow");

    value = value * 10 + digit;
  }

  // Internal invariant: value must be > 0 if input isn't "0".
  // If the project defines "0" as invalid, enforce as validation:
```

```
  if (value <= 0) return Error::make(Error::Kind::validation, 1004, "non-positive");

  return value;
}

} // namespace core
```

Key idea:

- Use **validation errors** for user-controlled inputs.
- Use **contracts** for internal invariants and programmer misuse.

26.3.6 6) Policy: Exceptions vs Result Types in a Core Library

A core library must pick a clear boundary rule. A widely robust rule is:

Rule: Cross-Boundary APIs Do Not Throw

Public core APIs that are consumed broadly should:

- be `noexcept` where practical,
- return `Result<T>` (or `Result<void>`).

Exceptions, if used, are confined to:

- internal implementation code,
- truly unrecoverable invariants (terminate),
- local algorithms where exception-based unwinding simplifies cleanup (still hidden behind boundary conversion).

Boundary Conversion: Exceptions to Result

```
#include "core/result.hpp"
#include <exception>
#include <string>

namespace core {

template <class F>
auto to_result(F&& f) noexcept -> Result<decltype(f())> {
  using T = decltype(f());
  try {
    return Result<T>(f());
  } catch (const std::bad_alloc&) {
    return Error::make(Error::Kind::internal, 2001, "out of memory");
  } catch (const std::exception& e) {
    return Error::make(Error::Kind::internal, 2002, std::string("exception: ") + e.what());
  } catch (...) {
    return Error::make(Error::Kind::internal, 2003, "unknown exception");
  }
}

} // namespace core
```

This allows internal code freedom while preserving stable boundary behavior.

26.3.7 7) Composing Errors and Preserving Context

When errors propagate upward, callers need enough context to diagnose without leaking sensitive data.

Add Context at Boundaries

```
#include "core/result.hpp"
```

```
#include <utility>

namespace core {

inline Error with_context(Error e, std::string detail) {
  if (!detail.empty()) {
    if (!e.detail.empty()) e.detail += "; ";
    e.detail += std::move(detail);
  }
  return e;
}

template <class T>
inline Result<T> add_context(Result<T> r, std::string detail) {
  if (r.ok()) return r;
  return with_context(r.error(), std::move(detail));
}

} // namespace core
```

Usage pattern:

```
core::Result<int> parse_configured_timeout(const core::IConfig& cfg) noexcept {
  auto s = cfg.get("timeout_ms");
  if (!s.has_value())
    return core::Error::make(core::Error::Kind::not_found, 3001, "missing timeout_ms");

  auto r = core::parse_positive_int(*s);
  return core::add_context(std::move(r), "key=timeout_ms");
}
```

26.3.8 8) Contracts for Object Lifetimes and State Machines

Core libraries often provide types with states. Contracts must ensure invalid transitions are detected.

Explicit State and Invariants

```
#include "core/contract.hpp"

class Channel {
public:
  void open() {
    CORE_EXPECT(state_ == State::closed);
    state_ = State::open;
  }

  void close() noexcept {
    // closing twice might be allowed; policy must be explicit.
    state_ = State::closed;
  }

  void send(std::string_view msg) {
    CORE_EXPECT(state_ == State::open);
    // send...
  }

private:
  enum class State { closed, open };
  State state_ = State::closed;
};
```

This ensures:

- invalid usage fails fast in debug (and possibly release per policy),

- invariants are concentrated and explicit,
- refactoring does not silently relax contracts.

26.3.9 9) Error Strategy and Performance Predictability

A core library must avoid error handling mechanisms that produce hidden costs.

Avoid Implicit Allocations in Error Paths (When Possible)

If errors are frequent (validation-heavy APIs), consider:

- using small-string optimization friendly messages,
- storing only codes/kinds in hot paths and formatting later,
- keeping `detail` optional and added at higher layers.

Avoid Exceptions as Control Flow Across Hot Boundaries

Even if exceptions are enabled, the policy should avoid throwing as part of normal operations (timeouts, not-found, validation). Use result types for expected failures.

26.3.10 10) A Core-Library Contract and Error Checklist

Contracts

- every public API states preconditions and postconditions,
- invariants are centralized and enforced,
- contract violations are treated as bugs (fail fast).

Errors

- a stable taxonomy exists (validation, transient, timeout, not-found, conflict, internal),
- errors carry stable numeric codes for telemetry,
- error propagation is explicit and composable,
- context is added at boundaries without leaking sensitive data.

Boundaries

- cross-boundary APIs do not throw (or the policy is explicit and consistent),
- exception-to-result conversion exists where needed,
- thread-safety and lifetime rules are part of the contract.

26.3.11 Closing Principle: Contracts Define Correctness; Errors Enable Survival

In a long-lived Modern C++ system, contracts and errors serve different purposes. Contracts define what it means to use the API correctly and keep invariants strong. Errors represent the reality that environments fail and inputs are imperfect. A core library succeeds when it makes this distinction explicit, enforces contracts consistently, propagates errors predictably, and preserves diagnosability and performance across decades of evolution.

Designing a High-Performance Service

27.1 Concurrency Model Selection

A high-performance service is defined less by its algorithms than by its concurrency model. The concurrency model determines how work is scheduled, how state is shared or isolated, how backpressure is applied, how latency is bounded, and how failures and shutdown are handled. In Modern C++, the model also determines where undefined behavior risks appear (data races, lifetime issues), and how difficult the service is to test and evolve.
Selecting a concurrency model is therefore an architectural decision. It must be driven by measurable requirements:

- throughput target and request mix,
- tail-latency requirements (p95/p99/p999),
- memory constraints and load shedding behavior,
- I/O profile (network-bound vs CPU-bound),
- state profile (stateless vs stateful, shared vs sharded),
- deployment environment (cores, NUMA, container limits),
- operational requirements (graceful shutdown, observability).

This section presents a practical selection framework, common concurrency models, their trade-offs, and a Modern C++ implementation-oriented walkthrough suitable for long-lived high-performance services.

27.1.1 1) First Principles: What You Must Decide Explicitly

Before choosing a model, define these invariants:

Work Unit Definition

What is the unit of scheduling?

- a request,
- an event (socket readable, timer fired),
- a message in a queue,
- a batch of operations.

State Ownership

Where does mutable state live?

- per-request (ephemeral),
- per-connection,
- global shared state,
- sharded state (partitioned by key).

Backpressure Strategy

What happens under overload?

- queue grows (dangerous unless bounded),
- shed load (reject early),
- degrade features,
- apply admission control (token bucket, concurrency limits).

Latency Budget and Tail Behavior

Do you prioritize:

- maximum throughput,
- predictable latency,
- fairness and isolation?

Failure and Shutdown Semantics

How do you guarantee:

- cancellation of in-flight work,
- draining of queues,
- deterministic shutdown of threads,
- no background work after stop?

A concurrency model that cannot support your shutdown semantics will produce production incidents even if it is fast.

27.1.2 2) Model Landscape: Common Concurrency Models in Services

Model A: Thread-Per-Request (or Thread-Per-Connection)

Each request/connection is handled by a dedicated OS thread.

Strengths

- simple mental model,
- blocking I/O is straightforward,
- isolation per thread can reduce shared-state coupling.

Weaknesses

- poor scalability under high connection counts,
- high context-switch and stack memory overhead,
- tail latency can degrade under load,
- harder to bound resource usage.

Use when:

- concurrency level is modest,
- operations are long-running and mostly blocking,
- simplicity is more important than extreme scale.

Model B: Fixed Thread Pool + Work Queue

A bounded number of worker threads pull tasks from a queue.

Strengths

- predictable CPU usage and thread count,
- good for CPU-bound workloads,
- easier to tune for core count and NUMA.

Weaknesses

- queue can become a bottleneck or unbounded memory risk,
- shared queue increases contention at high scale,
- blocking tasks reduce effective parallelism unless separate pools exist.

Use when:

- the service has mostly CPU work per request,
- I/O is limited or offloaded,
- a simple bounded concurrency model is desired.

Model C: Event Loop (Reactor) + Non-Blocking I/O

One (or a few) event loops handle I/O readiness events; work is executed as callbacks.

Strengths

- handles massive I/O concurrency efficiently,
- minimal thread count,
- good latency when callbacks are short.

Weaknesses

- callback complexity (inversion of control),
- blocking work stalls the loop (must offload CPU work),
- state machines become complex without careful design.

Use when:

- workload is I/O-bound (network services, proxies, gateways),
- you need many concurrent connections,
- request handlers can be structured as async flows.

Model D: Proactor (Async I/O Completion)

Work is scheduled by completion notifications (async I/O completes, callback fires).

Strengths

- efficient for async I/O,
- can reduce readiness polling overhead depending on platform,
- naturally fits completion-based programming.

Weaknesses

- platform differences and complexity,
- careful lifetime management required for in-flight operations,
- cancellation semantics can be tricky.

Use when:

- async I/O APIs are available and desired,
- you can invest in strict lifetime/cancellation discipline.

Model E: Sharded State + N Event Loops (or N Worker Groups)

Partition state by key and route each request to a shard (often single-threaded per shard).

Strengths

- reduces lock contention by avoiding shared mutable state,
- improves cache locality,
- predictable latency by isolating hot keys (if sharding is balanced).

Weaknesses

- cross-shard operations are complex,
- requires routing and partitioning strategy,
- rebalancing and hot-shard issues must be addressed.

Use when:

- there is significant mutable state,
- lock contention would dominate,
- you can partition by user/session/key.

Model F: Actor Model / Message Passing

State is encapsulated in actors that process messages sequentially.

Strengths

- avoids data races by construction (single-threaded actor state),
- clear ownership and isolation,
- naturally supports backpressure via mailbox limits.

Weaknesses

- requires messaging infrastructure,
- can complicate synchronous APIs,
- debugging flows requires correlation and tracing.

Use when:

- the service is stateful,
- correctness and isolation are priorities,
- you can model workflows as message flows.

27.1.3 3) A Selection Framework (Decision Procedure)

Step 1: Classify Workload

- **CPU-bound:** heavy computation dominates time.
- **I/O-bound:** waiting on network/disk dominates time.
- **Mixed:** both significant; needs separation of concerns.

Step 2: Determine Concurrency Scale

- low: tens to hundreds of concurrent requests,
- medium: thousands,
- high: tens of thousands to millions (connections).

Step 3: Determine State Topology

- stateless handlers $\rightarrow$ easier: thread pool or event loop,
- shared mutable state $\rightarrow$ consider sharding/actors,
- per-connection state $\rightarrow$ event-driven or actor-like.

Step 4: Choose Backpressure Mechanisms

Whichever model you choose, you must bound:

- queues,
- in-flight operations,
- memory per connection/request.

Step 5: Validate With Tail Latency and Shutdown

A model that looks good in average throughput but fails p99 under load is not acceptable for many services. Also ensure deterministic shutdown is implementable.

27.1.4 4) A Practical Modern C++ Baseline: Hybrid Reactor + Worker Pool

A robust design for many high-performance services is a hybrid:

- **I/O layer:** event loops handle socket readiness and protocol framing.
- **CPU layer:** a worker pool processes expensive computation.
- **State:** either per-connection state in the event loop thread, or sharded state in worker groups.

Key Rule: Never Block the Event Loop

Blocking the event loop destroys latency and throughput. Offload CPU work.

A Minimal Thread Pool With Bounded Queue

```
#include <condition_variable>
#include <functional>
#include <mutex>
#include <queue>
#include <thread>
#include <vector>
#include <atomic>

class ThreadPool {
public:
  explicit ThreadPool(std::size_t n) : stop_(false) {
    workers_.reserve(n);
    for (std::size_t i = 0; i < n; ++i) {
      workers_.emplace_back([this]{ worker_loop(); });
    }
  }
```

```
  ~ThreadPool() { stop(); }

  // Returns false if queue is full (backpressure).
  bool try_submit(std::function<void()> f) {
    {
      std::lock_guard<std::mutex> lk(m_);
      if (stop_) return false;
      if (q_.size() >= max_q_) return false;
      q_.push(std::move(f));
    }
    cv_.notify_one();
    return true;
  }

  void stop() {
    {
      std::lock_guard<std::mutex> lk(m_);
      if (stop_) return;
      stop_ = true;
    }
    cv_.notify_all();
    for (auto& t : workers_) if (t.joinable()) t.join();
  }

  void set_max_queue(std::size_t n) { max_q_ = n; }

private:
  void worker_loop() {
    while (true) {
      std::function<void()> task;
      {
        std::unique_lock<std::mutex> lk(m_);
        cv_.wait(lk, [this]{ return stop_ || !q_.empty(); });
```

```
        if (stop_ && q_.empty()) return;
        task = std::move(q_.front());
        q_.pop();
      }
      task();
    }
  }

  std::mutex m_;
  std::condition_variable cv_;
  std::queue<std::function<void()>> q_;
  std::vector<std::thread> workers_;
  std::atomic<bool> stop_;
  std::size_t max_q_ = 10000; // tune; must be bounded
};
```

This illustrates the essential architectural property: a bounded queue provides backpressure rather than unlimited memory growth.

Event Loop Offload Pattern (Conceptual)

The I/O thread parses requests, then offloads CPU work to the pool.

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

struct Response { std::string payload; };

class Connection {
public:
  void send(Response r); // must be called in the I/O thread
};
```

```
class Dispatcher {
public:
  explicit Dispatcher(ThreadPool& pool) : pool_(pool) {}

  // Called on I/O thread:
  void on_request(Connection& c, std::string req) {
    // Capture stable data; do not capture references that may dangle.
    auto conn = &c; // valid only if Connection lifetime is guaranteed by I/O layer policy
    bool ok = pool_.try_submit([conn, req = std::move(req)]() mutable {
      Response r = handle_cpu_heavy(std::string_view(req));
      // In a real design, post the send back to the I/O thread.
      // This example shows the separation of concerns only.
      conn->send(std::move(r));
    });

    if (!ok) {
      // Backpressure: reject early or degrade.
      c.send(Response{"busy"});
    }
  }

private:
  static Response handle_cpu_heavy(std::string_view req);

  ThreadPool& pool_;
};
```

The architectural lesson:

- capture ownership safely (`req = std::move(req)`),
- avoid calling connection methods from worker threads unless the connection is thread-safe,
- post results back to the event loop for I/O operations.

27.1.5 5) Sharding as a Concurrency Model for Stateful Services

If the service maintains significant mutable state (caches, sessions, counters), sharding avoids lock contention.

Shard Routing by Key

```
#include <cstddef>
#include <string_view>
#include <functional>

static std::size_t shard_for(std::string_view key, std::size_t shards) {
  return std::hash<std::string_view>{}(key) % shards;
}
```

Single-Threaded Shards (No Locks Inside)

Each shard runs on one thread, processing messages sequentially.

```
#include <functional>
#include <queue>
#include <mutex>

class Mailbox {
public:
  void push(std::function<void()> f) {
    std::lock_guard<std::mutex> lk(m_);
    q_.push(std::move(f));
  }

  bool try_pop(std::function<void()>& out) {
    std::lock_guard<std::mutex> lk(m_);
    if (q_.empty()) return false;
```

```
    out = std::move(q_.front());
    q_.pop();
    return true;
  }

private:
  std::mutex m_;
  std::queue<std::function<void()>> q_;
};
```

The key architectural property: inside a shard, state can be mutated without locks because only that shard's thread executes shard logic.

27.1.6 6) Backpressure and Load Shedding: Non-Negotiable for Performance

No concurrency model is safe without overload behavior. Decide early:

- maximum queued requests,
- maximum in-flight async operations,
- per-connection memory bounds,
- admission control thresholds (reject early).

Common patterns:

- bounded queue in thread pools,
- per-connection request limits,
- token bucket rate limiting,
- circuit breakers for downstream dependencies.

27.1.7 7) Concurrency and Testability

A concurrency model is only sustainable if it is testable.

- isolate scheduling behind an interface,
- provide deterministic executors for tests,
- avoid time-based sleeps; inject clocks,
- keep most logic in pure functions.

Deterministic Step Executor for Tests

```
#include <functional>
#include <queue>

class StepExecutor {
public:
  void post(std::function<void()> f) { q_.push(std::move(f)); }
  bool step_one() {
    if (q_.empty()) return false;
    auto f = std::move(q_.front());
    q_.pop();
    f();
    return true;
  }
private:
  std::queue<std::function<void()>> q_;
};
```

Design your service so core scheduling can be swapped with `StepExecutor` in tests.

27.1.8 8) Common Mistakes in Concurrency Model Selection

Mistake A: Choosing a Model Without Backpressure

Unbounded queues are slow memory leaks under load.

Mistake B: Mixing Blocking Work Into Event Loops

Any blocking call in the loop becomes a tail-latency multiplier.

Mistake C: Using Shared Mutable State by Default

Locks everywhere create contention, unpredictable latency, and deadlocks. Prefer sharding or message passing for stateful services.

Mistake D: Undefined Lifetime Across Threads

Capturing references into asynchronous tasks without guaranteed lifetimes causes use-after-free bugs. Prefer ownership capture and explicit lifetimes.

Mistake E: No Deterministic Shutdown Protocol

Threads must have a clear stop signal, join behavior, and resource cleanup guarantees. "Detached threads" are a long-term operational hazard.

27.1.9 9) Selection Summary: Practical Recommendations

For many high-performance services, a safe default is:

- event-driven I/O (reactor/proactor) for connection handling,
- bounded worker pools for CPU-heavy work,

- sharded state where mutable shared state would otherwise require heavy locking,
- explicit backpressure at every boundary.

When the service is mostly CPU-bound and stateless:

- a tuned thread pool with bounded queues may suffice.

When the service is massively I/O-bound:

- event loops with strict “no blocking” discipline are essential.

When the service is strongly stateful:

- sharding/actors provide correctness and performance stability by avoiding shared mutable state.

27.1.10 Closing Principle: Concurrency Model Is Architecture

In a high-performance Modern C++ service, concurrency is not an implementation detail; it is the architecture that governs latency, throughput, safety, and evolvability. Selecting the right model requires explicit decisions about state ownership, scheduling, backpressure, and shutdown. When those decisions are made deliberately and encoded into the structure of the code (bounded queues, explicit lifetimes, isolated side effects), the service can scale and survive for years without becoming fragile or untestable.

27.2 Resource Management

A high-performance service does not fail first because of algorithms. It fails because of resource collapse: memory growth, file-descriptor exhaustion, thread explosion, queue overload, connection storms, allocator contention, cache thrashing, or unbounded work accumulation. In

long-lived production systems, resource management is therefore an architectural responsibility, not a low-level implementation detail.
Modern C++ gives powerful tools for resource safety (RAII, deterministic destruction, strong type systems), but performance services require more: *explicit budgets, bounded structures, predictable lifetimes, and overload behavior*. The goal is to make resource usage:

- **bounded** (no unbounded queues, no silent growth),
- **predictable** (stable latency under load),
- **recoverable** (graceful degradation, shedding),
- **observable** (metrics for every critical resource),
- **testable** (stress tests and fault injection).

This section walks through a practical resource management strategy for a Modern C++ high-performance service: budgets, RAII wrappers, pooling, backpressure, cancellation, and operational guardrails.

27.2.1 1) First-Class Resource Budgets

The most important design shift is to treat budgets as explicit API inputs, not hidden assumptions.

Resource Budget Categories

- **CPU:** worker threads, CPU time per request, batch sizes.
- **Memory:** per-request allocations, queue memory, caches, buffers.
- **Descriptors:** sockets, files, epoll/kqueue handles.

- **Concurrency:** in-flight requests, per-connection outstanding ops.
- **Queues:** maximum length, maximum bytes, maximum age.
- **Time:** deadlines, timeouts, cancellation semantics.

Make Budgets Concrete and Centralized

```
#pragma once
#include <cstddef>
#include <chrono>

struct ServiceBudget {
  std::size_t max_in_flight_requests = 20000;
  std::size_t max_queue_tasks        = 100000;
  std::size_t max_conn_count         = 50000;

  std::size_t max_request_bytes      = 1 * 1024 * 1024;
  std::size_t max_response_bytes     = 4 * 1024 * 1024;

  std::chrono::milliseconds request_deadline{1000};
  std::chrono::milliseconds io_timeout{200};
};
```

Budgets should be validated on startup and exported as metrics to ensure production matches intent.

27.2.2 2) RAII as the Foundation: Every Resource Has an Owner

C++ enables deterministic release; the design goal is to ensure every resource has one obvious owner.

File Descriptor Wrapper (Illustrative POSIX Example)

```
#include <unistd.h>

class UniqueFd {
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() { reset(); }

  int get() const noexcept { return fd_; }
  int release() noexcept { int t = fd_; fd_ = -1; return t; }
  void reset(int fd = -1) noexcept {
    if (fd_ != -1) ::close(fd_);
    fd_ = fd;
  }

private:
  int fd_ = -1;
```

```
};
```

This pattern prevents descriptor leaks and makes shutdown and refactoring safer.

Thread Ownership Must Be Deterministic

Detached threads are a long-term hazard. Always define stop/join ownership.

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

class OwnedThread {
public:
  template <class F>
  explicit OwnedThread(F&& f) : th_(std::forward<F>(f)) {}

  OwnedThread(OwnedThread&&) noexcept = default;
  OwnedThread& operator=(OwnedThread&&) noexcept = default;

  OwnedThread(const OwnedThread&) = delete;
  OwnedThread& operator=(const OwnedThread&) = delete;

  ~OwnedThread() {
    if (th_.joinable()) th_.join();
  }

  void join() { if (th_.joinable()) th_.join(); }

private:
  std::thread th_;
};
```

A service should own all long-lived threads and join them during shutdown.

27.2.3 3) Bounded Queues and Backpressure

In performance services, unbounded queues are memory leaks under load. Your design must define what happens when the service is overloaded.

Bounded Queue Contract

A bounded queue API should:

- return failure when full (so callers can shed load),
- optionally support time-bounded wait,
- expose metrics: size, drops, max observed.

```
#include <condition_variable>
#include <mutex>
#include <queue>
#include <optional>

template <class T>
class BoundedQueue {
public:
  explicit BoundedQueue(std::size_t cap) : cap_(cap) {}

  bool try_push(T v) {
    std::lock_guard<std::mutex> lk(m_);
    if (q_.size() >= cap_) return false;
    q_.push(std::move(v));
    cv_.notify_one();
    return true;
  }

  std::optional<T> pop() {
```

```
    std::unique_lock<std::mutex> lk(m_);
    cv_.wait(lk, [this]{ return !q_.empty() || stopped_; });
    if (q_.empty()) return std::nullopt;
    T v = std::move(q_.front());
    q_.pop();
    return v;
  }

  void stop() {
    std::lock_guard<std::mutex> lk(m_);
    stopped_ = true;
    cv_.notify_all();
  }

  std::size_t size() const {
    std::lock_guard<std::mutex> lk(m_);
    return q_.size();
  }

private:
  mutable std::mutex m_;
  std::condition_variable cv_;
  std::queue<T> q_;
  std::size_t cap_;
  bool stopped_ = false;
};
```

Overload Response Patterns

When `try_push` fails, the caller must apply policy:

- reject the request early (fast fail),
- degrade features (e.g., skip expensive operations),

- shed by priority (keep critical traffic),
- apply admission control at the edge.

27.2.4 4) Memory Management for High-Performance Services

Avoid Accidental Allocation in Hot Paths

The most reliable strategy is structural:

- use `std::string_view` and `std::span` for input views,
- pre-allocate buffers per connection or per worker,
- keep objects small and movable,
- avoid repeated dynamic allocation inside loops.

Per-Request Allocation Budget

If requests can trigger unbounded allocation, your service becomes vulnerable to memory DoS. Enforce:

- maximum request size,
- maximum parsed elements,
- maximum recursion depth,
- maximum output size.

Buffer Pooling and Reuse

A practical approach is to reuse fixed-size buffers for common payload sizes.

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

class BufferPool {
public:
  explicit BufferPool(std::size_t buf_size, std::size_t initial)
    : buf_size_(buf_size) {
    free_.reserve(initial);
    for (std::size_t i = 0; i < initial; ++i)
      free_.push_back(std::vector<std::byte>(buf_size_));
  }

  std::vector<std::byte> acquire() {
    std::lock_guard<std::mutex> lk(m_);
    if (free_.empty()) return std::vector<std::byte>(buf_size_);
    auto b = std::move(free_.back());
    free_.pop_back();
    return b;
  }

  void release(std::vector<std::byte> b) {
    if (b.size() != buf_size_) return; // policy: drop unexpected sizes
    std::lock_guard<std::mutex> lk(m_);
    free_.push_back(std::move(b));
  }

private:
  std::mutex m_;
  std::vector<std::vector<std::byte>> free_;
```

```
  std::size_t buf_size_;
};
```

Guideline:

- pooling helps when allocation cost and fragmentation matter,
- it must be bounded (pool size must not grow unbounded),
- it must be observable (pool hits/misses).

Be Careful With Object Pools

Object pools can harm performance if they cause:

- contention on the pool lock,
- false sharing,
- cache-unfriendly reuse patterns.

Prefer pooling where it is measurable and justified.

27.2.5 5) Connection and Descriptor Management

High-performance services often fail by exhausting file descriptors.

Explicit Connection Limits

- cap maximum accepted connections,
- cap per-connection outstanding requests,
- cap per-connection buffer memory.

Slowloris and Idle Connection Protection

You must enforce:

- read timeouts for headers and bodies,
- minimum progress (bytes per time window),
- maximum idle time per connection.

These are not optional in production internet-facing services.

27.2.6 6) Cancellation, Deadlines, and Time Budgets

Without deadlines, resource usage becomes unbounded because slow dependencies hold onto memory and concurrency slots.

