https://simplifycpp.org

Modern C++ Encyclopedia

MODERN OOP, RAII & POLYMORPHISM

Volume 3

Prepared by Ayman Alheraki

DRAFT

Modern C++ Encyclopedia

MODERN OOP, RAII & POLYMORPHISM

Volume 3

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

Prepared by Ayman Alheraki

Contents

Authors Introduction **25**

Preface **32**

The evolution of C++ object-oriented design 34

From C++98 OOP to Modern C++ architecture 35

RAII as the foundation of safe systems programming 39

The shift from inheritance-heavy design to composition 43

Concluding perspective 49

What this volume teaches 50

Understanding the C++ object model 51

Designing safe and efficient ownership models 55

Building reliable resource-management systems 61

Concluding perspective 66

Who this volume is for 67

Systems programmers 67

Library designers 70

Engineers building large-scale C++ applications 72

Conclusion 74

I Foundations of Classes, Objects & C++ Object Model 75

1 Anatomy of an Object in Modern C++ 76

1.1 What Exactly Is an Object? . . . 76
1.1.1 Object identity . . . 77
1.1.2 Object vs value vs type . . . 80
1.1.3 Storage & representation . . . 82
1.1.4 Strict aliasing rules . . . 86
1.1.5 Closing perspective . . . 89
1.2 Object Memory Layout . . . 89
1.2.1 Layout rules . . . 90
1.2.2 Alignment & padding . . . 93
1.2.3 Bitfields and layout traps . . . 96
1.2.4 Field reordering for performance . . . 100
1.2.5 Windows-oriented notes for layout-sensitive code . . . 103
1.2.6 Closing perspective . . . 104
1.3 Object Lifetime & Construction Model . . . 105
1.3.1 Pre-construction memory state . . . 106
1.3.2 Initialization sequence . . . 109
1.3.3 Order of destruction . . . 113
1.3.4 Temporaries & lifetime extension . . . 118
1.3.5 Practical Windows notes . . . 121
1.3.6 Closing perspective . . . 122

2 Classes & Member Initialization 123

2.1 Member Initializer Lists (Deep) . . . 123
2.1.1 Best practices . . . 124
2.1.2 Common pitfalls . . . 130

2.1.3 Dependency initialization order . . . 135
2.1.4 Windows-oriented guidance . . . 139
2.1.5 Closing perspective . . . 140
2.2 Special Member Functions . . . 141
2.2.1 What counts as a special member function . . . 142
2.2.2 `= default` . . . 143
2.2.3 `= delete` . . . 146
2.2.4 `explicit` . . . 148
2.2.5 `noexcept` constructors and move operations . . . 150
2.2.6 Rule of Zero . . . 154
2.2.7 Rule of Three . . . 155
2.2.8 Rule of Five . . . 157
2.2.9 Rule of Six . . . 160
2.2.10 Best practices for special member functions . . . 163
2.2.11 Windows-oriented guidance . . . 164
2.2.12 Closing perspective . . . 165

3 Constructors & Initialization Taxonomy 166
3.1 All Constructor Kinds . . . 166
3.1.1 Default constructors . . . 167
3.1.2 Copy constructors . . . 169
3.1.3 Move constructors . . . 171
3.1.4 Delegating constructors . . . 173
3.1.5 In-place constructors . . . 174
3.1.6 Practical perspective . . . 177
3.2 Aggregate vs Non-Aggregate Types . . . 177
3.2.1 What an aggregate means in practice . . . 178
3.2.2 New C++20 aggregate rules . . . 181

3.2.3 Design strategies . . . 184
3.2.4 Pitfalls & gotchas . . . 187
3.2.5 Aggregate versus non-aggregate in real design . . . 190
3.2.6 Windows-oriented notes . . . 192
3.2.7 Closing perspective . . . 193

II Move Semantics, Perfect Forwarding & Lifetime Design 194

4 Deep Dive into Move Semantics 195
4.1 Why Move Semantics? . . . 195
4.1.1 Performance history . . . 196
4.1.2 Copy-elision evolution . . . 201
4.1.3 Move vs Copy cost models . . . 206
4.1.4 Design conclusions . . . 212
4.1.5 Extensive examples . . . 213
4.1.6 Windows build notes . . . 218
4.1.7 Practical rules to remember . . . 218
4.2 Implementing Move Constructors . . . 219
4.2.1 Designing optimal move operations . . . 219
4.2.2 Exception guarantees . . . 228
4.2.3 Move-only types . . . 236
4.2.4 Design checklist for move constructors . . . 244
4.2.5 Extensive examples . . . 245
4.2.6 Windows build and test notes . . . 251
4.2.7 Concluding design rules . . . 251
4.3 Perfect Forwarding Theory . . . 252
4.3.1 Reference collapsing . . . 253

4.3.2 Universal / forwarding references 255
4.3.3 Forwarding constructors 257
4.3.4 Factory pattern with forwarding 259
4.3.5 Perfect forwarding with variadic templates 260
4.3.6 Common pitfalls 261
4.3.7 Design guidelines 262
4.3.8 Summary 262

5 Ownership Models & Lifetime Control 264
5.1 Raw Ownership 264
5.1.1 Manual ownership design 265
5.1.2 Borrow semantics 272
5.1.3 Non-owning views 278
5.1.4 Advanced raw lifetime pitfalls 286
5.1.5 Practical design rules 288
5.1.6 Extensive examples 289
5.1.7 Windows build and analysis notes 293
5.1.8 Final design conclusions 294
5.2 Modern Smart Pointer Ownership 295
5.2.1 unique_ptr architecture 295
5.2.2 Implementing unique_ptr from scratch 301
5.2.3 shared_ptr reference counting internals 309
5.2.4 weak_ptr & cycle-breaking 317
5.2.5 Design guidance 323
5.2.6 Extensive examples 325
5.2.7 Windows build notes 329
5.2.8 Concluding rules 330
5.3 Hybrid Ownership Patterns 331

5.3.1 Shared + borrowed . . . 332
5.3.2 Borrowed + RAII guard . . . 340
5.3.3 Handle + body patterns . . . 348
5.3.4 Comparative design map . . . 359
5.3.5 Extensive examples . . . 360
5.3.6 Windows build notes . . . 366
5.3.7 Concluding design rules . . . 367

III RAII, Resource Wrappers & Exception-Safe Design 368

6 RAII Fundamentals & Deep Theory 369
6.1 Why RAII Works . . . 369
6.1.1 Deterministic destruction . . . 370
6.1.2 Comparisons with GC languages . . . 376
6.1.3 Preventing leaks and maintaining invariants . . . 377
6.1.4 Summary . . . 382
6.2 Resources Eligible for RAII . . . 383
6.2.1 Memory wrappers . . . 384
6.2.2 OS handles (Windows/Linux/Mac) . . . 390
6.2.3 File descriptors . . . 395
6.2.4 Network sockets . . . 399
6.2.5 GPU/Graphics handles . . . 406
6.2.6 Cross-platform RAII patterns . . . 412
6.2.7 What should not be modeled as owning RAII blindly . . . 413
6.2.8 Design checklist . . . 415
6.2.9 Extensive examples . . . 415
6.2.10 Windows build notes . . . 418

6.2.11 Concluding rules 419

7 Exception Safety in Resource Management 421

7.1 Levels of Exception Safety 421

7.1.1 Basic guarantee 422

7.1.2 Strong guarantee 424

7.1.3 No-throw guarantee 428

7.1.4 Relationship between RAII and exception guarantees 431

7.1.5 Combining guarantees in real systems 432

7.1.6 Example: layered guarantees 432

7.1.7 Windows compilation guide 433

7.1.8 Summary 434

7.2 Designing Exception-Safe Classes 434

7.2.1 Copy-and-swap 435

7.2.2 Commit/rollback pattern 439

7.2.3 Scoped guards 442

7.2.4 Design guidelines 446

7.2.5 Example: complete exception-safe class 447

7.2.6 Windows compilation guide 448

7.2.7 Summary 448

8 Advanced RAII Patterns 450

8.1 Monadic Resource Patterns 450

8.1.1 Resource pipelines 451

8.1.2 Deferred operations 462

8.1.3 RAII-based transactions 468

8.1.4 Comparing the three patterns 477

8.1.5 Windows build notes 478

8.1.6 Concluding design rules 479
8.2 Multi-Resource Structures 480
8.2.1 RAII graphs 481
8.2.2 Resource slicing 490
8.2.3 Resource bundles 498
8.2.4 Multi-resource failure handling 509
8.2.5 Comparative design map 511
8.2.6 Extensive examples 512
8.2.7 Windows build notes 518
8.2.8 Concluding design rules 519

IV Dynamic Polymorphism, Vtables & Runtime Architecture 521

9 Dynamic Polymorphism Basics 522
9.1 Why OOP Exists in C++ 522
9.1.1 Encapsulation and abstraction 522
9.1.2 Interface-based design 523
9.1.3 Extensibility 525
9.2 Virtual Methods & Late Binding 526
9.2.1 Basic virtual dispatch 526
9.2.2 Runtime polymorphism in containers 528
9.2.3 Virtual destructors 529
9.3 Abstract Classes & Interfaces 530
9.3.1 Pure virtual functions 530
9.3.2 Concrete implementations 530
9.3.3 Interface design principles 532
9.4 Overriding, final, override 532

9.4.1 The override specifier . . . 533
9.4.2 The final specifier . . . 534
9.4.3 Combining override and final . . . 535
9.4.4 Best practices . . . 535
9.4.5 Compilation guide for Windows . . . 536

10 Vtable Internals & Layout (Deep) 537
10.1 Vtable Structure . . . 537
10.1.1 What a vtable conceptually is . . . 538
10.1.2 How vtables are generated . . . 539
10.1.3 Single inheritance layout . . . 543
10.1.4 Memory layout and offsets . . . 545
10.1.5 How overrides map into slots . . . 551
10.1.6 RTTI relationship . . . 552
10.1.7 ABI differences between GCC/Clang and MSVC . . . 553
10.1.8 Illustrative comparison mindset . . . 556
10.1.9 Practical implications for Windows developers . . . 556
10.1.10 Pedagogical object-layout examples . . . 557
10.1.11 Deep design lessons . . . 562
10.1.12 Windows compilation guide . . . 564
10.1.13 Summary . . . 565
10.2 Multiple/Virtual Inheritance . . . 565
10.2.1 Basic multiple inheritance . . . 566
10.2.2 Virtual inheritance . . . 567
10.2.3 Diamond problems . . . 568
10.2.4 Adjustor thunks . . . 575
10.2.5 Performance implications . . . 581
10.2.6 Design patterns and guidance . . . 587

10.2.7 Extensive examples 590
10.2.8 Windows build notes 599
10.2.9 Concluding design rules 599

11 Dynamic Polymorphism Best Practices 601
11.1 When to use virtual 601
11.1.1 The fundamental purpose of virtual functions 601
11.1.2 Use virtual when behavior varies by derived type 603
11.1.3 Avoid virtual when behavior is fixed 604
11.1.4 Prefer non-virtual interfaces when polymorphism is unnecessary 604
11.1.5 Guideline for deciding 605
11.2 Avoiding OOP anti-patterns 605
11.2.1 Deep inheritance hierarchies 606
11.2.2 The fragile base class problem 607
11.2.3 Inheritance used only for code reuse 608
11.2.4 Virtual functions used unnecessarily 609
11.2.5 Ignoring the rule of zero 609
11.3 Virtual destructor rules 609
11.3.1 Why virtual destructors are required 610
11.3.2 Correct polymorphic destructor design 611
11.3.3 Protected destructors for non-polymorphic deletion 611
11.3.4 Pure virtual destructors 612
11.4 Polymorphism + RAII interactions 612
11.4.1 Polymorphic objects with RAII-managed resources 612
11.4.2 RAII with smart pointers and polymorphism 613
11.4.3 Polymorphic RAII wrappers 614
11.4.4 RAII and polymorphic destruction chains 616
11.4.5 Design guidance 617

11.4.6 Compilation guide for Windows 617
11.4.7 Summary 618

V Static Polymorphism, Templates & Type Erasure 619

12 CRTP Deep Dive 620

12.1 Static polymorphism mechanics 620
12.1.1 Why CRTP works 621
12.1.2 The role of `static_cast` 622
12.1.3 A full static polymorphism example 623
12.1.4 CRTP versus virtual dispatch 625
12.1.5 A modern note: explicit object parameters 626
12.2 Compile-time interface design 627
12.2.1 Unconstrained interface design 627
12.2.2 Constrained interface design with concepts 628
12.2.3 Separating public interface from customization points 630
12.2.4 Compile-time interface validation with `static_assert` 632
12.2.5 A standard-library-adjacent example mindset 633
12.2.6 When compile-time interface design is appropriate 633
12.3 CRTP mixins & policy classes 634
12.3.1 CRTP mixins 634
12.3.2 Design advice for mixins 637
12.3.3 Policy classes 637
12.3.4 Combining CRTP with policy classes 640
12.3.5 Mixin pitfalls 643
12.3.6 Using `std::enable_shared_from_this` as a familiar pattern 643
12.4 CRTP performance advantages 643

12.4.1 Direct call opportunities 644
12.4.2 No vptr, no vtable lookup, no indirect branch 645
12.4.3 Example: static versus dynamic dispatch 645
12.4.4 Better optimization visibility 648
12.4.5 Data layout benefits 649
12.4.6 The cost side of CRTP 650
12.4.7 When CRTP performance advantages are most meaningful 651
12.4.8 Practical benchmark guidance on Windows 651
12.4.9 Windows build guidance for the examples 652
12.5 Best practices for CRTP in large systems 653
12.5.1 Prefer CRTP for behavior reuse, not domain modeling 653
12.5.2 Constrain everything important 653
12.5.3 Keep customization points minimal 653
12.5.4 Avoid hidden semantic traps 653
12.5.5 Do not force CRTP where virtual dispatch is the right model 653
12.6 CRTP and type erasure in the bigger picture 654
12.6.1 CRTP 654
12.6.2 Virtual functions 654
12.6.3 Type erasure 654
12.6.4 Choosing correctly 654
12.7 Conclusion 655

13 Variant-Based Polymorphism **656**
13.1 std::variant as a type-safe union 656
13.1.1 Basic usage 656
13.1.2 Safe access mechanisms 657
13.1.3 Variant as algebraic data type 658
13.1.4 Valueless state 658

13.2 Visiting patterns 658
13.2.1 Basic visitation 659
13.2.2 Overloaded visitor pattern 659
13.2.3 Returning values from visitors 660
13.2.4 Multiple variant visitation 661
13.2.5 C++26 member visit 661
13.3 Pattern matching (C++23/26) 662
13.3.1 Variant-based pattern matching today 662
13.3.2 Limitations of current approach 662
13.3.3 Emerging pattern matching proposals 663
13.3.4 Conceptual future syntax 663
13.3.5 Relationship with variant 664
13.3.6 Current best practice 664
13.4 Eliminating inheritance hierarchies 664
13.4.1 Traditional OOP approach 664
13.4.2 Variant-based approach 665
13.4.3 Advantages 666
13.4.4 Trade-offs 666
13.4.5 Solving the expression problem 667
13.4.6 Real-world design pattern 667
13.5 Variant vs CRTP vs Virtual dispatch 668
13.5.1 Variant-based polymorphism 668
13.5.2 CRTP 668
13.5.3 Virtual dispatch 668
13.5.4 Decision guideline 669
13.6 Conclusion 669

14 Type Erasure Techniques **670**

14.1 Function types 670

14.1.1 What a function type is 670

14.1.2 Function types versus callable objects 671

14.1.3 Examples of function types 671

14.1.4 noexcept and function types 672

14.1.5 Lambdas as function objects 673

14.1.6 Why this matters for type erasure 674

14.2 std::function internals 674

14.2.1 Observable contract of std::function 674

14.2.2 The standard specifies behavior, not layout 676

14.2.3 Typical implementation model 676

14.2.4 Common small-object optimization 677

14.2.5 Invocation path 677

14.2.6 Target access 678

14.2.7 Empty-call behavior 679

14.2.8 A practical example 680

14.2.9 What std::function is not ideal for 681

14.3 Custom type-erased wrappers 681

14.3.1 The essence of type erasure 682

14.3.2 A minimal owning callable wrapper 682

14.3.3 What this example teaches 685

14.3.4 A non-owning callable view 686

14.3.5 Important lifetime warning 687

14.3.6 Policy-based erased wrapper 688

14.3.7 A custom type-erased drawable wrapper 688

14.4 The modern callable-wrapper family 692

14.4.1 std::function . . . 692
14.4.2 std::move_only_function . . . 692
14.4.3 std::copyable_function . . . 692
14.4.4 std::function_ref . . . 693
14.4.5 Practical selection guide . . . 693
14.5 Type erasure vs inheritance vs variant . . . 693
14.5.1 Type erasure . . . 693
14.5.2 Inheritance-based polymorphism . . . 694
14.5.3 Variant-based polymorphism . . . 695
14.5.4 A side-by-side conceptual comparison . . . 696
14.5.5 The expression-problem perspective . . . 697
14.5.6 Performance perspective . . . 697
14.5.7 A decision table . . . 698
14.6 Advanced design guidance . . . 698
14.6.1 Do not overuse std::function in hot paths . . . 698
14.6.2 Use non-owning wrappers for parameters when ownership is not needed 699
14.6.3 Use owning wrappers for storage . . . 699
14.6.4 Keep erased interfaces narrow . . . 699
14.6.5 Prefer semantic clarity over cleverness . . . 699
14.7 Windows build guidance . . . 699
14.7.1 Visual Studio 2022 . . . 699
14.7.2 MSVC command line . . . 700
14.7.3 MinGW-w64 g++ on Windows . . . 700
14.8 Conclusion . . . 701

VI GOF Patterns Modernized 702

15 GOF Patterns Modernized 703

15.1 Strategy Pattern . . . 703
15.1.1 Classical dynamic strategy . . . 703
15.1.2 Modern functional strategy with lambdas . . . 706
15.1.3 Static strategy with templates . . . 708
15.1.4 Modern guidance . . . 710
15.2 Observer Pattern . . . 710
15.2.1 Classical interface-based observer . . . 710
15.2.2 Observer with callback registration . . . 713
15.2.3 Modern guidance . . . 715
15.3 Adapter, Decorator, Bridge . . . 715
15.3.1 Adapter Pattern . . . 716
15.3.2 Decorator Pattern . . . 717
15.3.3 Functional decorator . . . 720
15.3.4 Bridge Pattern . . . 721
15.4 Command Pattern . . . 724
15.4.1 Classical command hierarchy . . . 724
15.4.2 Modern command with callables . . . 726
15.4.3 Undo-capable command . . . 727
15.5 Prototype Pattern (modern take) . . . 729
15.5.1 Classical modernized prototype . . . 730
15.5.2 Prototype with value types . . . 731
15.5.3 Modern guidance . . . 732
15.6 Visitor Pattern (dynamic, static, variant-based) . . . 733
15.6.1 Classical dynamic visitor . . . 733
15.6.2 Static visitor with templates . . . 736

15.6.3 Variant-based visitor . . . 738
15.6.4 Comparing the three forms . . . 740
15.7 Pattern composition in Modern C++ . . . 740
15.8 Design guidance for modernized patterns . . . 741
15.8.1 Prefer value semantics where possible . . . 741
15.8.2 Use smart pointers intentionally . . . 741
15.8.3 Prefer lambdas and small function objects for local behavior . . . 741
15.8.4 Prefer templates or concepts for static policy variation . . . 742
15.8.5 Use runtime polymorphism only when runtime openness is truly required 742
15.9 Windows build guidance . . . 742
15.9.1 Visual Studio 2022 project settings . . . 742
15.10 Conclusion . . . 743

16 C++-Specific Design Patterns 744
16.1 Pimpl . . . 744
16.1.1 Motivation . . . 744
16.1.2 Basic Pimpl structure . . . 745
16.1.3 Important rule: destructor must be defined out-of-line . . . 746
16.1.4 Copy and move semantics . . . 746
16.1.5 Pimpl trade-offs . . . 747
16.2 RAII patterns . . . 747
16.2.1 Basic RAII example . . . 748
16.2.2 Custom RAII wrapper . . . 748
16.2.3 Scoped lock RAII . . . 749
16.2.4 RAII for exception safety . . . 749
16.2.5 Scope guard pattern . . . 749
16.3 Resource acquisition filters . . . 750
16.3.1 Validated resource wrapper . . . 751

16.3.2 Factory with validation 751
16.3.3 RAII + filtering pipeline 752
16.3.4 Design intent 753
16.4 Policy-based design 753
16.4.1 Basic policy pattern 753
16.4.2 Policy composition 754
16.4.3 Benefits 755
16.4.4 Trade-offs 755
16.5 Compile-time polymorphism patterns 756
16.5.1 CRTP pattern 756
16.5.2 Concept-based polymorphism 757
16.5.3 Tag dispatch 758
16.5.4 If constexpr dispatch 758
16.5.5 Modern insights 759
16.6 Conclusion 759