Deadline Propagation

Represent time budgets explicitly and pass them through calls.

```
#include <chrono>

struct Deadline {
  std::chrono::steady_clock::time_point t;
  bool expired(std::chrono::steady_clock::time_point now) const noexcept { return now >= t;
  ↪ }
};
```

Cancellation in Work Queues

A practical policy:

- if a request times out, its queued work must be skipped,

- in-flight work must observe cancellation points,
- cancellation should be cooperative, not forced termination.

```
#include <atomic>

class CancelToken {
public:
  void cancel() noexcept { cancelled_.store(true); }
  bool cancelled() const noexcept { return cancelled_.load(); }
private:
  std::atomic<bool> cancelled_{false};
};
```

27.2.7 7) Thread and CPU Resource Management

Thread Count Must Be Bounded and Purpose-Driven

Separate pools by workload type:

- I/O threads (event loops),
- CPU worker threads,
- blocking dependency pool (if unavoidable),
- background maintenance (low priority).

Mixing blocking tasks into CPU pools causes tail latency spikes.

Avoid Oversubscription

If you spawn too many runnable threads:

- context switching increases,

- cache locality degrades,
- tail latency increases under load.

27.2.8 8) Resource Observability (If You Cannot Measure It, You Cannot Control It)

A service should export at minimum:

- current and peak in-flight requests,
- queue sizes and drop counts,
- connection count and accept/reject rates,
- memory usage and allocator stats (where possible),
- latency percentiles,
- timeouts and retries (if higher layers retry),
- file descriptor usage (open handles).

Resource metrics are part of the design contract. They turn resource collapse into detectable trends rather than surprise outages.

27.2.9 9) Shutdown and Draining: Resources Must Be Reclaimed Predictably

Graceful shutdown in performance services is mostly resource management:

- stop accepting new connections,
- stop admitting new requests,

- drain or cancel queued work,
- wait for in-flight work to complete up to a deadline,
- close descriptors and join threads deterministically.

Stop Sequence Skeleton

```
class Service {
public:
  void stop() {
    accepting_ = false;       // stop new work at edge
    queue_.stop();            // unblock workers
    cancel_all_in_flight();   // cooperative cancellation
    pool_.stop();             // join worker threads
    close_all_connections();  // release descriptors
  }

private:
  void cancel_all_in_flight();
  void close_all_connections();

  bool accepting_ = true;
  BoundedQueue<int> queue_{1000};
  ThreadPool pool_{8};
};
```

27.2.10 10) Common Resource Management Failure Modes

Unbounded Queues

Hidden memory growth and catastrophic tail latency.

Missing Timeouts

Requests accumulate and exhaust concurrency slots.

Detached Threads

Resource leaks and shutdown hangs.

Shared Ownership Everywhere

Hidden retention and unpredictable memory release.

Implicit Allocation in Hot Paths

Latency spikes due to allocator contention and fragmentation.

No Per-Connection Limits

Single slow client can consume disproportionate resources.

27.2.11 Resource Management Checklist for a High-Performance C++ Service

- Every critical resource has an explicit budget and metric.
- Every long-lived resource is owned by an RAII type.
- All queues are bounded; overload behavior is defined.
- Every request has a deadline; timeouts are enforced.
- Concurrency is bounded (in-flight caps); backpressure exists.
- Connections have per-connection memory and outstanding-work limits.

- Thread counts are fixed and purposeful; no detached threads.
- Shutdown drains and joins deterministically within a time budget.

27.2.12 Closing Principle: Performance Is Resource Discipline

High performance in Modern C++ is not primarily about micro-optimizations; it is about disciplined, explicit resource management. When budgets, ownership, backpressure, and deadlines are designed into the architecture, the service remains fast under load, predictable in tail latency, robust under failures, and maintainable for years of evolution.

27.3 Performance Tradeoffs

Performance engineering in a high-performance service is the art of choosing the right tradeoffs under real constraints. A system can usually be made faster in one dimension by paying in another: throughput vs. tail latency, memory vs. CPU, determinism vs. peak speed, simplicity vs. specialization, abstraction vs. locality. In long-lived Modern C++ services, the objective is not "fast on a microbenchmark," but *stable performance under production load with predictable cost of evolution*.

This section organizes performance tradeoffs that matter most in Modern C++ services into concrete decision points. Each tradeoff is presented as:

- what you gain,
- what you pay,
- how to decide safely,
- how to encode the decision in design.

27.3.1 1) Throughput vs. Tail Latency

The Core Tension

Optimizing for maximum throughput often increases queueing and batching, which can harm p99/p999 latency. Optimizing for tail latency often requires tighter budgets, smaller queues, and earlier load shedding, which can reduce peak throughput.

Design Lever: Bounded Queues and Admission Control

- Large queues increase throughput under bursts but amplify tail latency (queueing delay).
- Small queues enforce stability but force early rejection under load.

Bounded Queue Admission Control (Illustrative)

```
bool admit_request(std::size_t in_flight, std::size_t max_in_flight) noexcept {
  return in_flight < max_in_flight; // stable latency, controlled throughput
}
```

Design Lever: Batching

Batching amortizes overhead (syscalls, locks, cache misses) but adds waiting time to form batches.

- Use batching for throughput paths (logging, replication, bulk encoding).
- Avoid batching on latency-critical request completion paths unless bounded by strict deadlines.

27.3.2 2) CPU vs. Memory

Caching vs. Recomputing

- caching reduces CPU but increases memory and invalidation complexity,
- recomputing reduces memory but increases CPU cost and may impact latency.

The correct choice depends on:

- data reuse frequency,
- cache hit ratio under load,
- memory limits and GC-like retention risks (in C++: shared ownership, long-lived containers),
- invalidation complexity and correctness.

Compression vs. Decompression

- compressing reduces bandwidth/memory but consumes CPU and can increase tail latency,
- leaving data uncompressed reduces CPU but increases memory pressure and network time.

Use compression where:

- network is the bottleneck,
- data is large and repetitive,
- CPU headroom exists.

Allocation Strategy: Fewer Allocations vs. Larger Buffers

- fewer allocations (pooling, reuse) reduces allocator contention,
- larger buffers reduce fragmentation of many small allocations but may waste memory.

27.3.3 3) Abstraction vs. Predictability

Virtual Dispatch vs. Static Polymorphism

- virtual dispatch improves modularity and reduces compile-time coupling in some designs,
- static polymorphism (templates) can inline and optimize away indirection.

Tradeoffs:

- templates increase build time and binary size,
- virtual interfaces can stabilize ABI but may cost branch mispredicts and inhibit inlining.

Virtual Interface (Stable Boundary)

```
struct ICodec {
  virtual ~ICodec() = default;
  virtual void encode(std::span<const std::byte> in, std::vector<std::byte>& out) = 0;
};
```

Template Strategy (Max Inlining)

```
template <class Codec>
void process(Codec& c, std::span<const std::byte> in) {
  // can inline encode, specialized per Codec
  std::vector<std::byte> out;
  c.encode(in, out);
}
```

Generic Containers vs. Data-Oriented Layout

- object graphs of heap allocations are flexible but cache-unfriendly,
- contiguous arrays (`std::vector` of POD-like structs) are cache-friendly and SIMD-friendly.

Cache-Friendly Layout

```
struct Item { int id; float score; };
std::vector<Item> items; // contiguous, predictable iteration
```

When performance matters, designs should make hot data contiguous and cold data indirect.

27.3.4 4) Locking vs. Sharding vs. Message Passing

Locking

- simple to implement locally,
- becomes complex at scale (contention, deadlocks, priority inversion),
- tail latency suffers when contention spikes.

Sharding

- reduces contention by partitioning state,
- improves cache locality,
- complicates cross-shard operations and rebalancing.

Message Passing / Actors

- avoids data races by isolating state,
- provides natural backpressure via bounded mailboxes,
- adds messaging overhead and requires flow observability.

The decision depends on:

- proportion of shared mutable state,
- hot key distribution,
- latency requirements,
- operational complexity tolerance.

27.3.5 5) Copying vs. Borrowing vs. Ownership Transfer

Borrowing (`string_view`/`span`)

- avoids copies and allocations,
- risks dangling if stored beyond call,
- requires strict lifetime conventions.

Copying

- safer lifetime, simpler reasoning,
- costs CPU and memory bandwidth.

Ownership Transfer (Move Semantics)

- avoids deep copies,
- expresses ownership clearly,
- requires careful design to preserve invariants and avoid use-after-move.

Move-Based Handoff

```
#include <string>

struct Request {
  std::string payload;
};

void enqueue(Request r); // by value (moves at call site)

void on_io(Request r) {
  enqueue(std::move(r)); // ownership transferred
}
```

27.3.6 6) Synchronous vs. Asynchronous Boundaries

Synchronous (Blocking) APIs

- simpler control flow,
- easier to reason about errors and lifetimes,
- can waste threads under I/O waits and reduce concurrency.

Asynchronous APIs

- scales well for I/O-bound workloads,
- reduces thread count,
- increases complexity (state machines, cancellation, lifetime discipline).

The correct approach is often hybrid:

- async at I/O boundaries,
- synchronous CPU processing within a bounded worker pool.

27.3.7 7) Determinism vs. Peak Speed

Some techniques maximize peak speed but reduce determinism:

- aggressive caching (variable hit rates),
- adaptive batching,
- lock-free data structures under contention,
- speculative parallelism.

For services with strict SLOs, determinism often matters more than peak throughput. Prefer mechanisms with bounded worst-case behavior:

- bounded queues,
- bounded retries and deadlines,
- deterministic shutdown,
- stable memory usage patterns.

27.3.8 8) Lock-Free vs. Lock-Based Data Structures

Lock-Free

- can reduce contention in specific hot paths,
- may improve throughput when contention is moderate,
- introduces complexity (memory reclamation, ABA problems, subtle correctness risks).

Lock-Based

- simpler and often sufficient,
- can be optimized with sharding and reduced critical sections,
- may suffer under high contention.

A safe policy in long-lived systems:

- start with lock-based or shard-based designs,
- only adopt lock-free when profiling proves it is necessary,
- wrap lock-free structures behind stable interfaces with thorough tests.

27.3.9 9) Inlining and Header-Only vs. Build Times and Binary Size

Header-Only and Aggressive Inlining

- can improve performance by enabling inlining and optimization across call sites,
- increases compile time and rebuild cascades,
- increases binary size, which can harm instruction cache locality.

Out-of-Line Implementation

- reduces build coupling and rebuild times,
- supports ABI stability (especially with pImpl),
- may reduce inlining opportunities.

Design implication:

- keep hot, small functions inline,
- keep large implementations out-of-line,
- avoid templating everything in a core service library unless it is truly needed.

27.3.10 10) Observability vs. Overhead

Metrics, logs, and tracing are essential but have costs.

Costs

- additional CPU for formatting and export,
- allocation overhead,
- lock contention in logging subsystems,
- increased I/O.

Design Strategy

- structured events with minimal allocation,
- sampling for high-rate paths,

- separate hot-path counters from slow-path detailed logs,
- avoid synchronous disk I/O on request threads.

Minimal Hot-Path Counter

```
#include <atomic>

struct Counters {
  std::atomic<std::uint64_t> requests{0};
  std::atomic<std::uint64_t> rejected{0};
};
```

27.3.11 11) Security and Validation vs. Speed

Validation and safety checks cost CPU, but removing them can enable denial-of-service and correctness bugs.
Tradeoffs must be explicit:

- enforce maximum sizes and parsing limits,
- validate external inputs at boundaries,
- keep internal invariants as contracts.

High-performance services should prefer:

- O(n) validation with bounded constants,
- early rejection,
- streaming parsing for large inputs to avoid full buffering.

27.3.12 12) A Practical Decision Toolkit (How to Choose)

Rule 1: Optimize the Architecture Before the Micro-Details

Most wins come from:

- removing contention,
- improving locality,
- bounding queues and applying backpressure,
- avoiding unnecessary work.

Rule 2: Measure Under Production-Like Load

Tradeoffs cannot be decided by single-thread microbenchmarks alone. Use:

- representative request mixes,
- realistic concurrency and payload sizes,
- tail latency metrics.

Rule 3: Prefer Bounded Worst-Case Behavior

If you must pick, choose the design that:

- fails safely under overload,
- preserves latency predictability,
- is easier to operate and debug.

Rule 4: Encode Tradeoffs in Types and Budgets

Budgets and ownership models should be explicit so they cannot be ignored.

27.3.13 Closing Principle: Performance Is a Set of Explicit Choices

In high-performance Modern C++ services, performance tradeoffs are inevitable. The engineering skill is to choose tradeoffs that support the service's goals: stable tail latency, bounded resource use, correctness, and long-term maintainability. When tradeoffs are made explicit—via bounded queues, explicit deadlines, clear ownership, sharded state, and measured observability—the service remains fast not only today, but across years of evolution.

Designing a Long-Lived Framework

28.1 Extension Points

A long-lived framework survives not by predicting every future feature, but by providing stable extension points that allow new behaviors to be added without rewriting the core. The more successful the framework becomes, the more it must absorb:

- new product requirements,
- new integrations and platforms,
- new performance constraints,
- new compliance and security demands,
- new teams and code ownership boundaries.

Without extension points, growth expresses itself as modification of core code. That leads to:

- unstable internals (every new feature touches everything),
- increased regression risk,
- slower releases,
- fragmentation (teams fork the framework to move faster).

In Modern C++, extension points must be designed with special care:

- ownership and lifetime across boundaries,
- exception and error propagation across plugin boundaries,
- ABI stability (when binary plugins exist),
- thread-safety of callbacks,
- performance overhead of indirection,
- security and sandboxing concerns.

This section provides a practical, production-minded strategy for defining extension points that keep a framework stable for many years.

28.1.1 1) What Is an Extension Point?

An extension point is a *stable contract* that allows external code to influence behavior without changing framework core logic. It is defined by:

- a contract (inputs, outputs, failure semantics),
- a lifecycle model (construction, configuration, shutdown),
- an execution model (threading, ordering, reentrancy),
- a compatibility strategy (versioning and deprecation),
- a security model (trust boundaries and validation).

A good extension point is:

- **narrow:** minimal surface area,

- **composable:** multiple extensions can coexist,
- **testable:** extensions can be validated in isolation,
- **stable:** evolution is additive and versioned.

28.1.2 2) Classes of Extension Points

A long-lived C++ framework typically needs several distinct kinds.

A) Interface/Strategy Injection (Polymorphic)

A stable interface defines a variation point.

```
#include <string_view>
#include <optional>

struct IRequestAuthorizer {
  virtual ~IRequestAuthorizer() = default;
  virtual bool allowed(std::string_view user, std::string_view action) = 0;
};
```

Use When

- behavior must be substituted (auth, routing, encoding, storage),
- multiple implementations may exist,
- you need a stable boundary for long-term evolution.

B) Callback Hooks (Observer)

The framework emits events; extensions observe or react.

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

struct Event {
  std::string name;
  std::string message;
};

using EventCallback = void(*)(const Event&); // C ABI friendly option
```

Use When

- you need notifications (metrics, tracing, logging, auditing),
- the extension should not control core execution,
- ordering can be explicit and deterministic.

C) Data-Driven Extension (Registration)

Extensions register handlers by key.

```
#include <functional>
#include <string>
#include <unordered_map>

using Handler = std::function<int(const std::string&)>;

class Registry {
public:
  bool add(std::string name, Handler h) {
    return map_.emplace(std::move(name), std::move(h)).second;
  }
```

```
  Handler* find(const std::string& name) {
    auto it = map_.find(name);
    if (it == map_.end()) return nullptr;
    return &it->second;
  }

private:
  std::unordered_map<std::string, Handler> map_;
};
```

Use When

- you have named extension points (commands, routes, codecs),
- you want dynamic composition without deep inheritance.

D) Plugin Modules (Binary Boundary)

Extensions are built and distributed separately and loaded at runtime. This requires ABI discipline.

Use When

- third parties must extend the framework,
- independent deployment cadence is required,
- features must be optional at runtime.

This is the hardest extension style in C++ and must be treated as a full subsystem (versioning, ABI, error isolation).

28.1.3 3) Designing an Extension Point Contract

For each extension point, define these explicitly.

Inputs and Outputs

- use value types or non-owning views (`string_view`, `span`),
- avoid passing ownership ambiguously,
- avoid exposing internal framework types that will evolve frequently.

Failure Semantics

- define whether errors are recoverable and how they are represented,
- define whether exceptions are allowed to cross the boundary (often: no),
- define fallback behavior if an extension fails.

Performance Semantics

- define whether it is called on hot paths,
- define allocation expectations,
- define maximum allowed work per call (or budget).

Threading and Reentrancy

- which thread calls the extension (I/O thread, worker pool, main thread),
- whether calls can be concurrent,
- whether callbacks can call back into the framework (reentrancy),

- ordering guarantees (at-most-once, at-least-once, deterministic ordering).

28.1.4 4) Ownership and Lifetime Across Extension Boundaries

Ownership rules are the main source of instability in long-lived frameworks.

Rule: Prefer Borrowing for Inputs, Owning for Stored State

The extension sees borrowed inputs; if it must store something, it owns a copy.

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

struct RequestContext {
  std::string user_id; // owned, stable
  std::string route;   // owned, stable
};

struct IHook {
  virtual ~IHook() = default;
  virtual void on_request(std::string_view route, const RequestContext& ctx) = 0;
};
```

Avoid Passing Raw Pointers to Internal Mutable State

If you pass internal pointers, you freeze internal layout and lifetime rules. Prefer handles, views, or stable facades.

Lifecycle Control

The framework should control:

- when an extension is created,

- when it is configured,
- when it starts receiving calls,
- when it is stopped and destroyed.

28.1.5 5) Extension Point Patterns That Scale Over Years

Pattern 1: Ports and Adapters (Stable Core, Replaceable Edges)

Define narrow interfaces as ports. Provide adapters for environments.

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

struct IConfigPort {
  virtual ~IConfigPort() = default;
  virtual std::optional<std::string> get(std::string_view key) const = 0;
};
```

The framework depends on `IConfigPort`, not on concrete config systems.

Pattern 2: Middleware Pipeline (Composable Extension)

A pipeline allows many extensions to participate with deterministic ordering.

```
#include <functional>
#include <vector>

struct Request { /*...*/ };
struct Response { /*...*/ };

using Next = std::function<Response(Request&)>;
```

```
using Middleware = std::function<Response(Request&, Next)>;

class Pipeline {
public:
  void add(Middleware m) { chain_.push_back(std::move(m)); }

  Response run(Request& r, Next terminal) {
    Next next = std::move(terminal);
    for (auto it = chain_.rbegin(); it != chain_.rend(); ++it) {
      Middleware m = *it;
      Next prev = next;
      next = [m = std::move(m), prev = std::move(prev)](Request& req) mutable {
        return m(req, prev);
      };
    }
    return next(r);
  }

private:
  std::vector<Middleware> chain_;
};
```

Use this for:

- auth, rate limiting, tracing, validation, caching,
- cross-cutting concerns with clear ordering.

Pattern 3: Registration by Type/Key With Versioned Capabilities

Extensions register capabilities rather than overriding large base classes.

```
#include <string>
#include <unordered_map>
```

```
#include <memory>

struct ICodec {
  virtual ~ICodec() = default;
  virtual std::string encode(std::string_view in) = 0;
};

class CodecRegistry {
public:
  bool register_codec(std::string name, std::unique_ptr<ICodec> c) {
    return codecs_.emplace(std::move(name), std::move(c)).second;
  }

  ICodec* find(std::string_view name) const {
    auto it = codecs_.find(std::string(name));
    if (it == codecs_.end()) return nullptr;
    return it->second.get();
  }

private:
  std::unordered_map<std::string, std::unique_ptr<ICodec>> codecs_;
};
```

This keeps extension surfaces narrow and reduces fragile inheritance.

28.1.6 6) Binary Plugins: Designing a Safe C++ Plugin ABI

If you must support third-party binary plugins, avoid exposing C++ ABI details. Use a C ABI boundary.

C ABI Factory Function

```
extern "C" {
```

```
// Versioning: the framework queries plugin ABI version.
int plugin_abi_version();

// Create/destroy functions use opaque pointers.
void* plugin_create();
void  plugin_destroy(void*);

}
```

Opaque Interface Table (Stable Function Pointers)

```
#include <cstddef>

extern "C" {

struct PluginApi {
  int (*init)(void* self);
  int (*handle)(void* self, const char* req, std::size_t n, int* out_code);
  void(*shutdown)(void* self);
};

const PluginApi* plugin_api();

}
```

This avoids C++ vtable and name mangling instability across compilers and standard library versions.

Error Isolation at Plugin Boundary

Policy should require:

- plugin functions must not throw across boundary,

- framework catches and converts failures to structured errors,
- plugin time budgets are enforced (if possible),
- plugin calls are guarded by backpressure or circuit breakers.

28.1.7 7) Versioning and Evolution of Extension Points

Extension points must evolve without breaking existing extensions.

Prefer Additive Evolution

- add new optional methods via new interfaces,
- add new capabilities rather than changing old semantics,
- keep old extension points working for a defined window.

Capability Query Pattern

Extensions declare supported features; the framework adapts.

```
#include <cstdint>

struct Capabilities {
  std::uint32_t bits = 0;
  static constexpr std::uint32_t supports_async = 1u << 0;
  static constexpr std::uint32_t supports_batch = 1u << 1;
};

struct IExtension {
  virtual ~IExtension() = default;
  virtual Capabilities caps() const noexcept = 0;
};
```

Deprecation Policy for Extension APIs

A framework must define:

- how long deprecated hooks remain,
- how warnings are communicated,
- compatibility layers and migration guides.

28.1.8 8) Security and Trust Boundaries

Extension points create a trust boundary. The framework must decide:

- are extensions trusted (same team) or untrusted (third party)?
- can an extension read sensitive data?
- can it block or crash the process?

For untrusted extensions, consider:

- out-of-process plugins (IPC),
- strict input validation,
- timeouts and resource budgets,
- least-privilege data exposure.

Even for trusted extensions, enforce:

- bounded work per call,
- no blocking on critical threads,
- deterministic shutdown and cleanup.

28.1.9 9) Testability of Extension Points

A long-lived framework must test extension behavior as part of its CI.

Contract Tests

For each extension point:

- define minimal contract tests,
- test default behavior without extensions,
- test composition order and failure modes,
- test concurrency rules (no reentrancy violations).

Golden Extension Harness

Provide a harness that can load an extension in isolation and validate:

- lifecycle ordering,
- error handling,
- performance budgets (basic),
- shutdown correctness.

28.1.10 10) Extension Point Checklist (Framework-Grade)

- Narrow interface with stable types and clear contracts.
- Explicit ownership and lifetime rules (borrow inputs, own stored state).
- Defined threading model, ordering, and reentrancy policy.

- Failure semantics defined; exceptions do not cross boundaries.
- Performance expectations and budgets stated; hot paths minimized.
- Versioning and deprecation plan is explicit and enforceable.
- Security model defined; data exposure is least privilege.
- Test harness and contract tests exist for extension behavior.

28.1.11 Closing Principle: Extension Points Are the Framework's Survival Mechanism

A long-lived C++ framework is shaped by growth. Extension points are the mechanism that allows growth without destabilizing the core. The most durable extension points are narrow, versioned, contract-driven, testable, and disciplined in ownership and threading. When designed well, the framework can absorb new requirements and new teams for many years without turning into an unmaintainable patchwork.

28.2 Stability Guarantees

A framework that aims to live for many years must define *stability guarantees* as a first-class product feature. Stability is not a vague promise of "we will not break you"; it is a precise contract that answers, for every upgrade:

- what will keep working without changes,
- what may require recompilation,
- what may require source modifications,
- what may change in runtime behavior or performance,

- what is explicitly *not* guaranteed.

Without explicit guarantees, the framework forces each user team to discover breakage by surprise. Over time, that creates upgrade fear, version pinning, forks, and duplicated infrastructure. With explicit guarantees, upgrades become routine engineering.
This section defines a practical, Modern C++-oriented stability model for a long-lived framework, including: API/ABI/behavior contracts, versioning, deprecation, compatibility testing, and the techniques used to enforce guarantees over decades.

28.2.1 1) The Three Stability Axes: Source, Binary, and Behavior

Most stability discussions fail because they mix three different things.

Source (API) Stability

Definition: client code continues to compile without changes against a newer framework version.
Source stability depends on:

- preserving function signatures, names, namespaces,
- preserving template constraints and visible overload sets,
- avoiding breaking changes to required includes or macros,
- avoiding semantic changes that force different call patterns.

Binary (ABI) Stability

Definition: already-compiled client binaries continue to run with a newer framework binary, without recompilation.
ABI stability depends on:

- object layout (size, alignment, member order),

- vtable layout and virtual function order,
- name mangling and calling conventions,
- exception and RTTI ABI,
- standard library ABI compatibility,
- symbol visibility and linking model.

Behavioral Stability

Definition: runtime semantics remain consistent: outcomes, error conditions, ordering, and guarantees do not change unexpectedly.
Behavioral stability includes:

- correctness semantics (postconditions, invariants),
- error semantics (what fails, how it is reported),
- concurrency semantics (thread-safety, ordering, progress),
- performance semantics (complexity and tail-latency expectations),
- operational semantics (shutdown, resource limits, logging events).

A framework must state which axes it guarantees and under what conditions.

28.2.2 2) A Framework-Grade Stability Policy (What You Promise)

A robust policy is explicit and tiered.

Guarantee Levels

- **Level 1: Source-stable upgrades** within a major version.
- **Level 2: ABI-stable upgrades** within an LTS line or within a major version on supported toolchains.
- **Level 3: Behavioral stability** for documented contracts; changes require an explicit migration story.

What Must Be Explicitly *Not* Guaranteed

To avoid implicit promises, state non-guarantees such as:

- internal/private headers are not stable,
- undefined behavior is not stabilized (misuse remains misuse),
- performance may improve or regress within bounded expectations,
- non-documented side effects are not stable,
- debug-only diagnostics are not stable.

28.2.3 3) Versioning: Making Stability Machine-Checkable

Stability requires a versioning scheme that maps changes to expectations.

Semantic Version Triples

Use `MAJOR.MINOR.PATCH` with strict meaning:

- **MAJOR:** may break source and/or ABI, may remove deprecated features.

- **MINOR:** additive changes; preserves source compatibility for supported usage, preserves ABI if promised.
- **PATCH:** bug fixes only; no API/ABI changes; behavior changes only to correct documented bugs.

Compile-Time Version Macros

```
#pragma once

#define FW_VERSION_MAJOR 2
#define FW_VERSION_MINOR 7
#define FW_VERSION_PATCH 3

#define FW_VERSION_ENCODE(M,m,p) ((M)*10000 + (m)*100 + (p))
#define FW_VERSION FW_VERSION_ENCODE(FW_VERSION_MAJOR, FW_VERSION_MINOR, FW_VERSION_PATCH)
```

Feature Detection Over Version Checks

Prefer capability macros over hard version compares.