17 Architecture & Framework-Level Patterns 761
17.1 Dependency Injection 761
17.1.1 Why Dependency Injection matters in C++ 761
17.1.2 Constructor injection 762
17.1.3 Injection by reference versus pointer 764
17.1.4 Setter injection 766
17.1.5 Template-based injection 767
17.1.6 Interface injection versus value injection 769
17.1.7 Modern guidance for Dependency Injection 770
17.2 Composition over inheritance 770
17.2.1 Why composition is preferred 770
17.2.2 Composition and testability 772

17.2.3 Mixing composition and polymorphism 774
17.2.4 Composition with policies 774
17.2.5 Modern guidance for composition 776
17.3 Plugin architectures (dynamic & static) 776
17.3.1 Core goals of a plugin architecture 777
17.3.2 Dynamic plugin architecture 777
17.3.3 Dynamic plugin design cautions 780
17.3.4 Static plugin architecture 781
17.3.5 Dynamic versus static plugins 783
17.3.6 A practical hybrid design 784
17.3.7 Windows build guidance for dynamic plugins 784
17.4 Event-driven architecture 785
17.4.1 Basic event-driven model 785
17.4.2 Simple event bus example 786
17.4.3 Typed event dispatch 787
17.4.4 Signal-slot style 789
17.4.5 Event loops 791
17.4.6 Architectural benefits of event-driven systems 792
17.4.7 Architectural risks 793
17.5 Modern ECS (Entity Component System) overview 793
17.5.1 Core ideas of ECS 793
17.5.2 Why ECS exists 794
17.5.3 A minimal ECS sketch 795
17.5.4 Modern ECS storage models 797
17.5.5 Entities are identifiers, not objects 798
17.5.6 Components should usually be simple data 798
17.5.7 Systems operate over views 799

17.5.8 ECS versus classical OOP . . . 799
17.5.9 ECS and event-driven systems . . . 800
17.5.10 A practical modern ECS mindset . . . 800
17.6 Architectural comparison and selection . . . 800
17.6.1 Dependency Injection versus direct construction . . . 800
17.6.2 Composition versus inheritance . . . 801
17.6.3 Dynamic versus static plugins . . . 801
17.6.4 Event-driven architecture versus direct calls . . . 801
17.6.5 ECS versus object hierarchies . . . 801
17.7 Windows build guidance . . . 802
17.8 Conclusion . . . 803

VII Full Project: Config System & Object-Oriented Architecture 804

18 Designing a Production-Grade Config Library 805
18.1 Architecture . . . 805
18.1.1 Core architectural layers . . . 805
18.1.2 High-level architecture . . . 806
18.1.3 Design principles . . . 806
18.2 Value hierarchy using variant . . . 806
18.2.1 Basic value type . . . 806
18.2.2 Tree structure . . . 807
18.2.3 Safe access helpers . . . 808
18.2.4 Visitor-based operations . . . 808
18.2.5 Advantages of variant-based hierarchy . . . 809
18.3 Polymorphic parsers . . . 809
18.3.1 Abstract parser interface . . . 809

18.3.2 JSON-like parser example 810
18.3.3 Parser selection via factory 810
18.3.4 Template-based parser (static polymorphism) 811
18.3.5 Combining runtime and compile-time polymorphism 812
18.4 RAII wrappers 812
18.4.1 File reader RAII wrapper 812
18.4.2 Scoped parsing context 813
18.4.3 Combining RAII with parsing 814
18.4.4 Error-safe resource management 814
18.4.5 Custom deleter example 814
18.5 Integration example 815
18.6 Production considerations 816
18.6.1 Thread safety 816
18.6.2 Error handling 816
18.6.3 Extensibility 816
18.6.4 Performance 817
18.7 Conclusion 817

19 Implementation **818**
19.1 JSON parser 818
19.1.1 Design goals 818
19.1.2 Core value model 819
19.1.3 Lexer design 820
19.1.4 Parser structure 829
19.1.5 Front-end convenience API 834
19.1.6 Example usage 834
19.1.7 Production notes 835
19.2 File watcher RAII 835

19.2.1 Design goals . . . 835
19.2.2 RAII wrapper for HANDLE . . . 836
19.2.3 Directory watcher interface . . . 838
19.2.4 Windows implementation . . . 839
19.2.5 Config-specific filtering . . . 843
19.2.6 Hot-reload loop . . . 844
19.2.7 Why RAII matters here . . . 845
19.3 Merge engine . . . 845
19.3.1 Core merge strategies . . . 845
19.3.2 Merge policy model . . . 846
19.3.3 Helper predicates . . . 846
19.3.4 Recursive deep merge . . . 847
19.3.5 Example . . . 849
19.3.6 Layered configuration stack . . . 850
19.3.7 Design considerations . . . 850
19.3.8 Optional provenance tracking . . . 851
19.4 Visitor-based evaluation . . . 851
19.4.1 Utility visitor helper . . . 852
19.4.2 Pretty-print evaluation . . . 852
19.4.3 Path-based lookup evaluation . . . 854
19.4.4 Typed conversion evaluation . . . 856
19.4.5 Example query API . . . 858
19.4.6 Validation as a visitor . . . 859
19.4.7 Why visitor-based evaluation is a strong fit . . . 859
19.5 End-to-end implementation sketch . . . 860
19.5.1 Using the config system . . . 861
19.6 Windows build guidance . . . 862

19.7 Conclusion 862

20 Testing, Documentation & Extensions **864**

20.1 Unit tests 864

20.1.1 Testing principles 864

20.1.2 Minimal test framework approach 864

20.1.3 Testing JSON parsing 865

20.1.4 Testing merge engine 866

20.1.5 Testing path queries 866

20.1.6 Test runner 867

20.1.7 Modern testing extensions 867

20.2 Error handling 868

20.2.1 Error categories 868

20.2.2 Exception-based handling 868

20.2.3 Throwing structured errors 869

20.2.4 Recoverable errors using optional 869

20.2.5 Error propagation design 870

20.2.6 Error reporting example 870

20.3 Thread safety 870

20.3.1 Immutable configuration model 871

20.3.2 Atomic snapshot swapping 871

20.3.3 Reader threads 872

20.3.4 Thread safety guarantees 872

20.3.5 Alternative: mutex-based model 873

20.3.6 Modern recommendation 874

20.4 Exporting the library as modules 874

20.4.1 Module interface unit 874

20.4.2 Module implementation unit 875

20.4.3 Using the module 875
20.4.4 MSVC build instructions 876
20.4.5 CMake with modules (basic) 876
20.4.6 Advantages of modules 876
20.5 Conclusion 877

VIII Ending of the Volume **878**

Conclusion The Future of Object-Oriented Design in Modern C++ **879**
Lessons learned from modern C++ object design 879
Balancing dynamic and static polymorphism 894
Designing future-proof C++ libraries 899
Where modern C++ architecture is heading 903
Final reflection 906

Appendices **908**
Appendix A The C++ Object Model and ABI Details 908
Appendix B Memory Alignment and Cache Behavior 934
Appendix C Smart Pointer Internals 947
Appendix D RAII Patterns Library 979

References **996**

Authors Introduction

The evolution of C++ has never been linear. It is a language that has grown through decades of real-world engineering, shaped not by theoretical purity alone, but by the demands of performance, reliability, compatibility, and long-term maintainability. For those who have worked closely with it, C++ is not merely a programming language; it is a system of thoughta way of modeling problems where control, abstraction, and efficiency coexist without compromise.

This volume, *Modern C++ Encyclopedia Volume 3: Modern OOP, RAII & Polymorphism*, is written with a very specific purpose: to redefine how object-oriented programming in C++ should be understood in the modern era.

Why This Volume Exists

Object-oriented programming in C++ has often been misunderstood. Many developers approach it through the lens of other languages, where inheritance hierarchies and runtime polymorphism dominate design decisions. However, C++ was never meant to be confined to a single paradigm. From its earliest design, it has supported a richer and more flexible modelone where object orientation is only one part of a broader system that includes value semantics, generic programming, and low-level control.

In modern C++, the meaning of object-oriented design has fundamentally changed.

It is no longer about building deep inheritance trees or relying excessively on virtual dispatch. Instead, it is about:

1. Expressing ownership explicitly
2. Encoding lifetime into types
3. Leveraging RAII for correctness
4. Using composition as the primary structuring tool
5. Applying static polymorphism where it is more efficient and expressive
6. Reserving dynamic polymorphism for true runtime abstraction boundaries

This volume exists to present that modern understanding clearly, precisely, and in a way that connects directly to real-world engineering.

From Classical OOP to Modern C++ Design

Traditional object-oriented programming emphasized inheritance, base classes, and virtual functions as primary tools. While these remain important, they are no longer sufficient to describe modern C++ design.
The shift has been driven by several major developments:

1. The introduction of move semantics and value-oriented design
2. The maturation of RAII as the dominant resource-management model
3. The rise of templates, concepts, and compile-time polymorphism
4. The availability of type-safe alternatives such as `std::variant`

5. The increased importance of performance, cache behavior, and memory layout

As a result, modern C++ design is no longer centered on inheritance hierarchies. It is centered on:

1. Types as carriers of meaning and ownership
2. Composition as the primary method of assembling systems
3. Clear separation between interface boundaries and implementation details
4. Hybrid use of static and dynamic polymorphism

This transformation is at the heart of this volume.

The Role of RAII

If there is a single concept that defines C++ as a systems programming language, it is RAII. RAII is not merely a technique for managing memory. It is a principle that connects resource management with object lifetime in a deterministic and verifiable way. It ensures that:

1. resources are acquired in constructors
2. resources are released in destructors
3. cleanup happens automatically, even in the presence of exceptions

In this volume, RAII is treated not as an isolated topic, but as the backbone of reliable software design. It appears in:

1. smart pointers and ownership models
2. file and handle management

3. synchronization primitives
4. transaction and rollback patterns
5. scope guards and deferred execution

Understanding RAII deeply is essential for mastering modern C++.

Polymorphism Revisited

Polymorphism in C++ is far richer than the traditional notion of virtual functions.
Modern C++ offers multiple forms of polymorphism:

1. Dynamic polymorphism using virtual functions
2. Static polymorphism using templates and CRTP
3. Variant-based polymorphism using `std::variant` and visitation
4. Type erasure techniques for runtime flexibility without inheritance

Each of these techniques has its place. The challenge is not choosing one universally, but understanding when each is appropriate.
This volume explores these forms in depth and shows how they can be combined to produce flexible, efficient, and maintainable systems.

A Systems-Oriented Perspective

One of the distinguishing features of C++ is that it does not separate high-level design from low-level reality. Concepts such as object layout, alignment, cache behavior, and ABI compatibility are not abstract concernsthey directly affect how programs behave in production.
For this reason, this volume integrates topics that are often treated separately:

1. object-oriented design
2. memory layout and alignment
3. cache efficiency and performance
4. compiler ABI and object model details

A modern C++ programmer must understand how abstractions translate into machine-level behavior. Only then can design decisions be made with full awareness of their cost and consequences.

Who This Volume Is For

This volume is written for serious C++ programmers who want to move beyond surface-level understanding and develop a deeper, more precise command of the language.
It is especially suited for:

1. developers with experience in C or C++ who want to master modern design techniques
2. engineers building performance-critical or large-scale systems
3. library authors who need to design stable and expressive APIs
4. programmers transitioning from other languages who want to understand what makes C++ fundamentally different

It assumes familiarity with basic C++ syntax and concepts, but it does not assume prior mastery of modern features.

How to Read This Volume

This book is structured to move from foundational concepts to advanced architectural patterns and finally to real-world system design.
A recommended reading approach is:

1. Study the foundations of object models, constructors, and RAII carefully
2. Pay close attention to sections on polymorphism and type systems
3. Explore design patterns with a critical mindsetunderstand not only how they work, but when they should be avoided
4. Follow the full project sections to see how concepts integrate into real systems
5. Review the appendices for deeper understanding of ABI, memory layout, and performance

This is not a book to read passively. It is meant to be studied, revisited, and applied.

Philosophy of This Work

The philosophy behind this volume can be summarized in a few key principles:

1. **Clarity over tradition**: not all traditional practices remain optimal in modern C++
2. **Semantics over syntax**: types should express meaning, not just structure
3. **Ownership over convenience**: resource management must be explicit
4. **Composition over inheritance**: structure systems by assembling parts, not by forcing hierarchies

5. **Abstraction with awareness**: every abstraction has a cost, and that cost must be understood

These principles guide every chapter in this volume.

A Personal Note

For many years, I have worked with C++ across different generations of compilers, architectures, and application domains. I have seen the language evolve from a tool primarily associated with manual memory management into a sophisticated system capable of expressing high-level abstractions without sacrificing performance.
What has remained constant throughout this evolution is the importance of understandingnot just usingthe language.
This volume is the result of that perspective. It is written to help readers not only write C++ code, but understand why that code works, how it maps to the machine, and how to design systems that remain robust, efficient, and maintainable over time.

Closing

Modern C++ is not a language that rewards superficial knowledge. It rewards precision, discipline, and thoughtful design.
Object-oriented programming in C++ is no longer about following patterns blindly. It is about choosing the right abstractions, expressing intent clearly, and building systems that respect both high-level design and low-level reality.
This volume is an invitation to that level of understanding, It is not the end of the journeybut it is a solid foundation for those who are serious about mastering Modern C++.

Ayman Alheraki

Preface

This volume was written to present **modern C++ object-oriented design** as it is understood in serious contemporary systems development, not merely as it appeared in older introductory textbooks. In the early years of C++, object-oriented programming was often taught through a narrow lens: classes, inheritance trees, virtual functions, abstract base classes, and a strong tendency to model software as deep hierarchies of objects. That historical approach played an important educational role, but modern C++ has grown far beyond that limited interpretation. Today, the most reliable and high-performance C++ systems are designed with a broader and more disciplined architectural mindset. The language is no longer used only as C with classes, nor is modern object-oriented design in C++ defined by maximizing inheritance or the number of virtual interfaces in a codebase. Instead, modern practice emphasizes **ownership clarity**, **deterministic lifetime management**, **resource safety**, **value-oriented design where appropriate**, **careful composition of behavior**, and **precise use of polymorphism only when it is truly justified**.

This development is visible in the engineering culture surrounding **LLVM** and **Clang**. LLVMs current coding guidance and manuals reflect a mature style of large-scale software design: use standard C++ effectively, document intent clearly, reduce unnecessary coupling, and build code that remains maintainable across very large systems. LLVMs published coding standards describe practices for a large, library-based codebase, and LLVM documents its subprojects as using standard **C++17**. At the same time, current Clang documentation shows support across the historical range from **C++98** through **C++26 language modes**. This combination is

important for understanding the purpose of this volume: modern C++ architecture did not appear by rejecting the past, but by refining it through decades of practical experience.
This book therefore treats object-oriented design as part of a larger engineering discipline. It explains not only how classes and polymorphism work, but also how objects behave in memory, how ownership should be represented, how resources should be acquired and released safely, and why a well-designed C++ system must make lifetime rules visible in its types and interfaces. In modern code, correctness depends not just on syntax, but on the relationship between **construction**, **destruction**, **move operations**, **copy control**, **exception safety**, **API boundaries**, and **resource ownership semantics**.
The reader will also see throughout this volume that modern C++ object design is closely related to the practical reality of systems programming. Files must be closed. Memory must be released. locks must be unlocked. Handles must be invalidated correctly. Threads, buffers, streams, allocators, and operating-system resources must be controlled predictably. C++ remains one of the most important languages for these tasks because it provides direct control while still allowing programmers to encode safety through abstractions such as **RAII**, smart pointers, containers, and strongly designed classes.
This volume is especially relevant for programmers working on Windows. All examples are written in standard modern C++ and can be compiled in a Windows environment using **clang-cl**, Visual Studio, or a recent MSVC-compatible toolchain. LLVMs official Windows getting-started guidance documents the Visual Studio based workflow, and Clangs user documentation describes the compiler front-end behavior, options, and supported language modes. For this reason, the examples in this chapter are designed to feel natural in a real Windows development environment rather than in an abstract platform-neutral vacuum.
The purpose of this preface is not only to introduce the subject, but to define the educational philosophy of the volume. The reader is not expected merely to memorize rules about classes or to repeat slogans such as use inheritance carefully or prefer RAII. Instead, the reader is expected to understand **why** these principles emerged, **how** they relate to the C++ object

model, and **where** they change the design of real systems. Each important concept is therefore explained in terms of architecture, language behavior, correctness, and maintainability.

The evolution of C++ object-oriented design

The evolution of object-oriented design in C++ is a movement from **structure-first thinking** toward **lifetime-first thinking**. Earlier code often focused on representing the domain as a hierarchy of classes. Modern code focuses first on ownership, invariants, and correctness boundaries, and only then decides whether inheritance is needed. In other words, the modern question is not How many classes can represent this idea? but rather What owns what, what must be cleaned up, what varies, and how can the design remain stable over time?

This evolution was influenced by both language growth and industrial experience. In older C++ code, a great deal of software relied on manual memory management, raw owning pointers, hand-written cleanup logic, and broad inheritance trees that mixed interfaces with implementation reuse. As modern C++ matured, the language gained stronger standard library facilities, smart pointers, move semantics, stronger type utilities, and better support for expressing ownership explicitly. In parallel, large codebases taught developers that many inheritance-heavy designs were hard to maintain and that deterministic cleanup through RAII was one of the greatest architectural strengths of C++.

LLVM and Clang are especially instructive in this regard. Their documentation does not promote decorative abstraction. Instead, it reflects a practical engineering culture centered on clear APIs, explicit design decisions, low coupling, and maintainable code boundaries. LLVM coding guidance explicitly discusses large library-based code design, practical class rules, virtual method considerations, and the importance of disciplined coding standards in a long-lived codebase. These are not academic preferences; they are responses to the realities of scale.

From C++98 OOP to Modern C++ architecture

The **C++98** era established the language as a serious industrial tool, but its typical object-oriented style was shaped by the limitations and habits of that period. Many programmers wrote code where ownership was implicit, resource cleanup was manual, and inheritance served as the main mechanism for reuse. The common teaching style emphasized base classes, virtual methods, protected members, and runtime polymorphism far more than deterministic lifetime or ownership visibility.

A representative old-style example may look like this:

```
class Device {
public:
    virtual ~Device() {}
    virtual void start() = 0;
    virtual void stop() = 0;
};

class FileDevice : public Device {
private:
    FILE* handle_;

public:
    FileDevice(const char* path) : handle_(std::fopen(path, "wb")) {}

    ~FileDevice() override {
        if (handle_) {
            std::fclose(handle_);
        }
    }
```

```
    void start() override {
        if (handle_) {
            std::fputs("started\n", handle_);
        }
    }

    void stop() override {
        if (handle_) {
            std::fputs("stopped\n", handle_);
        }
    }
};
```

This design is historically understandable, but it reflects several weaknesses common to older code:

- ownership is present but not modeled explicitly as a first-class concept,
- resource cleanup is manual and tied to the programmers discipline,
- copying and assignment behavior may be dangerous unless carefully defined,
- the hierarchy is used by default even when the main challenge is lifetime safety rather than polymorphism.

Modern C++ architecture approaches the same kind of problem differently. It prefers library types that encode destruction automatically, narrow interfaces, composition of focused parts, and explicit ownership semantics. A modern redesign might look like this:

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

```
#include <vector>

class IDevice {
public:
    virtual ~IDevice() = default;
    virtual void start() = 0;
    virtual void stop() = 0;
};

class LogBuffer {
private:
    std::vector<char> data_;

public:
    explicit LogBuffer(std::size_t size = 256) : data_(size) {}

    char* data() noexcept { return data_.data(); }
    const char* data() const noexcept { return data_.data(); }
    std::size_t size() const noexcept { return data_.size(); }
};

class FileDevice final : public IDevice {
private:
    std::ofstream out_;
    LogBuffer buffer_;

public:
    explicit FileDevice(const std::string& path)
        : out_(path, std::ios::binary), buffer_(512) {}
```

```
    void start() override {
        if (out_) {
            out_ << "started\n";
        }
    }

    void stop() override {
        if (out_) {
            out_ << "stopped\n";
        }
    }
};
```

This version is architecturally different in several ways. The file stream already obeys deterministic destruction. The buffer is an owned member with clear lifetime. The interface is narrow. The type is marked `final` because further subclassing is not intended. The class design now communicates intent more clearly than the earlier raw-pointer version.
This shift matches modern official guidance well. LLVMs coding guidance emphasizes maintainability in a large codebase and documents special rules around virtual classes, class design, and interface behavior. Clangs current language documentation also makes it clear that modern compilers operate across a very large spectrum of C++ language modes, allowing developers to understand older code while still writing new code in a more disciplined style.
For Windows users, the same example can be compiled with a current LLVM toolchain through `clang-cl`. A straightforward command is:

```
clang-cl /std:c++20 /EHsc /W4 /O2 modern_device.cpp
```

LLVMs Windows getting-started guidance is centered on Visual Studio workflows, so this style of command is natural in a normal Windows developer environment.