```
#pragma once

// Example: new non-throwing API becomes available
#define FW_HAS_RESULT_API 1
```

28.2.4 4) Deprecation as Part of Stability

Deprecation is the mechanism that allows evolution without surprise breakage.

Deprecation Rules

- Deprecations are **additive** (warnings first).

- Every deprecation includes a **replacement path**.
- Deprecated APIs remain available for a **minimum window** (e.g., two minor releases or one LTS cycle).
- Removal happens only in a **major** release (or a defined break window).

A Portable Deprecation Macro

```
#pragma once

#if defined(__clang__) || defined(__GNUC__)
  #define FW_DEPRECATED(msg) __attribute__((deprecated(msg)))
#elif defined(_MSC_VER)
  #define FW_DEPRECATED(msg) __declspec(deprecated(msg))
#else
  #define FW_DEPRECATED(msg)
#endif
```

Using Deprecation Without Breaking Builds

```
#include "fw/deprecated.hpp"

namespace fw {

FW_DEPRECATED("Use fw::ClientOptions::timeout_ms and fw::Client::create().")
int create_client_legacy(const char* endpoint);

} // namespace fw
```

28.2.5 5) Techniques to Deliver ABI Stability in Modern C++

If you promise ABI stability, you must design for it from day one.

Technique A: pImpl for Public Classes

pImpl decouples public layout from private evolution.

```
#pragma once
#include <memory>
#include <string_view>

namespace fw {

class Client {
public:
  Client();
  ~Client();

  Client(Client&&) noexcept;
  Client& operator=(Client&&) noexcept;

  Client(const Client&) = delete;
  Client& operator=(const Client&) = delete;

  bool send(std::string_view payload) noexcept;

private:
  struct Impl;
  std::unique_ptr<Impl> p_;
};

} // namespace fw
```

Technique B: Avoid Exposing STL Types in ABI-Critical Boundaries (When Needed)

Across binary boundaries, `std::string`, `std::vector`, and exception types can be fragile if toolchain ABIs differ. If the framework is shipped as a binary for many toolchains, consider:

- opaque handles,
- C ABI boundaries for plugins,
- stable POD-like structs with explicit ownership rules.

Technique C: Freeze Virtual Interfaces

If you expose a polymorphic interface and promise ABI stability:

- do not reorder virtual functions,
- do not insert new virtual functions in the middle,
- prefer adding new interfaces (e.g., `IFoo2`) rather than mutating old ones.

```
namespace fw {

struct IStorage {
  virtual ~IStorage() = default;
  virtual bool put(std::string_view k, std::string_view v) = 0;
  virtual bool get(std::string_view k, /*out*/ void* sink) = 0; // illustrative
};

// Additive evolution: new interface rather than changing IStorage
struct IStorage2 : IStorage {
  virtual bool erase(std::string_view k) = 0;
};

} // namespace fw
```

Technique D: Inline Namespace for ABI Tagging

An inline namespace can encode ABI generations.

```cpp
namespace fw {
inline namespace abi_v2 {

class Client { /*...*/ };

} // namespace abi_v2
} // namespace fw
```

This helps prevent accidental mixing of incompatible binaries at link time.

28.2.6 6) Behavioral Stability: Contracts You Must State

Behavioral stability is what users feel. It must be documented as contracts.

Error Semantics

Define:

- what is a recoverable error vs a contract violation,
- how errors are represented (result types, error codes),
- whether exceptions can cross the boundary,
- which errors are stable and which may vary with environment.

Thread-Safety and Concurrency Guarantees

For every type, state one of:

- **thread-safe:** concurrent calls are safe,
- **thread-compatible:** safe if callers synchronize,
- **thread-hostile:** not safe for cross-thread use.

Also state:

- reentrancy rules,
- callback threading model,
- ordering guarantees (FIFO, best-effort),
- progress guarantees (blocking, lock-free, wait-free if claimed).

Resource and Shutdown Semantics

A long-lived framework must guarantee:

- deterministic cleanup (RAII, stop/join),
- bounded queues or explicit overload behavior,
- shutdown ordering (no callbacks after stop),
- cancellation behavior where applicable.

Performance Semantics (Bounded Expectations)

Do not promise absolute speed. Promise *properties*:

- complexity class (e.g., O(n) in input size),
- allocation behavior on hot paths (“no dynamic allocation” where guaranteed),
- bounded memory growth with configured limits,
- latency-sensitive thread rules (no blocking on event loop).

28.2.7 7) Compatibility Testing: Enforcing Stability Automatically

Stability guarantees are meaningless without automated enforcement.

API Stability Tests

- compile representative client code against each release,
- include “public surface” smoke tests: headers compile standalone,
- enforce include hygiene: public headers include what they use.

ABI Stability Tests

If ABI stability is promised:

- build a client against an older version and run it against the newer binary,
- verify symbol sets and exported signatures remain compatible,
- freeze layout for ABI-critical classes (pImpl, stable handles).

Behavioral Contract Tests

Maintain a suite of tests that represent your public contracts:

- error classification and messages (at least error codes/kinds),
- ordering and threading guarantees,
- boundary conditions and limits,
- shutdown and cancellation correctness.

Example: A Simple Contract Test Shape

```
template <class Client>
void client_contract(Client& c) {
  // must not throw across boundary
  bool ok = c.send("ping");
  (void)ok;
  // add checks for error reporting, idempotence, ordering, etc.
}
```

28.2.8 8) Migration Guarantees: How Users Upgrade Safely

A stability promise must include an upgrade story.

Supported Upgrade Modes

- **in-place upgrade:** replace binaries (ABI-stable promise required),
- **recompile upgrade:** rebuild clients (API-stable promise),
- **dual-run window:** run old and new side-by-side (protocol/data compatibility),
- **compatibility layer:** adapters that support both old and new APIs.

Strangler-Friendly Design

Provide stable facades so users can migrate incrementally:

- adapters for old interfaces,
- feature flags for new behavior behind stable call sites,
- explicit compatibility shims with deadlines for removal.

28.2.9 9) The Stability Contract Document (What You Publish)

A long-lived framework should publish a short stability contract (even internally), containing:

- supported toolchains and language levels,
- what is guaranteed within MAJOR/MINOR/PATCH,
- ABI policy and its scope (which libraries, which platforms),
- threading policy and callback rules,
- error and contract strategy (throwing policy, error categories),
- deprecation window and removal policy,
- performance properties that are guaranteed,
- the compatibility test matrix used to enforce guarantees.

28.2.10 Closing Principle: Stability Is an Engineering System

Stability is not achieved by avoiding change. It is achieved by engineering the framework to evolve safely:

- separate public surface from private evolution (pImpl, narrow interfaces),
- version and deprecate deliberately (no surprise removals),
- define behavioral contracts (errors, threads, shutdown),
- enforce with automated compatibility tests.

A framework that treats stability guarantees as an explicit product contract becomes upgradeable, adoptable, and trustworthy—the defining properties of a truly long-lived Modern C++ framework.

28.3 Versioning Strategy

Versioning is not a cosmetic numbering scheme; it is a *compatibility protocol* between the framework and its users. In a long-lived framework, version numbers must be meaningful enough that:

- upgrade risk can be estimated before building and deploying,
- compatibility promises are enforceable by process and automation,
- different teams can coordinate upgrades without fear or forks,
- deprecations and removals happen predictably,
- the framework can evolve without freezing innovation.

This section defines a framework-grade versioning strategy for Modern C++ systems, covering: semantic versioning discipline, API/ABI/behavior boundaries, feature flags and capability detection, deprecation windows, LTS lines, package/build integration, and the automation needed to make versioning real.

28.3.1 1) The Core Problem Versioning Must Solve

A framework that lives for many years faces two persistent tensions:

- **Users demand stability:** upgrades must be safe, predictable, and incremental.
- **Framework demands evolution:** new features, platforms, and performance work must ship.

A versioning strategy must reconcile these tensions by defining:

- what changes are allowed in each release class,

- how compatibility is preserved or intentionally broken,
- how users discover and migrate off old APIs,
- how multiple versions are supported concurrently (if needed).

28.3.2 2) Define What "Compatibility" Means in Your Framework

Before adopting any numbering scheme, explicitly define the compatibility axes you care about.

Source Compatibility (API)

Client code compiles without changes against the newer version (within the defined compatibility window).

Binary Compatibility (ABI)

Already-compiled binaries run with the newer framework binary (only if you ship/consume prebuilt binaries and you promise ABI stability).

Behavioral Compatibility

Runtime semantics remain consistent for documented contracts (errors, ordering, thread-safety, shutdown, complexity).

Rule Versioning must map changes to these compatibility axes. If the version number does not predict compatibility, it has no value.

28.3.3 3) Semantic Versioning as a Discipline (Not a Slogan)

A practical framework strategy uses a strict `MAJOR.MINOR.PATCH` triple with clear rules.

PATCH: Fixes Only

- bug fixes,
- security fixes,
- performance improvements that do not change documented semantics,
- no source-breaking changes,
- no ABI-breaking changes (if ABI stability is promised),
- no new required configuration.

MINOR: Additive Evolution

- new APIs are added (additive),
- new features are introduced behind defaults (no surprise behavior changes),
- deprecations may be introduced (warnings only),
- source compatibility preserved for supported usage,
- ABI compatibility preserved if promised (requires strict engineering discipline),
- behavioral changes are allowed only when explicitly gated (feature flags) or when correcting a documented bug.

MAJOR: Compatibility Reset Window

- deprecated APIs may be removed,
- semantics may be simplified and contracts tightened,

- ABI may change (unless you run multi-ABI or special compatibility layers),
- migration must be documented and tool-supported where possible.

Non-negotiable If you ship a breaking change in MINOR or PATCH, you destroy user trust and force version pinning. The entire strategy collapses.

28.3.4 4) A Change Classification Matrix

To make versioning enforceable, define a change matrix that maps change types to version increments.

API Surface Changes

- add a new function or overload: **MINOR**
- remove a public symbol: **MAJOR**
- change a function signature: **MAJOR**
- change template constraints such that code stops compiling: **MAJOR**
- change default template parameters: **MAJOR** (often breaks behavior or overload resolution)

ABI-Relevant Changes (When ABI Stability Is Promised)

- change public class size/layout: **MAJOR**
- reorder virtual functions: **MAJOR**
- add virtual functions to a stable interface: **MAJOR** (or define `IFoo2` instead)
- change inline namespace ABI tag: **MAJOR** (explicit ABI generation)

Behavior Changes

- change documented error category or return semantics: **MAJOR** (unless behind feature flag)
- change ordering guarantees, threading guarantees, or shutdown semantics: **MAJOR**
- bug fix that changes results previously incorrect: **PATCH** (if documented as bug fix; still requires careful release notes)

Performance Changes

- optimization with same semantics: **PATCH** or **MINOR**
- algorithmic complexity change on hot path: **MAJOR** if it violates documented performance contract

28.3.5 5) Versioning the Public Surface: API, ABI, and SPI

A long-lived framework has at least three surfaces:

Public API

Guaranteed stable under the version rules.

ABI-Safe Surface (Optional)

If you promise ABI stability, explicitly define which headers/types are ABI-stable. This is usually a small subset.

SPI (Service Provider Interface) / Internal APIs

Not stable. Used for internal integration or plugin internals. Must be clearly separated by:

- namespaces (`fw::detail`, `fw::internal`),
- directory structure (`include/fw/internal/...`),
- documentation and build rules (not installed, not exported).

Rule If users depend on SPI, upgrades become impossible. Your build system should make SPI use noisy (warnings or separate packages).

28.3.6 6) Deprecation Windows as Part of Versioning

Deprecation is the bridge between evolution and stability.

Deprecation Policy

- deprecate in **MINOR**,
- keep for a minimum window (e.g., **two MINOR releases** or **one LTS cycle**),
- remove only in **MAJOR**.

Deprecation Macro

```
#pragma once

#if defined(__clang__) || defined(__GNUC__)
  #define FW_DEPRECATED(msg) __attribute__((deprecated(msg)))
#elif defined(_MSC_VER)
  #define FW_DEPRECATED(msg) __declspec(deprecated(msg))
#else
```

```
  #define FW_DEPRECATED(msg)
#endif
```

Deprecate With a Replacement Path

```
namespace fw {

FW_DEPRECATED("Use fw::Client::create(const ClientOptions&) returning Result<Client>.")
Client create_client_legacy(const char* endpoint);

} // namespace fw
```

Best practice Provide an automated migration where possible:

- mechanical rename wrappers,
- compatibility adapters,
- compile-time warnings that point to the new API.

28.3.7 7) Capability Detection Over Version Checks

In large ecosystems, users should prefer *feature detection* over hard-coded version comparisons.

Why

- forks and downstream backports exist,
- LTS patches may introduce features,
- version checks create brittle code paths.

Feature Macros

```
#pragma once

#define FW_VERSION_MAJOR 2
#define FW_VERSION_MINOR 8
#define FW_VERSION_PATCH 0

#define FW_HAS_NONTHROWING_FACTORY 1
#define FW_HAS_PIPELINE_MIDDLEWARE 1
#define FW_HAS_SHARDED_EXECUTOR    0
```

Using Capability Checks

```
#if FW_HAS_NONTHROWING_FACTORY
  auto r = fw::Client::create(opts);
#else
  auto c = fw::create_client_legacy(opts.endpoint.c_str());
#endif
```

Rule Version macros exist for logging and diagnostics; capabilities drive conditional compilation.

28.3.8 8) ABI Versioning for Binary Distribution (When Relevant)

If your framework ships binaries, versioning must also prevent accidental binary mixing.

Inline Namespace ABI Tags

```
namespace fw {
inline namespace abi_v2 {

class Client { /* ABI generation v2 */ };
```

```
} // namespace abi_v2
} // namespace fw
```

Symbol Versioning (Platform/Toolchain Dependent)

If you use symbol versioning, treat it as a build/distribution policy:

- keep it deterministic and automated,
- validate the exported symbol set in CI,
- never silently remove exported symbols except in MAJOR.

"Stable Handle" Boundaries

For plugins and third-party binary extensions, prefer C ABI boundaries and opaque handles to avoid C++ ABI fragility.

28.3.9 9) Release Lines: LTS, Feature Releases, and Support Windows

Long-lived frameworks benefit from predictable release trains.

Common Model

- **Feature line (rolling):** frequent MINOR releases.
- **LTS line:** selected MINOR becomes LTS; only PATCHes after that.
- **Major transitions:** less frequent; include removal of deprecated APIs.

Support Commitments

Define:

- how long each LTS is supported,
- whether security fixes are backported,
- the maximum number of supported lines concurrently.

Rule If support windows are vague, users will pin forever.

28.3.10 10) Versioning Configuration and Defaults

Defaults are part of your compatibility contract.

Compatibility via Default Stability

In MINOR releases:

- avoid changing defaults that alter behavior significantly,
- if necessary, gate new behavior behind a feature flag with a stable default.

Feature Flags with Explicit Opt-In

```
enum class FeatureFlag {
  strict_validation,
  new_scheduler,
  deterministic_shutdown
};

struct FeatureSet {
  bool strict_validation = false;      // stable default
```

```
  bool new_scheduler = false;          // opt-in during migration window
  bool deterministic_shutdown = true;  // may be default if already promised
};
```

Migration pattern

- introduce feature in MINOR as opt-in,
- validate in production,
- flip default in next MAJOR (or after an explicit compatibility window).

28.3.11 11) Automation: Making Versioning Enforceable

Versioning fails without automated enforcement.

API Surface Checks

- compile public headers as standalone translation units,
- compile “representative client” code against every release,
- ensure public headers include what they use (include hygiene).

ABI Compatibility Checks (If Promised)

- run binary compatibility validation between releases,
- verify exported symbols and class layouts for ABI-stable types,
- test an old client binary against the new framework binary.

Behavioral Contract Tests

- error kinds/codes remain stable for documented conditions,
- threading and shutdown semantics are preserved,
- performance invariants (no unbounded growth, bounded queues) remain true.

Release Gate

A release should not be published unless:

- change classification matches MAJOR/MINOR/PATCH,
- deprecations meet window policy,
- compatibility tests pass for the promised guarantees.

28.3.12 12) A Practical Versioning Blueprint (Recommended)

A robust blueprint for a long-lived Modern C++ framework:

- Use strict `MAJOR.MINOR.PATCH`.
- Within a MAJOR:
  - MINOR is additive, may deprecate, no breaking changes.
  - PATCH is fixes only.
- Define a clear deprecation window and remove only in MAJOR.
- Prefer capability macros over version comparisons.
- Separate API vs SPI in headers, namespaces, and packaging.

- If ABI stability is promised:
  - limit ABI-stable surface area,
  - use pImpl/opaque handles and/or inline namespace ABI tags,
  - enforce ABI checks in CI.
- Offer an LTS line with PATCH-only updates and published support windows.
- Gate behavior-changing improvements behind feature flags until a MAJOR boundary.

28.3.13 Closing Principle: Version Numbers Are a Contract You Must Enforce

A long-lived framework wins adoption when upgrades are boring. That requires version numbers that predict reality, deprecation that provides time and tooling, capabilities that prevent brittle checks, and automated enforcement that prevents accidental breaking changes. When versioning is treated as an engineering system rather than a label, the framework can evolve for many years without losing its users to fear, forks, or stagnation.

Part X

Design Wisdom

Common Design Failures in C++

29.1 Overengineering

Overengineering is a design failure where a system is built for complexity it does not yet have, using abstractions and mechanisms whose costs exceed their present value. In Modern C++, overengineering is especially dangerous because the language makes it easy to construct extremely powerful (and extremely subtle) abstractions: template metaprogramming, policy-based design, deep type hierarchies, intrusive frameworks, custom allocators, lock-free structures, and elaborate build systems. These tools are not wrong; they are expensive. Overengineering happens when cost is paid *now* for benefits that remain hypothetical.
A long-lived system must remain:

- understandable by new engineers,
- safe to refactor,
- predictable in performance,
- stable under incremental change,
- economical to maintain.

Overengineering undermines all five.

29.1.1 1) The True Cost Model of Overengineering

To avoid overengineering you must measure cost correctly. The real cost is not the extra lines of code; it is the extra *states and degrees of freedom* the system introduces.

Cost Dimensions

- **Cognitive cost:** how many concepts must an engineer hold to make a small change?
- **Change amplification:** how many files and modules must be touched to add a feature?
- **Coupling cost:** how many dependencies become implicit and non-local?
- **Testing cost:** how much behavior becomes hard to simulate or isolate?
- **Build cost:** compile times, link times, incremental rebuild cascades.
- **Operational cost:** deployment risk, debugging difficulty, observability gaps.

Overengineering increases these costs even if runtime performance is excellent.

29.1.2 2) Common Forms of Overengineering in Modern C++

A) Over-Abstracting Too Early

Symptoms:

- abstract base classes for things that have only one implementation,
- factories and registries everywhere with no real variability,
- “plugin-ready” designs before there is a plugin ecosystem.

A Typical Smell: Unnecessary Interface

```
struct IDataStore {
  virtual ~IDataStore() = default;
  virtual void put(int key, int value) = 0;
  virtual int  get(int key) const = 0;
};

class DataStore : public IDataStore {
  // only implementation; no real variability exists
};
```

When there is only one implementation and no near-term plan for another, an interface adds:

- an extra type layer,
- indirection and lifetime concerns,
- mocking temptation (often masking better design).

Prefer a Concrete Type Until Variation Is Real

```
class DataStore {
public:
  void put(int key, int value);
  int  get(int key) const;
};
```

Introduce an interface only when a second implementation is actually needed, or when you need a stable boundary for independent compilation/ABI.

B) Template Overuse and "Type-Level Architecture"

Symptoms:

- heavy policy-based design where runtime configuration would suffice,
- templates used to encode business rules in types,
- error messages become a barrier to entry.

Smell: Policy Explosion

```
template <class StoragePolicy, class RetryPolicy, class LogPolicy, class CachePolicy>
class Service { /* ... */ };
```

This can be appropriate in low-level libraries, but in application frameworks it often:

- increases compile-time coupling,
- increases binary size,
- makes incremental changes expensive.

Prefer Runtime Composition for Application-Level Concerns

```
struct ServiceOptions {
  int retry_count = 0;
  bool enable_cache = false;
  bool enable_tracing = true;
};

class Service {
public:
  explicit Service(ServiceOptions opts);
};
```

C) Premature Generalization of Data Models

Symptoms:

- building a universal “entity” system early,
- over-normalized models for domains that are still unknown,
- making everything dynamic (string keys, variant values) without need.

The cost is loss of invariants: compile-time safety is replaced by runtime checks.

D) Framework Building Instead of Product Building

Symptoms:

- large internal frameworks where plain libraries would do,
- custom dependency injection containers for small codebases,
- elaborate configuration systems with minimal configuration needs.

A framework is justified only when multiple products or teams need the same structure *and* the framework is cheaper than duplication.

E) Premature Optimization That Hardens Design

Symptoms:

- custom allocators everywhere without profiling evidence,
- lock-free data structures without contention measurements,
- manual memory management patterns replacing RAII,
- micro-optimizations that obscure the architecture.

Overengineering often disguises itself as “performance work”.

29.1.3 3) Why Overengineering Is Worse in C++

Modern C++ supports extremely powerful abstractions with:

- templates and compile-time computation,
- multiple forms of polymorphism,
- custom allocators and memory-resource interfaces,
- concurrency primitives and atomics,
- UB pitfalls when invariants are unclear.

Therefore:

- complexity can become *non-local*: changing one header triggers rebuilds everywhere,
- abstraction leaks can become *undefined behavior*: lifetime and concurrency mistakes are catastrophic,
- maintaining ABI boundaries becomes hard when internal types leak into public headers.

29.1.4 4) How Overengineering Manifests Over Time (Long-Lived Failure Pattern)

A typical lifecycle:

1. The system is designed with many extension points “for the future”.
2. Few extension points are used; most become dead weight.
3. Engineers bypass abstractions to ship features.

4. The framework becomes inconsistent: some code follows the architecture, some bypasses it.
5. Refactoring is blocked because nobody trusts the abstraction boundaries.
6. Upgrades and improvements become slow; teams fork or rewrite.

Overengineering often causes the rewrite it tried to avoid.

29.1.5 5) Design Heuristics to Avoid Overengineering

Heuristic A: Build the Simplest Thing That Preserves Invariants

Simplicity is not fewer lines; it is fewer invalid states.

Heuristic B: Delay Abstraction Until You Have at Least Two Real Uses

An abstraction is justified when:

- you have at least two concrete use-cases,
- they share a stable common core,
- divergence is meaningful and likely to persist.

Heuristic C: Prefer Libraries Over Frameworks

Libraries are pull-based (you call them) and often compose better over time. Frameworks are push-based (they call you) and impose structure. Build a framework only when the structure itself is your product.

Heuristic D: Keep Interfaces Narrow and Stable

If you must abstract, keep the surface minimal. A large interface is a promise you will regret.

Heuristic E: Optimize Only After Profiling

Performance work must be evidence-driven. Without measurement, "optimization" is often just complexity.

Heuristic F: Prefer Value Semantics and Explicit Ownership

Overengineered designs often rely on shared ownership everywhere and complicated lifetimes. Prefer:

- values and RAII,
- `unique_ptr` for ownership transfer,
- non-owning views for inputs.

29.1.6 6) Concrete Refactoring: De-Overengineering Patterns

Pattern 1: Replace Inheritance with Composition

Overengineered Deep Hierarchy

```
struct Component { virtual ~Component() = default; virtual void tick() = 0; };
struct NetworkComponent : Component { /*...*/ };
struct AuthNetworkComponent : NetworkComponent { /*...*/ };
struct CachedAuthNetworkComponent : AuthNetworkComponent { /*...*/ };
```

Prefer a Flat Core + Composable Decorators

```
class Network { public: void tick(); };

class AuthDecorator {
public:
  explicit AuthDecorator(Network& n) : n_(n) {}
```

```
  void tick();
private:
  Network& n_;
};

class CacheDecorator {
public:
  explicit CacheDecorator(AuthDecorator& a) : a_(a) {}
  void tick();
private:
  AuthDecorator& a_;
};
```

Pattern 2: Collapse Unused Extension Points

If an extension point has no users:

- remove it (if allowed),
- or internalize it behind a single concrete implementation,
- keep the public surface minimal.

Pattern 3: Move Complex Policy to Runtime Options

Replace template policy explosion with configuration objects.

```
struct Options { bool cache = false; int retries = 0; };
class Service { public: explicit Service(Options o); };
```

Pattern 4: Localize Performance Specialization

If a hot path requires specialized optimization:

- isolate it in a small component,

- hide complexity behind a stable interface,
- keep the rest of the system clean.

29.1.7 7) A Practical Overengineering Checklist

Before adding any abstraction, answer:

- What concrete second use-case exists *now*?
- What invariant does this abstraction protect?
- Can the same goal be achieved by a simpler convention or helper function?
- What is the cost to test and debug this abstraction?
- Does it increase compile-time coupling or ABI fragility?
- Will it complicate ownership, lifetime, or concurrency?
- Can it be removed later without a rewrite?

If these questions do not have strong answers, it is likely overengineering.

29.1.8 8) The Correct Alternative: Intentional Evolution

A long-lived C++ system should be designed to evolve by:

- keeping the core small and stable,
- adding features incrementally,
- introducing extension points only when pressure is real,
- enforcing contracts and ownership rules,
- measuring performance and optimizing only where needed.

29.1.9 Closing Principle: Complexity Must Pay Rent

Overengineering is not "too much code"; it is complexity that does not pay rent. In Modern C++, each additional layer of abstraction has compounding costs in comprehension, build time, debugging, performance predictability, and long-term maintenance. A long-lived design chooses simplicity that preserves invariants today, and adds complexity only when real requirements demand it.

29.2 Leaky Abstractions

A leaky abstraction is an interface that claims to hide complexity but cannot keep that promise in practice. The hidden details "leak" into user code as soon as you scale, optimize, debug, port, or operate the system. Instead of simplifying design, the abstraction creates two simultaneous burdens:

- you must learn the abstraction *and* the underlying system it tried to hide,
- you must debug failures through multiple layers that do not align.

In Modern C++, leaky abstractions are especially harmful because:

- undefined behavior can hide behind innocent-looking APIs,
- performance costs (allocations, cache misses, atomics) are often non-obvious,
- concurrency and lifetime semantics are difficult to fully encapsulate,
- ABI and build-time coupling can leak across layers.

This section explains how leaky abstractions appear in C++ systems, why they persist, how to detect them, and how to design abstractions that stay honest and robust over long lifetimes.

29.2.1 1) What “Leak” Means in Engineering Terms

An abstraction leaks when at least one of these holds:

- **Correctness leak:** users must know hidden invariants to avoid bugs.
- **Performance leak:** users must know internal costs to avoid slowdowns.
- **Resource leak:** users must manage memory, threads, or handles the abstraction claimed to manage.
- **Concurrency leak:** users must know internal threading, ordering, or locking behavior.
- **Error leak:** users must know internal failure modes and recovery details.
- **Portability leak:** users must know OS/toolchain specifics despite a portable facade.

A framework can survive some leakage, but it must be deliberate:

- either the abstraction becomes *honest* and exposes the relevant dimensions,
- or it becomes *strict* and enforces the hidden constraints mechanically.

29.2.2 2) Why Leaky Abstractions Are Common in C++

A) C++ Cannot Magically Remove Hardware and OS Reality

Latency is not uniform, memory is hierarchical, I/O blocks, and scheduling is nondeterministic. No interface can fully hide:

- cache effects,
- memory allocation patterns,
- syscalls and kernel queues,
- thread scheduling and contention.

B) Lifetime and Ownership Are Semantics, Not Implementation Details

If an abstraction returns a view, the user must know lifetime. If it returns an owning object, the cost and move/copy semantics matter.

C) Generic Abstractions Often Encode Ambiguity

"One API for all uses" tends to leak because different uses require different guarantees (latency vs throughput, safety vs speed).

D) ABI, Toolchains, and Build Systems Leak Easily

Header-only templates, inline functions, and exposing STL types across boundaries can turn internal details into externally observable commitments.

29.2.3 3) Common Leaky Abstractions in C++ Systems

A) "Async" APIs That Leak Threading and Lifetimes

A typical failure is an async API that appears simple but requires callers to know:

- which thread runs callbacks,
- whether callbacks may run concurrently,
- whether callbacks may run after shutdown,
- whether data captured by callbacks remains valid.

Leaky Example: Callback Without Clear Thread/Lifetime Contract

```
#include <functional>
#include <string_view>
```

```
class Client {
public:
  using Callback = std::function<void(std::string_view)>;

  void async_get(std::string_view key, Callback cb);
};
```

This leaks immediately:

- Is `key` copied or referenced?
- Is the callback invoked on an I/O thread? worker thread? caller thread?
- Can it run after `Client` is destroyed?
- Is `string_view` valid after return?

Make It Honest: Explicit Ownership, Threading, and Cancellation

```
#include <string>
#include <functional>
#include <chrono>

struct CancelToken {
  bool cancelled() const noexcept;
};

class Client {
public:
  using Callback = std::function<void(std::string /*value*/)>;

  // Contract:
  // - key is copied (value semantics)
```

```
  // - callback is invoked at most once
  // - callback is invoked on the client's executor
  // - callback will not run after stop() completes
  // - cancellation is cooperative
  void async_get(std::string key,
                 std::chrono::milliseconds deadline,
                 CancelToken token,
                 Callback cb);

  void stop(); // drains/cancels pending ops with defined semantics
};
```

B) "Portable" File/Network Abstractions That Leak OS Semantics

Examples of leaks:

- path rules and encoding differences,
- file locking semantics,
- socket options and backpressure,
- timer granularity and wakeup behavior,
- signal handling and interruption behavior.

A portable abstraction must either:

- expose these differences as explicit capabilities/options, or
- restrict itself to a safe common subset and enforce it.

C) Error Abstractions That Leak Implementation Details

A leaky error abstraction forces users to interpret strings or undocumented codes, or to know internal layers to decide whether an error is retryable.

Leaky Example: Stringly-Typed Errors

```
struct Error {
  const char* message; // users parse message to decide actions
};
```

Better: Stable Categories + Codes

```
#include <cstdint>
#include <string>

struct Error {
  enum class Kind : std::uint8_t { validation, not_found, timeout, transient, internal };
  Kind kind = Kind::internal;
  int  code = 0;          // stable numeric identity for metrics and policy
  std::string message;    // human-readable
};
```

D) "Simplified" Ownership That Leaks Shared Ownership Costs

Overuse of `std::shared_ptr` is often justified as "simpler lifetime." The leak is performance and semantics:

- atomic reference counting cost,
- non-deterministic destruction time (when last ref drops),
- hidden object graphs that prevent cleanup,
- cycles requiring `weak_ptr` discipline.

An honest abstraction:

- uses value semantics where possible,

- uses `unique_ptr` for ownership transfer,
- uses `shared_ptr` only where shared lifetime is a requirement,
- documents the lifetime model explicitly.

E) Container Abstractions That Leak Allocation and Locality

Returning containers by value is safe, but can be expensive. Returning views is cheap, but leaks lifetime. The correct choice depends on:

- frequency of calls,
- data size distribution,
- whether the caller needs stable storage,
- thread-safety and invalidation rules.

29.2.4 4) How Leaky Abstractions Create Long-Term Design Damage

A) Hidden Constraints Become Tribal Knowledge

When users must know internal details to use an API correctly, the framework becomes dependent on undocumented human memory.

B) Debugging Becomes Cross-Layer Archaeology

Failures require stepping through abstraction layers that were not designed to align with underlying causality.

C) Performance “Surprises” Become Normal

If performance costs are hidden, users learn to distrust the abstraction and bypass it, fragmenting the architecture.

D) Portability Promises Collapse

Every platform difference forces either a patchwork of special cases or a rewrite.

29.2.5 5) How to Detect Leaky Abstractions Early

A) Ask: “What Must the User Know That the API Does Not Say?”

If correct usage requires unstated knowledge about:

- thread-safety,
- lifetimes,
- performance,
- ordering,

then the abstraction is leaking.

B) Look for “Works in Tests, Fails in Production”

Leaks often appear only under:

- high concurrency,
- overload and backpressure,
- partial failures,
- slow I/O and timeouts.

C) Look for "Everyone Wraps It Again"

If every team builds a wrapper around the framework API, it likely lacks the real control points that users need.

29.2.6 6) Designing Non-Leaky (Honest) Abstractions

Principle 1: Make the Contract Explicit

Every stable abstraction must state:

- ownership and lifetime,
- threading and reentrancy rules,
- error/failure semantics,
- performance expectations (at least qualitatively),
- shutdown and cancellation behavior.

Principle 2: Separate Hot Path and Convenience Path

Offer:

- a low-level API with explicit control (costs visible),
- a higher-level convenience wrapper built on top.

Two-Level API Sketch

```
struct RawBufferView {
  const std::byte* data;
  std::size_t size;
```

```
};

class Parser {
public:
  // hot path: explicit view, no allocation contract can be stated
  bool parse(RawBufferView in) noexcept;

  // convenience: can allocate, can copy, easier for casual callers
  bool parse(std::string_view in);
};
```

Principle 3: Encode Invariants in Types

If something must be non-null, represent it. If something has a deadline, pass a deadline type. If something must be bounded, pass a bounded buffer type.

Non-Null Wrapper

```
template <class T>
class NotNull {
public:
  explicit NotNull(T* p) : p_(p) { if (!p_) std::abort(); }
  T* get() const noexcept { return p_; }
  T& operator*() const noexcept { return *p_; }
  T* operator->() const noexcept { return p_; }
private:
  T* p_;
};
```

Principle 4: Be Honest About What You Cannot Hide

Some realities must be exposed:

- backpressure and queue bounds,

- memory usage and allocation behavior,
- concurrency model (callbacks thread, ordering),
- cancellation and timeouts.

When exposed, users can reason correctly and build reliable systems.

29.2.7 7) Repairing a Leaky Abstraction in a Long-Lived System

Step 1: Write the Missing Contract

Document what the code already does (threading, lifetimes, errors). This stops the bleeding.

Step 2: Add Diagnostics and Guardrails

- assert invariants in debug builds,
- detect misuse (callbacks after stop),
- add structured error categories.

Step 3: Introduce a New Honest API Alongside the Old

Keep backward compatibility:

- add `V2` APIs with explicit semantics,
- deprecate old ambiguous APIs,
- migrate internal framework code first.

Step 4: Remove or Freeze the Old API at a Major Boundary

Only after a defined deprecation window.

29.2.8 Leaky Abstraction Checklist

- Does correct usage require knowledge not stated in the signature or docs?
- Are lifetimes explicit, or do callers guess when views remain valid?
- Are threading and callback rules explicit and testable?
- Are error categories stable and actionable without parsing strings?
- Are resource limits and backpressure exposed (or enforced)?
- Do users frequently bypass or wrap the API?
- Does the abstraction hide performance costs that matter in production?

29.2.9 Closing Principle: Abstractions Must Be Either Tight or Honest

In Modern C++ systems, you cannot hide reality forever. Hardware, OS semantics, allocation costs, and concurrency rules will eventually surface. A durable framework therefore chooses one of two strategies:

- **tight abstractions:** restrict inputs and enforce invariants strongly, or
- **honest abstractions:** expose the dimensions users must control (lifetimes, threading, backpressure).

Leaky abstractions fail because they try to do neither. They promise simplicity while requiring hidden expertise. Long-lived design replaces that illusion with explicit contracts and disciplined boundaries.

29.3 Hidden Ownership

Hidden ownership is a design failure where the lifetime of objects and resources is not obvious from the code that uses them. The program still compiles, tests may pass, and performance may look fine at first. But the system becomes fragile over time because developers cannot reliably answer the most important questions:

- Who owns this object?
- Who is responsible for destroying it?
- How long is it guaranteed to live?
- Can it be shared across threads?
- Can it outlive the component that created it?

In Modern C++, hidden ownership is not a style issue; it is a correctness and maintainability issue. Ambiguous lifetimes lead to:

- use-after-free and dangling references,
- memory leaks through unintended retention,
- double deletes or inconsistent cleanup,
- subtle concurrency bugs when lifetimes cross threads,
- inability to refactor because nobody trusts the object graph.

This section explains how hidden ownership appears in C++ designs, why it persists, how to detect it, and how to replace it with explicit, stable ownership contracts suitable for long-lived systems.

29.3.1 1) What Counts as Hidden Ownership?

Ownership is hidden whenever the code's surface does not reveal lifetime responsibility.

Typical Hidden Ownership Signals

- returning raw pointers without stating ownership (owning or observing?),
- storing pointers or references to external objects without lifetime guarantees,
- using `std::shared_ptr` as a default "fix" for lifetime,
- implicit global registries that retain objects,
- callback-based async code that captures references to short-lived data,
- "manager" classes that silently delete objects they did not create,
- containers of polymorphic pointers with unclear destruction policy.

The Resulting Pathology

Engineers begin to treat lifetimes as folklore:

- "this pointer is probably valid here",
- "do not call that after init",
- "shutdown order matters but nobody remembers why".

Long-lived systems cannot rely on folklore.

29.3.2 2) Why Hidden Ownership Happens in C++

A) Raw Pointers Are Ambiguous by Default

A raw pointer can mean:

- owning pointer (must delete),
- observer pointer (do not delete),
- optional pointer (nullable),
- out-parameter (callee writes into it),
- borrowed view with limited lifetime.

The type alone does not communicate which meaning applies.

B) "Convenient" Sharing Masks Real Boundaries

`std::shared_ptr` provides shared lifetime, but if used by default it hides:

- who actually controls destruction,
- whether retention is intentional or accidental,
- whether a component can shut down deterministically.

C) Async and Callback Designs Amplify Lifetime Risk

When work is deferred, references captured today may be used after the referenced object is gone.

D) Registries, Singletons, and Global Services Retain Objects

Global state often becomes a hidden owner. Objects "never die" because they remain registered.

29.3.3 3) Common Hidden Ownership Anti-Patterns

Anti-Pattern A: Returning Raw Pointers With Unclear Policy

```
struct Widget;

class Factory {
public:
  Widget* create(); // owning or observing? unclear
};
```

Fix: Express Ownership in the Type

```
#include <memory>

struct Widget;

class Factory {
public:
  std::unique_ptr<Widget> create(); // clear: caller owns
};
```

Anti-Pattern B: Storing Borrowed References Without a Lifetime Contract

```
class Logger;

class Service {
public:
  void set_logger(Logger& l) { logger_ = &l; } // borrowed, but for how long?
private:
  Logger* logger_ = nullptr; // may dangle
};
```

Fix Options

- require the logger to outlive the service by construction (inject in ctor),
- store `shared_ptr` only if shared lifetime is required,
- store an interface reference with well-defined owner at a higher level.

Constructor Injection (Lifetime by Structure)

```
class Logger;

class Service {
public:
  explicit Service(Logger& l) : logger_(l) {}
private:
  Logger& logger_; // non-null, lifetime guaranteed by architecture
};
```

Anti-Pattern C: "shared_ptr Everywhere"

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

struct Node {
  std::vector<std::shared_ptr<Node>> children; // implicit retention graph
  std::shared_ptr<Node> parent;                // cycle risk
};
```

Problems:

- potential cycles (`parent` must be `weak_ptr`),
- unpredictable destruction time,
- atomic refcount overhead,
- hidden global retention through long-lived owners.

Fix: Use Unique Ownership + Non-Owning Links

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

struct Node {
  std::vector<std::unique_ptr<Node>> children; // tree owns children
  Node* parent = nullptr;                      // observer (non-owning)
};
```

This is explicit: ownership is in `unique_ptr`; observer pointers are non-owning.

Anti-Pattern D: Async Callbacks Capturing References

```
#include <functional>
#include <string>

class Executor {
public:
  void post(std::function<void()> f);
};

void schedule_work(Executor& ex, const std::string& s) {
  ex.post([&] { /* uses s later */ }); // s may be destroyed
}
```

Fix: Capture Ownership Explicitly

```
void schedule_work(Executor& ex, std::string s) {
  ex.post([s = std::move(s)]() mutable {
    /* safe: owns data */
  });
}
```

Anti-Pattern E: Registries as Hidden Owners

```
#include <vector>

struct Handler { virtual ~Handler() = default; };

class Registry {
public:
  void add(Handler* h) { handlers_.push_back(h); } // who deletes?
private:
  std::vector<Handler*> handlers_;
};
```

Fix: Registry Owns Explicitly or Borrows Explicitly

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

struct Handler { virtual ~Handler() = default; };

class OwningRegistry {
public:
  void add(std::unique_ptr<Handler> h) { handlers_.push_back(std::move(h)); }
private:
  std::vector<std::unique_ptr<Handler>> handlers_;
};
```

If borrowing is required, encode the lifetime contract in architecture and use references, not raw pointer ambiguity.

29.3.4 4) Hidden Ownership and Concurrency

Hidden ownership becomes catastrophic with threads:

- an object is destroyed on one thread while another thread still uses it,
- shared ownership hides the true synchronization boundaries,
- weak references fail at runtime under load and become complex.

Rule: Cross-Thread Objects Need Explicit Ownership + Synchronization

If an object crosses threads, decide:

- does it remain owned by one thread and accessed via messages?
- is it shared with locks?
- is it immutable after publication?

Prefer Message Passing for Mutable State

Instead of sharing mutable objects, keep ownership within a thread and communicate by value or by move.

```
#include <functional>
#include <queue>
#include <mutex>

class Mailbox {
public:
  void post(std::function<void()> f) {
    std::lock_guard<std::mutex> lk(m_);
    q_.push(std::move(f));
  }
private:
  std::mutex m_;
  std::queue<std::function<void()>> q_;
};
```

The point is architectural: state stays where it is owned; work is routed to it.

29.3.5 5) How to Make Ownership Visible (Framework-Grade Rules)

Rule 1: Express Ownership in Types

- `T` for values (owned by value),
- `std::unique_ptr<T>` for unique ownership transfer,
- `std::shared_ptr<T>` only for true shared lifetime,
- `std::weak_ptr<T>` only for non-owning links into shared ownership,
- `T&` for required non-null borrowed dependencies,
- `T*` for optional borrowed dependencies (documented).

Rule 2: Make Borrowing Local

Borrowed references and views (`string_view`, `span`) should usually:

- not be stored,
- not escape the scope where they are valid,
- be used as parameters, not as members, unless lifetime is structurally guaranteed.

Rule 3: Avoid Shared Ownership as a Default

Shared ownership is a global decision:

- it changes shutdown semantics,
- it makes memory retention hard to reason about,

- it spreads lifetime responsibility across many call sites.

Use shared ownership only when:

- multiple independent owners truly exist,
- destruction must occur when the last user releases,
- deterministic shutdown is still achievable by design.

Rule 4: Centralize Lifetimes for Long-Lived Objects

Long-lived services (thread pools, schedulers, caches) should be owned by an application root and injected as references or handles.

```
struct AppRoot {
  Logger logger;
  Executor exec;
  Service service{logger, exec}; // lifetimes made obvious
};
```

Rule 5: Treat “Stop” and “Shutdown” as Ownership Operations

A component that owns background work must provide:

- explicit `stop()` / `shutdown()` semantics,
- guarantee that callbacks do not run after stop completes,
- join threads and release resources deterministically.

29.3.6 6) Detecting Hidden Ownership in Existing Code

Audit Questions

- Where are raw pointers stored in members or containers?
- Which APIs return pointers or references? Are they owning or borrowing?
- Which callbacks capture references? Can they outlive captured data?
- Which objects “never die”? Are registries retaining them?
- Do we rely on destruction order between globals/statics?

Smells in Practice

- intermittent crashes on shutdown,
- leaks that disappear when disabling one feature,
- fixes that add `shared_ptr` rather than clarifying ownership,
- unclear invariants around initialization and teardown order.

29.3.7 7) Refactoring Strategy: From Hidden to Explicit Ownership

Step 1: Classify Every Pointer as Owning or Non-Owning

Document and encode it in the type.

Step 2: Replace Owning Raw Pointers With `unique_ptr`

This makes ownership transfer explicit and deterministic.

Step 3: Replace Stored Borrowed Pointers With Structural Lifetimes

Inject dependencies by reference via constructors. If optional, keep pointer but document policy.

Step 4: Limit `shared_ptr` to True Shared Lifetime

When used:

- avoid cycles,
- keep graphs shallow,
- ensure shutdown can still revoke references or stop background work.

Step 5: Make Async Capture Ownership Explicit

Capture by value for deferred execution. If sharing is needed, capture `shared_ptr` intentionally, not accidentally.

29.3.8 Hidden Ownership Checklist

- Every resource and object has a clearly identifiable owner.
- Owning relationships are encoded in types, not in comments.
- Borrowed references/views do not outlive their source.
- Shared ownership is rare and justified; cycles are prevented.
- Registries either own objects explicitly or borrow via structural lifetimes.
- Async work does not capture dangling references; shutdown is deterministic.

29.3.9 Closing Principle: Ownership Must Be Visible to Be Maintainable

A long-lived C++ system survives refactoring, scaling, and team changes only when ownership is obvious. Hidden ownership turns lifetime into guesswork, and guesswork turns into undefined behavior, memory retention, and shutdown fragility. Modern C++ provides the tools to make ownership explicit; design wisdom is using them to make lifetimes a stable part of architecture rather than a hidden property of implementation.

29.4 Premature Generalization

Premature generalization is the design failure of building a system for hypothetical future requirements rather than the concrete requirements that exist today. It happens when engineers attempt to design "the general solution" early, before the domain has stabilized and before they have enough real use-cases to justify the abstraction. The result is not a more flexible system; it is usually a more fragile one.

In Modern C++, premature generalization is especially costly because the language makes it easy to encode flexibility at many levels:

- template-based policy injection,
- generic containers and type erasure,
- extensive indirection via interfaces,
- feature flags and configuration-driven behavior,
- plugin architectures and registries.

These mechanisms are powerful, but they introduce many degrees of freedom. If the freedom is not needed yet, it becomes overhead and ambiguity.

A long-lived system must evolve safely. Safe evolution is not achieved by guessing the future; it is achieved by creating *stable seams* that let you change the system when the future arrives.

29.4.1 1) What Premature Generalization Looks Like

A system is prematurely generalized when:

- the design supports many variations, but only one is used,
- the general interface is larger than any concrete implementation needs,
- adding a simple feature requires touching a generic framework,
- the abstraction tries to predict domain rules that are not yet understood,
- the configuration surface becomes larger than the product itself.

The most important symptom is **negative leverage**:

- every change requires more work than it should,
- and the extra work does not buy real reuse.

29.4.2 2) Why Premature Generalization Is Common in C++

A) C++ Rewards Clever Abstraction

Templates, metaprogramming, and policy-based design enable elegant general solutions. But elegance is not the same as longevity.

B) Fear of Refactoring Leads to "Design It Right Once" Thinking

Teams generalize early to avoid future refactors, but premature generalization often causes *more* refactoring because the wrong general shape must later be undone.

C) Requirements Are Unstable Early

Early requirements are often incomplete or wrong. Generalizing around unstable requirements locks incorrect assumptions into architecture.

D) Reuse Is Overestimated

Developers assume future reuse will appear naturally. In practice, reuse is expensive and requires strong alignment between multiple consumers.

29.4.3 3) The Real Cost Model: Degrees of Freedom

Generalization adds degrees of freedom:

- more types, more indirection, more configuration,
- more potential states,
- more interaction surfaces between components,
- more test cases to cover.

The cost compounds over time:

- cognitive load for new engineers,
- build time and binary size (templates),
- operational risk (misconfiguration),
- refactoring resistance (framework entanglement).

29.4.4 4) Common Premature Generalization Anti-Patterns in Modern C++

Anti-Pattern A: Abstract Base Classes With Only One Implementation

```
struct IRepository {
  virtual ~IRepository() = default;
  virtual int load(int id) = 0;
  virtual void save(int id, int value) = 0;
};

class SqlRepository : public IRepository {
  // only implementation exists; interface adds cost without benefit
};
```

Prefer Concrete Types Until Variation Is Real

```
class SqlRepository {
public:
  int load(int id);
  void save(int id, int value);
};
```

Introduce `IRepository` only when a second implementation is needed or when you need a stable boundary for independent compilation/ABI.

Anti-Pattern B: Template Policy Explosion for Business Logic

```
template <class AuthPolicy, class RetryPolicy, class CachePolicy, class LogPolicy>
class Service { /* ... */ };
```

This kind of generality can be appropriate for low-level libraries, but in application-level services it often causes:

- slow builds,

- cryptic diagnostics,
- large binaries,
- hard-to-control combinations.

Prefer Runtime Options for Application-Level Variation

```
struct ServiceOptions {
  bool enable_cache = false;
  int  retry_count = 0;
};

class Service {
public:
  explicit Service(ServiceOptions opts);
};
```

Anti-Pattern C: “Universal Value” (Type-Erased Everything)

```
#include <variant>
#include <string>
#include <unordered_map>

using AnyValue = std::variant<int, double, std::string, bool>;

using Record = std::unordered_map<std::string, AnyValue>; // domain erased
```

This looks flexible but often leads to:

- loss of invariants,
- runtime type checks everywhere,
- hard-to-refactor schemas,

- weaker tooling and compile-time verification.

Prefer Domain Types With Explicit Evolution

```
struct UserRecord {
  int id;
  std::string name;
  bool active;
};
```

Anti-Pattern D: Plugin Architecture "Just in Case"

```
class PluginManager {
public:
  void load_all_from_directory(const char* path);
  void register_factory(const char* name, void* (*factory)());
};
```

If you do not actually have:

- third-party extensions,
- independent deployment cadence,
- a stable ABI strategy,

then plugins introduce complexity and security risks with no immediate payoff.

Anti-Pattern E: Over-Configurable Systems

A framework that allows every behavior to be configured creates:

- huge testing matrices,
- operational fragility (misconfiguration),
- unclear defaults and semantics.

29.4.5 5) How Premature Generalization Damages Long-Lived Systems

A) It Locks in Wrong Concepts

The first general model is rarely correct. Once exposed publicly, it becomes difficult to change without breakage.

B) It Replaces Refactoring With Configuration

Instead of improving design, teams add flags and optional paths, producing branching logic and accumulating technical debt.

C) It Reduces Local Reasoning

Behavior becomes dependent on global configuration and hidden registries. Engineers cannot understand a function by reading it.

D) It Slows Delivery and Increases Regression Risk

Every change touches generic infrastructure and must preserve unused features.

29.4.6 6) How to Decide When Generalization Is Justified

Criterion 1: Two or More Real Consumers

An abstraction is justified when you have at least two concrete consumers with meaningful differences that are expected to persist.