RAII as the foundation of safe systems programming

RAII is the principle that a resource should be acquired during object initialization and released automatically during destruction. This idea is so central to modern C++ that it is impossible to understand professional C++ design without it. RAII is not limited to memory. It applies equally to files, sockets, mutexes, operating-system handles, transactions, temporary state changes, and any other resource that must be released reliably.

The fundamental strength of RAII is that it binds correctness to lifetime. A resource-owning object becomes responsible for cleanup, and scope exit becomes the mechanism that guarantees release. This eliminates large classes of bugs caused by forgotten cleanup, duplicated cleanup paths, early returns, and exception-related resource leaks.

A manual style is fragile:

```
#include <cstdio>

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

    std::fputs("header\n", f);

    if (std::ferror(f)) {
        std::fclose(f);
        return;
    }

    std::fputs("body\n", f);
```

```
    std::fclose(f);
}
```

In this code, every path must remember to clean up manually. As the logic becomes more complex, this approach becomes increasingly error-prone.
The RAII version is simpler and safer:

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

void write_report(const std::string& path) {
    std::ofstream out(path);
    if (!out) {
        return;
    }

    out << "header\n";
    if (!out) {
        return;
    }

    out << "body\n";
}
```

The cleanup is automatic. The programmer no longer has to manually close the file on every exit path. This is the essence of RAII: destruction is part of the design, not an afterthought.
The same principle becomes even more important with synchronization:

```
#include <iostream>
#include <mutex>
```

```cpp
#include <string>

class Logger {
private:
    std::mutex mutex_;

public:
    void write(const std::string& message) {
        std::lock_guard<std::mutex> lock(mutex_);
        std::cout << message << '\n';
    }
};
```

Here the lock is released automatically when the guard object leaves scope. This is one of the clearest demonstrations of how C++ turns scope and destruction into correctness tools.
The current C++ Core Guidelines make this ownership-oriented design explicit. They recommend using `unique_ptr` or `shared_ptr` to represent ownership, preferring `unique_ptr` when ownership is not shared, and using smart pointers in interfaces only when lifetime semantics are part of the function contract. Those rules reflect the modern understanding that lifetime and ownership should be visible directly in types.
A Windows-specific RAII wrapper is often useful when working with platform handles:

```cpp
#include <windows.h>
#include <stdexcept>

class UniqueHandle {
private:
    HANDLE handle_ = INVALID_HANDLE_VALUE;

public:
```

```
    explicit UniqueHandle(HANDLE h) : handle_(h) {
        if (handle_ == INVALID_HANDLE_VALUE || handle_ == nullptr) {
            throw std::runtime_error("invalid handle");
        }
    }

    ~UniqueHandle() {
        if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
            CloseHandle(handle_);
        }
    }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept : handle_(other.handle_) {
        other.handle_ = INVALID_HANDLE_VALUE;
    }

    UniqueHandle& operator=(UniqueHandle&& other) noexcept {
        if (this != &other) {
            if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
                CloseHandle(handle_);
            }
            handle_ = other.handle_;
            other.handle_ = INVALID_HANDLE_VALUE;
        }
        return *this;
    }
```

```
    HANDLE get() const noexcept {
        return handle_;
    }
};
```

This example illustrates several modern lessons at once:

- ownership is unique and explicit,
- copying is disabled to prevent double release,
- movement transfers ownership safely,
- destruction performs cleanup automatically.

A Windows build command using LLVM tools is straightforward:

```
clang-cl /std:c++20 /EHsc /W4 /O2 handle_raii.cpp
```

LLVMs Windows documentation and Clang user documentation make this kind of workflow a normal part of a Visual Studio based toolchain.

The shift from inheritance-heavy design to composition

One of the clearest signs of the maturity of modern C++ design is the move away from inheritance as the default reuse mechanism. In earlier styles, developers often created deep hierarchies even when the real need was simply to share a few behaviors or state components. This usually led to rigid base classes, excessive use of `protected` members, unclear extension rules, and fragile coupling between derived classes and base implementation details.
An inheritance-heavy design may look like this:

```
#include <iostream>
#include <string>

class Window {
protected:
    int width_;
    int height_;
    std::string title_;

public:
    Window(int w, int h, std::string t)
        : width_(w), height_(h), title_(std::move(t)) {}

    virtual ~Window() = default;

    virtual void draw() = 0;

    virtual void resize(int w, int h) {
        width_ = w;
        height_ = h;
    }
};

class DialogWindow : public Window {
public:
    DialogWindow() : Window(400, 300, "Dialog") {}

    void draw() override {
        std::cout << "Drawing dialog: " << title_ << '\n';
```

```
    }
};

class ToolWindow : public Window {
public:
    ToolWindow() : Window(250, 500, "Tool") {}

    void draw() override {
        std::cout << "Drawing tool: " << title_ << '\n';
    }
};
```

This style is not always wrong, but it encourages a common mistake: mixing interface abstraction with implementation reuse in the same base type. The base class carries state and behavior that derived classes inherit whether or not they truly want that representation.
A composition-oriented design separates concerns more cleanly:

```
#include <iostream>
#include <string>
#include <utility>

class Geometry {
private:
    int width_;
    int height_;

public:
    Geometry(int w, int h) : width_(w), height_(h) {}

    void resize(int w, int h) noexcept {
```

```
        width_ = w;
        height_ = h;
    }

    int width() const noexcept { return width_; }
    int height() const noexcept { return height_; }
};

class Caption {
private:
    std::string text_;

public:
    explicit Caption(std::string text) : text_(std::move(text)) {}

    const std::string& text() const noexcept { return text_; }
};

class IRenderable {
public:
    virtual ~IRenderable() = default;
    virtual void draw() const = 0;
};

class DialogWindow final : public IRenderable {
private:
    Geometry geometry_;
    Caption caption_;
```

```
public:
    DialogWindow() : geometry_(400, 300), caption_("Dialog") {}

    void draw() const override {
        std::cout << "Drawing dialog: " << caption_.text()
                  << " [" << geometry_.width()
                  << "x" << geometry_.height() << "]\n";
    }
};

class ToolWindow final : public IRenderable {
private:
    Geometry geometry_;
    Caption caption_;

public:
    ToolWindow() : geometry_(250, 500), caption_("Tool") {}

    void draw() const override {
        std::cout << "Drawing tool: " << caption_.text()
                  << " [" << geometry_.width()
                  << "x" << geometry_.height() << "]\n";
    }
};
```

This second version uses inheritance only for a narrow rendering interface. Reusable state such as geometry and caption handling is provided through contained components. This structure is often easier to extend, test, and maintain. It also keeps the abstraction hierarchy conceptually honest: the renderable interface represents a true polymorphic contract, while shared implementation details remain local and composable.

LLVMs coding standards are particularly useful here because they remind developers that virtual methods and class design choices have real code-generation and maintenance consequences. LLVMs class-related guidance includes special notes about virtual methods and out-of-line virtual anchors, reflecting the fact that virtual hierarchies carry binary and architectural costs. That is precisely why modern C++ uses inheritance more selectively than older style guides often encouraged.

A further composition example uses strategies:

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

class LayoutStrategy {
public:
    virtual ~LayoutStrategy() = default;
    virtual void apply() const = 0;
};

class GridLayout final : public LayoutStrategy {
public:
    void apply() const override {
        std::cout << "Applying grid layout\n";
    }
};

class FlowLayout final : public LayoutStrategy {
public:
    void apply() const override {
        std::cout << "Applying flow layout\n";
```

```
    }
};

class Widget {
private:
    std::string name_;
    std::unique_ptr<LayoutStrategy> layout_;

public:
    Widget(std::string name, std::unique_ptr<LayoutStrategy> layout)
        : name_(std::move(name)), layout_(std::move(layout)) {}

    void render() const {
        std::cout << "Rendering widget: " << name_ << '\n';
        layout_->apply();
    }
};
```

This avoids an explosion of subclasses such as `GridButtonWidget`, `FlowButtonWidget`, `GridDialogWidget`, and many similar combinations. Composition allows orthogonal concerns to remain orthogonal.

For Windows compilation:

```
clang-cl /std:c++20 /EHsc /W4 /O2 composition_example.cpp
```

Concluding perspective

The modern history of C++ object-oriented design can be understood as a steady refinement of responsibility. Early C++ taught programmers how to build classes and hierarchies. Modern

C++ teaches them how to design ownership, control lifetime, protect invariants, and choose the right abstraction mechanism for each problem.

This is why **RAII** became the foundation of safe systems programming. It turns object lifetime into a correctness tool. It is why ownership models became explicit through smart pointers, containers, and move-aware designs. It is why composition gained importance over inheritance-heavy reuse. And it is why current industrial codebases such as LLVM continue to emphasize careful coding standards, maintainable interfaces, and disciplined class design rather than abstraction for its own sake.

The reader should therefore approach modern C++ OOP not as an outdated textbook category, but as a living engineering discipline. In the strongest modern design, every class has a clear purpose, every resource has a visible owner, every cleanup action is deterministic, and every use of inheritance must justify itself against simpler and safer alternatives. When these principles are followed, C++ becomes one of the most powerful languages available for building efficient, reliable, and long-lived software systems.

What this volume teaches

This volume teaches the reader how modern C++ object-oriented design must be understood as a complete engineering model rather than a collection of isolated language features. The reader is guided through the conceptual and practical foundations of class-based design, but always in the broader context of correctness, maintainability, and systems-level reliability.

The subject matter is organized around three major educational goals. First, the reader must understand the **C++ object model**. Second, the reader must learn how to design **safe and efficient ownership models**. Third, the reader must learn how to build **reliable resource-management systems**. These themes are deeply connected. Without understanding the object model, ownership becomes vague. Without ownership clarity, resource management becomes fragile. Without reliable resource management, object-oriented design becomes

unsafe regardless of how elegant its class hierarchy may appear.

Understanding the C++ object model

A central goal of this volume is to help the reader understand what a C++ object really is, both as a language concept and as a practical implementation entity. In ordinary teaching material, the word *object* is often used loosely, as though it simply means an instance of a class. While that is not wrong, it is incomplete. A serious C++ programmer must understand the object model at a deeper level.

From the compilers perspective, class types are represented structurally. In Clangs internal representation, records and C++ class types are modeled through declaration nodes such as `RecordDecl` and `CXXRecordDecl`, while member functions are represented by declarations such as `CXXMethodDecl`. This is valuable not only to compiler developers, but also to advanced C++ programmers, because it reveals that a C++ class is not magical. It is a precise composition of fields, member functions, constructors, destructors, base-class relationships, access rules, and special semantic properties. Understanding this mental model helps the programmer design classes with more discipline.

The object model includes several important realities:

- an object has lifetime, not merely storage,
- a class may define invariants that must hold after construction and before destruction,
- member functions operate in the context of an object state,
- special member functions shape copyability, movability, and destruction semantics,
- polymorphic classes carry additional conceptual and binary implications,
- object layout and representation influence efficiency, ABI behavior, and maintainability.

The practical consequence is that a good C++ class is not just a container of functions. It is a boundary of meaning. It controls initialization, preserves invariants, defines ownership responsibilities, and restricts how state may evolve. If these responsibilities are neglected, the class may compile successfully while still being poorly designed.

A simple educational example may begin with a small class whose state and invariants are explicit:

```
#include <string>
#include <stdexcept>
#include <utility>

class Account {
private:
    std::string owner_;
    double balance_;

public:
    Account(std::string owner, double initial_balance)
        : owner_(std::move(owner)), balance_(initial_balance) {
        if (balance_ < 0.0) {
            throw std::invalid_argument("negative initial balance");
        }
    }

    const std::string& owner() const noexcept {
        return owner_;
    }

    double balance() const noexcept {
        return balance_;
```

```
    }

    void deposit(double amount) {
        if (amount <= 0.0) {
            throw std::invalid_argument("deposit must be positive");
        }
        balance_ += amount;
    }

    void withdraw(double amount) {
        if (amount <= 0.0 || amount > balance_) {
            throw std::invalid_argument("invalid withdrawal");
        }
        balance_ -= amount;
    }
};
```

This example is modest, but it illustrates a major principle of the object model: the object owns its state, its constructor establishes validity, and its member functions preserve the class invariant. A class should not merely expose data; it should guard meaning.

The reader will also learn that understanding the object model requires attention to copy and move behavior. In modern C++, objects are often transferred rather than copied. A programmer who ignores move semantics or special member function rules may accidentally create inefficient or dangerous designs. This is especially important when a class manages a resource or expresses unique ownership.

Consider the following example:

```
#include <memory>
#include <string>
#include <utility>
```

```
class DocumentBuffer {
private:
    std::string name_;
    std::unique_ptr<char[]> data_;
    std::size_t size_;

public:
    DocumentBuffer(std::string name, std::size_t size)
        : name_(std::move(name)),
          data_(std::make_unique<char[]>(size)),
          size_(size) {}

    DocumentBuffer(const DocumentBuffer&) = delete;
    DocumentBuffer& operator=(const DocumentBuffer&) = delete;

    DocumentBuffer(DocumentBuffer&&) noexcept = default;
    DocumentBuffer& operator=(DocumentBuffer&&) noexcept = default;

    const std::string& name() const noexcept {
        return name_;
    }

    std::size_t size() const noexcept {
        return size_;
    }

    char* data() noexcept {
        return data_.get();
```

```
    }

    const char* data() const noexcept {
        return data_.get();
    }
};
```

This example teaches several lessons about the object model:

- the class has unique ownership of its storage,
- copying is forbidden because duplicating ownership here would be unsafe or expensive,
- moving is permitted because ownership transfer is meaningful,
- the type itself communicates lifecycle rules.

This is precisely the kind of object-model thinking that modern C++ requires. The object is not only a bundle of methods. It is a domain-specific owner with carefully chosen semantic operations.
On Windows, a reader may compile this example with:

```
clang-cl /std:c++20 /EHsc /W4 /O2 object_model_example.cpp
```

When using Visual Studio Developer Command Prompt, this command is natural and consistent with LLVMs documented Windows workflow. If desired, the same source can also be placed inside a normal Visual Studio project configured to use LLVM tools.

Designing safe and efficient ownership models

A second major purpose of this volume is to teach how ownership should be represented explicitly in modern C++. Ownership is one of the most important architectural ideas in the

language. Without a clear ownership model, systems become difficult to reason about, object lifetime becomes ambiguous, and resource cleanup becomes unreliable.

In older code, ownership was often hidden behind raw pointers. A pointer field might or might not own the memory it referenced. A function might return a pointer that the caller was expected to delete, or perhaps not. Such designs forced programmers to rely on comments, memory discipline, and shared assumptions. Modern C++ rejects that ambiguity. Ownership should be visible in types and interfaces whenever possible.

The current ownership-oriented style aligns strongly with the guidance widely taught in modern C++ practice. `std::unique_ptr` expresses exclusive ownership. `std::shared_ptr` expresses shared ownership. References and raw pointers may express non-owning access, but they should not silently carry ownership unless an interface makes that intention exceptionally clear.

A weak older style may look like this:

```
class Widget {
public:
    void draw() {}
};

class Screen {
private:
    Widget* widget_;

public:
    Screen() : widget_(new Widget) {}

    ~Screen() {
        delete widget_;
    }
```

```
    Widget* widget() {
        return widget_;
    }
};
```

At first glance, this seems manageable, but it contains several risks:

- copy behavior is dangerous unless explicitly controlled,
- ownership is encoded manually,
- exception safety must be reasoned about carefully,
- interfaces may expose raw owning state too freely.

A stronger modern redesign is:

```
#include <memory>

class Widget {
public:
    void draw() const {}
};

class Screen {
private:
    std::unique_ptr<Widget> widget_;

public:
    Screen() : widget_(std::make_unique<Widget>()) {}

    Screen(const Screen&) = delete;
```

```
    Screen& operator=(const Screen&) = delete;

    Screen(Screen&&) noexcept = default;
    Screen& operator=(Screen&&) noexcept = default;

    Widget& widget() noexcept {
        return *widget_;
    }

    const Widget& widget() const noexcept {
        return *widget_;
    }
};
```

This version teaches modern ownership discipline:

- the ownership is explicit,
- destruction is automatic,
- move semantics are natural,
- the interface exposes access without exposing ownership confusion.

The reader will also learn that efficient ownership models are not merely about choosing a smart pointer. They require architectural thought. A type should own only what it is responsible for. Ownership should be as local as possible. Shared ownership should be used only when there is a real shared-lifetime requirement. Non-owning relationships should remain clearly non-owning.
A more realistic composition example is the following:

```
#include <memory>
#include <string>
#include <utility>
#include <vector>

class Texture {
private:
    std::string file_name_;

public:
    explicit Texture(std::string file_name)
        : file_name_(std::move(file_name)) {}

    const std::string& file_name() const noexcept {
        return file_name_;
    }
};

class Sprite {
private:
    Texture* texture_;

public:
    explicit Sprite(Texture* texture) : texture_(texture) {}

    void render() const {
        if (texture_) {
        }
    }
```

```
};

class Scene {
private:
    std::vector<std::unique_ptr<Texture>> textures_;
    std::vector<Sprite> sprites_;

public:
    Texture& create_texture(const std::string& file_name) {
        textures_.push_back(std::make_unique<Texture>(file_name));
        return *textures_.back();
    }

    void create_sprite(Texture& texture) {
        sprites_.emplace_back(&texture);
    }
};
```

This example is educational because it separates owning and non-owning roles. The `Scene` owns the textures. The `Sprite` merely refers to a texture that must outlive it. Such distinctions are at the heart of safe ownership design.

This volume also teaches that ownership decisions affect API design. A function parameter passed by reference is not the same as a transferred `unique_ptr`. A returned object by value is not the same as a borrowed pointer. A stored `shared_ptr` has lifetime implications that can spread through the entire architecture. Therefore, ownership is not an implementation detail. It is one of the most important design decisions in a C++ system.

For Windows compilation:

```
clang-cl /std:c++20 /EHsc /W4 /O2 ownership_models.cpp
```

A Visual Studio user may place such examples in a console project, select the LLVM toolset if installed, and compile them directly in the IDE or from the Developer Command Prompt.

Building reliable resource-management systems

The third major teaching goal of this volume is to show how reliable systems are built by making resource management structural rather than manual. In modern C++, the principal mechanism for this is **RAII**, the discipline that binds resource acquisition to object initialization and resource release to object destruction.

A resource-management system becomes reliable when the programmer no longer depends on memory of cleanup steps scattered across control flow. Instead, resources are wrapped in objects whose destructors enforce cleanup automatically. This applies not only to heap memory, but also to files, locks, network resources, database handles, operating-system objects, temporary states, and every other resource that must be released exactly once.

A manual style is fragile:

```
#include <cstdio>

void save_text(const char* path, const char* text) {
    FILE* file = std::fopen(path, "w");
    if (!file) {
        return;
    }

    std::fputs(text, file);

    if (std::ferror(file)) {
        std::fclose(file);
        return;
```

```
    }

    std::fclose(file);
}
```

The code above is simple, but it scales poorly as logic grows. Every exit path must remember cleanup. Every change increases the chance of human error.
A more reliable RAII-based version is:

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

void save_text(const std::string& path, const std::string& text) {
    std::ofstream file(path);
    if (!file) {
        return;
    }

    file << text;
}
```

This is a major design improvement. Cleanup is automatic, and the lifetime of the file resource is tied directly to the lifetime of the stream object.
The same principle becomes even more important with synchronization:

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

class SafePrinter {
```

```
private:
    std::mutex mutex_;

public:
    void print(const std::string& text) {
        std::lock_guard<std::mutex> lock(mutex_);
        std::cout << text << '\n';
    }
};
```

This example teaches a profound lesson: scope itself becomes part of the correctness mechanism. The lock is released at scope exit automatically, regardless of how control leaves the function.

A robust resource-management system also often requires custom RAII wrappers. This is especially relevant on Windows, where a program may acquire operating-system handles that must be closed correctly. A minimal educational wrapper is:

```
#include <windows.h>
#include <stdexcept>

class FileHandle {
private:
    HANDLE handle_ = INVALID_HANDLE_VALUE;

public:
    explicit FileHandle(HANDLE handle) : handle_(handle) {
        if (handle_ == INVALID_HANDLE_VALUE || handle_ == nullptr) {
            throw std::runtime_error("failed to acquire HANDLE");
        }
    }
```

```
    ~FileHandle() {
        if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
            CloseHandle(handle_);
        }
    }

    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    FileHandle(FileHandle&& other) noexcept : handle_(other.handle_) {
        other.handle_ = INVALID_HANDLE_VALUE;
    }

    FileHandle& operator=(FileHandle&& other) noexcept {
        if (this != &other) {
            if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
                CloseHandle(handle_);
            }
            handle_ = other.handle_;
            other.handle_ = INVALID_HANDLE_VALUE;
        }
        return *this;
    }

    HANDLE get() const noexcept {
        return handle_;
    }
};
```

This class demonstrates the architecture of reliable resource management:

- acquisition happens under control,
- destruction guarantees cleanup,
- copying is forbidden to prevent double release,
- moving transfers responsibility safely.

A small usage example may be written as follows:

```
#include <windows.h>

int main() {
    HANDLE raw = CreateFileA(
        "sample.txt",
        GENERIC_WRITE,
        0,
        nullptr,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        nullptr
    );

    FileHandle file(raw);

    const char text[] = "Hello from RAII on Windows\n";
    DWORD written = 0;

    WriteFile(file.get(), text, sizeof(text) - 1, &written, nullptr);
}
```

This is the style of system construction that this volume teaches: resources must not float through a program as unmanaged values. They must be embedded inside lifetime-aware types. On Windows, a reader may compile such code with:

```
clang-cl /std:c++20 /EHsc /W4 /O2 windows_raii.cpp
```

If the source uses Windows APIs, it should be compiled from a Visual Studio Developer Command Prompt or an IDE environment where the Windows SDK is available.

Concluding perspective

What this volume teaches, therefore, is not a narrow or outdated picture of C++ object-oriented programming. It teaches the professional foundations of modern C++ design. The reader will learn how objects are structured conceptually, how class semantics influence correctness, how ownership should be expressed directly in the type system, and how resource management must be built into the architecture of the program itself.

To understand C++ well is to understand far more than class syntax. It is to understand object lifetime, invariants, interface discipline, ownership transfer, destruction semantics, and the relationship between abstraction and correctness. This volume is intended to help the reader build that understanding in a systematic and practical way.

By the end of this volume, the reader should be able to analyze a class not merely by asking whether it compiles, but by asking deeper questions. What does it own. What invariants does it preserve. What operations are safe. What resources does it manage. Is its interface honest about lifetime. Is its design efficient. Is its polymorphism justified. Is its cleanup deterministic. Those are the questions that shape modern C++ engineering, and they are the questions this volume is written to teach.

Who this volume is for

This volume is written for programmers who already possess a basic understanding of C++ syntax and who want to deepen their mastery of the language as an engineering tool for building robust and scalable systems. While the concepts discussed here are valuable for all C++ developers, the material is especially relevant to three professional groups: systems programmers, library designers, and engineers responsible for large-scale C++ applications.

Systems programmers

Systems programmers work close to the underlying hardware or operating system. Their work often involves writing components such as compilers, operating system services, device drivers, networking frameworks, high-performance servers, and low-level runtime libraries. In such environments, correctness, efficiency, and deterministic control of resources are essential. Modern C++ provides systems programmers with a powerful combination of low-level control and high-level abstraction. Through careful class design and RAII-based techniques, resources such as memory buffers, file descriptors, sockets, synchronization primitives, and operating-system handles can be managed safely without sacrificing performance.
Consider a simple example where a systems component must manage a dynamically allocated buffer used for binary processing.

```
#include <memory>
#include <cstddef>
#include <cstring>

class BinaryBuffer {
private:
    std::unique_ptr<unsigned char[]> data_;
    std::size_t size_;
```

```
public:
    explicit BinaryBuffer(std::size_t size)
        : data_(std::make_unique<unsigned char[]>(size)),
          size_(size) {}

    std::size_t size() const noexcept {
        return size_;
    }

    unsigned char* data() noexcept {
        return data_.get();
    }

    const unsigned char* data() const noexcept {
        return data_.get();
    }

    void clear() noexcept {
        std::memset(data_.get(), 0, size_);
    }
};
```

In this example, the class expresses ownership explicitly using `std::unique_ptr`. The lifetime of the allocated memory is tied directly to the lifetime of the object. When the object is destroyed, the memory is automatically released. This approach prevents memory leaks and ensures deterministic cleanup without requiring manual calls to `delete[]`.

Systems programmers also frequently interact with operating-system resources. RAII wrappers allow these resources to be handled safely. A Windows-specific example demonstrates this

principle.

```
#include <windows.h>
#include <stdexcept>

class UniqueHandle {
private:
    HANDLE handle_;

public:
    explicit UniqueHandle(HANDLE handle)
        : handle_(handle) {
        if (handle_ == INVALID_HANDLE_VALUE || handle_ == nullptr) {
            throw std::runtime_error("invalid handle");
        }
    }

    ~UniqueHandle() {
        if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
            CloseHandle(handle_);
        }
    }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept
        : handle_(other.handle_) {
        other.handle_ = INVALID_HANDLE_VALUE;
    }
```

```
    UniqueHandle& operator=(UniqueHandle&& other) noexcept {
        if (this != &other) {
            if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
                CloseHandle(handle_);
            }
            handle_ = other.handle_;
            other.handle_ = INVALID_HANDLE_VALUE;
        }
        return *this;
    }

    HANDLE get() const noexcept {
        return handle_;
    }
};
```

Systems developers who master these patterns gain the ability to build reliable infrastructure software where resource lifetime is predictable and failures are easier to diagnose.
To compile such code on Windows:

```
clang-cl /std:c++20 /EHsc /W4 /O2 systems_example.cpp
```

Library designers

Library designers occupy a special role in the C++ ecosystem. Instead of writing applications directly, they create reusable components that other programmers depend upon. These libraries may provide data structures, networking facilities, serialization frameworks, graphics engines, mathematical tools, or application frameworks.

Designing a C++ library requires careful attention to interface stability, ownership semantics, and separation between implementation details and public APIs. A poorly designed library can impose hidden lifetime rules, expose internal structures unnecessarily, or create performance bottlenecks through unnecessary abstraction.

One important principle is that a library interface should clearly communicate ownership relationships. The following example illustrates a small library-style resource manager that provides controlled access to objects.

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

class Resource {
public:
    void initialize() {}
};

class ResourceManager {
private:
    std::vector<std::unique_ptr<Resource>> resources_;

public:
    Resource& create_resource() {
        resources_.push_back(std::make_unique<Resource>());
        return *resources_.back();
    }

    std::size_t resource_count() const noexcept {
        return resources_.size();
    }
};
```

In this design:

- the manager owns all resources,
- ownership is expressed through `std::unique_ptr`,
- callers receive references rather than ownership transfers,
- the internal container remains hidden from the public API.

Such patterns are essential in large library frameworks where memory ownership must remain predictable and interfaces must remain stable across many versions.
Library designers must also consider compile-time dependencies and ABI stability. A well-designed C++ library often hides implementation details behind private members, avoids exposing unnecessary headers, and carefully structures its class hierarchy to minimize coupling between modules.

Engineers building large-scale C++ applications

Large-scale C++ applications present unique challenges. These systems may contain hundreds or thousands of source files, multiple subsystems, and complex interactions between modules. Without disciplined design principles, such codebases can quickly become difficult to maintain.
In large systems, the importance of clear ownership rules and resource management cannot be overstated. Objects must have well-defined lifetimes, subsystems must communicate through stable interfaces, and resource cleanup must occur automatically to prevent leaks and undefined behavior.
A simple example of subsystem composition illustrates this principle.

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

```
class Service {
public:
    void start() {}
};

class Application {
private:
    std::vector<std::unique_ptr<Service>> services_;

public:
    void add_service(std::unique_ptr<Service> service) {
        services_.push_back(std::move(service));
    }

    void start_all() {
        for (auto& service : services_) {
            service->start();
        }
    }
};
```

In this design, the application object owns the services it manages. The ownership transfer occurs through `std::unique_ptr`. Once the service is added to the application, its lifetime becomes tied to the application itself. When the application is destroyed, all services are automatically cleaned up.

This approach scales well in large architectures because ownership relationships remain explicit and deterministic.

To compile the example on Windows:

```
clang-cl /std:c++20 /EHsc /W4 /O2 application_example.cpp
```

Large systems also benefit from strong separation between modules, disciplined use of RAII, and avoidance of unnecessary global state. Engineers working on such projects must treat resource lifetime and object ownership as architectural concerns rather than implementation details.

Conclusion

The readers of this volume share a common goal: to understand how modern C++ can be used to build reliable, efficient, and maintainable software systems. Whether working on system infrastructure, reusable libraries, or complex applications, programmers must develop a deep understanding of the C++ object model, ownership semantics, and deterministic resource management.

By studying these topics in detail, the reader will gain the ability to design classes that express clear responsibilities, manage resources safely, and integrate smoothly into large software architectures. These capabilities are essential for anyone who wishes to use C++ as a professional engineering language rather than merely as a programming tool.

Part I

Foundations of Classes, Objects & C++ Object Model

Anatomy of an Object in Modern C++

1.1 What Exactly Is an Object?

In modern C++, an **object** is not merely something created from a class. That popular simplification is useful for beginners, but it is far too shallow for serious systems work, library design, compiler-oriented reasoning, or large-scale architectural development. In the language itself, the notion of an object is tied to **identity**, **lifetime**, **storage**, **representation**, and **type**. A correct understanding of these ideas is essential for writing safe, efficient, and standards-conforming code.

An object in C++ occupies storage during some portion of program execution, has a type, participates in initialization and destruction rules, and may be referred to through expressions whose value categories determine whether they denote identity or merely compute values. This distinction is one of the foundations of the modern C++ object model. A programmer who understands only syntax may write code that compiles, yet still fails to reason correctly about aliasing, lifetime, object layout, or resource ownership. By contrast, a programmer who understands the anatomy of an object can design stronger classes, detect undefined behavior earlier, and make better decisions about interfaces, memory management, and performance. This chapter therefore examines the object from four closely connected perspectives:

- object identity,
- object versus value versus type,

- storage and representation,
- strict aliasing rules.

These are not isolated topics. They form a single conceptual system. Identity explains what it means for an expression to designate a specific entity. Type explains what kind of entity it is and what operations are valid. Storage and representation explain how the object inhabits memory. Strict aliasing explains the rules that constrain how that object may be accessed through typed expressions. Together, these topics form the real foundation of object-oriented and systems-level programming in modern C++.

1.1.1 Object identity

Object identity is one of the most important and most misunderstood ideas in C++. An object has identity when it is treated as a distinct entity in the program, not merely as a temporary value. Identity answers the question: *which exact object is this expression talking about?*
In the modern language model, the distinction between **glvalues** and **prvalues** is central. A glvalue is associated with identity. A prvalue, by contrast, typically represents a value to be used for computation or initialization rather than directly naming a persistent object. This distinction matters because object-oriented reasoning depends on identity. You do not assign to a pure abstract value in memory; you assign to an object. You do not mutate a temporary mathematical concept; you mutate a particular entity whose lifetime and state exist in the program.
Consider the following example:

```
#include <iostream>
#include <string>

class Counter {
private:
```

```
    int value_;

public:
    explicit Counter(int value = 0) : value_(value) {}

    void increment() noexcept {
        ++value_;
    }

    int value() const noexcept {
        return value_;
    }
};

int main() {
    Counter c{10};
    Counter& ref = c;

    ref.increment();

    std::cout << c.value() << '\n';
}
```

In this example, `c` is an object with identity. The reference `ref` is not a new object of type `Counter`; it is another name bound to the same existing object. When `ref.increment()` is called, the state change occurs in that same object. Identity is what makes this reasoning possible.

Now compare this with a pure value computation:

```
int x = 10;
```

```
int y = x + 5;
```

The expression `x + 5` computes a value. It does not designate a distinct object in the same sense that `x` does. This is why value category matters. In serious C++ programming, especially where temporaries, move semantics, overload resolution, and object lifetime all interact, confusing identity with mere value computation leads to subtle mistakes.
Identity also matters in object-oriented design because class member functions generally operate on objects that have stable state and lifetime. The meaning of `this` inside a non-static member function depends entirely on the fact that the function is acting on a particular object with identity.

```
#include <iostream>
#include <string>

class Employee {
private:
    std::string name_;

public:
    explicit Employee(std::string name) : name_(std::move(name)) {}

    void rename(const std::string& new_name) {
        this->name_ = new_name;
    }

    const std::string& name() const noexcept {
        return name_;
    }
};
```

```
int main() {
    Employee e{"Ayman"};
    e.rename("Engineer Ayman");
    std::cout << e.name() << '\n';
}
```

The member function `rename()` is not operating on an abstract type. It is operating on a specific object designated by `this`. The idea of identity is therefore woven directly into the structure of class-based programming.

For practical work on Windows, this example may be compiled from the Visual Studio Developer Command Prompt with:

```
clang-cl /std:c++20 /EHsc /W4 /O2 object_identity.cpp
```

1.1.2 Object vs value vs type

A second essential distinction in C++ is the difference between an **object**, a **value**, and a **type**. These three terms are often mixed together in casual discussion, but they are not interchangeable.

A **type** is a specification. It describes what kind of entity something is, what operations are valid, what representation constraints may exist, and how the compiler should interpret expressions involving that entity. A class is a type. An `int` is a type. A pointer to `double` is a type. A function type is a type. Types belong to the static structure of the language.

A **value** is the abstract content associated with an expression or stored in an object. For an integer object, the value may be `42`. For a boolean object, it may be `true`. For a class type, the notion of value may be more structured and is often understood through the objects observable state.

An **object** is a runtime entity that occupies storage during some period of execution and has a type. The object may contain or represent a value, but it is not the same thing as the value itself. The following example shows the difference clearly:

```cpp
#include <iostream>

int main() {
    int a = 42;
    int b = 42;

    std::cout << "a = " << a << '\n';
    std::cout << "b = " << b << '\n';
}
```

Here:

- `int` is the type,
- `42` is the value,
- `a` and `b` are two distinct objects.

They hold equal values, but they are not the same object. They occupy different storage and have different identities. This distinction is fundamental in all serious reasoning about assignment, aliasing, mutation, references, and ownership.
For class types, the distinction becomes even more important:

```cpp
#include <iostream>
#include <string>

class Point {
private:
    int x_;
    int y_;
```

```
public:
    Point(int x, int y) : x_(x), y_(y) {}

    int x() const noexcept { return x_; }
    int y() const noexcept { return y_; }
};

int main() {
    Point p1{3, 4};
    Point p2{3, 4};

    std::cout << "(" << p1.x() << ", " << p1.y() << ")\n";
    std::cout << "(" << p2.x() << ", " << p2.y() << ")\n";
}
```

The type is `Point`. The two objects are `p1` and `p2`. Their values, in the ordinary state-based sense, are equal. Yet they remain different objects with different identities and different storage. This matters whenever references, addresses, mutation, ownership, or polymorphic behavior enter the design.

This distinction also clarifies why a class itself is not an object. A class is a type definition. Objects are instances of that type, created during execution with storage and lifetime. A type exists at the level of language structure; an object exists at runtime; a value is the state or abstract content associated with the object or expression.

1.1.3 Storage & representation

Every real object in C++ lives in storage. Storage is the memory made available to the program as sequences of contiguous bytes, each with a unique address. The object occupies some portion of that storage according to its type and representation rules. This is one of the most important bridges between the abstract language and the physical machine.

Storage should be understood in two related senses. First, there is the **region of memory** used for the object. Second, there is the **object representation**, meaning the set of bits and bytes that encode the objects stored state. A correct understanding of storage and representation is especially important for low-level programming, serialization, ABI-sensitive code, compiler work, binary protocols, and performance engineering.
Consider a simple scalar object:

```
#include <iostream>

int main() {
    int value = 123;
    std::cout << "Address: " << &value << '\n';
    std::cout << "Value: " << value << '\n';
}
```

The object `value` has type `int`, occupies storage somewhere in program memory, and has an implementation-defined representation as bits in that memory. Its address identifies the start of the storage used for that object.
For class types, representation includes members and possibly base-class subobjects:

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

class Record {
private:
    int id_;
    double score_;

public:
```

```
    Record(int id, double score) : id_(id), score_(score) {}

    int id() const noexcept { return id_; }
    double score() const noexcept { return score_; }
};

int main() {
    std::cout << "sizeof(Record) = " << sizeof(Record) << '\n';
    std::cout << "alignof(Record) = " << alignof(Record) << '\n';

    Record r{7, 98.5};
    std::cout << "id = " << r.id() << ", score = " << r.score() << '\n';
}
```

This example teaches several important facts:

- an objects storage size may include padding,
- alignment requirements are part of the representation story,
- class objects consist of subobjects, not merely a flat mathematical value,
- the physical representation is constrained by the implementation and ABI.

For a systems programmer, representation is not an academic concern. It affects interoperability, memory layout, caching behavior, binary compatibility, and optimization. Yet modern C++ also requires discipline: not every byte-level manipulation that appears possible is permitted by the language rules. That is why one must distinguish between what memory physically contains and what access patterns the type system actually allows.
If byte-level inspection is needed, C++ permits careful access through character-like byte types. A safe illustrative pattern is:

```cpp
#include <cstddef>
#include <cstdint>
#include <iomanip>
#include <iostream>

int main() {
    std::uint32_t number = 0x12345678u;

    const unsigned char* bytes =
        reinterpret_cast<const unsigned char*>(&number);

    for (std::size_t i = 0; i < sizeof(number); ++i) {
        std::cout << std::hex
                  << std::setw(2)
                  << std::setfill('0')
                  << static_cast<unsigned>(bytes[i])
                  << ' ';
    }

    std::cout << '\n';
}
```

This code inspects the object representation byte by byte. It is useful for education, debugging, binary protocols, and systems work. It also prepares the reader for a deeper understanding of aliasing rules, because not every kind of pointer reinterpretation is valid simply because bytes exist in memory.
On Windows, compile with:

```
clang-cl /std:c++20 /EHsc /W4 /O2 storage_representation.cpp
```

1.1.4 Strict aliasing rules

Strict aliasing is one of the most important correctness rules in low-level C++ programming. It is also one of the most common sources of subtle undefined behavior. The rule exists because compilers, including Clang and LLVM-based optimizers, rely on **type-based alias analysis** to reason about whether two expressions may refer to the same memory. If the programmer violates those rules, the optimizer may legally transform the code in ways that produce unexpected results.

In practical terms, strict aliasing means that an object in memory must generally be accessed through an lvalue or glvalue of an appropriate type, with only limited exceptions. The fact that two different pointer types can be forced to point to the same address does not make such access valid in the language.

A classic unsafe example is:

```
#include <iostream>

int main() {
    float f = 1.0f;
    int* p = reinterpret_cast<int*>(&f);

    std::cout << *p << '\n';
}
```

This code may compile, but it is not a sound way to reinterpret the stored bits of `f` as an `int`. The problem is not merely style. It is that such access can violate the languages aliasing rules and therefore invoke undefined behavior.

A safer modern alternative is to copy the representation rather than alias the storage as a different type. For trivially copyable value-to-value reinterpretation, the modern language provides `std::bit_cast`.

```cpp
#include <bit>
#include <cstdint>
#include <iostream>

int main() {
    float f = 1.0f;
    std::uint32_t bits = std::bit_cast<std::uint32_t>(f);

    std::cout << bits << '\n';
}
```

This performs a value-level bit reinterpretation safely for suitable types, without pretending that the object itself may simply be accessed as an unrelated type. Another traditional safe technique is `std::memcpy` when byte copying is the real intent.

```cpp
#include <cstdint>
#include <cstring>
#include <iostream>

int main() {
    float f = 1.0f;
    std::uint32_t bits = 0;

    std::memcpy(&bits, &f, sizeof(bits));

    std::cout << bits << '\n';
}
```

These approaches are important because they preserve the distinction between copying representation and violating aliasing rules through typed access.

Clangs documentation is especially clear that code violating strict aliasing has undefined behavior, that optimizer behavior may change from one version to another, and that code known to violate these rules should either be fixed or, if necessary as a last resort, built with `-fno-strict-aliasing`. Clang also documents an experimental `TypeSanitizer` that can help detect strict type-aliasing violations in instrumented builds. For practical investigation on Windows with Clang-based workflows, the programmer should therefore treat aliasing correctness as a first-class design concern, not an optional detail.
A safe practical lesson is this:

- do not assume that two unrelated pointer types may legally inspect the same object,
- use references and pointers that match the true type of the object,
- use `std::bit_cast` for value reinterpretation where appropriate,
- use `std::memcpy` for raw byte transfer,
- reserve low-level casts for carefully justified cases,
- when debugging suspicious low-level code, consider a Clang-based build strategy that checks for aliasing-related problems.

For Windows development, a normal build command is:

```
clang-cl /std:c++20 /EHsc /W4 /O2 strict_aliasing_examples.cpp
```

If a codebase contains legacy constructs that are known to depend on aliasing violations and cannot be corrected immediately, a reduced-optimization compatibility build may use:

```
clang-cl /std:c++20 /EHsc /W4 /O2 /clang:-fno-strict-aliasing legacy_code.cpp
```

That switch should not be treated as a design solution. The correct long-term solution is to rewrite the code so that the object is accessed through valid types and safe representation-copying techniques.