Criterion 2: The Variation Axis Is Stable

If you cannot clearly name the axis of variation (e.g., “storage backend”, “transport”, “serialization”), generalization is premature.

Criterion 3: The Common Core Is Large

If the shared part is small and the differences are large, an abstraction may create more coupling than reuse.

Criterion 4: You Can Test the Abstraction Independently

If you cannot test it without the whole system, it is not a stable seam.

Criterion 5: The Abstraction Does Not Create an Explosion of Combinations

If a generic design yields too many combinations of options, you will not be able to validate it in a long-lived project.

29.4.7 7) Prefer "Stable Seams" Over "General Solutions"

Instead of building a generalized framework early, build stable seams:

- clear module boundaries,
- narrow interfaces around I/O,
- explicit ownership and lifetimes,
- explicit error contracts,
- testable adapters.

Example: Seam Around Storage Without Premature Abstraction Start with one implementation:

```
class Storage {
public:
  bool put(std::string_view k, std::string_view v);
```

```
  bool get(std::string_view k, std::string& out);
};
```

When a second backend appears, introduce an interface as a seam:

```
struct IStorage {
  virtual ~IStorage() = default;
  virtual bool put(std::string_view k, std::string_view v) = 0;
  virtual bool get(std::string_view k, std::string& out) = 0;
};
```

29.4.8 8) Refactoring Out Premature Generalization

If the codebase is already over-generalized, a safe approach:

Step 1: Identify Unused Variation

List:

- interfaces with one implementation,
- configuration options never used,
- template parameters that never vary,
- extension points with no extensions.

Step 2: Collapse to the Dominant Path

- internalize generic mechanisms,
- remove unused parameters,
- replace indirection with direct calls.

Step 3: Preserve a Seam Where Future Variation Is Plausible

Keep a narrow adapter boundary rather than a broad framework.

Step 4: Introduce Generalization Later (When Real)

When a second consumer appears, generalize with real evidence.

29.4.9 Premature Generalization Checklist

- Do we have at least two real consumers today?
- Can we name the stable axis of variation?
- Does the abstraction reduce total code and total risk, or increase it?
- Is behavior dependent on global registries and configuration?
- Does the design create a combinatorial testing matrix?
- Are invariants lost due to type erasure and dynamic schemas?
- Would a simple seam (module boundary + adapter) be enough for now?

29.4.10 Closing Principle: Generalize Only Under Real Pressure

A long-lived Modern C++ system is not future-proofed by guessing future requirements. It is made durable by:

- keeping the core simple and explicit,
- encoding invariants in types and ownership,
- designing seams that enable refactoring,

- generalizing only when multiple real uses force it.

Premature generalization trades today's clarity for tomorrow's hypothetical flexibility—and usually loses both.

Principles That Actually Survive Reality

30.1 Simplicity Is Not Minimalism

In real systems, **simplicity** is often misunderstood as **minimalism**. Minimalism is the pursuit of fewer lines, fewer classes, fewer files, fewer layers, fewer features. Simplicity, however, is the pursuit of:

- fewer invalid states,
- clearer ownership and responsibility,
- predictable behavior under stress,
- stable boundaries for change,
- local reasoning without hidden context.

A minimalist design can be deceptively small yet operationally complex: hidden constraints, implicit global state, non-local coupling, and sharp edges that only appear in production. A truly simple design may contain *more* structure—more types, more modules, more explicit contracts—because that structure *removes ambiguity*.
This section explains the difference in a Modern C++ context, where type safety, RAII, concurrency, performance, and long-lived evolution all force us to prefer *explicitness* over superficial smallness.

30.1.1 1) The Difference: Minimal Surface vs. Minimal Surprise

Minimalism

- Optimize for small code size.
- Reduce visible structure.
- Prefer implicit conventions over explicit contracts.
- Often compresses concerns into fewer constructs.

Simplicity

- Optimize for comprehension and correctness over time.
- Make invariants explicit and enforceable.
- Prefer clear boundaries and ownership.
- Reduce the number of *surprising behaviors*.

Core claim Simplicity minimizes *surprise*. Minimalism minimizes *surface*. In long-lived systems, surprise is the true cost.

30.1.2 2) Why Minimalism Fails in Real C++ Systems

A) Minimalism Often Hides Ownership

A small API that returns raw pointers may look “minimal” but it pushes lifetime complexity onto every caller.

Minimal But Ambiguous

```
struct Widget;

Widget* get_widget(int id); // who owns? who deletes? how long valid?
```

Slightly Larger, Actually Simple

```
#include <memory>

struct Widget;

std::unique_ptr<Widget> make_widget(int id); // explicit ownership transfer
```

The second is not minimalist, but it is simpler: fewer invalid states.

B) Minimalism Creates Hidden Coupling

A single global accessor can collapse code size while creating non-local dependencies.

Minimal but Globally Coupled

```
class Config {
public:
  static Config& instance();
  int get_int(const char* key) const;
};
```

This looks simple until testing, concurrency, initialization order, and multi-tenant runtime appear. Then the hidden global lifetime becomes the complexity.

C) Minimalism Turns Errors Into Folklore

A minimal API that returns `bool` or a generic error string forces callers to guess and invent recovery logic.

Minimal Error Channel

```cpp
bool send_request(const char* payload); // why did it fail?
```

Simple, Stable, Actionable Error Semantics

```cpp
#include <string>
#include <cstdint>

struct Error {
  enum class Kind : std::uint8_t { timeout, transient, validation, internal };
  Kind kind = Kind::internal;
  int  code = 0;          // stable identity for metrics/policy
  std::string message;    // human readable
};

struct Result {
  bool ok = false;
  Error err{};
};

Result send_request(std::string_view payload) noexcept;
```

This increases surface but reduces downstream complexity drastically.

D) Minimalism Breaks Under Concurrency

The most dangerous “minimal” design is one that hides threading rules.

```cpp
class Cache {
public:
  void put(std::string_view k, std::string_view v);
  std::string_view get(std::string_view k) const; // view lifetime? thread-safe?
};
```

A minimal interface is not simple if it cannot express:

- thread-safety guarantees,
- view invalidation rules,
- memory ownership constraints.

30.1.3 3) A More Accurate Definition of Simplicity

In systems that must survive reality, simplicity means:

A) Local Reasoning

A function can be understood without reading 20 other files or knowing hidden global state.

B) Explicit Contracts

Inputs, outputs, and failure semantics are precise, stable, and testable.

C) Explicit Ownership and Lifetimes

Who destroys what is visible in type signatures and architecture.

D) Bounded Resource Behavior

Queues are bounded, timeouts exist, and overload behavior is defined.

E) Predictable Change

The design has seams: you can change one part without cascading changes everywhere.

30.1.4 4) Structural Simplicity: Adding the Right Kind of “More”

A long-lived C++ design is often simpler if it has *more structure* in the right places.

A) More Types (To Remove Invalid States)

Represent invariants explicitly so misuse becomes hard.

Explicit Non-Null Dependency

```
template <class T>
class NotNull {
public:
  explicit NotNull(T* p) : p_(p) { if (!p_) std::abort(); }
  T* get() const noexcept { return p_; }
  T& operator*() const noexcept { return *p_; }
  T* operator->() const noexcept { return p_; }
private:
  T* p_;
};

class Logger;

class Service {
public:
  explicit Service(NotNull<Logger> logger) : logger_(logger) {}
private:
  NotNull<Logger> logger_;
};
```

Yes, more code. Fewer invalid states. Less surprise.

B) More Boundaries (To Reduce Coupling)

A few well-chosen modules reduce cognitive load by isolating domains:

- transport boundary,
- serialization boundary,
- persistence boundary,
- domain boundary,
- orchestration boundary.

C) More Tests (As Design Clarification)

Tests are not just verification; they are executable documentation of contracts. A simple system has contracts that can be tested without complex scaffolding.

D) More Observability (To Reduce Operational Mystery)

Counters and structured logs reduce the need for heroic debugging. Minimalism that omits observability is not simplicity; it is deferred complexity.

30.1.5 5) When Minimalism *Is* Simplicity

Minimalism becomes simplicity only when:

- invariants are naturally enforced by the domain,
- lifetimes are trivially scoped,
- concurrency and distribution are absent,
- performance constraints are forgiving,

- the system is not expected to evolve significantly.

In other words: minimalism works when reality is small. Long-lived systems rarely stay small.

30.1.6 6) Practical Heuristics: How to Choose "Simple" Over "Minimal"

Heuristic 1: Prefer Fewer *States*, Not Fewer *Classes*

If adding a type removes invalid states, it is simplifying.

Heuristic 2: Make the Cost Model Visible

If an operation allocates, blocks, or may throw, the API should make that clear:

- `noexcept` when you promise non-throwing,
- explicit `deadline` parameters for time budgets,
- explicit `span`/`string_view` lifetimes for borrowing.

Heuristic 3: Separate Convenience From Control

Offer:

- a low-level API with explicit control and clear costs,
- a convenience wrapper for common paths.

Heuristic 4: Do Not Hide Overload Behavior

Define backpressure and bounds. Minimal APIs that silently queue forever are complex in production.

Heuristic 5: Simplicity Requires "Boring" Upgrade Paths

A simple framework has:

- stable public surfaces,
- explicit versioning and deprecations,
- predictable migration steps.

30.1.7 7) A C++ Case Study Pattern: "Small API" vs "Simple System"

A small API can be a trap:

- one `Manager` class that does everything,
- one global service locator,
- one "universal" container of `variant`/`any`.

A simple system tends to have:

- a few cohesive types with clear roles,
- explicit construction order (composition root),
- explicit lifetimes and shutdown,
- bounded queues and timeouts,
- testable seams.

30.1.8 Closing Principle: Simplicity Is the Removal of Ambiguity

Simplicity is not a contest for fewer lines. In long-lived Modern C++ systems, simplicity is the deliberate removal of ambiguity:

- ambiguity about ownership,
- ambiguity about errors,
- ambiguity about concurrency,
- ambiguity about performance cost,
- ambiguity about evolution and stability.

Minimalism can hide ambiguity. Simplicity exposes and resolves it. That is why simplicity survives reality.

30.2 Explicit Is Better Than Clever

Modern C++ gives engineers exceptional expressive power: templates, constexpr evaluation, operator overloading, implicit conversions, ADL, type deduction, and sophisticated metaprogramming techniques. This power can produce elegant solutions—but it also makes it easy to write code that is *clever* rather than *clear*. In long-lived systems, cleverness is rarely a net win. The principle that survives reality is:

> **Prefer explicit code and explicit design contracts over clever, implicit, or surprising mechanisms.**

This principle is not anti-intellectual. It is pro-maintainability. A system survives when:

- new engineers can safely change code after reading it once,

- behavior is predictable under stress and evolution,
- correctness does not depend on hidden conventions,
- tools (static analysis, tests, reviews) can reason about the code.

Explicitness turns design intent into something the compiler, the reviewer, and the future maintainer can all verify.

30.2.1 1) What "Explicit" Means in Real C++ Design

Explicitness is not verbosity for its own sake. It is the deliberate act of making these properties visible:

- **Ownership:** who owns and destroys resources?
- **Lifetime:** how long is a reference or view valid?
- **Cost:** does this allocate, block, throw, or synchronize?
- **Contract:** what are valid inputs and guaranteed outputs?
- **Concurrency:** which thread is this called on, and is it safe?
- **Intent:** why does the code exist, not just how it works?

Cleverness often hides at least one of these.

30.2.2 2) Why Clever Code Fails Over Time

A) Clever Code Is Hard to Modify Safely

Clever constructs are optimized for the author's mental model at a point in time. Maintainers must reconstruct that mental model before making changes.

B) Clever Code Defeats Reviews

Reviewers cannot reliably detect correctness issues when behavior is implicit or non-local (ADL, operator tricks, SFINAE-heavy overload sets).

C) Clever Code Is Fragile Under Evolution

Subtle interactions break when:

- a new overload is added,
- a template constraint changes,
- a type becomes movable-only,
- a new header introduces ADL candidates,
- a refactor changes an include order.

D) Clever Code Often Masks Undefined Behavior Risk

Hidden lifetimes, aliasing assumptions, and concurrency subtleties can yield UB under optimization or new compilers.

30.2.3 3) Explicitness in the Language: Use the Features That Prevent Surprise

A) `explicit` Constructors to Prevent Implicit Conversions

Clever (Implicit) Conversion Surprise

```
struct Timeout {
  Timeout(int ms) : ms(ms) {}
  int ms;
```

```
};

void set_timeout(Timeout t);

set_timeout(5); // silently constructs Timeout{5}
```

Explicit (Intent Visible)

```
struct Timeout {
  explicit Timeout(int ms) : ms(ms) {}
  int ms;
};

void set_timeout(Timeout t);

set_timeout(Timeout{5});
```

B) Avoid User-Defined Conversions Unless They Are Unavoidable

Implicit conversions create non-local behavior. Prefer named functions:

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

Bytes bytes(std::size_t n) { return Bytes{n}; }
```

C) Prefer Named Operations Over Operator Overloading for Non-Obvious Semantics

Operator overloading is appropriate when it matches mathematical intuition and is unsurprising. It becomes cleverness when it encodes domain-specific operations with hidden costs.

Prefer

```
price = price.add_tax(rate);
```

over a surprising `operator+` that performs I/O or lookup.

D) Make Move/Copy Semantics Explicit at Boundaries

C++ move semantics are powerful but can become "clever" when ownership transfer is implicit.

Explicit Ownership Transfer

```
#include <string>
#include <utility>

struct Request {
  std::string payload;
};

void enqueue(Request r); // by value => caller may move

void on_input(Request r) {
  enqueue(std::move(r)); // explicit transfer
}
```

E) Prefer `noexcept` Where You Intend Non-Throwing Contracts

A non-throwing API is a contract. Make it visible:

```
bool try_send(std::string_view payload) noexcept;
```

Avoid clever reliance on "it probably won't throw".

30.2.4 4) Explicitness in Ownership and Lifetimes

Hidden ownership is one of the most expensive sources of defects in C++ systems.

A) Use `unique_ptr` for Ownership Transfer

```
#include <memory>

struct Task;

std::unique_ptr<Task> make_task(); // caller owns
```

B) Use References for Required Dependencies

Constructor injection makes the lifetime requirement structural:

```
struct Logger;

class Service {
public:
  explicit Service(Logger& log) : log_(log) {}
private:
  Logger& log_;
};
```

C) Use Views Only for Parameters, Not Stored State

`string_view` and `span` are explicit about non-ownership, but storing them is often a lifetime trap.

```
#include <string_view>

void log_line(std::string_view s); // good: used during the call
```

If it must be stored, copy into an owning type.

30.2.5 5) Explicitness in Error Handling and Contracts

Clever error handling hides what can fail and how.

A) Prefer Structured Errors Over Strings

```
#include <cstdint>
#include <string>

struct Error {
  enum class Kind : std::uint8_t { timeout, transient, validation, not_found, internal };
  Kind kind = Kind::internal;
  int  code = 0;
  std::string message;
};
```

B) Make Preconditions Visible and Enforced

Preconditions can be expressed through:

- types (non-null wrapper, bounded integer),
- assertions in debug builds,
- explicit validation at boundaries.

Non-Null Wrapper

```
template <class T>
class NotNull {
public:
  explicit NotNull(T* p) : p_(p) { if (!p_) std::abort(); }
  T* get() const noexcept { return p_; }
private:
  T* p_;
};
```

C) Avoid Clever Control Flow

Exceptions are not inherently clever; they become clever when used for normal control flow. Likewise, `goto`-like macros or hidden early returns can obscure invariants. Prefer direct, visible flow with explicit error propagation.

30.2.6 6) Explicitness in Concurrency

Concurrency is where cleverness becomes operational risk.

A) Make Threading Models Explicit

If callbacks may run concurrently, state it and encode it by design. If a component is single-thread confined, express it and enforce it.

B) Prefer Message Passing for Shared Mutable State

Rather than clever lock-free tricks or ad-hoc synchronization, keep ownership in one place and route work explicitly.

C) Prefer RAII for Locks and Resources

```
#include <mutex>

class Counter {
public:
  void inc() {
    std::lock_guard<std::mutex> lk(m_);
    ++value_;
  }
private:
  std::mutex m_;
```

```
  int value_ = 0;
};
```

This is not the fastest possible design, but it is explicit, safe, and refactorable. Optimize only when measurement demands it.

30.2.7 7) Explicitness in Templates and Generic Code

Templates become “clever” when the intent is hidden behind metaprogramming.

A) Use Concepts to Make Requirements Explicit

```
#include <concepts>

template <class T>
concept Hashable = requires(T a) {
  { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};
```

Concepts replace implicit SFINAE tricks with explicit requirements.

B) Prefer Simple Overload Sets

Avoid complicated ADL and enable_if combinations when a clear overload or a tag type can express intent.

C) Keep Generic Code Localized

If the generic machinery is not the product, keep it small and near its use. Do not let meta-architecture become the architecture.

30.2.8 8) Explicitness in Architecture: Make the “Seams” Visible

At the system level, explicitness means:

- explicit module boundaries,
- explicit dependency directions,
- explicit initialization/shutdown order,
- explicit extension points and versioning.

Composition Root Centralize object graph creation so ownership is visible:

```
struct App {
  Logger logger;
  Executor exec;
  Service service{logger, exec}; // lifetimes and dependencies are explicit
};
```

30.2.9 9) A Practical Checklist: Replace Cleverness With Explicitness

When reviewing or designing code, ask:

- Can a new engineer predict what this does without knowing hidden rules?
- Is ownership visible in types and construction?
- Are lifetimes safe without “remembering” constraints?
- Are errors structured and actionable without parsing strings?
- Are threading rules explicit and testable?
- Does this rely on ADL, implicit conversions, or macro tricks?

- If this breaks, can we debug it without stepping through five layers of indirection?

If the answer is “no,” the code may be clever, but it is not robust.

30.2.10 Closing Principle: Explicitness Scales With Time and Teams

Cleverness can impress in the moment. Explicitness survives for years:

- it makes contracts visible,
- it makes ownership and costs predictable,
- it makes code review and refactoring safe,
- it makes behavior stable under evolution.

In Modern C++ long-lived systems, explicit is not “less advanced.” It is the most advanced form of engineering discipline: writing code that remains correct and understandable long after the original author is gone.

30.3 Design for Humans First

A long-lived software system is not sustained by compilers, CPUs, or frameworks. It is sustained by **humans**: engineers who read, review, debug, refactor, extend, and operate the system for years. Hardware evolves, toolchains change, requirements shift, and teams rotate. The code that survives is the code that remains understandable, teachable, and safely changeable.
Therefore, a principle that survives reality is:

> **Design for humans first. Performance and elegance matter, but human comprehension is the dominant constraint in long-lived systems.**

This is not an argument for slow code or weak engineering. It is an argument for choosing designs that maximize *correctness and velocity over time*. In Modern C++, where undefined behavior, lifetime errors, concurrency mistakes, and complex compilation models exist, designing for humans is a primary safety mechanism.

30.3.1 1) What "Design for Humans" Means in Practice

Designing for humans means optimizing for:

- **readability:** can a competent engineer understand the intent quickly?
- **predictability:** do similar things look and behave similarly?
- **local reasoning:** can a change be made without global knowledge?
- **discoverability:** can engineers find the right place to add/change behavior?
- **debuggability:** can failures be diagnosed without archaeology?
- **reviewability:** can reviewers confidently validate correctness?
- **operability:** can the system be deployed, monitored, and safely shut down?

It also means reducing *cognitive load*. Cognitive load is the number of things an engineer must keep in mind to avoid breaking the system. High cognitive load is the direct enemy of longevity.

30.3.2 2) Why Human-Centered Design Is Critical in Modern C++

C++ has characteristics that amplify human error:

- **lifetime complexity:** references, views, raw pointers, aliasing.
- **undefined behavior:** incorrect code may appear to work and later fail under optimization.

- **concurrency hazards:** data races are undefined behavior; ordering is subtle.
- **template complexity:** generic code can become hard to interpret and debug.
- **build coupling:** header inclusion and templates can make change expensive.

Design for humans reduces these failure modes by making intent and contracts explicit.

30.3.3 3) Human-First Principles That Scale

A) Make Invariants Visible

If something must always be true, encode it:

- in types,
- in constructors,
- in explicit validation at boundaries.

Example: Non-null Dependency

```
#include <cstdlib>

template <class T>
class NotNull {
public:
  explicit NotNull(T* p) : p_(p) { if (!p_) std::abort(); }
  T* get() const noexcept { return p_; }
  T& operator*() const noexcept { return *p_; }
  T* operator->() const noexcept { return p_; }
private:
  T* p_;
};
```

```
struct Logger;

class Service {
public:
  explicit Service(NotNull<Logger> logger) : logger_(logger) {}
private:
  NotNull<Logger> logger_;
};
```

This helps humans by removing an entire category of “maybe null” reasoning.

B) Prefer Explicit Ownership and Lifetimes

Humans struggle most with hidden ownership. Use the type system:

- `std::unique_ptr` for ownership transfer,
- references for required non-owning dependencies,
- views (`string_view`, `span`) mainly for parameters,
- `std::shared_ptr` only when shared lifetime is truly required.

Ownership Transfer Is Obvious

```
#include <memory>

struct Task;

std::unique_ptr<Task> make_task(); // caller owns, deterministic destruction
```

C) Optimize for "Obvious Paths"

A human-first API has one obvious way to do the common thing. If there are many ways, engineers will pick inconsistent ones.

Example: One Constructor Path + Named Factory

```
struct ClientOptions { int timeout_ms = 1000; };

class Client {
public:
  static Client create(ClientOptions opts); // explicit and discoverable
};
```

D) Prefer Consistency Over Cleverness

Consistent conventions reduce mental effort:

- naming,
- error strategy,
- threading rules,
- ownership patterns,
- logging/metrics patterns.

The system should feel like one mind wrote it.

E) Reduce Context Switching With Cohesive Modules

Humans understand systems as *maps*. A human-first architecture has:

- clear directories,

- clear module boundaries,
- stable ownership of components.

A good module:

- has one purpose,
- has a small public surface,
- hides details,
- is testable without the whole system.

30.3.4 4) Human-First Error Handling: Make Failure Actionable

Humans can manage failure when it is structured and predictable.

A) Stable Error Categories

```
#include <cstdint>
#include <string>

struct Error {
  enum class Kind : std::uint8_t { validation, timeout, transient, not_found, internal };
  Kind kind = Kind::internal;
  int  code = 0;          // stable identity for policy/metrics
  std::string message;    // human readable
};
```

B) Prefer Fail-Fast at Boundaries, Fail-Safe Internally

At system boundaries (I/O, parsing, user input), validate early and reject invalid states. Internally, maintain invariants so that deep code does not need defensive paranoia.

30.3.5 5) Human-First Concurrency: Make Threading Understandable

A) State the Threading Model

For each component, document and enforce:

- which thread owns state,
- what is safe concurrently,
- callback execution rules,
- shutdown guarantees.

B) Prefer Concurrency Models That Humans Can Reason About

Message passing and ownership confinement often produce fewer mistakes than widespread shared mutable state.

Mailbox Style (Ownership by Thread)

```
#include <functional>
#include <mutex>
#include <queue>

class Mailbox {
public:
  void post(std::function<void()> f) {
    std::lock_guard<std::mutex> lk(m_);
    q_.push(std::move(f));
  }
  bool try_pop(std::function<void()>& out) {
    std::lock_guard<std::mutex> lk(m_);
    if (q_.empty()) return false;
```

```
    out = std::move(q_.front());
    q_.pop();
    return true;
  }
private:
  std::mutex m_;
  std::queue<std::function<void()>> q_;
};
```

This is not the only model, but it is one that humans can explain and test.

30.3.6 6) Human-First Performance: Make the Cost Model Visible

Humans make correct decisions when costs are visible.

A) Separate Convenience APIs from Hot-Path APIs

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

class Parser {
public:
  bool parse(std::string_view in); // convenience
  bool parse_bytes(const unsigned char* data, std::size_t n) noexcept; // explicit hot path
};
```

B) Avoid Hidden Allocation and Blocking on Critical Paths

If an operation might allocate or block, either:

- make it explicit in naming (`load_from_disk`, `resolve_dns`),
- or document/encode it in contract (timeouts, buffer ownership).

30.3.7 7) Human-First Evolution: Make Change Safe

Humans maintain code. The architecture must make change safe:

- narrow, stable interfaces,
- explicit deprecation windows,
- compatibility tests,
- seams for refactoring.

A) Prefer Additive Change

Within a compatibility window:

- add new APIs rather than mutating semantics of old ones,
- deprecate before removing,
- provide migration paths.

B) Keep a Composition Root

Make construction and lifetimes visible in one place:

```
struct App {
  Logger logger;
  Executor exec;
  Service service{logger, exec};
};
```

This reduces the need for global state and clarifies shutdown order.

30.3.8 8) Human-First Code Review and Team Growth

Design for humans directly supports team scaling:

- reviewers can validate changes quickly,
- new engineers learn the system without tribal knowledge,
- ownership boundaries reduce merge conflicts,
- invariants reduce regressions.

Human-first design uses:

- consistent conventions and style,
- small pull requests,
- tests that reflect contracts,
- observability that turns production into feedback.

30.3.9 9) A Practical Checklist

Before accepting a design, ask:

- Can a new engineer discover where to implement a change?
- Are ownership and lifetimes obvious from types and construction?
- Are error semantics structured and actionable?
- Is the threading model explicit, testable, and enforceable?
- Can we reason locally about correctness, without global context?

- Does the API have one obvious path for common use?
- Are costs visible (allocation, blocking, synchronization)?
- Is shutdown deterministic, or does lifetime leak across components?

If the answers are unclear, the design is not human-first.