1.1.5 Closing perspective

A modern C++ object is not merely a programming convenience. It is a precisely constrained runtime entity with type, identity, storage, representation, lifetime, and access rules. Once this is understood, many parts of modern C++ become easier to reason about. References make more sense because identity matters. Move semantics make more sense because objects own resources and lifetimes. Class invariants make more sense because an object is more than a bag of fields. Strict aliasing makes more sense because the optimizer depends on type-correct access to memory.

This is why the anatomy of an object stands at the beginning of serious C++ study. A programmer who understands objects only as instances of classes has learned the surface of the language. A programmer who understands object identity, value categories, storage, representation, and aliasing rules has begun to understand the language as an engineering system.

1.2 Object Memory Layout

Object memory layout is one of the most important foundations of modern C++ systems programming. A programmer may write classes, structs, and polymorphic types at a high level, but every real object still occupies storage, obeys alignment constraints, may contain padding, and is represented in memory according to rules that are partly defined by the language and partly determined by the target ABI and implementation. For this reason, any serious discussion of modern C++ object-oriented design must include a precise understanding of layout.

In practice, object memory layout affects far more than curiosity about `sizeof`. It influences cache behavior, memory bandwidth, ABI compatibility, binary serialization, interoperability with C and operating-system APIs, correctness of low-level casts, and the reliability of performance-sensitive systems. The design of a class is therefore not only a matter of interface

and semantics. It is also a matter of representation.
This section explains object memory layout through four closely related themes:

- layout rules,
- alignment and padding,
- bit-fields and layout traps,
- field reordering for performance.

These topics must be understood together. Layout rules explain the formal and implementation-governed structure of an object. Alignment and padding explain why objects often occupy more bytes than the sum of their visible members. Bit-fields show that compact representation can introduce portability and correctness hazards. Field reordering demonstrates that physical arrangement inside a class can materially affect performance and memory efficiency.

1.2.1 Layout rules

The layout of an object in C++ is constrained by both the language and the implementation. At the language level, the programmer knows that a class object consists of its non-static data members and any base-class subobjects. At the implementation level, the compiler and target ABI determine exact offsets, padding, alignment boundaries, vptr placement for polymorphic classes, and other representation details.
This means that C++ offers **partial predictability**, not unlimited predictability. Some things are guaranteed or strongly constrained. Other things are implementation-defined or ABI-dependent. A disciplined C++ programmer must know the difference.
A simple example is a plain aggregate-like structure:

```
#include <cstddef>
```

```cpp
#include <iostream>

struct PlainRecord {
    int id;
    double score;
    char flag;
};

int main() {
    std::cout << "sizeof(PlainRecord) = " << sizeof(PlainRecord) << '\n';
    std::cout << "alignof(PlainRecord) = " << alignof(PlainRecord) << '\n';
    std::cout << "offsetof(id) = " << offsetof(PlainRecord, id) << '\n';
    std::cout << "offsetof(score) = " << offsetof(PlainRecord, score) << '\n';
    std::cout << "offsetof(flag) = " << offsetof(PlainRecord, flag) << '\n';
}
```

This example introduces several practical rules:

- each non-static data member has an offset within the object,
- the object as a whole has an alignment requirement,
- the compiler may insert padding between members or at the end of the object,
- the total size of the object is generally rounded to satisfy alignment constraints.

The language does not promise that the size will equal the simple arithmetic sum of member sizes. If `int` is 4 bytes, `double` is 8 bytes, and `char` is 1 byte, the apparent total is 13 bytes. Yet the actual `sizeof(PlainRecord)` may be larger because of alignment and tail padding.
The concept of **standard-layout** types is important here. A standard-layout class is subject to stronger structural restrictions than an arbitrary class, and these restrictions exist partly to

support more predictable interoperability and layout reasoning. However, even standard-layout does not mean no padding or fully portable byte-for-byte layout across all compilers and ABIs. It only means the type satisfies certain structural properties that the language recognizes. Another important rule is that declaration order of non-static data members matters. The compiler does not arbitrarily permute the order of members inside a class definition to make layout smaller. The programmer controls member order in source code, and that order strongly influences object layout.

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

struct ExampleA {
    char a;
    int b;
    char c;
};

struct ExampleB {
    int b;
    char a;
    char c;
};

int main() {
    std::cout << "sizeof(ExampleA) = " << sizeof(ExampleA) << '\n';
    std::cout << "sizeof(ExampleB) = " << sizeof(ExampleB) << '\n';
}
```

In many environments, these two structures differ in size because the order of fields changes the amount of padding that must be inserted. This is one of the first concrete lessons of

memory layout: source-level ordering decisions produce real physical effects.
For Windows users, a straightforward build command with LLVM tools is:

```
clang-cl /std:c++20 /EHsc /W4 /O2 layout_rules.cpp
```

If you want to inspect layout-related record details while experimenting, it is often useful to compile in a console project and compare `sizeof`, `alignof`, and `offsetof` results across related class definitions.

1.2.2 Alignment & padding

Alignment is the rule that certain object types must begin at addresses that are multiples of some power-of-two boundary or implementation-defined requirement. Alignment exists because many processors access naturally aligned data more efficiently, and some architectures impose stricter correctness requirements for misaligned access.
Every object type therefore has an alignment requirement. The alignment of a class is generally influenced by the strictest alignment requirement among its members. To satisfy those requirements, the compiler may insert **padding** bytes between members and after the final member of the object.
Consider this simple example:

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

struct MisalignedLooking {
    char tag;
    double value;
    int count;
};
```

```
int main() {
    std::cout << "sizeof(MisalignedLooking) = "
              << sizeof(MisalignedLooking) << '\n';

    std::cout << "alignof(MisalignedLooking) = "
              << alignof(MisalignedLooking) << '\n';

    std::cout << "offsetof(tag) = "
              << offsetof(MisalignedLooking, tag) << '\n';

    std::cout << "offsetof(value) = "
              << offsetof(MisalignedLooking, value) << '\n';

    std::cout << "offsetof(count) = "
              << offsetof(MisalignedLooking, count) << '\n';
}
```

The member `tag` uses only one byte, but if `double` requires stronger alignment, the compiler may insert several padding bytes after `tag` so that `value` starts at a suitable boundary. Then the total object size may again be rounded upward so that arrays of this type maintain proper alignment for each element.
Padding must be understood in two different ways:

- **internal padding**, inserted between members,
- **tail padding**, inserted after the last member.

Internal padding ensures the next member begins at the correct offset. Tail padding ensures that the overall object size is compatible with alignment when multiple objects are placed consecutively, such as in arrays.
A practical demonstration with arrays makes this clearer:

```cpp
#include <iostream>

struct Node {
    char state;
    int id;
};

int main() {
    Node nodes[3]{};

    std::cout << "sizeof(Node) = " << sizeof(Node) << '\n';
    std::cout << "Address of nodes[0] = " << &nodes[0] << '\n';
    std::cout << "Address of nodes[1] = " << &nodes[1] << '\n';
    std::cout << "Address of nodes[2] = " << &nodes[2] << '\n';
}
```

When the addresses are printed, the distance between adjacent elements is `sizeof(Node)`, not the arithmetic sum of its visible fields. This is the real operational meaning of tail padding in an array context.

Modern C++ also allows stronger explicit alignment when needed through `alignas`:

```cpp
#include <iostream>

struct alignas(32) CacheAlignedCounter {
    long long value;
};

int main() {
    std::cout << "sizeof(CacheAlignedCounter) = "
              << sizeof(CacheAlignedCounter) << '\n';
```

```
    std::cout << "alignof(CacheAlignedCounter) = "
              << alignof(CacheAlignedCounter) << '\n';
}
```

This can be useful in high-performance code, lock-free structures, SIMD-oriented data, or false-sharing avoidance, but it must be applied carefully. Increasing alignment may increase object size and memory pressure, so alignment is not automatically beneficial simply because it sounds more optimized.

1.2.3 Bitfields and layout traps

Bit-fields are often introduced as a way to pack small integer values into fewer bits, potentially reducing memory usage. They can be useful in narrow domains such as hardware control registers, protocol representations, compact flags, and special storage layouts. However, bit-fields come with major portability and maintainability traps. They must therefore be approached carefully.
A simple example is:

```
#include <iostream>

struct Flags {
    unsigned read  : 1;
    unsigned write : 1;
    unsigned exec  : 1;
    unsigned mode  : 3;
};

int main() {
    Flags f{};
    f.read = 1;
```

```
    f.write = 0;
    f.exec = 1;
    f.mode = 5;

    std::cout << "sizeof(Flags) = " << sizeof(Flags) << '\n';
    std::cout << "read = " << f.read << '\n';
    std::cout << "write = " << f.write << '\n';
    std::cout << "exec = " << f.exec << '\n';
    std::cout << "mode = " << f.mode << '\n';
}
```

At first sight, the programmer may expect this structure to occupy exactly one byte, because the visible widths add up to only 6 bits. But the actual size is implementation-dependent and commonly larger than the raw bit count suggests. This is because bit-fields are allocated within storage units according to implementation and ABI rules.
Bit-fields introduce several traps:

- allocation unit sizes are implementation-dependent,
- ordering of bits inside storage units is implementation-dependent,
- crossing of byte boundaries may differ by platform,
- signedness behavior of plain bit-fields can be subtle,
- addresses of bit-fields cannot be taken as normal objects,
- mixing bit-fields with normal fields may produce surprising padding and alignment effects.

A classic trap is assuming a stable portable binary layout:

```
struct PacketHeader {
    unsigned version : 4;
    unsigned type    : 4;
    unsigned length  : 8;
};
```

This may look like a perfect network or file format definition, but it is not a reliable cross-platform binary specification by itself. The compiler may pack and order the fields according to target ABI rules that differ from the external format you intend to represent. In portable protocol code, explicit byte-level encoding is usually safer than relying on bit-fields for binary interchange.

Another trap involves zero-width unnamed bit-fields, which can force the next bit-field to begin in a new allocation unit:

```
#include <iostream>

struct SplitFlags {
    unsigned a : 3;
    unsigned   : 0;
    unsigned b : 3;
};

int main() {
    std::cout << "sizeof(SplitFlags) = " << sizeof(SplitFlags) << '\n';
}
```

This kind of construct can be useful in rare low-level situations, but it also makes layout more delicate and less obvious to readers.

If a programmer truly needs compact portable bit manipulation, ordinary integer fields with explicit masks and shifts are often clearer and more controllable:

```
#include <cstdint>
#include <iostream>

class PackedFlags {
private:
    std::uint8_t bits_ = 0;

public:
    void set_read(bool v) {
        if (v) bits_ |= 0x01;
        else   bits_ &= static_cast<std::uint8_t>(~0x01);
    }

    void set_write(bool v) {
        if (v) bits_ |= 0x02;
        else   bits_ &= static_cast<std::uint8_t>(~0x02);
    }

    bool read() const noexcept {
        return (bits_ & 0x01) != 0;
    }

    bool write() const noexcept {
        return (bits_ & 0x02) != 0;
    }
};

int main() {
    PackedFlags f;
```

```
    f.set_read(true);
    f.set_write(false);

    std::cout << "read = " << f.read() << '\n';
    std::cout << "write = " << f.write() << '\n';
}
```

This style is often preferable in portable library and application code because the representation is explicit and the semantics are easier to audit.

1.2.4 Field reordering for performance

Because the compiler preserves member declaration order, the programmer can often improve object size and memory locality by reordering fields. This is one of the simplest and most valuable layout-aware optimization techniques in C++.
Consider the following two class definitions:

```
#include <iostream>

struct PoorOrder {
    char  state;
    double total;
    int   count;
    char  enabled;
};

struct BetterOrder {
    double total;
    int    count;
    char   state;
```

```
    char   enabled;
};

int main() {
    std::cout << "sizeof(PoorOrder) = " << sizeof(PoorOrder) << '\n';
    std::cout << "sizeof(BetterOrder) = " << sizeof(BetterOrder) << '\n';
}
```

In many implementations, `BetterOrder` is smaller because fields with stronger alignment are placed first, reducing internal fragmentation. This does not always produce dramatic savings for one object, but it can matter enormously when millions of objects are stored in containers, caches, job systems, or memory pools.

The practical reasoning behind field reordering is simple:

- place strongly aligned fields earlier,
- group fields of similar size together where it reduces padding,
- consider hot fields together for cache locality,
- avoid thoughtless interleaving of tiny and large members.

For a more realistic example, compare two entity records:

```
#include <iostream>
#include <string>

struct EntityA {
    bool active;
    double x;
    int id;
    double y;
```

```
    bool visible;
};

struct EntityB {
    double x;
    double y;
    int id;
    bool active;
    bool visible;
};

int main() {
    std::cout << "sizeof(EntityA) = " << sizeof(EntityA) << '\n';
    std::cout << "sizeof(EntityB) = " << sizeof(EntityB) << '\n';
}
```

The reordered version often reduces wasted space. In performance-sensitive systems, this may improve not only memory consumption but also effective cache density, allowing more objects per cache line.

However, field reordering is not merely about minimizing `sizeof`. It should be guided by actual access patterns. Sometimes a slightly larger object with better clustering of frequently accessed fields can outperform a smaller one. For example, if two fields are read together in a hot loop, keeping them adjacent may be more valuable than minimizing total size by a few bytes.

A practical workflow for field layout tuning on Windows is:

- first measure `sizeof` and `alignof`,
- inspect `offsetof` for important members,
- compare alternative field orders,

- benchmark real workloads rather than trusting appearance alone.

An example utility for inspecting offsets is:

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

struct Profiled {
    double hot1;
    double hot2;
    int cold1;
    char flag;
};

int main() {
    std::cout << "sizeof(Profiled) = " << sizeof(Profiled) << '\n';
    std::cout << "alignof(Profiled) = " << alignof(Profiled) << '\n';
    std::cout << "offsetof(hot1) = " << offsetof(Profiled, hot1) << '\n';
    std::cout << "offsetof(hot2) = " << offsetof(Profiled, hot2) << '\n';
    std::cout << "offsetof(cold1) = " << offsetof(Profiled, cold1) << '\n';
    std::cout << "offsetof(flag) = " << offsetof(Profiled, flag) << '\n';
}
```

This type of inspection is especially valuable before committing to serialization formats, memory pools, custom allocators, or FFI boundaries.

1.2.5 Windows-oriented notes for layout-sensitive code

When experimenting with object layout on Windows, the recommended development approach is to use a modern Visual Studio environment together with LLVM tools if available. Compile

from the Developer Command Prompt so that the Windows SDK and include paths are set correctly.
Typical commands are:

```
clang-cl /std:c++20 /EHsc /W4 /O2 object_layout.cpp
```

If you are testing ABI-sensitive behavior or comparing structure layout between Microsoft-style and non-Microsoft-style conventions in special cases, be aware that compilers may provide compatibility modes and attributes that affect record layout. Such features should be used only when interoperability requirements truly demand them, because they make code more ABI-sensitive and less transparent.
For day-to-day application and library development on Windows, the safest strategy is:

- write ordinary standard-conforming layouts first,
- inspect size, alignment, and offsets explicitly,
- avoid assuming hidden packing behavior,
- avoid bit-fields for portable external binary formats,
- reorder fields only when measurement or design analysis justifies it.

1.2.6 Closing perspective

Object memory layout is where high-level class design meets machine reality. A class definition is not just a semantic abstraction; it is also a concrete memory layout decision. Once this is understood, many advanced C++ topics become clearer. Padding stops being mysterious. Alignment stops looking like a compiler quirk. Field ordering becomes an engineering tool. Bit-fields are recognized as specialized mechanisms with serious portability caveats.
A mature C++ programmer therefore learns to reason about both meaning and representation at the same time. The strongest designs are those in which the object model is clean, the layout is

intentional, the memory cost is measured, and the code remains correct across optimization levels, compilers, and long-term maintenance.

1.3 Object Lifetime & Construction Model

Object lifetime is one of the deepest and most important parts of the C++ object model. A modern C++ programmer must understand not only how to declare and use objects, but also when an object formally begins to exist, when it becomes safe to access, how its subobjects are initialized, how destruction proceeds, and how temporaries behave when bound to references. These rules are central to safe class design, RAII, exception safety, and correct low-level reasoning.

An object is not fully understood merely by naming its type or by seeing where memory has been reserved. In C++, storage and lifetime are related but not identical ideas. Memory may be available before an object's lifetime begins, and memory may continue to exist after the object itself has ceased to exist. The language therefore distinguishes carefully between raw storage, initialization, active object lifetime, and destruction.

For modern object-oriented design, this distinction is decisive. Constructors are not merely convenient functions. They establish invariants. Destructors are not merely cleanup helpers. They mark the ordered shutdown of an object's substructure and resource ownership. Temporary objects are not merely compiler artifacts. They participate in precise rules that influence correctness, performance, and API design.

This section presents the construction model through four essential topics:

- pre-construction memory state,
- initialization sequence,
- order of destruction,
- temporaries and lifetime extension.

These topics form one continuous model. Storage comes first, then lifetime begins through initialization, then the object exists and may be used according to its type and invariants, and finally destruction ends its lifetime in a strictly defined order. Temporary objects fit into this model with special rules, especially when references are involved.

1.3.1 Pre-construction memory state

Before an object is fully constructed, there may already be memory reserved for it, but that does not automatically mean the object is ready to use as a complete object of its type. This distinction is especially important in low-level code, custom allocation systems, manual storage management, placement new scenarios, and advanced library work.

A common beginner misunderstanding is to assume that once bytes exist at some address, an object of the intended type is already fully present there. In reality, modern C++ treats object lifetime formally. Storage may exist without a live object of the final intended type yet being present. For ordinary automatic objects, this distinction is often hidden because declaration and initialization occur together. But in more advanced code, the difference becomes visible and essential.

Consider a simple automatic object:

```
#include <iostream>
#include <string>

class User {
private:
    std::string name_;

public:
    explicit User(std::string name) : name_(std::move(name)) {
        std::cout << "Constructing User: " << name_ << '\n';
```

```
    }

    ~User() {
        std::cout << "Destroying User: " << name_ << '\n';
    }

    const std::string& name() const noexcept {
        return name_;
    }
};

int main() {
    User u{"Ayman"};
    std::cout << u.name() << '\n';
}
```

For such an object, the language gives a clean model: storage is associated with the variable, initialization occurs, and once initialization is complete, the object's lifetime has begun. The important principle is that correctness begins only after construction has established the object. The distinction becomes more visible when raw storage is handled directly:

```
#include <new>
#include <iostream>
#include <string>

class Message {
private:
    std::string text_;

public:
```

```
    explicit Message(std::string text) : text_(std::move(text)) {
        std::cout << "Message constructed\n";
    }

    ~Message() {
        std::cout << "Message destroyed\n";
    }

    const std::string& text() const noexcept {
        return text_;
    }
};

int main() {
    alignas(Message) unsigned char buffer[sizeof(Message)];

    Message* ptr = new (buffer) Message("Hello from placement new");

    std::cout << ptr->text() << '\n';

    ptr->~Message();
}
```

This example is educational because it separates storage from lifetime. The byte array `buffer` exists before any `Message` object exists there. Only when placement new is executed does the `Message` lifetime begin in that storage. Likewise, calling the destructor ends the lifetime of the `Message`, even though the byte array storage still exists afterward.

This distinction is one of the foundations of serious RAII design. A resource-owning class is correct only after construction has established its invariant, and a destroyed object must no

longer be treated as a live instance of its former type merely because the bytes still exist in memory.
For Windows compilation with Clang's MSVC-compatible driver:

```
clang-cl /std:c++20 /EHsc /W4 /O2 pre_construction_state.cpp
```

1.3.2 Initialization sequence

The initialization sequence of a C++ object is not arbitrary. For class types, the language defines a strict order in which subobjects are initialized. Understanding this order is essential because many bugs arise when programmers assume that the order written in the constructor initializer list is the true initialization order. It is not.
For a complete most-derived object, initialization proceeds conceptually through the following structural layers:

- virtual base classes first, if any,
- then direct base classes,
- then non-static data members in declaration order,
- then the constructor body.

The crucial phrase is **declaration order**. Data members are initialized in the order they are declared in the class definition, not in the textual order in the constructor initializer list. This rule exists so that destruction can reliably proceed in reverse order.
A clear example is:

```
#include <iostream>
#include <string>

class Trace {
```

```
private:
    std::string name_;

public:
    explicit Trace(std::string name) : name_(std::move(name)) {
        std::cout << "Constructing " << name_ << '\n';
    }

    ~Trace() {
        std::cout << "Destroying " << name_ << '\n';
    }
};

class Example {
private:
    Trace first_;
    Trace second_;

public:
    Example() : second_("second_"), first_("first_") {
        std::cout << "Example constructor body\n";
    }
};

int main() {
    Example e;
}
```

Even though second_ appears first in the initializer list, first_ is constructed first because it is declared first in the class. Then second_ is constructed. Only after all base and member

subobjects are initialized does the constructor body execute.

This rule has major design implications. The safest engineering style is to write the initializer list in the same order as member declaration order. That reduces confusion and avoids warnings from good compilers.

A more realistic example involving member dependency is:

```
#include <iostream>
#include <string>

class Config {
private:
    std::string path_;

public:
    explicit Config(std::string path) : path_(std::move(path)) {}

    const std::string& path() const noexcept {
        return path_;
    }
};

class Engine {
private:
    Config config_;
    std::string summary_;

public:
    explicit Engine(std::string path)
        : config_(std::move(path)),
          summary_("Loaded from: " + config_.path()) {
```

```
        std::cout << summary_ << '\n';
    }
};
```

This design is valid because `config_` is declared before `summary_`. Therefore `config_` is initialized first, and its value may safely be used when initializing `summary_`. If the declaration order were reversed, the program would become incorrect even if the initializer list visually suggested the intended order.

Base-class initialization follows equally strict rules:

```
#include <iostream>

class Base {
public:
    Base() {
        std::cout << "Base constructed\n";
    }

    ~Base() {
        std::cout << "Base destroyed\n";
    }
};

class Derived : public Base {
private:
    int value_;

public:
    Derived() : Base(), value_(42) {
        std::cout << "Derived constructed\n";
```

```
    }

    ~Derived() {
        std::cout << "Derived destroyed\n";
    }
};

int main() {
    Derived d;
}
```

The `Base` subobject is initialized before the `Derived` part. This is one of the structural laws of the object model: the most-derived object is built on top of fully initialized base subobjects and fully initialized member subobjects.
For Windows development:

```
clang-cl /std:c++20 /EHsc /W4 /O2 initialization_sequence.cpp
```

1.3.3 Order of destruction

Destruction in C++ is the conceptual mirror image of construction. Once a class object's destructor begins, its body runs first, and then its members and base-class subobjects are destroyed automatically in reverse order of completed construction. This reverse ordering is one of the reasons the language mandates deterministic initialization order.
The reverse-destruction rule is not merely a formal curiosity. It is essential for safety. If one member depends on another during normal operation, then reversing construction order ensures dependencies unwind correctly.
Consider again a traced example:

```
#include <iostream>
```

```
#include <string>

class Trace {
private:
    std::string name_;

public:
    explicit Trace(std::string name) : name_(std::move(name)) {
        std::cout << "Constructing " << name_ << '\n';
    }

    ~Trace() {
        std::cout << "Destroying " << name_ << '\n';
    }
};

class Holder {
private:
    Trace alpha_;
    Trace beta_;

public:
    Holder() : alpha_("alpha_"), beta_("beta_") {
        std::cout << "Holder ready\n";
    }

    ~Holder() {
        std::cout << "Holder destructor body\n";
    }
```

```
};

int main() {
    Holder h;
}
```

The destruction behavior is:

- first the body of `Holder:: Holder()` runs,
- then `beta_` is destroyed,
- then `alpha_` is destroyed.

That is reverse member order relative to construction. Base classes are likewise destroyed after members, again in reverse order of construction completion.
This rule is one of the foundations of RAII. A member that owns a file, a lock, a socket, a transaction guard, or a buffer will be released automatically as part of object destruction. The programmer does not need to manually remember to destroy the members; the language itself enforces the shutdown sequence.
A resource-oriented example shows the architectural value clearly:

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

class LogFile {
private:
    std::ofstream out_;

public:
```

```
    explicit LogFile(const std::string& path) : out_(path) {
        std::cout << "LogFile opened\n";
    }

    ~LogFile() {
        std::cout << "LogFile closing\n";
    }

    void write(const std::string& text) {
        out_ << text << '\n';
    }
};

class Application {
private:
    LogFile log_;

public:
    Application() : log_("app.log") {
        log_.write("Application started");
    }

    ~Application() {
        log_.write("Application stopping");
    }
};

int main() {
    Application app;
```

```
}
```

This is precisely why destruction order matters. The `Application` destructor body runs while `log_` is still alive, so it can safely log a shutdown message. After the destructor body ends, `log_` is destroyed automatically. If the order were not defined this way, many RAII patterns would become unsafe or impossible.

Arrays follow the same reverse principle at the element level:

```
#include <iostream>

class Item {
private:
    int id_;

public:
    explicit Item(int id) : id_(id) {
        std::cout << "Constructing Item " << id_ << '\n';
    }

    ~Item() {
        std::cout << "Destroying Item " << id_ << '\n';
    }
};

int main() {
    Item items[] = { Item(1), Item(2), Item(3) };
}
```

The elements are destroyed in reverse order relative to completion of construction, which matches the broader destruction model of the language.

For Windows builds:

```
clang-cl /std:c++20 /EHsc /W4 /O2 destruction_order.cpp
```

1.3.4 Temporaries & lifetime extension

Temporary objects are central to modern C++. They arise in expression evaluation, return-value flow, conversions, aggregate expressions, and many other ordinary constructs. A temporary is still a real object with lifetime, but its lifetime is usually short unless the language extends it through a specific rule.

One of the most important rules is **lifetime extension by reference binding**. When a temporary is bound to a const reference, or in certain modern cases an rvalue reference, its lifetime can be extended to match the lifetime of that reference object. This rule is critical because without it, many natural and efficient APIs would be dangerous.

A classic example is:

```
#include <iostream>
#include <string>

int main() {
    const std::string& ref = std::string("extended temporary");
    std::cout << ref << '\n';
}
```

The temporary `std::string` would ordinarily die at the end of the full-expression, but because it is directly bound to a reference in this form, its lifetime is extended to match `ref`.

The same principle appears with user-defined types:

```
#include <iostream>

class Box {
private:
```

```
    int value_;

public:
    explicit Box(int value) : value_(value) {
        std::cout << "Box constructed\n";
    }

    ~Box() {
        std::cout << "Box destroyed\n";
    }

    int value() const noexcept {
        return value_;
    }
};

int main() {
    const Box& b = Box(99);
    std::cout << b.value() << '\n';
}
```

Here the temporary `Box(99)` remains alive as long as `b` remains alive.
An important modern point is that lifetime extension may apply recursively for reference members initialized from temporaries. This matters in aggregate-style constructions and reference-holding wrapper objects.

```
#include <iostream>

struct Holder {
    const int& ref;
```

```
};

int main() {
    const Holder& h = Holder{123};
    std::cout << h.ref << '\n';
}
```

In this pattern, both the temporary `Holder` and the temporary `int` bound through its reference member are kept alive as required by the language rule for this case.
However, lifetime extension is not universal. It has exceptions, and a professional programmer must not assume that every reference-related expression safely prolongs a temporary. For example, returning a reference to a temporary from a function remains wrong, and binding through intermediate expressions or through certain member-initialization contexts can fail to produce the desired lifetime.
A dangerous anti-pattern is:

```
#include <string>

const std::string& bad() {
    return std::string("temporary");
}
```

This returns a reference to a temporary that will not remain alive for the caller. The result is a dangling reference.
The safe style is to return by value:

```
#include <string>

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

Modern C++ makes returning by value efficient, and this style avoids dangling reference errors. Temporary lifetime also matters when objects are stored inside references or views. A good engineering rule is:

- bind references only when lifetime is clearly guaranteed,
- avoid storing references to temporary-producing expressions unless the rule is explicit and well understood,
- prefer value ownership when uncertainty exists,
- treat every borrowed view as a lifetime contract, not as a mere syntax convenience.

For Windows compilation:

```
clang-cl /std:c++20 /EHsc /W4 /O2 temporary_lifetime.cpp
```

1.3.5 Practical Windows notes

All examples in this section are intended to work naturally in a Windows environment using the Visual Studio Developer Command Prompt and a current Clang installation that provides `clang-cl`. A typical workflow is:

- open the Visual Studio Developer Command Prompt,
- save each example in its own `.cpp` file,
- compile with `clang-cl /std:c++20 /EHsc /W4 /O2 file.cpp`,
- run the produced executable and observe the construction and destruction trace.

For educational experiments involving lifetime order, it is often helpful to:

- print messages from constructors and destructors,

- compare member declaration order with initializer-list order,
- test scope exit behavior,
- experiment with temporary binding and reference-return mistakes.

This kind of tracing is extremely effective for building a correct mental model of object lifetime.

1.3.6 Closing perspective

The construction model of a C++ object is not an implementation accident. It is one of the core semantic systems of the language. Storage may exist before the object's active lifetime. Initialization gives structure and validity to that storage. Construction proceeds in a strict order through bases and members. Destruction unwinds that structure in reverse order. Temporary objects participate in the same model, but with special lifetime rules that can either preserve safety or create subtle bugs depending on how references are used.

This is why object lifetime stands at the center of modern C++ architecture. RAII depends on it. Exception safety depends on it. Ownership reasoning depends on it. Good class design depends on it. Once the programmer truly understands pre-construction state, initialization order, destruction order, and temporary lifetime extension, many of the most important design rules of modern C++ become not arbitrary rules to memorize, but logical consequences of a coherent object model.

Classes & Member Initialization

2.1 Member Initializer Lists (Deep)

Member initializer lists are one of the most important structural features of class construction in modern C++. They are not a cosmetic alternative to assignment inside the constructor body. They are the primary mechanism by which base classes and non-static data members are initialized before the constructor body begins execution. For this reason, a serious understanding of C++ object construction is impossible without a deep understanding of member initializer lists.

A constructor body executes only after the initialization phase of the object has already begun and after all direct base classes and non-static data members have been initialized according to the language rules. This means that when a programmer writes assignments inside the constructor body, those assignments usually do not replace initialization. Instead, they often perform a second step after an earlier default construction or earlier initialization has already happened. In many cases this is less efficient, less expressive, and more error-prone than constructing the members correctly in the first place.

Modern C++ guidance strongly favors direct initialization over post-construction assignment. It also emphasizes that member declaration order, not initializer-list text order, determines the actual order of initialization. These rules are essential for writing correct constructors, especially in classes with resource-owning members, dependent members, references,

const-qualified members, and non-default-constructible subobjects.
This section studies member initializer lists deeply through three major themes:

- best practices,
- common pitfalls,
- dependency initialization order.

These themes are tightly connected. Good practices exist because the construction model is strict. Most pitfalls come from misunderstanding that strict model. Dependency ordering matters because objects are built in a fixed order, not according to how the programmer visually arranges the initializer list.

2.1.1 Best practices

The best modern use of member initializer lists begins with one foundational rule: initialize every base and every non-static data member deliberately and clearly. This does not always mean that every member must appear explicitly in every constructors initializer list, because modern C++ also provides default member initializers. However, every member should have a clear initialization strategy.
A strong basic example is:

```
#include <string>
#include <utility>

class User {
private:
    std::string name_;
    int age_;
```

```
public:
    User(std::string name, int age)
        : name_(std::move(name)), age_(age) {
    }
};
```

This constructor is clear and direct. The string member is constructed directly from the provided argument, and the integer member is initialized immediately. No extra work is performed in the constructor body.

One of the strongest best practices is to prefer initialization to assignment inside constructors. Compare the following two forms:

```
#include <string>

class Good {
private:
    std::string text_;

public:
    explicit Good(const char* p)
        : text_(p) {
    }
};

class Bad {
private:
    std::string text_;

public:
    explicit Bad(const char* p) {
```

```
        text_ = p;
    }
};
```

The first version directly constructs `text_`. The second version first default-constructs `text_`, then assigns to it inside the body. The first is typically clearer and more efficient, especially for class-type members.

Another important best practice is to use **default member initializers** for values that are intended to be the same across constructors unless explicitly overridden. This reduces repetition and prevents accidental omission when additional constructors are added later.

```
#include <string>
#include <utility>

class ConnectionOptions {
private:
    int timeout_ms_ = 5000;
    bool keep_alive_ = true;
    std::string host_ = "localhost";

public:
    ConnectionOptions() = default;

    explicit ConnectionOptions(std::string host)
        : host_(std::move(host)) {
    }

    ConnectionOptions(std::string host, int timeout_ms)
        : timeout_ms_(timeout_ms), host_(std::move(host)) {
    }
```

```
};
```

This design is strong because the defaults are visible at the point of member declaration. Every constructor benefits from them automatically unless it chooses to override them explicitly. This style is especially valuable in large classes with many constructors.
Another best practice is to keep the order of the member initializer list the same as the order of member declaration. This is not required for correctness of the generated program when the initializer expressions themselves are otherwise valid, but it is strongly recommended because the actual initialization order always follows declaration order. Matching the source order to the real execution order reduces confusion and improves maintainability.

```
#include <string>
#include <utility>

class FileInfo {
private:
    std::string path_;
    std::string extension_;
    int size_;

public:
    FileInfo(std::string path, std::string extension, int size)
        : path_(std::move(path)),
          extension_(std::move(extension)),
          size_(size) {
    }
};
```

This style is easier to audit than a constructor whose initializer list is written in a different order from the member declarations.

A further best practice is to use the initializer list for members that **must** be initialized there:

- reference members,
- const-qualified members,
- members of types without usable default constructors,
- base classes requiring arguments.

For example:

```
#include <string>

class Config {
private:
    const int port_;
    std::string& output_;

public:
    Config(int port, std::string& output)
        : port_(port), output_(output) {
    }
};
```

Here, both members require initialization in the initializer list. The constructor body cannot later fix this.
Delegating constructors are also a strong modern practice when multiple constructors share common initialization logic:

```
#include <string>
#include <utility>
```

```
class Logger {
private:
    std::string file_name_;
    int level_;

public:
    Logger()
        : Logger("default.log", 1) {
    }

    explicit Logger(std::string file_name)
        : Logger(std::move(file_name), 1) {
    }

    Logger(std::string file_name, int level)
        : file_name_(std::move(file_name)), level_(level) {
    }
};
```

This prevents duplication and centralizes the true initialization logic in one place.
For Windows developers using LLVM tools, a good compilation command is:

```
clang-cl /std:c++20 /EHsc /W4 /O2 member_initializer_best_practices.cpp
```

To ask Clang for stronger help with initializer-order issues, it is useful to enable reorder diagnostics explicitly:

```
clang-cl /std:c++20 /EHsc /W4 /clang:-Wreorder /clang:-Wreorder-ctor
↪  member_initializer_best_practices.cpp
```

If clang-tidy is available, one practical check for constructor-body assignments that should become member initializers is:

```
clang-tidy member_initializer_best_practices.cpp
↪  --checks=-*,cppcoreguidelines-prefer-member-initializer --
```

2.1.2 Common pitfalls

Member initializer lists are conceptually simple, but the mistakes programmers make with them are often subtle and serious. The most common pitfall is assuming that the order written in the initializer list controls the order of initialization. It does not. The order of initialization is determined by:

- virtual bases first, where applicable,
- direct bases next,
- non-static data members in the order of declaration,
- then the constructor body.

This pitfall appears in code like this:

```
class WrongOrder {
private:
    int first_;
    int second_;

public:
    explicit WrongOrder(int x)
        : second_(x), first_(second_ + 1) {
    }
};
```

At first glance, a reader might think `second_` is initialized first because it appears first in the initializer list. But `first_` is declared first, so it is initialized first. The expression `second_ + 1` is therefore problematic because it tries to depend on a member that is not yet initialized. This is exactly the kind of bug that reorder diagnostics and declaration-order discipline are meant to prevent.

A second common pitfall is using assignment in the constructor body instead of initialization. For trivial scalar types this may appear harmless, but for class types it often creates extra work and can even be impossible for certain members.

```
#include <memory>

class BadOwner {
private:
    std::unique_ptr<int> value_;

public:
    BadOwner() {
        value_ = std::make_unique<int>(42);
    }
};
```

This compiles, but it still default-constructs `value_` first and only then assigns a new owned object in the body. The stronger style is:

```
#include <memory>

class GoodOwner {
private:
    std::unique_ptr<int> value_;
```

```
public:
    GoodOwner()
        : value_(std::make_unique<int>(42)) {
    }
};
```

A third pitfall is forgetting that default member initializers are overridden by explicit constructor member initializers for the same member. This is usually desirable, but the programmer must understand the precedence clearly.

```
class Counter {
private:
    int start_ = 100;

public:
    Counter() = default;

    explicit Counter(int start)
        : start_(start) {
    }
};
```

The default member initializer is used in the default constructor, but the explicit initializer in the second constructor overrides it. This is the correct and intended model, but misunderstanding it can lead to wrong assumptions about when defaults apply.
Another pitfall is forgetting to initialize every member consistently when no default member initializer exists. This often happens when new members are added later to an existing class:

```
#include <string>
```

```
class Risky {
private:
    int id_;
    std::string name_;
    int flags_;

public:
    Risky()
        : id_(0), name_("unknown") {
    }

    Risky(int id, std::string name)
        : id_(id), name_(std::move(name)) {
    }
};
```

Here `flags_` is not initialized by either constructor. For some types and contexts, this can leave the member indeterminate. This is one reason modern guidelines favor default member initializers when a stable default exists.

A further pitfall concerns exceptions and partially initialized objects. If a constructor throws during initialization of a later member, already constructed subobjects are destroyed automatically, but the complete object never becomes fully constructed. This behavior is correct and powerful, but it means member initialization must be written with full awareness of resource ownership and dependency.

```
#include <iostream>
#include <stdexcept>
#include <string>

class Trace {
```

```
private:
    std::string name_;

public:
    explicit Trace(std::string name) : name_(std::move(name)) {
        std::cout << "Constructing " << name_ << '\n';
    }

    ~Trace() {
        std::cout << "Destroying " << name_ << '\n';
    }
};

class Demo {
private:
    Trace a_;
    Trace b_;

public:
    Demo()
        : a_("a_"), b_("b_") {
        throw std::runtime_error("construction failure");
    }
};

int main() {
    try {
        Demo d;
    } catch (const std::exception&) {
```

```
    }
}
```

In this example, both members are constructed before the constructor body throws. Their destructors run during stack unwinding even though the complete `Demo` object was never successfully established. This is part of the deep construction model that gives RAII its strength.

One more pitfall is trying to perform virtual initialization through constructors in a class hierarchy. During construction, virtual dispatch does not behave as though the most-derived object were already fully available in the ordinary runtime sense. If initialization truly depends on derived behavior, a factory-based or post-construction design is usually safer.

2.1.3 Dependency initialization order

Dependency initialization order is the deepest practical issue in member initializer lists. It arises whenever one base class or member logically depends on another already being initialized. The language model is strict: dependencies must follow the real construction order, not a visually desired order.

The essential rule is this: if member `B` depends on member `A`, then `A` must be declared before `B`. Writing `A` first in the initializer list is not enough if declaration order disagrees.

A safe example is:

```
#include <string>
#include <utility>

class Engine {
private:
    std::string file_name_;
    std::string description_;
```

```
public:
    explicit Engine(std::string file_name)
        : file_name_(std::move(file_name)),
          description_("Engine loaded from: " + file_name_) {
    }
};
```

This is correct because `file_name_` is declared before `description_`. Therefore `file_name_` is initialized first, and its value is available when `description_` is initialized. Now consider the broken version:

```
#include <string>
#include <utility>

class BrokenEngine {
private:
    std::string description_;
    std::string file_name_;

public:
    explicit BrokenEngine(std::string file_name)
        : file_name_(std::move(file_name)),
          description_("Engine loaded from: " + file_name_) {
    }
};
```

Although the initializer list appears to initialize `file_name_` before `description_`, the actual member construction order follows declaration order. Because `description_` is declared first, its initializer runs first, and the expression that depends on `file_name_` is not semantically safe. The same principle applies to base classes:

```
#include <iostream>
#include <string>

class BaseConfig {
protected:
    std::string path_;

public:
    explicit BaseConfig(std::string path)
        : path_(std::move(path)) {
        std::cout << "BaseConfig ready\n";
    }
};

class DerivedSystem : public BaseConfig {
private:
    std::string summary_;

public:
    explicit DerivedSystem(std::string path)
        : BaseConfig(std::move(path)),
          summary_("System path: " + path_) {
    }
};
```

This is valid because the direct base `BaseConfig` is initialized before the derived class members. Therefore the derived member `summary_` can depend on base state that has already been established.

Dependency order is especially important with RAII wrappers and owned resources. Consider a class where one member represents a file and another member depends on that file path or

state:

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

class FileSession {
private:
    std::string path_;
    std::ofstream out_;

public:
    explicit FileSession(std::string path)
        : path_(std::move(path)),
          out_(path_) {
    }
};
```

This is structurally sound because `path_` is declared before `out_`. The file stream can safely use the path during its own initialization.
If the declarations are reversed, the code becomes logically broken even if the initializer list appears in the intended order:

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

class BrokenFileSession {
private:
    std::ofstream out_;
```

```
    std::string path_;

public:
    explicit BrokenFileSession(std::string path)
        : path_(std::move(path)),
          out_(path_) {
    }
};
```

Because `out_` is declared first, it initializes first. Its initializer expression tries to use `path_` before `path_` has been initialized. This is exactly the kind of dependency bug that can survive code review if the reviewer trusts the visual order of the initializer list instead of the declaration order.

The best engineering rule is therefore:

- declare members in dependency order,
- write initializer lists in the same order,
- let earlier members provide stable inputs for later ones,
- never rely on textual initializer-list order to override the construction model.

For larger designs, this often leads to better class structure. Members that should not depend on each other become easier to isolate. Members that do depend on each other become easier to place and reason about. In well-designed classes, the declaration order itself often documents the construction dependencies.

2.1.4 Windows-oriented guidance

All examples in this section are suitable for a Windows development environment using Visual Studio Developer Command Prompt and LLVM tools. A practical workflow is:

- save each example in a separate `.cpp` file,
- compile with `clang-cl`,
- enable warnings aggressively,
- enable reorder diagnostics during constructor-development work,
- run clang-tidy when available for member-initializer guidance.

Typical build commands are:

```
clang-cl /std:c++20 /EHsc /W4 /O2 member_initializer_lists_deep.cpp
```

For stronger constructor-order checking:

```
clang-cl /std:c++20 /EHsc /W4 /clang:-Wreorder /clang:-Wreorder-ctor
↪  member_initializer_lists_deep.cpp
```

For modernizing constructor-body assignments into proper member initializers:

```
clang-tidy member_initializer_lists_deep.cpp
↪  --checks=-*,cppcoreguidelines-prefer-member-initializer --
```

This is especially useful in older Windows codebases where constructors historically relied on assignment in the constructor body.

2.1.5 Closing perspective

Member initializer lists are one of the places where the true nature of C++ object construction becomes visible. They show that initialization is not an optional decorative syntax around construction. It is the construction model itself. Base classes and members come into existence before the constructor body runs, and they do so according to strict language rules.

That is why best practices around member initializer lists are so important. Direct initialization is usually clearer and more efficient than assignment. Default member initializers reduce repetition and omission bugs. Declaration-order discipline prevents dependency mistakes. Reorder diagnostics and modern tooling make these rules easier to enforce, but the real goal is deeper than warning suppression: it is to design classes whose construction logic is structurally correct, explicit, and maintainable.

A programmer who understands member initializer lists deeply does not merely write prettier constructors. Such a programmer writes classes whose invariants are established correctly from the first moment of object lifetime, whose dependencies are visible in source order, and whose construction model supports the larger goals of modern C++: correctness, efficiency, and robust resource management.

2.2 Special Member Functions

Special member functions are among the most important mechanisms in the C++ object model because they define how objects are created, copied, moved, assigned, and destroyed. They sit at the exact point where class design meets ownership, lifetime, exception safety, value semantics, and resource management. For this reason, no serious study of modern C++ classes is complete without a deep understanding of these functions and the rules that govern them.

The language provides a small set of operations that receive special treatment from the compiler. These operations may be implicitly declared, implicitly defined, defaulted by the programmer, or defined as deleted. They are called **special member functions** because they form the default operational contract of a class. They determine whether a type is default-constructible, copyable, movable, assignable, and destructible in the ordinary class sense.

In practice, modern C++ design revolves around knowing when to:

- allow the compiler to generate the correct behavior,

- explicitly request that behavior with `= default`,
- forbid an operation with `= delete`,
- control implicit conversions with `explicit`,
- declare constructor and move operations `noexcept` when they truly cannot throw,
- apply the Rule of Zero, Rule of Three, or Rule of Five correctly.

This section explores these ideas in depth because they are directly connected to the broader goals of this volume: safe ownership, deterministic lifetime, and maintainable class architecture.

2.2.1 What counts as a special member function

The language treats the following operations as special member functions:

- the default constructor,
- the copy constructor,
- the move constructor,
- the copy assignment operator,
- the move assignment operator,
- the destructor.

These six operations define the ordinary life cycle of a class object. Not every class needs to declare them manually. In modern C++, a large amount of high-quality class design comes from **not** writing them unless ownership or invariants truly require it.
A simple example is:

```cpp
#include <string>

class Person {
private:
    std::string name_;
    int age_ = 0;

public:
    Person() = default;

    Person(std::string name, int age)
        : name_(std::move(name)), age_(age) {
    }
};
```

This class does not define copy, move, assignment, or destruction manually. That is usually correct because the data members already know how to manage themselves. The standard library types supply their own correct behavior, so the class can safely rely on the compiler-generated operations.

That idea leads directly to one of the most important modern design rules: **if the members already manage themselves correctly, do not add manual special member functions without a real reason**.

2.2.2 = default

The `= default` form tells the compiler that the programmer wants the standard generated behavior for a special member function. This is not the same as writing an empty body. A defaulted function is still treated according to the language rules for implicit generation and eligibility, whereas an empty user-written body changes the semantics of the type in important ways.

A minimal example is:

```
class Point {
private:
    int x_ = 0;
    int y_ = 0;

public:
    Point() = default;
    Point(const Point&) = default;
    Point(Point&&) = default;
    Point& operator=(const Point&) = default;
    Point& operator=(Point&&) = default;
    ~Point() = default;
};
```

This makes the intent fully explicit: the class wants the normal operations. Such code is often useful in teaching material, ABI-sensitive headers, or public library interfaces where the author wants the interface contract to be visible in the class definition.

A key modern point is that defaulting can preserve efficient and correct compiler-generated behavior while still making the design explicit. This is especially helpful in class templates, framework code, and exported APIs.

It is also important to understand that explicitly defaulting a function after its first declaration has slightly different language consequences from defaulting it on the first declaration. For ordinary class design, the main practical lesson is simple: if the natural operation is correct, prefer `= default` over writing an empty function body.

Compare these two forms:

```
class Good {
public:
```

```
    Good() = default;
};

class Misleading {
public:
    Misleading() {
    }
};
```

The empty constructor body in `Misleading` is not the same design signal as `= default`. In modern C++, `= default` says use the standard generated semantics. An empty body says this is a user-provided constructor, which can affect triviality, implicit generation, and sometimes optimization-relevant properties.

A realistic resource-owning wrapper may also default only some operations:

```
#include <memory>

class Buffer {
private:
    std::unique_ptr<int[]> data_;
    std::size_t size_ = 0;

public:
    Buffer() = default;

    explicit Buffer(std::size_t size)
        : data_(std::make_unique<int[]>(size)), size_(size) {
    }

    Buffer(Buffer&&) noexcept = default;
```

```
    Buffer& operator=(Buffer&&) noexcept = default;

    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;

    ~Buffer() = default;
};
```

This class demonstrates a strong modern pattern: move operations are defaulted, copy operations are deleted, and destruction is left to the members. This is often exactly right for unique ownership.

2.2.3 = delete

The = delete form tells the compiler and the reader that a function exists as part of overload resolution or class interface structure, but it is intentionally forbidden. Deleted functions are one of the cleanest mechanisms in modern C++ for preventing invalid operations.
A classic example is a unique-ownership type:

```
#include <memory>

class FileOwner {
private:
    std::unique_ptr<int> handle_;

public:
    FileOwner() : handle_(std::make_unique<int>(42)) {}

    FileOwner(const FileOwner&) = delete;
    FileOwner& operator=(const FileOwner&) = delete;
```

```
    FileOwner(FileOwner&&) noexcept = default;
    FileOwner& operator=(FileOwner&&) noexcept = default;

    ~FileOwner() = default;
};
```

Here, copying is forbidden because duplicating ownership would be incorrect or at least unclear. Moving is permitted because ownership transfer is meaningful.
Deleted functions are also useful for preventing undesired conversions:

```
class Token {
public:
    Token(int) = delete;
    Token(const char*) = delete;

    Token() = default;
};
```

In this case the type refuses construction from those source forms entirely.
One of the most important guideline-level rules in modern C++ is that if you define or delete one of the major ownership-related operations, you usually need to consider all of them together. This is the heart of the Rule of Five guidance discussed later in this section.
A practical Windows-oriented note is that Clang and modern analysis tools are particularly good at helping detect classes that defined a destructor or deleted some operations but forgot to address the rest. This is one reason why aggressive warnings and static analysis are so valuable in class design work.

2.2.4 `explicit`

Strictly speaking, `explicit` is not itself a special member function. It is a function specifier that is especially important for constructors and conversion operators. It belongs in this chapter because it directly controls how objects may be created from other values and therefore affects the safety and honesty of class interfaces.

An `explicit` constructor prevents unwanted implicit conversions and accidental copy-initialization. This is crucial for types that model resources, sizes, handles, IDs, policies, units, or domain-specific values.

A simple example is:

```
class Size {
private:
    int value_;

public:
    explicit Size(int value) : value_(value) {}

    int value() const noexcept {
        return value_;
    }
};
```

Now compare the effect:

```
void set_size(Size s) {
}

int main() {
    Size a(10);      // OK
```

```
    Size b{20};      // OK
    // Size c = 30;  // error because constructor is explicit

    set_size(Size{40});   // OK
    // set_size(50);       // error because implicit conversion is blocked
}
```

This is often the correct design for strong types. Without `explicit`, classes can silently accept conversions that weaken readability and may introduce bugs.
Modern C++ also supports `explicit(bool)` for conditional explicitness in templates and generic wrappers, but for core class design the main lesson remains the same: **if a one-argument constructor should not behave like an implicit conversion, mark it explicit**.
A type that models an operating-system or domain handle is a strong candidate:

```
class PortNumber {
private:
    int port_;

public:
    explicit PortNumber(int port) : port_(port) {
    }

    int value() const noexcept {
        return port_;
    }
};
```

This keeps the API honest. A raw integer is not automatically treated as a port object unless the programmer says so clearly.

2.2.5 noexcept constructors and move operations

Constructors may have exception specifications, and move operations in particular are often expected to be `noexcept` when that guarantee is correct. In modern C++, `noexcept` is not decorative. It affects program behavior, optimization opportunities, container strategy, and exception safety design.

The fundamental rule is straightforward: declare an operation `noexcept` only if it truly cannot throw.

A simple example is:

```
class Counter {
private:
    int value_ = 0;

public:
    Counter() noexcept = default;

    explicit Counter(int value) noexcept : value_(value) {
    }

    int value() const noexcept {
        return value_;
    }
};
```

For scalar-only state like this, `noexcept` is usually correct and expressive.

Move operations are especially important:

```
#include <utility>
```

```
class Handle {
private:
    int id_ = -1;

public:
    Handle() noexcept = default;

    explicit Handle(int id) noexcept : id_(id) {
    }

    Handle(const Handle&) = delete;
    Handle& operator=(const Handle&) = delete;

    Handle(Handle&& other) noexcept
        : id_(other.id_) {
        other.id_ = -1;
    }

    Handle& operator=(Handle&& other) noexcept {
        if (this != &other) {
            id_ = other.id_;
            other.id_ = -1;
        }
        return *this;
    }

    ~Handle() = default;
};
```

Here, move construction and move assignment are naturally `noexcept`. That matters because

standard containers often prefer move operations only when they are non-throwing or otherwise sufficiently safe according to the type traits and library rules.
A more realistic RAII wrapper on Windows might follow the same structure:

```
#include <windows.h>

class UniqueHandle {
private:
    HANDLE handle_ = INVALID_HANDLE_VALUE;

public:
    UniqueHandle() noexcept = default;

    explicit UniqueHandle(HANDLE handle) noexcept
        : handle_(handle) {
    }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept
        : handle_(other.handle_) {
        other.handle_ = INVALID_HANDLE_VALUE;
    }

    UniqueHandle& operator=(UniqueHandle&& other) noexcept {
        if (this != &other) {
            if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
                CloseHandle(handle_);
            }
```

```
            handle_ = other.handle_;
            other.handle_ = INVALID_HANDLE_VALUE;
        }
        return *this;
    }

    ~UniqueHandle() {
        if (handle_ != INVALID_HANDLE_VALUE && handle_ != nullptr) {
            CloseHandle(handle_);
        }
    }

    HANDLE get() const noexcept {
        return handle_;
    }
};
```

This design separates three ideas clearly:

- acquisition may be ordinary construction,
- copying is forbidden,
- moving is cheap and non-throwing.

That is exactly the sort of class architecture modern C++ encourages.
As a practical design guideline, low-level moves, swaps, destructors, and tiny scalar constructors are often good candidates for `noexcept`. But the keyword should be applied only when justified by the true behavior of the operation and its members.

2.2.6 Rule of Zero

The Rule of Zero is one of the most important modern C++ design principles. It says that if a class does not directly own a raw resource and its members already manage themselves correctly, then the class should avoid defining special member functions manually.
This is the preferred modern style because it maximizes correctness, reduces boilerplate, and allows the compiler and the member types to provide the natural operations.
A strong Rule-of-Zero example is:

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

class Project {
private:
    std::string name_;
    std::vector<std::string> files_;
    int version_ = 1;

public:
    Project() = default;

    Project(std::string name, std::vector<std::string> files, int version)
        : name_(std::move(name)),
          files_(std::move(files)),
          version_(version) {
    }
};
```

This class owns meaningful state, but it does not need custom copy, move, or destruction logic because its members already know how to manage themselves. This is ideal modern C++

design.
The Rule of Zero encourages a separation of concerns:

- classes that model business concepts, values, and aggregates should usually rely on their members,
- only dedicated ownership wrappers or invariant-heavy types should define custom special member functions.

This rule is closely aligned with the C++ Core Guidelines recommendation to avoid defining default operations when you can avoid it.

2.2.7 Rule of Three

The Rule of Three is the traditional pre-C++11 guidance: if a class needs a user-defined destructor, copy constructor, or copy assignment operator, then it probably needs all three. This rule arose because old-style classes that manually owned raw resources would often become incorrect if only one of these functions was written. For example:

```
class BadString {
private:
    char* data_;

public:
    explicit BadString(char* data) : data_(data) {
    }

    ~BadString() {
        delete[] data_;
    }
};
```

This class is dangerous because the compiler-generated copy constructor and copy assignment operator would copy the raw pointer, leading to multiple objects thinking they own the same allocation. Double deletion would follow.

The Rule of Three says that if such a class defines a destructor, it also needs custom copy behavior.

A corrected teaching example is:

```
#include <cstring>

class LegacyBuffer {
private:
    char* data_;

public:
    LegacyBuffer()
        : data_(new char[1]{'\0'}) {
    }

    explicit LegacyBuffer(const char* text) {
        std::size_t n = std::strlen(text);
        data_ = new char[n + 1];
        std::memcpy(data_, text, n + 1);
    }

    LegacyBuffer(const LegacyBuffer& other) {
        std::size_t n = std::strlen(other.data_);
        data_ = new char[n + 1];
        std::memcpy(data_, other.data_, n + 1);
    }
```

```
    LegacyBuffer& operator=(const LegacyBuffer& other) {
        if (this != &other) {
            std::size_t n = std::strlen(other.data_);
            char* new_data = new char[n + 1];
            std::memcpy(new_data, other.data_, n + 1);
            delete[] data_;
            data_ = new_data;
        }
        return *this;
    }

    ~LegacyBuffer() {
        delete[] data_;
    }
};
```

Modern code should usually prefer `std::string`, `std::vector`, or smart pointers instead of writing such classes directly. But the Rule of Three remains historically and conceptually important because it explains why careless ownership classes are dangerous.

2.2.8 Rule of Five

The Rule of Five extends the Rule of Three into modern C++. Because C++11 and later added move construction and move assignment, a class that truly manages ownership often needs to consider five operations together:

- destructor,
- copy constructor,
- copy assignment operator,

- move constructor,
- move assignment operator.

The C++ Core Guidelines express this idea in an even stronger form: if you define or delete any copy, move, or destructor function, define or delete them all consistently.
A modern ownership type may intentionally forbid copy and support move:

```
#include <memory>

class ImageBuffer {
private:
    std::unique_ptr<unsigned char[]> data_;
    std::size_t size_ = 0;

public:
    ImageBuffer() = default;

    explicit ImageBuffer(std::size_t size)
        : data_(std::make_unique<unsigned char[]>(size)),
          size_(size) {
    }

    ImageBuffer(const ImageBuffer&) = delete;
    ImageBuffer& operator=(const ImageBuffer&) = delete;

    ImageBuffer(ImageBuffer&&) noexcept = default;
    ImageBuffer& operator=(ImageBuffer&&) noexcept = default;

    ~ImageBuffer() = default;
};
```

This is an excellent modern example. The class has a clear ownership policy:

- unique ownership,
- no copy,
- efficient move,
- automatic destruction.

A different class may choose deep-copy semantics instead:

```
#include <algorithm>
#include <memory>

class CopyableBuffer {
private:
    std::unique_ptr<int[]> data_;
    std::size_t size_ = 0;

public:
    CopyableBuffer() = default;

    explicit CopyableBuffer(std::size_t size)
        : data_(std::make_unique<int[]>(size)),
          size_(size) {
    }

    CopyableBuffer(const CopyableBuffer& other)
        : data_(other.size_ ? std::make_unique<int[]>(other.size_) : nullptr),
          size_(other.size_) {
        std::copy(other.data_.get(), other.data_.get() + size_, data_.get());
```

```
    }

    CopyableBuffer& operator=(const CopyableBuffer& other) {
        if (this != &other) {
            auto new_data =
                other.size_ ? std::make_unique<int[]>(other.size_) : nullptr;
            std::copy(other.data_.get(),
                      other.data_.get() + other.size_,
                      new_data.get());
            data_ = std::move(new_data);
            size_ = other.size_;
        }
        return *this;
    }

    CopyableBuffer(CopyableBuffer&&) noexcept = default;
    CopyableBuffer& operator=(CopyableBuffer&&) noexcept = default;

    ~CopyableBuffer() = default;
};
```

This class explicitly designs all five relevant operations. That is the essence of the Rule of Five: ownership classes must make those semantics deliberate.

2.2.9 Rule of Six

The phrase **Rule of Six** is not a formal rule of the C++ language and is not a standardized guideline in the same sense as Rule of Zero or the Core Guidelines C.20 and C.21. It is an informal teaching extension used in some codebases and courses.
When people say Rule of Six, they often mean one of two things:

- the five special member functions plus a user-defined `swap`,
- the five special member functions plus one more important class-level operation required by the local design style.

In practical modern C++, if a class truly manages ownership and defines custom copy and move logic, a carefully designed `swap` is often useful. That is probably the most defensible informal interpretation of Rule of Six.
For example:

```
#include <memory>
#include <utility>

class ResourceBlock {
private:
    std::unique_ptr<int[]> data_;
    std::size_t size_ = 0;

public:
    ResourceBlock() = default;

    explicit ResourceBlock(std::size_t size)
        : data_(std::make_unique<int[]>(size)),
          size_(size) {
    }

    ResourceBlock(const ResourceBlock& other)
        : data_(other.size_ ? std::make_unique<int[]>(other.size_) : nullptr),
          size_(other.size_) {
        for (std::size_t i = 0; i < size_; ++i) {
```

```
            data_[i] = other.data_[i];
        }
    }

    ResourceBlock& operator=(const ResourceBlock& other) {
        if (this != &other) {
            ResourceBlock temp(other);
            swap(temp);
        }
        return *this;
    }

    ResourceBlock(ResourceBlock&&) noexcept = default;
    ResourceBlock& operator=(ResourceBlock&&) noexcept = default;

    ~ResourceBlock() = default;

    void swap(ResourceBlock& other) noexcept {
        using std::swap;
        swap(data_, other.data_);
        swap(size_, other.size_);
    }
};
```

This is a reasonable five plus swap design. However, it should be taught honestly: **Rule of Six is an informal style phrase, not a language rule**. The real design lesson is that ownership types often need one more explicitly designed operation beyond the classic five, and `swap` is the most common candidate.

2.2.10 Best practices for special member functions

The most important modern best practices are:

- prefer the Rule of Zero whenever possible,
- use `= default` when the natural generated behavior is the correct public design,
- use `= delete` to forbid invalid ownership or conversion operations,
- mark single-argument constructors `explicit` unless implicit conversion is truly intended,
- mark move operations and tiny scalar constructors `noexcept` only when that guarantee is true,
- if you define or delete one ownership-related special member function, review all of them together.

A particularly strong class is often very small:

```
#include <memory>

class Session {
private:
    std::unique_ptr<int> token_;

public:
    Session() : token_(std::make_unique<int>(1)) {}

    Session(const Session&) = delete;
    Session& operator=(const Session&) = delete;
```

```
    Session(Session&&) noexcept = default;
    Session& operator=(Session&&) noexcept = default;

    ~Session() = default;
};
```

This class does not attempt to outsmart the language. It expresses ownership directly and lets the compiler and the member type do most of the work.

2.2.11 Windows-oriented guidance

All examples in this section are suitable for a Windows development workflow using Visual Studio Developer Command Prompt together with LLVM tools. A typical build command is:

```
clang-cl /std:c++20 /EHsc /W4 /O2 special_member_functions.cpp
```

For deeper checking during development, it is useful to:

- enable high warning levels,
- inspect generated errors when copy or move is deleted,
- test container behavior with movable and non-movable types,
- verify that `noexcept` specifications match true behavior.

A practical experiment on Windows is to place a custom movable type inside `std::vector` and observe how design choices around copy, move, and `noexcept` affect usability and performance characteristics.