30.3.10 Closing Principle: The System That Survives Is the One People Can Maintain

A long-lived C++ system survives not because it is minimal, clever, or perfectly abstract. It survives because it is designed to be understood and safely changed by humans over many years. Human-first design is therefore not a soft preference; it is a core engineering strategy for correctness, velocity, and longevity.

30.4 When to Say "No"

Long-lived systems fail more often from *uncontrolled growth* than from missing features. Each new feature, configuration flag, extension point, or abstraction adds permanent complexity:

- more states to reason about,
- more interactions to test,
- more compatibility promises to maintain,
- more operational failure modes to handle,
- more cognitive load for every future engineer.

Therefore, one of the most practical principles that survives reality is learning when to say **"No"**—not as stubbornness, but as disciplined design governance. "No" protects:

- architectural integrity,
- correctness and safety,
- long-term velocity,
- predictable operations,
- the ability to evolve without rewriting.

In Modern C++, saying "No" is also a safety measure: complexity multiplies lifetime hazards, concurrency risks, and undefined behavior exposure.

30.4.1 1) What "No" Means in Engineering Terms

Saying "No" does not mean rejecting value. It means one of the following deliberate choices:

- **No, not now:** defer until requirements stabilize and evidence exists.
- **No, not here:** place the feature at a different layer (app code, plugin, separate service).
- **No, not like that:** accept the goal, reject the proposed design.
- **No, because it breaks the contract:** reject changes that violate stability guarantees.
- **No, because it is unsafe:** reject changes that introduce UB, race risks, or unclear ownership.

The skill is not refusing; it is choosing the right boundary and preserving long-term correctness.

30.4.2 2) The Dominant Reasons to Say “No”

A) The Feature Adds Degrees of Freedom Without Real Consumers

If a feature adds:

- new configuration matrices,
- new extension points,
- new polymorphic axes,

but has no clear second consumer or real pressure, it is likely premature generalization. “No” protects the system from future debt.

B) The Proposal Hides Ownership or Lifetimes

If the design introduces:

- raw owning pointers,
- shared ownership by default,
- stored views with unclear lifetimes,
- callbacks that can outlive their captured data,

say “No” until ownership is explicit.

Example: Reject Ambiguous Ownership

```
struct Handler;
void register_handler(Handler* h); // who owns? who deletes? can it outlive registry?
```

Accept Only With Explicit Contract

```cpp
#include <memory>

struct Handler;
void register_handler(std::unique_ptr<Handler> h); // registry owns
```

Or, if borrowing is intended:

```cpp
struct Handler;
void register_handler(Handler& h); // requires h outlives registry; architectural rule
```

C) The Feature Breaks Behavioral or API Contracts

A long-lived framework must protect established contracts:

- threading guarantees,
- error semantics,
- shutdown semantics,
- performance invariants (bounded queues, no hidden blocking),
- API/ABI stability promises.

If the change breaks a contract, the correct default is “No” unless a major version boundary or a controlled migration exists.

D) The Feature Is Operationally Dangerous

Some changes increase runtime risk disproportionally:

- unbounded queues,
- implicit retries without budgets,

- “auto” behaviors that hide overload,
- synchronous logging or disk I/O on hot paths,
- new dynamic loading or reflection-like mechanisms.

If you cannot explain failure modes and mitigation, say “No.”

E) The Feature Expands the Public Surface Too Much

Public APIs become commitments. The more surface you publish, the more you must support forever. Say “No” when:

- an API is too specialized,
- the domain is unstable,
- the feature can live in user code or a separate library.

F) The Feature Increases Compile-Time and Coupling Excessively

In Modern C++ large-scale systems, compile-time is a product feature. Say “No” when:

- a design forces templates everywhere,
- public headers pull in heavy dependencies,
- small changes trigger large rebuilds,
- build times become a tax on every engineer.

30.4.3 3) "No" as a Design Tool: Where Features Should Live

A common failure is adding features to the wrong layer. "No" is often correct because the request targets the framework core when it should be:

- **application-level policy** (not framework),
- **extension/plugin** (optional, isolated),
- **adapter layer** (bridge to environment),
- **separate service** (independent scaling and deployment),
- **configuration tooling** (outside runtime).

Example: Framework vs Policy If users want "custom retry rules," do not build a universal retry DSL into the core. Instead:

- offer a small retry hook (seam),
- keep default behavior simple,
- let applications implement sophisticated policies.

30.4.4 4) A Decision Framework: The "No" Checklist

Before accepting a feature into the core framework, require answers to:

A) Evidence

- What real use-cases exist today?
- How many independent consumers need it?
- What concrete pain does it remove?

B) Contract Impact

- Does it change API/ABI?
- Does it change error semantics?
- Does it change threading/shutdown guarantees?
- Does it change performance invariants?

C) Cost Model

- Does it allocate? block? synchronize?
- Does it increase compile time or binary size?
- Does it increase operational complexity?

D) Testability

- Can it be tested in isolation?
- Can it be fuzzed or property-tested?
- Can failures be diagnosed with logs/metrics?

E) Reversibility

- If wrong, can we remove it without breaking users?
- Can it be an internal experiment first?
- Can it be opt-in behind a feature flag?

If these answers are weak, the correct decision is often “No” or “Not now.”

30.4.5 5) Patterns for Saying “No” Without Blocking Progress

Pattern 1: Provide a Seam Instead of a Feature

Instead of adding complex behavior, add a stable seam where user code can plug in.

```
#include <functional>

struct RetryContext { int attempt = 0; };

using ShouldRetry = std::function<bool(const RetryContext&)>;

struct ClientOptions {
  int max_attempts = 3;
  ShouldRetry should_retry; // optional seam; default policy is simple
};
```

This keeps the core small while enabling extension.

Pattern 2: Make It a Separate Library

If a feature is valuable but not universal, ship it as an add-on library:

- core stays stable,
- add-on evolves faster,
- users opt in explicitly.

Pattern 3: Accept the Goal, Reject the Mechanism

Example: “We need faster serialization”. Reject: a complex compile-time reflection framework in the core. Accept: a focused, benchmarked serializer with explicit schema types and an adapter boundary.

Pattern 4: Opt-In Experimental Flags With Expiration

If you must experiment, do it with:

- opt-in flags,
- explicit semantics,
- a planned decision date (keep/remove/graduate),
- no silent default changes within a stable line.

30.4.6 6) The Most Important Places to Say "No" in C++

A) "Just Use shared_ptr Everywhere"

Say "No" unless shared lifetime is truly required. Shared ownership is an architectural decision, not a convenience.

B) "Add Another Configuration Option"

If options multiply, you get a testing explosion and fragile operations. Prefer a small set of coherent modes.

C) "We Need a Plugin System"

Say "No" unless you can commit to:

- ABI strategy,
- security model,
- versioning and deprecation,
- isolation and failure containment.

D) “Make It Generic With Templates”

Say “No” when the genericity is not essential. Generic code is powerful, but it taxes compile time and comprehension.

E) “Add Lock-Free for Speed”

Say “No” until profiling shows contention is the bottleneck and correctness strategy is clear.

30.4.7 7) Saying “No” Is Saying “Yes” to Longevity

Every “No” protects future engineers:

- from debugging fragile abstractions,
- from unpredictable shutdown behavior,
- from unbounded memory growth,
- from versioning nightmares,
- from slow builds and brittle dependencies.

The cost of saying “No” is short-term disappointment. The cost of not saying “No” is long-term stagnation and eventual rewrite.

30.4.8 Closing Principle: Guard the Core, Empower the Edges

A long-lived system stays healthy when the core remains small, explicit, and stable, while flexibility is provided at the edges:

- seams, adapters, and extensions,
- separate libraries for specialized features,

- opt-in experiments with clear contracts.

Knowing when to say “No” is therefore not negativity; it is the essential act of design leadership that keeps C++ systems correct, operable, and evolvable for years.

Appendices

Appendix A: Design Patterns Reinterpreted in Modern C++

Purpose and Context

The classic catalog of design patterns emerged in a historical context where:

- resource management was largely manual,
- language support for generic programming was limited,
- value semantics were difficult to express safely,
- concurrency primitives were low-level and error-prone,
- testing and refactoring tools were immature.

Modern C++ fundamentally reshaped this landscape. As a result, many classical patterns:

- collapse into language features,
- become simpler through value semantics and RAII,
- shift from inheritance-heavy structures to composition,
- move from type hierarchies to callables and sum types.

This appendix reinterprets well-known patterns through the lens of Modern C++ design principles suitable for long-lived, large-scale systems.

A.1 RAII as the Foundational Pattern

RAII is not merely one pattern among many. It is the structural foundation upon which safe Modern C++ design is built.

Intent Tie resource lifetime to object lifetime so that:

- cleanup is deterministic,
- failure paths are safe,
- ownership is explicit,
- APIs resist misuse.

```
#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("fopen failed");
  }

  ~File() { if (f_) std::fclose(f_); }

  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;
  }

private:
  std::FILE* f_ = nullptr;
};
```

RAII eliminates entire classes of bugs and replaces defensive programming with structural guarantees.

A.2 Singleton Reinterpreted as a Composition Root

The classic Singleton pattern attempts to guarantee a single instance and global access. In practice, it introduces:

- hidden dependencies,
- unclear initialization order,
- testing difficulties,
- implicit global state.

Modern Reinterpretation Replace singletons with:

- explicit object graphs,

- constructor injection,
- scoped lifetimes.

```
struct Logger {
  void info(const char* msg) noexcept;
};

struct Database {
  explicit Database(Logger& log) : log_(log) {}
  Logger& log_;
};

struct Application {
  Logger logger;
  Database db{logger};
};
```

The instance is still unique, but its existence and lifetime are explicit and testable.

A.3 Factory Patterns with Explicit Ownership

Factories remain valuable when they:

- enforce construction invariants,
- select implementations,
- return ownership explicitly.

```
#include <memory>

struct Transport {
  virtual ~Transport() = default;
```

```
  virtual void send(const char* msg) = 0;
};

struct TcpTransport : Transport {
  void send(const char* msg) override;
};

std::unique_ptr<Transport> make_transport() {
  return std::make_unique<TcpTransport>();
}
```

Returning `std::unique_ptr` makes ownership and destruction policy unambiguous.

A.4 Strategy Without Inheritance

In many Modern C++ designs, strategies are better expressed as callables rather than class hierarchies.

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

using Comparator = std::function<bool(int,int)>;

void sort_with(std::vector<int>& v, Comparator cmp) {
  std::sort(v.begin(), v.end(), cmp);
}
```

This approach:

- reduces type complexity,
- improves locality,
- avoids fragile inheritance trees.

A.5 Builder Replaced by Validated Options

Many classical Builder patterns can be replaced by:

- immutable option objects,
- single validation point,
- named construction.

```
#include <string>

struct ClientOptions {
  std::string endpoint;
  int timeout_ms = 1000;
};

class Client {
public:
  static Client create(ClientOptions opts) {
    if (opts.endpoint.empty())
      throw std::runtime_error("endpoint required");
    return Client(std::move(opts));
  }

private:
  explicit Client(ClientOptions opts) : opts_(std::move(opts)) {}
  ClientOptions opts_;
};
```

This preserves clarity while avoiding partially constructed states.

A.6 Observer with Explicit Lifetime Control

Classic Observer implementations often fail due to dangling callbacks and unclear ownership.

Modern Reinterpretation Use explicit subscription objects that control lifetime.

```
#include <functional>
#include <vector>

class Event {
public:
  using Callback = std::function<void(int)>;

  class Subscription {
  public:
    Subscription(Event* e, std::size_t i) : e_(e), id_(i) {}
    ~Subscription() { if (e_) e_->unsubscribe(id_); }

    Subscription(const Subscription&) = delete;
    Subscription& operator=(const Subscription&) = delete;

  private:
    Event* e_;
    std::size_t id_;
  };

  Subscription subscribe(Callback cb) {
    subs_.push_back(std::move(cb));
    return Subscription(this, subs_.size() - 1);
  }

private:
  void unsubscribe(std::size_t) {}
  std::vector<Callback> subs_;
};
```

Lifetime is explicit, scoped, and deterministic.

A.7 State Pattern via `std::variant`

Instead of polymorphic state hierarchies, Modern C++ favors closed, explicit state machines.

```
#include <variant>

struct Idle {};
struct Running { int job_id; };
struct Stopped {};

using State = std::variant<Idle, Running, Stopped>;
```

This approach:

- eliminates invalid states,
- makes transitions explicit,
- improves exhaustiveness checking.

A.8 Bridge and PImpl for Stability

When build-time isolation or ABI stability is required, the PImpl idiom remains valid.

```
#include <memory>

class Engine {
public:
  Engine();
  ~Engine();

  Engine(Engine&&) noexcept;
  Engine& operator=(Engine&&) noexcept;
```

```
private:
  struct Impl;
  std::unique_ptr<Impl> impl_;
};
```

Use this pattern deliberately, not reflexively.

Summary

Modern C++ does not discard design patterns; it refines them. Patterns that survive reality:

- encode ownership explicitly,
- reduce inheritance in favor of composition,
- favor values, RAII, and deterministic lifetimes,
- create stable seams rather than abstract frameworks.

Patterns fail when they increase cleverness. They endure when they reduce ambiguity.

Appendix B: Idioms Every C++ Designer Must Master

Purpose and Scope

A C++ “idiom” is a recurring design technique that encodes a principle: ownership, invariants, error boundaries, performance constraints, build hygiene, or extensibility. In long-lived systems, idioms are not stylistic preferences. They are **survivability tools**. They reduce ambiguity, eliminate invalid states, and make behavior predictable under maintenance, refactoring, concurrency, and evolving requirements.

This appendix collects idioms that every C++ designer should master for production-grade, long-lived systems. The selection emphasizes:

- explicit ownership and lifetime,
- deterministic resource management,
- stable interfaces and boundaries,
- safe generic programming,
- testability and observability,
- performance realism (without premature cleverness).

B.1 RAII Everywhere (Not Only for Memory)

Core idea Bind acquisition and release of *any* resource to object lifetime:

- file descriptors and handles,
- mutexes and locks,
- sockets and connections,
- transactions,
- temporary state changes (scoped flags),
- subscriptions and callbacks.

```
#include <mutex>

class GuardedCounter {
public:
  void inc() {
    std::lock_guard<std::mutex> lk(m_);
    ++value_;
```

```
  }
  int get() const {
    std::lock_guard<std::mutex> lk(m_);
    return value_;
  }
private:
  mutable std::mutex m_;
  int value_ = 0;
};
```

Design rule If a resource has a release function, the release must be **automatic**, **deterministic**, and **exception-safe**.

B.2 Rule of Zero (Prefer It), Rule of Five (When You Must)

Rule of Zero If a class owns resources, prefer using standard library types that already manage lifetimes. Then the class needs no custom destructor/copy/move.

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

class Report {
public:
  void add_line(std::string s) { lines_.push_back(std::move(s)); }
private:
  std::vector<std::string> lines_; // owns its memory; Rule of Zero
};
```

Rule of Five If you manage a raw resource directly, you must define or delete:

- destructor,
- copy constructor,

- copy assignment,
- move constructor,
- move assignment.

Design rule Rule of Zero is the default. Rule of Five is a deliberate choice justified by boundary needs.

B.3 Make Ownership Explicit: `unique_ptr`, `shared_ptr`, References, Views

A durable ownership model

- `T` (value): owns itself; simplest and safest.
- `std::unique_ptr<T>`: unique ownership transfer.
- `std::shared_ptr<T>`: shared lifetime *only when required*.
- `T&`: required non-owning dependency with structural lifetime.
- `T*`: optional non-owning dependency (policy must be explicit).
- `std::string_view`/`std::span`: borrowed views; usually *parameters only*.

Avoid hidden ownership If an API returns a raw pointer, the ownership rule must be encoded elsewhere. Prefer encoding it in the type.

```
#include <memory>

struct Widget { int id; };
```

```
std::unique_ptr<Widget> make_widget(int id) {
  return std::make_unique<Widget>(Widget{id});
}
```

B.4 Non-Null Idiom (Eliminate a Class of Bugs)

Problem Nullable pointers create repeated checks, hidden assumptions, and failure paths.

Solution Represent "must not be null" as a type-level constraint.

```
#include <cstdlib>

template <class T>
class NotNull {
public:
  explicit NotNull(T* p) : p_(p) { if (!p_) std::abort(); }
  T* get() const noexcept { return p_; }
  T& operator*() const noexcept { return *p_; }
  T* operator->() const noexcept { return p_; }
private:
  T* p_;
};
```

Design rule When null is not a valid state, do not represent it.

B.5 Prefer Value Semantics (Then Add Indirection Only With Evidence)

Why value semantics survive Values make:

- ownership obvious,
- copying/moving explicit,

- tests easier,
- concurrency safer (copy isolates state).

```
#include <string>

struct User {
  std::string name;
  int level = 0;
};
```

When not Use indirection when:

- polymorphism is required,
- object is large and copied frequently,
- stable addresses are required,
- sharing is semantically required.

B.6 Make Construction Enforce Invariants

Principle Objects should be born valid. If validity depends on runtime checks, validate once at construction and keep class invariants strong.

```
#include <string>
#include <stdexcept>

class Endpoint {
public:
  explicit Endpoint(std::string host) : host_(std::move(host)) {
    if (host_.empty()) throw std::runtime_error("host required");
  }
```

```
  const std::string& host() const noexcept { return host_; }
private:
  std::string host_;
};
```

Design rule Push validation to boundaries. Keep internal code free from repeated checks.

B.7 Strong Types for Units, IDs, and Domain Concepts

Problem Using `int` for everything creates accidental mixing of concepts.

Solution Wrap domain values in small strong types.

```
#include <cstdint>

struct UserId { std::uint64_t v; };
struct OrderId { std::uint64_t v; };

void load_user(UserId);
void load_order(OrderId);
```

This prevents a whole category of errors without runtime cost.

B.8 PImpl (Compilation Firewall and ABI Boundary)

Intent Hide implementation details to:

- reduce rebuilds when internals change,
- keep heavy dependencies out of headers,
- support stable ABI boundaries where necessary.

```
#include <memory>
#include <string_view>

class Engine {
public:
  Engine();
  ~Engine();
  Engine(Engine&&) noexcept;
  Engine& operator=(Engine&&) noexcept;

  Engine(const Engine&) = delete;
  Engine& operator=(const Engine&) = delete;

  void process(std::string_view input);

private:
  struct Impl;
  std::unique_ptr<Impl> impl_;
};
```

Design rule Use PImpl for boundaries and large frameworks. Avoid it for tiny types where indirection dominates.

B.9 The Composition Root (The Anti-Singleton Idiom)

Problem Global access hides dependencies and harms testing.

Solution Centralize wiring in one place (composition root). Inject dependencies explicitly.

```
struct Logger { void info(const char*) noexcept; };
struct Executor { void post(void(*fn)()); };
```

```
struct Service {
  Service(Logger& log, Executor& ex) : log_(log), ex_(ex) {}
  Logger& log_;
  Executor& ex_;
};

struct App {
  Logger log;
  Executor ex;
  Service svc{log, ex};
};
```

Design rule If you cannot see lifetimes and dependencies, the system will not scale in humans.

B.10 Narrow Interfaces and "Pass by View, Store by Value"

Core rule

- Accept data as a view (`string_view`, `span`) when you only need it during the call.
- Store owned data by value (`std::string`, `std::vector`) when it must outlive the call.

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

class LogLine {
public:
  explicit LogLine(std::string_view s) : text_(s) {} // stores by value
  const std::string& text() const noexcept { return text_; }
private:
  std::string text_;
};
```

Design rule Views are borrowed. Borrowing should be local and short-lived.

B.11 Prefer `constexpr` for Invariants and Compile-Time Validation

Intent Move stable rules and invariants to compile time when possible:

- eliminate runtime checks,
- improve correctness,
- document design constraints in code.

```
#include <array>
#include <cstddef>

constexpr std::size_t max_users = 1024;

constexpr bool is_power_of_two(std::size_t n) noexcept {
  return n && ((n & (n - 1)) == 0);
}

static_assert(is_power_of_two(1024), "must be power of two");
```

B.12 Explicit Error Semantics (Avoid Stringly-Typed Failure)

Goal Errors should be:

- structured,
- stable,
- actionable without parsing strings.

```
#include <cstdint>
#include <string>
```

```cpp
struct Error {
  enum class Kind : std::uint8_t { validation, timeout, transient, not_found, internal };
  Kind kind = Kind::internal;
  int  code = 0;
  std::string message;
};

template <class T>
struct Result {
  T value{};
  Error err{};
  bool ok = false;
};
```

Design rule A caller must know whether to retry, fail, or fix input without reading logs.

B.13 “noexcept” as a Contract (Not a Hint)

Meaning `noexcept` is a contract:

- it communicates intent,
- it enables optimizations,
- it stabilizes behavior under failure paths.

```cpp
bool try_push(int v) noexcept;
```

Design rule Use `noexcept` for operations that must not fail by throwing. Pair it with explicit error return.

B.14 The "Two-Level API" Idiom: Convenience and Control

Why it survives Long-lived systems need:

- an easy path for common usage,
- an explicit path for performance/low-level control.

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

class Parser {
public:
  bool parse(std::string_view in);                // convenience
  bool parse_bytes(const unsigned char* p,
                   std::size_t n) noexcept;       // explicit hot path
};
```

Design rule Do not force every caller into a single abstraction level.

B.15 "Make It Testable" Idioms

Testability is achieved by design:

- inject dependencies instead of creating them internally,
- hide I/O behind narrow seams,
- keep pure logic separate from side effects,
- make time and randomness injectable.

```
struct Clock {
  long now_ms() const noexcept;
```

```
};

class RateLimiter {
public:
  RateLimiter(Clock& clock, int max_per_sec)
    : clock_(clock), max_(max_per_sec) {}

private:
  Clock& clock_;
  int max_;
};
```

Design rule If you cannot test a component without real I/O, it is likely missing a seam.

B.16 Concurrency Idiom: Confine Ownership, Communicate by Messages

Principle Shared mutable state is difficult for humans. Confinement reduces complexity:

- one thread owns the state,
- other threads send messages/commands,
- ownership stays explicit.

```
#include <functional>
#include <mutex>
#include <queue>

class Mailbox {
public:
  void post(std::function<void()> f) {
    std::lock_guard<std::mutex> lk(m_);
    q_.push(std::move(f));
```

```
  }
  bool try_pop(std::function<void()>& out) {
    std::lock_guard<std::mutex> lk(m_);
    if (q_.empty()) return false;
    out = std::move(q_.front());
    q_.pop();
    return true;
  }
private:
  std::mutex m_;
  std::queue<std::function<void()>> q_;
};
```

Design rule Prefer designs that humans can explain without hand-waving about memory order.

B.17 Summary: The Idioms That Keep Systems Alive

The idioms that survive in long-lived C++ systems share a common theme:

- eliminate invalid states,
- make ownership and lifetimes explicit,
- reduce non-local coupling,
- keep interfaces narrow and stable,
- separate convenience from control,
- enforce invariants early and systematically,
- keep concurrency understandable.

Mastering these idioms is not about writing “modern” code for fashion. It is about building systems that remain correct, maintainable, and evolvable after years of growth, team changes, toolchain upgrades, and real-world operational pressure.

Appendix C: Checklist for Design Reviews

Purpose

Design reviews exist to prevent architectural debt before it becomes structural. Code reviews catch local defects; design reviews catch **system failures**:

- unclear ownership and lifetime rules,
- unstable boundaries and dependency cycles,
- fragile concurrency models,
- poor error semantics and weak invariants,
- hidden performance costs,
- hard-to-test designs,
- operational risks (observability gaps, unsafe shutdown).

This appendix provides a practical, systematic checklist for reviewing designs in Modern C++ projects intended to live for many years. The checklist is structured so reviewers can evaluate:

- **intent and scope**,
- **architecture and boundaries**,
- **APIs and contracts**,

- **ownership and lifetime**,
- **error strategy**,
- **concurrency model**,
- **performance cost model**,
- **testability and verification**,
- **operability and evolution**.

How to Use This Checklist

- Use it at **design time**, before major implementation.
- Ask reviewers to explicitly record **risks**, **assumptions**, and **open questions**.
- Require that each major subsystem has:
  - an explicit ownership model,
  - a concurrency story,
  - a failure story,
  - an operability story,
  - a versioning/evolution story.
- Treat “not specified” as a design defect for long-lived systems.

C.1 Intent, Scope, and Success Criteria

- Is the design goal stated as **one sentence** that a new engineer can repeat?

- Is the scope clear? What is explicitly **out of scope**?
- Are the primary use-cases enumerated (not hypothetical)?
- Are success metrics defined (latency, throughput, memory, uptime, correctness)?
- What constraints exist (platform, toolchain, ABI, backward compatibility, certification)?
- What assumptions are made about environment (hardware, network, storage, load shape)?

Red Flags

- "We will optimize later" with no performance budget.
- "We will add observability later" with no operational plan.
- "We may need plugins" with no ABI/versioning strategy.

C.2 Architecture, Decomposition, and Boundaries

- Is the system decomposed into cohesive modules with clear responsibilities?
- Are boundaries aligned with **change frequency** (volatile code isolated)?
- Are dependencies directed (no cycles)? Who may depend on whom?
- Are external integrations isolated behind adapters (I/O at the edges)?
- Is there a composition root showing construction order and lifetimes?
- Are build-time boundaries managed (headers, dependencies, transitive includes)?

Red Flags

- A “core” module that depends on everything.
- Cross-cutting globals or service locators.
- Domain logic mixed with networking/storage details.

C.3 Public API Design and Contracts

- Are APIs narrow, cohesive, and stable?
- Are responsibilities obvious from naming and type signatures?
- Do functions specify:
  - preconditions (what must hold before call),
  - postconditions (what is guaranteed after call),
  - invariants (what always holds for the object),
  - failure modes (what can go wrong and how represented)?
- Do APIs expose hidden costs (allocation, blocking, synchronization)?
- Are cancellation and deadlines explicit for operations that may block?

API Smell Checklist

- Returning raw pointers without ownership meaning.
- “God” interfaces with too many responsibilities.
- Implicit conversions that hide behavior.
- Configuration flags that create many runtime modes.

C.4 Ownership, Lifetimes, and Resource Management

- Can every significant object answer:
  - Who owns it?
  - Who destroys it?
  - How long does it live?
  - Can it be shared across threads?
- Is ownership encoded in types (`unique_ptr`, `shared_ptr`, references)?
- Are borrowed views (`string_view`, `span`) used only when lifetimes are local and safe?
- Are resources managed via RAII (no manual cleanup paths)?
- Is shutdown deterministic for background work (threads, timers, callbacks)?

Lifetime Hazard Questions

- Are callbacks capturing references that may dangle?
- Are registries holding objects longer than intended?
- Is shared ownership used as a default rather than a necessity?
- Are there cycles in `shared_ptr` graphs?

C.5 Error Strategy and Failure Semantics

- Is there a documented error strategy:
  - exceptions vs error returns,
  - recoverable vs unrecoverable failures,

  - retryable vs non-retryable classes,
  - boundary validation rules?

- Are errors structured (kind/code/message) rather than stringly-typed?
- Are failures observable (logged/metriced) without leaking sensitive data?
- Are invariants protected (fail-fast when violated)?
- Are partial-failure behaviors defined (degraded modes, backoff, circuit breaking)?

Red Flags

- `bool` return with no diagnostic path.
- Exceptions used for normal control flow.
- Silent failure (ignored errors).

C.6 Concurrency Model and Thread Safety

- What is the concurrency model (thread-per-request, event loop, worker pool, actor/message passing)?
- Which objects are thread-confined vs shared?
- If shared:
  - what synchronization is used,
  - what invariants does it protect,
  - how is contention measured?

- Are data races impossible by construction (ownership confinement) or by discipline (locks)?
- Are callbacks executed on defined threads? Is reentrancy defined?
- Is cancellation safe and deterministic?

Concurrency Red Flags

- "Mostly thread-safe" without a definition.
- Shared state accessed through raw pointers with unclear lifetime.
- Lock-free introduced without proof of necessity and correctness strategy.

C.7 Performance and Cost Model

- Are performance budgets stated (latency p99, throughput, memory)?
- Are the hot paths identified?
- Does the design avoid hidden allocations on hot paths?
- Are queues bounded? What is overload/backpressure behavior?
- Are timeouts/deadlines present for I/O operations?
- Are data structures chosen based on access patterns?
- Are compile times and incremental builds considered (templates, headers)?

Performance Red Flags

- Unbounded buffering or queues.
- Excessive dynamic allocation in inner loops.
- Premature “optimization” that complicates correctness.

C.8 Testability and Verification Strategy

- Can the design be unit-tested without real I/O?
- Are dependencies injectable (composition root)?
- Are pure computations separated from side effects?
- Are contracts testable (invariants, preconditions, error cases)?
- Is the test pyramid intentional:
  - unit tests for logic,
  - integration tests for boundaries,
  - end-to-end tests for workflows?
- Are property-based tests or fuzzing relevant for parsers, protocols, or state machines?
- Is regression testing planned for refactors?

Testing Red Flags

- Hard-coded globals (clock, RNG, filesystem).
- Cannot test without a database/network.
- Mocks required for everything because boundaries are unclear.

C.9 Observability, Debuggability, and Operations

- What telemetry exists:
  - logs (structured),
  - metrics (counters, histograms),
  - traces (request correlation)?
- Are error paths observable (with stable codes)?
- Can you diagnose:
  - timeouts,
  - overload,
  - memory growth,
  - thread pool exhaustion,
  - retry storms?
- Are startup and shutdown semantics defined?
- Can the system degrade gracefully? What is the policy?

Operational Red Flags

- Logs without correlation IDs.
- Failures without metrics.
- Shutdown that relies on destructor order of globals.

C.10 Evolution, Compatibility, and Technical Debt Control

- Is the design safe to refactor (clear seams, small modules)?
- Are public APIs versioned (even if only internally)?
- Is there a deprecation policy (how long, how communicated)?
- Are migrations supported with clear steps?
- Is backward compatibility required? If yes, what exactly is guaranteed?
- Is there a plan for removing experimental features?
- Are dependencies pinned and upgrade strategy defined (toolchain, third-party libs)?

Evolution Red Flags

- Public surfaces that expose internal types deeply.
- "Temporary" flags that never expire.
- No plan for upgrades until the day it breaks.

C.11 Security and Robustness at Boundaries

- Are untrusted inputs validated at boundaries (parsing, decoding, sizes)?
- Are limits defined (payload sizes, recursion depth, queue length)?
- Are timeouts used to avoid resource exhaustion?
- Are secrets handled correctly (no leakage into logs)?
- Is the failure behavior safe under malicious or malformed input?
- Is memory safety preserved under concurrency and lifetime rules?

C.12 Final Review Summary Template

Use this template at the end of a design review:

- **Decision:** approved / approved with changes / rejected
- **Key risks:**
  - Risk 1:
  - Risk 2:
  - Risk 3:
- **Required changes:**
  - Change 1:
  - Change 2:
  - Change 3:
- **Open questions:**
  - Question 1:
  - Question 2:
- **Follow-up validations:**
  - Benchmark:
  - Prototype:
  - Threat review:
  - Test plan:

Summary

A design review checklist is a force multiplier. It turns implicit assumptions into explicit decisions, exposes hidden ownership and concurrency hazards early, and prevents architectural debt from becoming permanent. In long-lived Modern C++ systems, the most expensive failures are not local bugs but design omissions—especially those that hide lifetimes, costs, and operational behavior. This checklist exists to ensure those fundamentals are always reviewed and always specified.

Appendix D: Migration from Legacy C++ to Modern Design

Purpose

Most production C++ systems are not greenfield. They are living artifacts:

- written across multiple language eras (C++98/03, C++11, C++14/17/20/23),
- shaped by old compiler limitations and past performance assumptions,
- constrained by ABI stability, platform support, and release cadence,
- entangled with build systems, third-party dependencies, and operational tooling.

Modernizing such systems is not “rewrite everything using modern features.” It is a controlled engineering process that targets **survivability**:

- reduce undefined behavior exposure,
- make ownership and lifetimes explicit,
- shrink coupling and rebuild impact,
- improve testability and observability,

- raise design clarity so the system can evolve safely.

This appendix presents a pragmatic, stepwise migration strategy from legacy C++ codebases to modern design.

D.1 Migration Philosophy: Evolution, Not Revolution

1) Treat Modernization as Risk Reduction

A modernization plan must be judged by measurable risk reduction:

- fewer crashes, fewer data races,
- fewer memory leaks and use-after-free,
- smaller blast radius of change,
- faster, more reliable builds,
- higher test coverage of stable contracts.

2) Protect Behavior First

Before changing design, preserve behavior:

- write characterization tests,
- freeze external interfaces,
- instrument runtime behavior (logs/metrics) to detect regressions.

3) Migrate Along Seams

Do not modernize by touching everything. Modernize by creating seams:

- isolate I/O,
- isolate dependencies,
- isolate concurrency,
- isolate heavy headers,
- isolate unstable domains.

D.2 Baseline Assessment: What You Must Know First

A modernization effort fails when it starts with tools and syntax rather than a baseline.

Required Baseline Questions

- What compiler versions and standards are in production today?
- What platforms (OS/arch) and toolchains must remain supported?
- What release cadence exists (weekly, monthly, quarterly)?
- What ABI compatibility constraints exist?
- What are the dominant failure modes (crashes, performance, deadlocks)?
- Where are the biggest coupling points (global state, singletons, macros)?
- What are the hottest paths (CPU, allocation, contention)?

Deliverables

- a dependency map (internal modules and third-party),
- a build-time profile (slow targets, heavy headers),
- a runtime profile (hot paths, allocation, contention),
- a risk register (UB hotspots, lifetime hazards, concurrency hazards).

D.3 Stabilize First: Tooling and Safety Nets

Modernization without safety nets becomes fear-driven and slow.

1) Enable Warnings and Treat Them as Design Feedback

- Increase warning levels incrementally.
- Fix high-signal warnings first (uninitialized, narrowing, virtual destructor issues).
- Separate “noise” from true risk by isolating third-party warnings.

2) Add Sanitizers and Runtime Checks in Non-Production Builds

Sanitizers are the fastest way to surface UB and lifetime errors.

3) Add Static Analysis and Formatting Consistency

- Use static analysis for lifetime, concurrency, nullability, and API misuse patterns.
- Enforce formatting to reduce diff noise and make reviews cheaper.

4) Create Characterization Tests

Legacy behavior is often undocumented. Tests capture reality.

```
#include <cassert>

// Minimal example: lock down observed behavior before refactoring.
int legacy_compute(int x);

void characterize() {
  assert(legacy_compute(0) == 42);
  assert(legacy_compute(1) >= 0);
}
```

D.4 The Ownership Migration: From Raw Pointers to Explicit Lifetimes

The highest ROI modernization is clarifying ownership. Many legacy failures are ownership failures.

1) Categorize Pointers

Every pointer in the system should be classified:

- owning pointer,
- non-owning observer,
- optional dependency,
- borrowed view.

2) Replace Owning Raw Pointers

- Use `std::unique_ptr` for unique ownership.

- Use `std::shared_ptr` only when shared lifetime is required by semantics.

Legacy

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

Node* make_node();           // who deletes?
void  destroy_node(Node*);   // manual cleanup required
```

Modern

```
#include <memory>

struct Node { std::unique_ptr<Node> next; };

std::unique_ptr<Node> make_node(); // ownership explicit
```

3) Replace “Maybe Null” With Explicit Contracts

- Use references for required dependencies.
- Use pointers for optional dependencies with explicit checks.

Legacy

```
void run(Logger* log); // is null allowed?
```

Modern

```
void run(Logger& log);        // required
void run(Logger* log) = delete; // forbid accidental null usage
```

4) Views: Pass by View, Store by Value

Legacy code often stores raw pointers to buffers. Prefer:

- accept input as `std::string_view` / `std::span`,
- store persistent data in owning containers.

D.5 The Resource Migration: Replace Manual Cleanup With RAII

Legacy C++ often has:

- `new/delete`,
- `malloc/free`,
- manual `close()` calls,
- custom error-path cleanup.

1) Remove Manual Cleanup Paths

- Wrap resources in objects with deterministic destructors.
- Prefer standard RAII wrappers (`std::lock_guard`, `std::unique_ptr` with deleters).

Legacy

```
FILE* f = fopen(path, "r");
if (!f) return -1;
// ...
fclose(f);
```

Modern RAII Wrapper (Minimal)

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

using FilePtr = std::unique_ptr<std::FILE, int(*)(std::FILE*)>;

FilePtr open_file(const char* path, const char* mode) {
  return FilePtr(std::fopen(path, mode), &std::fclose);
}
```

D.6 Replace Legacy Patterns: Macros, Globals, and Singletons

1) Reduce Macros to the Edges

Macros hide types, evaluation order, and control flow.

- Replace macro constants with `constexpr`.
- Replace macro functions with `inline` functions or templates.
- Replace macro logging with typed wrappers.

Legacy

```
#define MAX_USERS 1024
#define SQR(x) ((x)*(x))
```

Modern

```
constexpr int max_users = 1024;

template <class T>
constexpr T sqr(T x) { return x * x; }
```

2) Replace Globals With Composition Root

Global state makes testing and shutdown brittle. Move wiring into an explicit root.

```
struct App {
  Logger log;
  Database db{log};
  Service svc{db, log};
};
```

D.7 API Modernization: Contracts, Errors, and Semantics

1) Clarify Error Strategy

Legacy APIs often mix:

- error codes,
- exceptions,
- sentinel values,
- global error variables.

Choose a coherent strategy:

- exceptions for unrecoverable or boundary failures (with strict policy),
- error-return results for high-frequency operations or expected failures.

2) Replace Sentinel Values With Stronger Types

Legacy

```
int find_user(const char* name); // -1 means not found
```

Modern

```cpp
#include <optional>
#include <cstdint>

using UserId = std::uint64_t;

std::optional<UserId> find_user(std::string_view name);
```

3) Make Cost Visible

If an operation allocates or blocks, encode that in API naming and contracts.

D.8 Concurrency Migration: From Ad-Hoc Locks to a Model

Legacy concurrency often grows organically:

- locks scattered everywhere,
- unclear ownership,
- “mostly thread-safe” claims,
- deadlocks from lock ordering.

Migration Steps

1. Document current threading assumptions.
2. Classify objects: thread-confined vs shared.
3. Confine ownership where possible (message passing).
4. Centralize synchronization primitives and define lock ordering.
5. Add tests for shutdown and race conditions.

Prefer Ownership Confinement

```
#include <functional>
#include <mutex>
#include <queue>

class Mailbox {
public:
  void post(std::function<void()> f) {
    std::lock_guard<std::mutex> lk(m_);
    q_.push(std::move(f));
  }
  bool try_pop(std::function<void()>& out) {
    std::lock_guard<std::mutex> lk(m_);
    if (q_.empty()) return false;
    out = std::move(q_.front());
    q_.pop();
    return true;
  }
private:
  std::mutex m_;
  std::queue<std::function<void()>> q_;
};
```

This replaces many shared-state hazards with explicit message flow.

D.9 Build and Dependency Modernization

Legacy codebases often suffer from:

- huge transitive includes,
- poor header discipline,
- slow incremental builds,

- hard-to-upgrade third-party libraries.

Practical Steps

- Reduce includes in headers (forward declare where possible).
- Move heavy dependencies into `.cpp` files.
- Use PImpl for large stable interfaces.
- Define layering rules and enforce them.
- Introduce a dependency upgrade policy (pinned versions, tested upgrades).

D.10 Refactoring Strategy: Safe, Incremental, Reversible

A Safe Modernization Loop

1. Add tests and observability for the target area.
2. Isolate the seam (adapter or interface boundary).
3. Convert ownership and RAII in that seam.
4. Simplify the design (remove dead extension points).
5. Measure regressions (performance, memory, correctness).
6. Ship incrementally and repeat.

Avoid the Big Rewrite Trap

A rewrite is justified only when:

- the architecture blocks all safe evolution,

- the cost to refactor exceeds the cost to rebuild,
- you can freeze scope and deliver in small increments,
- you can maintain compatibility during transition.

If these conditions are not met, a rewrite is usually riskier than incremental modernization.

D.11 Modernization Checklist (Practical Gate)

Before considering a subsystem "modernized," verify:

- Ownership is explicit in APIs and members.
- No raw owning pointers remain in the core.
- RAII covers files, locks, threads, and subscriptions.
- Error semantics are consistent and structured.
- Threading model is documented and enforced.
- Observability exists for failures and performance.
- Tests cover contracts and critical behavior.
- Build impact is improved (less coupling, fewer heavy headers).

Summary

Migrating from legacy C++ to modern design is an engineering discipline, not a syntax update. The most effective modernization targets the deepest sources of long-term cost:

- hidden ownership,

- unclear lifetimes,
- ad-hoc concurrency,
- uncontrolled coupling,
- fragile failure semantics.

Modern C++ features are powerful, but they are most valuable when used to make systems **explicit**, **testable**, and **evolvable**. A long-lived system is modern not because it uses new keywords, but because its design reduces ambiguity and makes safe change the default.

Appendix E: Glossary of Design Terms

How to Read This Glossary

This glossary defines design and architecture terms as they are used in this book, with emphasis on Modern C++ and long-lived systems. Each entry aims to be:

- **operational**: useful in real design decisions and reviews,
- **precise**: avoiding vague "buzzword" definitions,
- **C++-aware**: reflecting ownership, lifetimes, concurrency, and performance realities.

A

Abstraction A simplification that hides detail to reduce cognitive load. Good abstractions expose stable concepts and preserve the cost model; bad abstractions hide essential constraints (allocation, blocking, ownership, concurrency) and create surprises.

Abstraction Leakage (Leaky Abstraction) A failure mode where the abstraction cannot fully hide underlying details, forcing callers to reason about internals (e.g., cache behavior, network retries, thread scheduling). Leaks are sometimes unavoidable; the design goal is to make leaks explicit and documented.

Adapter A boundary component that converts one interface or data representation into another. In long-lived systems, adapters isolate volatility: external APIs, OS calls, file formats, protocols, and third-party libraries.

Aggregate A type composed of multiple parts treated as a whole. In C++ design, aggregates often appear as plain structs for configuration, results, and data transfer objects (DTOs), especially when invariants are simple and validation occurs at boundaries.

Allocation The act of obtaining memory dynamically. The design-relevant concept is not "allocation exists," but where and how often it occurs (hot path vs cold path), whether it is bounded, and whether it is visible in the API contract.

API (Application Programming Interface) The public surface of a component: its functions, types, contracts, and observable behavior. An API is a promise. Changing it has a long-term cost that must be managed through versioning and deprecation policies.

API Contract The explicit or implicit rules governing correct usage: preconditions, postconditions, invariants, error semantics, performance expectations, thread-safety guarantees, and lifetime rules.

API Stability The degree to which an API can be relied upon not to change in breaking ways. Stability may apply to source compatibility, binary compatibility, or behavioral compatibility.

Architecture The set of fundamental design decisions that shape the system: decomposition, boundaries, dependency direction, data ownership, concurrency model, error strategy, and operational characteristics. Architecture is what is expensive to change later.

Architectural Boundary A seam where responsibilities, dependencies, and contracts are enforced. Examples: domain vs infrastructure, parsing boundary, persistence boundary, network boundary, UI boundary.

Architectural Debt Design shortcuts that reduce short-term cost but increase long-term cost by making change harder, riskier, or more expensive. Architectural debt is structural; it does not vanish by refactoring a few files.

Assertion A runtime check expressing an invariant or precondition in debug builds. Assertions should fail fast and localize blame. They complement, not replace, type-level design and boundary validation.

Asynchronous Operation An operation that begins and completes later, typically via callbacks, futures, event loops, or task systems. The design obligation is to specify lifetime rules, cancellation rules, execution context (thread), and shutdown semantics.

B

Backpressure A mechanism that prevents unbounded growth under load by slowing producers or rejecting work. Common forms include bounded queues, admission control, rate limiting, and explicit “try_push” APIs.

Behavioral Compatibility A property that the observable behavior of an API remains consistent across versions, even if types or signatures remain unchanged. Behavioral changes without versioning are a common source of production failures.

Binary Compatibility (ABI Compatibility) The ability for separately compiled components to link and run correctly without recompilation. ABI stability influences choices like inline functions, templates in public headers, PImpl, and virtual interfaces.

Boundary Validation Validation performed at the system edges (user input, network, file parsing). The goal is to keep internal subsystems working with validated, well-formed data so invariants remain strong and defensive checks are minimized.

Budget (Performance Budget) A quantifiable target for latency, throughput, memory, and other costs. A design without budgets cannot be evaluated objectively.

Build Hygiene Practices that keep builds fast and predictable: minimizing transitive includes, isolating dependencies, avoiding macro pollution, controlling template expansion in public interfaces, and enforcing layering.

C

Capability A specific permission or authority embodied by an object. Capability-based design passes explicit objects (tokens, handles) to grant access, rather than using global state or implicit privileges.

Change Amplification When a small change triggers many downstream changes (files, modules, builds). High amplification indicates tight coupling and weak boundaries.

Cohesion A measure of how strongly related the responsibilities within a module or type are. High cohesion improves maintainability because changes are localized.

Composition Root The single place where the application object graph is assembled and lifetimes are defined. A composition root reduces hidden dependencies, clarifies ownership, and makes testing easier.

Concept (C++ Concepts) A compile-time predicate expressing requirements on template parameters. Concepts make generic code more explicit, reduce error complexity, and improve maintainability.

Contract A specification of what a component requires and guarantees. Contracts include input validity, lifetime rules, failure modes, thread safety, and cost expectations.

Coupling The degree to which two components depend on each other. Coupling becomes harmful when it is non-local, implicit, or bidirectional.

CQRS (Command Query Responsibility Segregation) A design technique separating read operations (queries) from write operations (commands), often because they have different performance, consistency, or scaling needs.

Critical Section A region of code that must not be executed concurrently because it accesses shared mutable state. The design obligation is to keep critical sections short, explicit, and protected by well-defined synchronization.

D

Data Race Concurrent access to the same memory location where at least one access is a write and there is no synchronization establishing ordering. In C++ this is a correctness failure with undefined behavior at the language level.

Deadline A time bound for an operation. Deadline-driven design forces explicit handling of timeouts, cancellation, and overload behavior.

Deadlock A state where two or more threads wait indefinitely due to cyclic dependencies on resources (often locks). Deadlocks are prevented by strict lock ordering, confinement, or avoiding shared state.

Dependency Direction An architectural constraint specifying which layer or module may depend on which. Correct dependency direction is essential for stable, evolvable systems.

Dependency Injection Providing dependencies from the outside rather than constructing them internally. In C++ this is often simplest via constructors and composition roots, not necessarily via containers.

Design Invariant A property that must always hold for a component to be correct. Invariants should be established at construction and preserved by all operations.

Domain Model The representation of the problem space concepts and rules. A strong domain model reduces accidental complexity and keeps business rules independent of infrastructure.

E

Encapsulation Hiding internal representation so clients depend only on stable contracts. Encapsulation is not merely "private data": it is a boundary discipline.

Error Semantics The standardized meaning of failures: which errors are possible, how they are represented, whether they are retryable, and what recovery is expected.

Event Loop A concurrency model where work is driven by events (I/O readiness, timers) and handled by callbacks or tasks. The design obligations include execution context, reentrancy rules, and fairness.

Exception Safety Guarantees about program state in the presence of exceptions:

- basic guarantee: invariants preserved, no leaks,
- strong guarantee: operation either succeeds or has no effect,
- no-throw guarantee: operation never throws.

Explicit Ownership A design style where ownership and lifetimes are visible in types and interfaces (values, `unique_ptr`, references). This reduces ambiguity and prevents lifetime bugs.

F

Fail-Fast A strategy where the system detects invariant violations early and stops the faulty path immediately (assertion, termination, clear error), rather than continuing in an inconsistent state.

Fault Containment A design goal that limits the blast radius of failures. Examples include process isolation, thread confinement, bulkheads, and strict boundaries that prevent corruption from spreading.

Framework A reusable architecture that controls the flow and calls user code at extension points. Frameworks demand strong stability guarantees, careful versioning, and clear extension contracts.

Function Object (Functor) An object that can be called like a function. In modern design, function objects and lambdas often replace class-based Strategy and Command patterns.

I

Idempotency A property where repeating an operation produces the same effect as performing it once. Essential for retries in distributed systems and robust error recovery.

Immutability Objects whose state does not change after construction. Immutability reduces concurrency complexity and enables safe sharing.

Incremental Build A build mode where only changed components are rebuilt. Design choices that reduce header coupling and template exposure improve incremental build performance.

Interface Segregation Design principle: clients should not depend on methods they do not use. In C++ this often translates into smaller interfaces, narrower APIs, and separation of concerns.

L

Layer A set of components with similar responsibilities and a defined dependency direction. Good layering isolates volatility and makes evolution predictable.

Lifetime The period during which an object remains valid. Lifetime bugs in C++ often arise from non-owning references and callbacks that outlive captured state.

Link-Time Coupling Coupling introduced by binary dependencies and ABI constraints. It influences decisions about virtual interfaces, inline code, and shared library boundaries.

M

Maintainability The ease with which engineers can safely change the system. Maintainability is driven by clarity, boundaries, testability, and a stable cost model.

Message Passing A concurrency approach where state is owned by a single context and other contexts communicate via messages/commands. It reduces shared mutable state and makes reasoning simpler.

Mock A test double that simulates behavior of a dependency. Excessive mocking often indicates poor boundaries; good design reduces the need for pervasive mocks.

Module A unit of decomposition with a public surface and internal implementation. A module should have high cohesion, controlled dependencies, and testable contracts.

N

Narrow Interface An API that exposes only what clients need. Narrow interfaces reduce coupling and improve evolvability.

Non-Functional Requirements (NFRs) Requirements about how the system behaves: performance, reliability, security, availability, maintainability, observability. NFRs are architectural drivers.

noexcept A function contract indicating it will not throw exceptions. It affects optimization and failure behavior and should be applied deliberately with explicit error channels when needed.

O

Observability The ability to understand system state from outputs: logs, metrics, traces. Observability is not optional in long-lived systems; it is a design property.

Overengineering Adding flexibility, abstraction, or generality without evidence of need. Overengineering increases cognitive load and maintenance cost and often delays delivery.

Ownership Responsibility for managing an object's lifetime and releasing resources. In C++ design, ownership should be explicit, not implicit.

P

PImpl (Pointer to Implementation) An idiom that hides implementation details behind an opaque pointer to reduce build coupling and optionally help maintain ABI stability across versions.

Polymorphism Ability to treat different types uniformly through a common interface. In modern C++ design:

- prefer static polymorphism (templates/concepts) when types are known at compile time and performance matters,
- prefer dynamic polymorphism (virtual interfaces) when late binding and stable ABI boundaries are required,
- prefer type erasure when you need runtime polymorphism without exposing inheritance in public interfaces.

Precondition A requirement that must be true before calling a function. Preconditions should be enforced by types when possible, otherwise validated at boundaries or asserted in debug builds.

Postcondition A guarantee about the state after a function completes successfully. Postconditions define reliability and allow callers to reason safely.

Premature Optimization Optimization performed before measurement and budgeting. In long-lived design, optimization should be driven by profiling and cost models, not guesswork.

R

RAII Resource Acquisition Is Initialization. A foundational idiom ensuring deterministic cleanup through object lifetime. RAII applies to far more than memory: locks, threads, files, sockets, and transactional state.

Refactoring Changing internal structure without changing externally observable behavior. Safe refactoring depends on tests, boundaries, and explicit contracts.

Reentrancy A property where a function can be safely called again before a previous call completes (often via callbacks). Reentrancy hazards should be stated and designed for explicitly.

Resilience The ability to continue operating under failures or overload. Resilience requires explicit policies: retries with budgets, backoff, circuit breaking, and graceful degradation.

Risk Register A documented list of known technical risks and mitigation plans. Long-lived systems benefit from tracking risks explicitly, not informally.

S

Scalability Ability to maintain performance as load increases. Scalability is not just speed; it includes contention behavior, queueing, memory growth, and operational stability.

Seam A place where behavior can be changed or substituted without rewriting the system. Seams enable testing and safe evolution.

Separation of Concerns Keeping different responsibilities in different components. In C++ design, this often means isolating I/O, parsing, concurrency, and third-party dependencies.

Shutdown Semantics Rules defining what happens when the system stops:

- no callbacks after shutdown completes,
- background work drained or cancelled deterministically,
- resources released in a defined order.

Stability Guarantee A promise about what will remain compatible across releases (API, ABI, behavior, performance bounds). Guarantees must be explicit to be meaningful.

Strong Type A distinct type wrapping a primitive to prevent accidental mixing of concepts (IDs, units, money, sizes). Strong types improve correctness with negligible runtime cost.

T

Technical Debt A metaphor for the future cost of expedient design shortcuts. Debt is manageable when tracked and paid down; it becomes destructive when hidden and compounded.

Testability A design property describing how easily a component can be verified in isolation. Testability is improved by narrow interfaces, dependency injection, separation of pure logic from I/O, and explicit contracts.

Thread Safety A contract describing which operations can be called concurrently and under what rules. Thread safety must be explicit; otherwise it becomes folklore.

Time Budget An explicit bound on how long an operation may take. Time budgets force design clarity around timeouts, cancellation, and fallback behavior.

U

Undefined Behavior (UB) A situation where the language imposes no requirements on program behavior. UB can lead to crashes, silent corruption, or security vulnerabilities. Designs that clarify lifetimes, avoid data races, and enforce invariants reduce UB exposure.

Unit of Work A logically atomic set of operations, often tied to a transaction or request. A unit of work clarifies boundaries, lifetimes, and rollback semantics.

V

Value Semantics Design style where objects behave like values: copying produces independent state, and lifetimes are explicit. Value semantics simplify reasoning and improve safety.

Versioning A policy and mechanism to evolve APIs and behavior over time without breaking users unexpectedly. Versioning must be paired with deprecation policies and migration guidance.

W

Work Queue A structure that holds tasks/commands for asynchronous processing. Correct designs define:

- queue bounds,
- backpressure behavior,
- task ownership and lifetime,
- shutdown and draining policy.

Closing Note

The purpose of a glossary in a long-lived design book is not to impress but to align teams. Many design failures begin as vocabulary failures: engineers use the same word to mean different things. Consistent definitions improve reviews, reduce conflict, and turn architecture from opinion into shared reasoning.

Epilogue

Software Design as an Ethical Responsibility

Why Ethics Belongs in a C++ Design Book

Software design is often discussed as if it were purely technical: architectures, patterns, performance, concurrency, memory, and correctness. In reality, design choices have **ethical consequences** because software influences:

- safety (medical devices, cars, aviation, industrial systems),
- security (financial systems, identity infrastructure, critical services),
- fairness and inclusion (accessibility, language, usability constraints),
- stability of institutions (government services, health, education),
- economic outcomes (cost of maintenance, vendor lock-in, system failure costs),
- human dignity (privacy, abuse prevention, transparency).

This is amplified in C++ because the language enables:

- direct control over hardware resources,

- high performance and low latency execution,
- concurrency and parallelism at scale,
- integration at the core of systems where failure is costly,
- but also undefined behavior and memory safety hazards if misused.

A long-lived system is not merely code that compiles for many years. It is code that remains **safe to operate**, **safe to change**, and **trustworthy** under evolving demands. That is why ethics belongs here.

Ethical Design Is Design Under Constraints

Ethical responsibility does not require moral slogans. It requires disciplined engineering under real constraints. In practice, ethical design means:

- Do not ship unknown risk when it affects safety or security.
- Do not create systems that cannot be maintained responsibly.
- Do not hide complexity that will harm operators and future engineers.
- Do not externalize cost onto users through fragile designs.
- Do not treat correctness, observability, and security as optional.

Ethics becomes concrete through design properties:

- clear contracts,
- explicit ownership,
- deterministic shutdown,

- bounded resource use,
- measured performance budgets,
- auditable behavior,
- operational transparency.

1) Correctness Is a Moral Constraint

In long-lived systems, correctness is not an abstract ideal. Incorrect software wastes time, money, and sometimes lives.

A) Undefined Behavior Is Not "Just a Bug"

In C++, undefined behavior is uniquely dangerous because it can:

- pass tests today and fail tomorrow under optimization,
- cause silent corruption instead of obvious crashes,
- create security vulnerabilities without visible symptoms.

Ethical C++ design treats UB as a systemic hazard. The ethical position is simple:

> If you cannot explain why your code is correct under the language rules, you are shipping risk to others.

B) Strong Invariants Reduce Harm

The ethical design approach is to build systems where invalid states are difficult or impossible:

- enforce invariants at construction,

- avoid raw owning pointers,
- minimize shared mutable state,
- encode assumptions with types.

```
#include <string>
#include <stdexcept>

class Endpoint {
public:
  explicit Endpoint(std::string host) : host_(std::move(host)) {
    if (host_.empty()) throw std::runtime_error("host required");
  }
  const std::string& host() const noexcept { return host_; }
private:
  std::string host_;
};
```

This is not merely “clean code.” It is preventing entire failure classes from entering production.

2) Security Is a Design Obligation

Security cannot be patched in after architecture is set. Design decisions determine:

- attack surface,
- input validation boundaries,
- privilege separation,
- secret handling,
- observability and auditability.

Ethical responsibility includes designing for hostile conditions:

- treat all external input as untrusted,
- validate at boundaries,
- keep parsing strict and defensive,
- define resource limits (sizes, timeouts, retries),
- prefer fail-closed behavior for security-sensitive operations.

A) Resource Exhaustion Is a Security Problem

Unbounded queues, unbounded retries, and unbounded memory growth are not only reliability issues; they enable denial-of-service.
Ethical design requires backpressure and budgets:

```
#include <queue>
#include <mutex>

template <class T>
class BoundedQueue {
public:
  explicit BoundedQueue(std::size_t cap) : cap_(cap) {}

  bool try_push(T v) {
    std::lock_guard<std::mutex> lk(m_);
    if (q_.size() >= cap_) return false;
    q_.push(std::move(v));
    return true;
  }

  bool try_pop(T& out) {
```

```
    std::lock_guard<std::mutex> lk(m_);
    if (q_.empty()) return false;
    out = std::move(q_.front());
    q_.pop();
    return true;
  }

private:
  std::mutex m_;
  std::queue<T> q_;
  std::size_t cap_;
};
```

Bounding capacity is an ethical choice: it prevents the system from consuming resources until it collapses and harms users.

3) Reliability Is Respect for Users and Operators

Reliable systems respect:

- the user's time and trust,
- the operator's responsibility,
- the organization's mission,
- the public's dependency on services.

A reliable design defines:

- failure modes (what can fail and how it is represented),
- recovery modes (retry, degrade, stop),

- shutdown semantics (drain, cancel, stop),
- operational observability (metrics, logs, alerts).

A) A System Without Observability Is Ethically Incomplete

If a system fails silently, it offloads the cost of diagnosis to humans under pressure.
Ethical design makes failures visible:

- stable error codes,
- structured logging,
- metrics that capture saturation and backlog,
- trace correlation across boundaries.

4) Maintainability Is a Duty to Future Engineers

A long-lived system will outlast:

- its original team,
- its original assumptions,
- its initial platforms,
- perhaps even its original organization.

Design that is difficult to change is not just inconvenient. It creates:

- fear-driven development,
- fragile workarounds,

- permanent technical debt,
- eventual rewrites that waste public and private resources.

Ethical design therefore includes:

- clear module boundaries,
- explicit ownership and lifetimes,
- consistent error and concurrency models,
- stable APIs with deprecation policy,
- tests that validate contracts,
- build hygiene to keep iteration affordable.

5) Performance Honesty

Performance is a moral issue when it:

- determines fairness (slow systems exclude users),
- determines safety (timeouts in control systems),
- determines operational cost (wasted compute and energy),
- determines reliability (overload collapses).

But ethical performance is not obsession. It is honesty:

- define budgets,
- measure reality,

- optimize hot paths with evidence,
- preserve readability where it does not violate budgets.

The ethical violation is not “slow code.” It is shipping unknown cost under load and pretending it will be fine.

6) Responsible Abstraction: Do Not Hide the Truth

Abstraction is necessary. But abstraction becomes unethical when it hides:

- ownership,
- blocking behavior,
- allocation,
- failure modes,
- concurrency constraints.

Good abstraction compresses complexity without distorting reality. A long-lived design strives for:

- explicit costs,
- explicit failure semantics,
- explicit concurrency model,
- explicit shutdown behavior.

7) The Culture of Responsibility

Ethical software design is sustained by culture:

- “we do not ship known UB”,
- “we measure before optimizing”,
- “we do not expand APIs without contracts”,
- “we make shutdown deterministic”,
- “we do not accept silent failure”,
- “we protect future maintainers”.

Culture becomes enforceable through:

- design reviews with checklists,
- coding guidelines aligned with risks,
- continuous testing and static analysis,
- postmortems focused on learning, not blame,
- deprecation policies and migration support.

A Final Engineering Statement

The deepest truth of software design is that **someone else will pay**. If we choose shortcuts, they will pay later: users, operators, colleagues, and organizations.
In Modern C++, where power and danger coexist, ethical responsibility means:

- build systems that are correct by construction,
- make costs and contracts explicit,
- keep failure observable and recoverable,
- ensure the system can be maintained and evolved without fear.

A long-lived system is not an accident. It is the result of engineers who treat design as a responsibility, not merely a craft.

Why C++ Still Matters for Serious Systems

The Question Behind the Question

When engineers ask whether C++ still matters, they rarely mean “can it build software?” They mean:

- Can it still compete with modern languages in safety and productivity?
- Is it a rational choice for long-lived, large-scale systems?
- Does it still provide unique value beyond legacy inertia?

For serious systems, the correct way to answer is to separate **capability** from **responsibility**. C++ remains relevant not because it is comfortable or fashionable, but because it uniquely occupies a space where:

- performance is not a preference but a constraint,
- predictability matters more than convenience,
- resource control is part of correctness,
- interoperability and deployment realities dominate,
- the system must live longer than its original team.

C++ still matters because serious systems have serious constraints, and C++ remains one of the most capable tools for meeting them when used with modern design discipline.

1) Performance as a Design Constraint, Not a Micro-Optimization Hobby

In serious systems, performance is inseparable from correctness. A control system that misses deadlines, a trading system that exceeds latency budgets, or an embedded controller that exceeds memory limits is *incorrect* even if its logic is correct.
C++ offers:

- predictable low-level cost models,
- zero-overhead abstractions when used correctly,
- direct control over allocation, layout, and locality,
- fine-grained concurrency constructs,
- explicit tradeoffs between latency, throughput, and memory.

A) "Zero-Overhead" as a Contract

The zero-overhead principle is not magic. It is a discipline: the ability to write high-level code that, when compiled, does not impose unintended runtime cost compared to an equivalent hand-written low-level solution.

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

long sum(std::span<const int> xs) noexcept {
```

```
  long s = 0;
  for (int v : xs) s += v;
  return s;
}
```

This is expressive, type-safe, and (with optimization) compiles down to simple iteration over contiguous memory.

2) Deterministic Resource Management (RAII) Is Still a Competitive Advantage

Many serious systems depend on deterministic cleanup:

- releasing locks,
- closing file descriptors,
- returning buffers to pools,
- stopping threads and timers,
- ending transactions reliably.

C++ is built around RAII: deterministic lifetimes as a first-class design mechanism.

```
#include <mutex>

class SafeCounter {
public:
  void inc() {
    std::lock_guard<std::mutex> lk(m_);
    ++v_;
  }
```

```
private:
  std::mutex m_;
  int v_ = 0;
};
```

RAII is not just convenience. It is a correctness tool that keeps cleanup consistent under exceptions, early returns, and complex control flow.

3) Interoperability: The Reality of the World Is Not Greenfield

Serious systems rarely exist in isolation. They integrate with:

- operating systems and kernels,
- device drivers and hardware APIs,
- networking stacks and protocols,
- legacy libraries and vendor SDKs,
- other languages and runtimes (C, Rust, Python, Java, .NET).

C++ remains a dominant “integration language” because:

- it is deeply compatible with C interfaces,
- it can expose stable ABI-compatible entry points,
- it can embed or be embedded by other runtimes,
- it can sit at the boundary where performance and safety collide.

This matters for long-lived systems: rewrites are expensive, but integration is inevitable.

4) Control Over Memory Layout and Data-Oriented Design

Modern performance often depends on **data layout**. CPU caches, branch prediction, SIMD, NUMA, and memory bandwidth are design-level concerns.
C++ provides:

- explicit control over object layout (within language rules),
- contiguous containers and predictable iteration,
- the ability to choose representation with minimal overhead,
- templates and compile-time composition that can express efficient layouts.

A) Contiguity and Locality Are Architectural

```
#include <vector>
#include <cstdint>

struct Particle {
  float x, y, z;
  float vx, vy, vz;
  std::uint32_t flags;
};

void step(std::vector<Particle>& ps, float dt) noexcept {
  for (auto& p : ps) {
    p.x += p.vx * dt;
    p.y += p.vy * dt;
    p.z += p.vz * dt;
  }
}
```

This is a simple example, but the point is architectural: serious systems often live or die by data locality, and C++ gives first-class control over it.

5) Concurrency and Parallelism at the Systems Level

Serious systems must exploit multicore hardware responsibly. C++ provides:

- low-level atomics and memory model,
- high-level concurrency primitives (threads, mutexes, condition variables),
- task and future abstractions,
- the ability to build custom schedulers and execution models.

The real advantage is not that concurrency is easy. It is that C++ allows expressing exactly the model required when correctness, latency, and throughput must be balanced.

6) Modern C++ Has Evolved: The Language Supports Better Design

C++ is not frozen. Modern design practice in C++ is shaped by:

- move semantics for efficient value-oriented APIs,
- smart pointers and RAII for explicit lifetimes,
- stronger type tools (`enum class`, `variant`, `optional`),
- improved generic programming (concepts, constrained templates),
- safer library abstractions (ranges, spans, chrono improvements),
- clearer concurrency primitives.

A) Expressiveness Without Sacrificing Cost Transparency

```cpp
#include <optional>
#include <string_view>

std::optional<int> parse_port(std::string_view s) noexcept {
  int v = 0;
  if (s.empty()) return std::nullopt;
  for (char c : s) {
    if (c < '0' || c > '9') return std::nullopt;
    v = v * 10 + (c - '0');
  }
  if (v < 0 || v > 65535) return std::nullopt;
  return v;
}
```

The code is explicit about failure, does not allocate, and preserves a clear cost model.

7) The Most Honest Answer: C++ Matters When You Accept the Responsibility

C++ remains uniquely powerful, but that power comes with obligations:

- to avoid undefined behavior,
- to make ownership explicit,
- to define concurrency models precisely,
- to keep APIs stable and testable,
- to build observability and operational safety,

- to design for humans and maintenance.

Serious systems cannot rely on hope. They require explicit engineering.

8) When C++ Is the Wrong Choice

C++ is not universal. It is the wrong choice when:

- the dominant constraints are rapid iteration and short-lived code,
- the system is primarily glue between existing services,
- the team cannot enforce strong discipline around safety and design,
- the operational environment demands managed runtimes and strict sandboxing,
- performance and resource constraints are not central.

Choosing C++ ethically means choosing it for the right reasons, with the right maturity.

Closing Perspective

C++ still matters for serious systems because serious systems must solve problems at the boundary between:

- hardware reality and software intent,
- performance budgets and correctness guarantees,
- longevity and continuous change,
- interoperability and evolving ecosystems.

In that boundary space, C++ remains one of the most capable tools available. The language is not the reason a system survives; design is. But when design is disciplined, C++ gives engineers the control, performance, and expressiveness needed to build systems that remain correct and valuable for many years.

References

How These References Are Used

This book is grounded in resources that are:

- **normative** (language and library standards, official specifications),
- **authoritative** (committee and toolchain documentation, established guidelines),
- **industry-proven** (well-known architecture and design literature, long-lived practices),
- **verifiable** (publicly available or formally published sources that can be independently checked).

To keep the book practical, references are organized by the design questions they answer:

- what the language guarantees and forbids,
- what the standard library provides,
- how concurrency and memory ordering behave,
- how toolchains implement and diagnose behavior,
- which architecture principles endure in large systems,
- how engineering organizations manage long-term evolution.

1) ISO and Standardization (Normative Language/Library Foundations)

ISO C++ Standard

- ISO/IEC 14882:2024(E) — *Programming Languages — C++* (commonly referred to as "C++23").
- ISO/IEC 14882:2020(E) — *Programming Languages — C++* (commonly referred to as "C++20").
- ISO/IEC 14882:2017(E) — *Programming Languages — C++* (commonly referred to as "C++17").

WG21 Working Papers and Committee Materials

- WG21 Working Drafts and Editor Reports for successive standard iterations (working paper series typically identified by document numbers such as "Nxxxx").
- WG21 Papers (proposal documents, wording papers, issue resolutions) for language and library evolution, including concurrency, contracts discussions, reflection work, and library improvements.

ISO C++ Committee Status Summaries

- ISO C++ committee public status summaries for C++23 publication details and C++26 work-in-progress tracking (used for chronology, macro values, and standard naming conventions).

2) Core Guidelines and Modern C++ Design Guidance

C++ Core Guidelines

- *C++ Core Guidelines* (Bjarne Stroustrup, Herb Sutter, and contributors), maintained by the ISO C++ community.
- Focus areas used in this book:
  - lifetime and ownership rules,
  - resource management (RAII),
  - type safety and interface design,
  - concurrency and data-race avoidance,
  - error handling guidance,
  - performance rules and “don’t pessimize prematurely” principles.

ISO C++ Community Guidance

- ISO C++ community articles and guidance pages for:
  - standard naming, publication conventions, and macro values,
  - committee work summaries and milestones.

3) Language Reference and Standard Library Reference (Authoritative Technical Detail)

C++ Language and Library Reference

- *cppreference.com* — community-maintained reference widely used for standard library interfaces, semantics summaries, feature availability, and compiler support tables.
- Focus areas used in this book:
  - standard library components (`span`, `string_view`, ranges, chrono, concurrency primitives),
  - exception safety notes, complexity guarantees, iterator invalidation rules,
  - feature test macros and language version support tracking.

Primary Compiler Documentation (GCC, Clang/LLVM, MSVC)

- GCC documentation:
  - language standard modes and dialect flags,
  - diagnostics and warnings,
  - sanitizers integration notes (where applicable),
  - ABI notes and toolchain behavior relevant to large-scale builds.
- Clang/LLVM documentation:
  - language standard modes and flags,
  - sanitizers (ASan, UBSan, TSan) usage and limitations,

  - libc++ notes and conformance behavior.

- Microsoft Visual C++ (MSVC) documentation:
  - language conformance switches and standard mode selection,
  - diagnostics and warnings,
  - runtime library behavior and ABI-related notes,
  - Windows-specific toolchain integration guidance.

4) Concurrency, Memory Model, and Performance Foundations

C++ Concurrency and Memory Model (Normative + Explanatory)

- ISO/IEC 14882 (concurrency clauses) for:
  - data race definition and consequences,
  - atomics and memory ordering,
  - happens-before relationships,
  - thread and synchronization semantics.
- ISO C++ committee technical specifications and papers related to concurrency extensions (used as context for evolution and design intent).

Performance and Systems Reality (Architecture-Level Facts)

- CPU and memory hierarchy fundamentals (cache locality, contention, bandwidth constraints) as established in systems programming literature and vendor architecture manuals (conceptual grounding used in data-oriented and high-performance sections).

- Classic “cost model” principles for:
  - allocations vs stack storage,
  - indirection vs locality,
  - synchronization overhead,
  - queueing and backpressure behavior.

5) Testing, Verification, and Diagnostics (Tooling as Design Feedback)

Unit and Integration Testing Frameworks (C++ Ecosystem)

- GoogleTest / GoogleMock documentation and best practices (for test design, mocking boundaries, and test organization).
- Catch2 documentation (modern unit-testing patterns and lightweight workflow).
- Doctest documentation (header-only testing approaches and tradeoffs).

Property-Based Testing and Fuzzing

- Property-based testing concepts as established in the QuickCheck tradition (principles: invariants, generators, shrinking, counterexample minimization).
- LLVM libFuzzer and sanitizers documentation (fuzzing harness design, coverage guidance, typical pitfalls).

Static Analysis and Code Quality Automation

- clang-tidy documentation (checks for readability, performance, modernize, bugprone, concurrency-related patterns).
- Clang Static Analyzer documentation (path-sensitive analysis and defect classes).
- Compiler warning references and diagnostic catalogs for GCC/Clang/MSVC (used to define warning baselines and migration strategies).

6) Software Architecture and Design Literature (Long-Lived Principles)

Foundational Design and Architecture

- Gamma, Helm, Johnson, Vlissides — *Design Patterns: Elements of Reusable Object-Oriented Software*. (Patterns as historical baseline; reinterpreted for Modern C++).
- Robert C. Martin — *Clean Architecture*. (Boundaries, dependency direction, architectural rules as long-lived constraints).
- Martin Fowler — *Refactoring: Improving the Design of Existing Code* (2nd edition). (Safe refactoring principles, evolutionary design mechanics).
- Eric Evans — *Domain-Driven Design*. (Domain boundaries, ubiquitous language, modeling discipline).
- Len Bass, Paul Clements, Rick Kazman — *Software Architecture in Practice*. (Architectural tactics, quality attributes, tradeoffs).

Large-Scale Systems and Reliability Engineering

- *Site Reliability Engineering* and *The Site Reliability Workbook* (Google). (Operational discipline: observability, error budgets, reliability as design driver).
- Michael T. Nygard — *Release It!* (Stability patterns, failure modes, production readiness thinking).
- Jez Humble and David Farley — *Continuous Delivery* (deployment pipelines, release safety, feedback loops).

Interfaces, APIs, and Evolution

- Joshua Bloch — *Effective Java* (principles of API design that generalize across languages; used for public API stability thinking).
- API versioning and deprecation practices as established in mature platform documentation and long-lived library ecosystems (principles: semantic compatibility, migration windows, staged removals).

7) Security and Robustness (Engineering-Grade Guidance)

Secure Coding and Threat Modeling

- CERT C++ Secure Coding Standard (secure coding rules, vulnerability classes, and mitigation patterns).
- OWASP guidance for:
  - input validation and boundary defense,
  - authentication/authorization principles,

  - secure logging practices (avoid leaking secrets),
  - common risk categories relevant to service architectures.

Memory Safety and Risk Reduction

- ISO C++ standard rules around object lifetime and undefined behavior (normative).
- Sanitizer documentation (ASan/UBSan/TSan) for practical detection of:
  - use-after-free, buffer overflow,
  - undefined behavior hotspots,
  - data races and concurrency defects.

8) Build Systems, Packaging, and Toolchain Discipline

CMake

- CMake official documentation:
  - target-based design,
  - interface/private/public dependency modeling,
  - export/install patterns,
  - toolchain files and cross-compilation guidance.

Package and Dependency Management

- Conan documentation (dependency graphs, profiles, reproducible builds).
- vcpkg documentation (manifest mode, binary caching, CI integration).

ABI and Binary Boundaries

- Toolchain ABI references (platform-specific) for:
  - shared library boundaries,
  - symbol visibility,
  - ODR pitfalls in large builds,
  - compatibility tradeoffs (PImpl, stable C interfaces).

9) Practical Reference Tooling Used for Validation and Examples

Compiler Explorer

- Compiler Explorer (Godbolt) used for:
  - validating cost models and generated assembly in illustrative examples,
  - comparing code generation across compilers and standard modes,
  - confirming “zero-overhead” claims for selected idioms.

Profiling and Benchmarking

- Benchmarking methodology as established in performance engineering practice:
  - measure first, optimize second,
  - avoid micro-benchmark traps,
  - verify changes under representative load,
  - track p50/p95/p99 latency for services.

- Google Benchmark documentation (micro-benchmarking harness patterns and pitfalls).

10) Reference Map: Which Parts of the Book Rely on Which Sources

Language Rules, Undefined Behavior, Object Lifetime

- ISO/IEC 14882 (normative object model, lifetime, UB rules).
- C++ Core Guidelines (lifetime and safety rules).
- Compiler documentation and sanitizers (practical detection and enforcement).

Interfaces, APIs, and Contracts

- C++ Core Guidelines (interfaces, resource management, error handling).
- Architecture literature (stable boundaries, dependency direction).
- Testing sources (testability as a design property).

Concurrency and Scalability

- ISO standard concurrency clauses (atomics, races, synchronization).
- Toolchain/sanitizers (race detection guidance).
- Reliability literature (operational constraints, backpressure, budgets).

Large-Scale Design, Evolution, and Longevity

- Architecture and refactoring literature (evolutionary design, safe change).
- SRE and production reliability texts (operability, observability, incident learning).
- Tooling documentation (build discipline, CI feedback loops).

Closing Note

The purpose of listing references in a long-lived design book is alignment, not decoration. Serious systems are built on shared truth:

- the standard defines what the language guarantees,
- guidelines and toolchains define how we avoid common hazards,
- architecture literature defines how large systems survive change,
- reliability and security references define what must not be treated as optional.

These resources collectively form the foundation for the principles, patterns, and engineering practices presented throughout this book.