2.2.12 Closing perspective

Special member functions are where the languages object model becomes concrete. They determine whether a class behaves like a value, like a unique owner, like a movable handle, or like a non-copyable system resource. They are also the point where careless design most easily creates lifetime bugs, double deletion, accidental conversions, inconsistent semantics, and exception-safety weaknesses.

Modern C++ therefore treats these functions not as boilerplate to type mechanically, but as part of the semantic identity of the class. A well-designed class either relies on the Rule of Zero and lets its members do the work, or it explicitly states its ownership and transfer policy through defaulted, deleted, explicit, and noexcept operations. That is the real purpose of studying special member functions deeply: not to memorize syntax, but to design object behavior honestly, safely, and efficiently from the very beginning of the class definition.

Constructors & Initialization Taxonomy

3.1 All Constructor Kinds

Constructors are one of the most fundamental mechanisms of the C++ object model. They establish the initial state of an object, enforce class invariants, and determine how objects enter their valid lifetime. In modern C++, constructors are not merely functions that assign values to members. They are the structural entry point into the object's lifetime and therefore form the first stage of resource management, ownership semantics, and safe program design.

A constructor is a special member function that has the same name as the class and has no return type. Its responsibility is to initialize the base classes and non-static data members of the object and establish the invariants required for correct operation of the class. Once a constructor finishes execution, the object is considered fully initialized and safe to use according to the rules of the type.

Modern C++ supports multiple forms of constructors. Each form serves a different role in the lifecycle of an object. These constructor categories allow objects to be created in different contexts such as default initialization, copying, moving, delegating between constructors, or constructing objects directly inside containers or wrapper objects.

This section examines the most important constructor categories used in modern C++:

- Default constructors
- Copy constructors

- Move constructors
- Delegating constructors
- In-place constructors

Understanding these constructors is essential for designing classes that are safe, efficient, and compatible with modern C++ libraries and containers.

3.1.1 Default constructors

A default constructor is a constructor that can be called without providing any arguments. It initializes an object into a valid default state. If a class does not define any constructors, the compiler may implicitly generate a default constructor when possible.
Default constructors are commonly used when objects are created before their values are known or when objects are stored inside containers that require default construction.
A simple example is shown below:

```
#include <iostream>

class Counter {
private:
    int value_;

public:
    Counter() : value_(0) {
        std::cout << "Default constructor called\n";
    }

    int value() const noexcept {
        return value_;
```

```
    }
};

int main() {
    Counter c;
    std::cout << c.value() << '\n';
}
```

The constructor initializes the internal value to zero and ensures the object starts in a valid state. Modern C++ also allows default constructors to be explicitly requested using the `= default` syntax:

```
class Point {
private:
    int x_ = 0;
    int y_ = 0;

public:
    Point() = default;
};
```

Here the compiler-generated constructor initializes the members using their default member initializers.

Default constructors play an important role in container usage. Many containers create elements internally before assigning values to them.

Example using a vector:

```
#include <vector>

class Item {
```

```
public:
    Item() = default;
};

int main() {
    std::vector<Item> items(5);
}
```

The vector creates five objects using the default constructor.
For Windows compilation using LLVM tools:

```
clang-cl /std:c++20 /EHsc /W4 /O2 default_constructor.cpp
```

3.1.2 Copy constructors

The copy constructor creates a new object as a copy of an existing object of the same type. It is invoked whenever an object is initialized from another object of the same type.
Typical situations that trigger a copy constructor include:

- passing objects by value
- returning objects by value
- initializing one object from another

The signature of a copy constructor typically takes a reference to a const object of the same type.

```
#include <iostream>
#include <string>

class Document {
```

```
private:
    std::string text_;

public:
    Document(std::string text)
        : text_(std::move(text)) {}

    Document(const Document& other)
        : text_(other.text_) {
        std::cout << "Copy constructor\n";
    }

    const std::string& text() const noexcept {
        return text_;
    }
};

int main() {
    Document d1("example");
    Document d2 = d1;

    std::cout << d2.text() << '\n';
}
```

In many cases, modern C++ classes can rely on compiler-generated copy constructors:

```
class Record {
private:
    int id_;
    double value_;
```

```
public:
    Record(int id, double value)
        : id_(id), value_(value) {}

    Record(const Record&) = default;
};
```

If the class members are themselves copyable types, the compiler-generated copy constructor is usually correct.
However, classes that manage resources such as raw pointers, file handles, or operating-system objects must implement copying carefully or disable it entirely.

3.1.3 Move constructors

Move constructors were introduced in modern C++ to support efficient transfer of resources between objects. Instead of copying expensive resources such as dynamic memory or file handles, a move constructor transfers ownership from one object to another.
Move constructors typically accept an rvalue reference:

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

class Buffer {
private:
    std::unique_ptr<int[]> data_;
    std::size_t size_;

public:
    Buffer(std::size_t size)
```

```
        : data_(std::make_unique<int[]>(size)), size_(size) {}

    Buffer(Buffer&& other) noexcept
        : data_(std::move(other.data_)),
          size_(other.size_) {

        other.size_ = 0;
        std::cout << "Move constructor\n";
    }
};

int main() {
    Buffer a(100);
    Buffer b = std::move(a);
}
```

Here the internal memory is transferred instead of duplicated. The source object remains in a valid but unspecified state.

Modern C++ containers heavily rely on move constructors for efficiency. When objects are reallocated or transferred between containers, moving avoids unnecessary memory duplication.

When the default move behavior is correct, it can be requested using:

```
class DataBlock {
private:
    std::unique_ptr<int[]> data_;

public:
    DataBlock(DataBlock&&) noexcept = default;
};
```

Move semantics are particularly important in high-performance systems programming and resource-management classes.

3.1.4 Delegating constructors

Delegating constructors allow one constructor of a class to call another constructor of the same class. This feature helps avoid duplicated initialization logic and ensures consistent construction behavior.
Instead of repeating initialization code across multiple constructors, one constructor can delegate to another.

```
#include <string>
#include <iostream>

class Logger {
private:
    std::string file_;
    int level_;

public:
    Logger() : Logger("default.log", 1) {}

    explicit Logger(std::string file)
        : Logger(std::move(file), 1) {}

    Logger(std::string file, int level)
        : file_(std::move(file)), level_(level) {
        std::cout << "Logger initialized\n";
    }
};
```

```
int main() {
    Logger a;
    Logger b("system.log");
}
```

The first two constructors delegate to the third constructor, which performs the real initialization.
Delegating constructors simplify maintenance and reduce the risk of inconsistent initialization logic.
This technique is widely used in modern C++ frameworks and libraries to centralize initialization logic.

3.1.5 In-place constructors

In-place construction allows an object to be constructed directly in its final storage location rather than being created temporarily and then copied or moved. This technique is used heavily in containers, optional values, and variant types.
In-place construction avoids unnecessary object moves and can improve performance in many situations.
A typical example occurs with containers:

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

class User {
private:
    std::string name_;
    int id_;
```

```
public:
    User(std::string name, int id)
        : name_(std::move(name)), id_(id) {}
};

int main() {
    std::vector<User> users;

    users.emplace_back("Alice", 1);
    users.emplace_back("Bob", 2);
}
```

The `emplace_back` operation constructs the object directly inside the container storage without creating a temporary object.

Another common in-place construction pattern appears with optional values:

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

class Session {
private:
    std::string name_;

public:
    explicit Session(std::string name)
        : name_(std::move(name)) {}
};

int main() {
    std::optional<Session> session;
```

```
    session.emplace("admin");
}
```

The object is constructed directly inside the memory managed by the optional wrapper.
In-place construction may also be performed manually using placement new when working with low-level storage management:

```
#include <new>
#include <iostream>

class Item {
public:
    Item(int v) {
        std::cout << "Item constructed with " << v << '\n';
    }
};

int main() {
    alignas(Item) unsigned char storage[sizeof(Item)];

    Item* p = new (storage) Item(42);

    p->~Item();
}
```

Here the object is constructed directly inside a preallocated buffer.
This technique is frequently used in memory pools, custom allocators, and high-performance systems.
For Windows development environments the examples can be compiled using the following command:

```
clang-cl /std:c++20 /EHsc /W4 /O2 constructors_taxonomy.cpp
```

Developers can experiment with the examples by modifying constructors, observing copy and move behavior, and printing diagnostic messages from constructors to understand exactly which constructor is invoked.

3.1.6 Practical perspective

Constructors form the first stage of an object's lifecycle. Each constructor category addresses a different scenario in which an object may be created. Default constructors establish baseline state, copy constructors replicate existing objects, move constructors transfer ownership efficiently, delegating constructors simplify initialization logic, and in-place construction enables efficient object creation directly in container storage.
A well-designed modern C++ class uses these constructors deliberately. Many classes require only a small subset of them. Others rely on compiler-generated implementations. Resource-management classes may require careful custom implementations to enforce ownership rules and maintain class invariants.
Understanding the taxonomy of constructors therefore provides a clear mental model of how objects enter existence in C++ and how class design interacts with memory management, container behavior, and modern resource-safe programming practices.

3.2 Aggregate vs Non-Aggregate Types

In modern C++, the distinction between **aggregate** and **non-aggregate** types is one of the most important parts of initialization design. This distinction affects how objects may be initialized, how much control the class author retains over invariants, how readable object construction appears at call sites, and how easily a type participates in generic programming, serialization-oriented code, configuration objects, and value-like data transport structures.

An aggregate is a type that can be initialized directly from a brace-enclosed list of values without invoking an ordinary user-provided constructor. In contrast, a non-aggregate type is initialized according to constructor overload resolution and the ordinary constructor rules of the language. This difference is not superficial. It reflects two very different design philosophies:

- aggregate design emphasizes exposed structure and direct member-wise initialization,
- non-aggregate design emphasizes controlled construction and invariant enforcement through constructors.

Modern C++ has refined the aggregate model over several language revisions. In particular, the C++20 rules are broader and more practical than older historical descriptions of aggregates. For this reason, a modern C++ programmer must not rely only on old textbook rules such as an aggregate is just a plain old struct with no constructors. The actual model is more subtle, and the design consequences are important.
This section studies aggregate and non-aggregate types through three central themes:

- the newer C++20 aggregate rules,
- design strategies,
- pitfalls and gotchas.

These topics are strongly connected. The rules determine what the language permits. Design strategy determines whether aggregate-style exposure is desirable. Pitfalls arise when programmers assume that syntax alone decides meaning, or when a class evolves over time and silently stops being an aggregate.

3.2.1 What an aggregate means in practice

At a practical level, an aggregate is a type that supports direct brace-based member initialization according to its declared structure. The initialization logic is structural rather than

constructor-driven. This makes aggregates especially useful for simple value carriers, configuration objects, records, message structures, and lightweight bundles of data where the public state is intentionally visible and no complex construction logic is required.
A basic example is:

```
#include <iostream>
#include <string>

struct ServerConfig {
    std::string host;
    int port;
    bool secure;
};

int main() {
    ServerConfig cfg{"localhost", 443, true};

    std::cout << cfg.host << '\n';
    std::cout << cfg.port << '\n';
    std::cout << std::boolalpha << cfg.secure << '\n';
}
```

This form is clear and efficient. The object is initialized directly by its members in declaration order. No user-defined constructor is involved.
A non-aggregate version of a similar type may instead centralize initialization through constructors:

```
#include <iostream>
#include <string>
#include <utility>
```

```
class SecureConfig {
private:
    std::string host_;
    int port_;
    bool secure_;

public:
    SecureConfig(std::string host, int port, bool secure)
        : host_(std::move(host)), port_(port), secure_(secure) {
    }

    const std::string& host() const noexcept { return host_; }
    int port() const noexcept { return port_; }
    bool secure() const noexcept { return secure_; }
};

int main() {
    SecureConfig cfg("localhost", 443, true);

    std::cout << cfg.host() << '\n';
    std::cout << cfg.port() << '\n';
    std::cout << std::boolalpha << cfg.secure() << '\n';
}
```

This version is intentionally not aggregate-like in style. It uses constructor-based control and encapsulates representation.

The important lesson is that aggregate versus non-aggregate is not about one style being universally better. It is about choosing whether a type should be initialized structurally or through an explicit construction interface.

3.2.2 New C++20 aggregate rules

Older C++ teaching often described aggregates in a very narrow way: no constructors, no bases, no private members, and essentially a simple C-style structure. Modern C++ broadened this model substantially, especially by making aggregates more useful in realistic class hierarchies and by supporting designated initialization syntax in C++20.
A modern programmer should understand the following general direction of the C++20 aggregate model:

- aggregates are still types initialized structurally rather than through user-provided constructors,
- aggregate initialization remains brace-based and follows the declared structure of the type,
- aggregates may participate in more realistic modern designs than older pre-C++17 folklore would suggest,
- designated initializers in C++20 make aggregate initialization more expressive for suitable types.

A simple C++20-style designated initialization example is:

```
#include <iostream>

struct DisplayMode {
    int width;
    int height;
    bool fullscreen;
};
```

```
int main() {
    DisplayMode mode{
        .width = 1920,
        .height = 1080,
        .fullscreen = true
    };

    std::cout << mode.width << "x" << mode.height << '\n';
    std::cout << std::boolalpha << mode.fullscreen << '\n';
}
```

This syntax makes aggregate construction easier to read, especially when a type has many members of similar kinds. It is particularly useful for configuration records and test data setup. An important practical point is that designated initialization does not transform an arbitrary class into an aggregate. The type must already satisfy the aggregate conditions of the language. If it does not, designated initialization is not available.

C++20 also made aggregate usage more practical with inheritance in certain cases. A structurally simple derived type may still behave aggregate-like if it satisfies the relevant language conditions.

For example:

```
#include <iostream>
#include <string>

struct Address {
    std::string city;
    std::string street;
};

struct Office : Address {
```

```
    int room;
};

int main() {
    Office office{{"Riyadh", "Main Street"}, 203};

    std::cout << office.city << '\n';
    std::cout << office.street << '\n';
    std::cout << office.room << '\n';
}
```

This kind of design would not fit the oversimplified old rule that aggregates must not involve inheritance in practice. Modern aggregate rules are more expressive, though still structural. Another important C++20-era practical point is that default member initializers and aggregate-style construction interact naturally:

```
#include <iostream>
#include <string>

struct BuildOptions {
    std::string target = "x64";
    bool optimize = true;
    int level = 2;
};

int main() {
    BuildOptions a{};
    BuildOptions b{"arm64", false, 0};

    std::cout << a.target << ' ' << a.optimize << ' ' << a.level << '\n';
```

```
    std::cout << b.target << ' ' << b.optimize << ' ' << b.level << '\n';
}
```

This style is highly practical. It provides readable defaults while preserving brace-based structural construction.
The best way to think about the new aggregate rules is not as a list to memorize mechanically, but as a modernized structural-initialization model. The language permits aggregate construction in more useful scenarios than older folklore suggests, but the central idea remains the same: the type must still be fundamentally constructor-light and structurally initialized.

3.2.3 Design strategies

Choosing aggregate or non-aggregate design is an architectural decision. The correct choice depends on whether the type is primarily a **data structure** or an **abstraction boundary**.
A good candidate for aggregate design usually has these properties:

- it is mainly a transparent bundle of related values,
- its invariants are weak, obvious, or naturally enforced by the types of its members,
- direct member visibility is acceptable,
- brace-based construction improves readability,
- value-like usage is more important than encapsulated behavior.

A good example is a simple request object:

```
#include <string>

struct LoginRequest {
    std::string user;
```

```
    std::string password;
    bool remember_me;
};
```

This kind of type is often ideal as an aggregate because it acts as a data carrier rather than as an active object with hidden invariants.
Another strong aggregate use case is configuration transport:

```
#include <string>

struct CompilerOptions {
    std::string standard = "c++20";
    bool warnings_as_errors = false;
    bool optimize = true;
    int optimization_level = 2;
};
```

This is readable, flexible, and easy to use in tests and setup code.
By contrast, a type should often be non-aggregate when:

- it must enforce invariants at construction,
- some states must be forbidden,
- the representation should remain hidden,
- ownership or lifetime rules require controlled construction,
- member order should not leak into the public interface,
- future evolution of the implementation should not break callers.

A good non-aggregate example is a validated range:

```cpp
#include <stdexcept>

class Range {
private:
    int low_;
    int high_;

public:
    Range(int low, int high)
        : low_(low), high_(high) {
        if (low_ > high_) {
            throw std::invalid_argument("invalid range");
        }
    }

    int low() const noexcept { return low_; }
    int high() const noexcept { return high_; }
};
```

This type should not be aggregate-like because unrestricted structural initialization would weaken invariant enforcement.
A useful design strategy in modern C++ is therefore:

- use aggregates for transparent value carriers,
- use non-aggregates for types with semantics stronger than public data bundle,
- avoid turning a conceptually rich abstraction into an aggregate merely for syntactic convenience,
- avoid overengineering a pure record type into a constructor-heavy abstraction when direct structure is clearer.

For performance-sensitive and systems-oriented work, aggregate design can also be attractive because it is easy to inspect, easy to initialize in tests, and often friendly to simple value-oriented patterns. However, this should never come at the cost of losing necessary invariants.

3.2.4 Pitfalls & gotchas

One of the most important aggregate-related pitfalls is that a type may stop being an aggregate after a seemingly small class evolution. For example, adding a constructor, changing access control, or introducing certain other structural features can silently change how initialization works.
Consider this type:

```
struct Options {
    int a;
    int b;
};
```

It can be initialized as:

```
Options x{1, 2};
```

Now suppose someone later adds a constructor:

```
struct Options {
    int a;
    int b;

    Options(int x, int y) : a(x), b(y) {
    }
};
```

The syntax may still look similar at some call sites, but the semantics have changed fundamentally. Construction is now constructor-driven rather than aggregate-driven. This matters for overload resolution, narrowing behavior, designated initialization availability, and API expectations.

A second major pitfall is assuming that aggregate initialization is immune to declaration-order concerns. It is not. Aggregate initialization proceeds structurally according to the declared order of elements. This means the member order becomes part of the visible initialization contract.

```
struct Triple {
    int x;
    int y;
    int z;
};

int main() {
    Triple t{1, 2, 3};
}
```

This is obvious in a tiny type, but in larger records it becomes fragile. Reordering members changes the meaning of existing brace-based call sites. That is one of the long-term maintenance costs of aggregates.

A third pitfall is forgetting that aggregates with public members may have weaker invariant protection. If a type must always satisfy a relationship between fields, aggregate exposure may make it too easy to create invalid objects.

```
struct Rectangle {
    int width;
    int height;
};
```

This is fine if negative values are acceptable or harmless. But if the real abstraction requires strictly positive dimensions, aggregate design may be too permissive. A constructor-based type would be stronger.

Another pitfall involves default member initializers combined with partial aggregate initialization. Members not explicitly initialized in a brace list follow the aggregate rules and default member initializers where appropriate. This is useful, but the programmer must understand exactly which defaults are being relied upon.

```
#include <iostream>

struct Options {
    int width = 800;
    int height = 600;
    bool fullscreen = false;
};

int main() {
    Options a{};
    Options b{1024};

    std::cout << a.width << ' ' << a.height << ' ' << a.fullscreen << '\n';
    std::cout << b.width << ' ' << b.height << ' ' << b.fullscreen << '\n';
}
```

This can be extremely useful, but it also means that changing member order later may unexpectedly change which values are being supplied positionally.

A further gotcha is assuming that designated initialization in C++ behaves exactly like C designated initialization. Modern C++ designated initializers are more constrained and must follow the C++ rules for aggregates. A programmer moving between C and C++ low-level code should not assume the languages behave identically here.

Another long-term design gotcha is API stability. Aggregates expose their structure directly to callers. That means any evolution of member order, member presence, or structural form is more likely to affect call sites. Constructor-based non-aggregate classes can hide those changes more effectively behind a stable abstraction layer.
For this reason, aggregates are often excellent for:

- local value carriers,
- internal configuration structs,
- test fixtures,
- simple public record types that are intentionally transparent.

They are often less appropriate for:

- long-lived public library abstractions whose representation may evolve,
- types with nontrivial invariants,
- ownership-heavy classes,
- handles and RAII wrappers,
- abstractions where future implementation flexibility matters.

3.2.5 Aggregate versus non-aggregate in real design

A useful practical comparison is the difference between a transport object and a semantic object.
A transport object may be:

```
#include <string>

struct BuildCommand {
    std::string source;
    std::string output;
    bool debug;
};
```

This is naturally aggregate-like. It is just a bundle of values.

A semantic object may be:

```
#include <stdexcept>
#include <string>
#include <utility>

class OutputPath {
private:
    std::string path_;

public:
    explicit OutputPath(std::string path)
        : path_(std::move(path)) {
        if (path_.empty()) {
            throw std::invalid_argument("output path must not be empty");
        }
    }

    const std::string& value() const noexcept {
        return path_;
    }
```

```
};
```

This should clearly be non-aggregate. The abstraction has a validity rule that deserves controlled construction.
The strongest modern design discipline is to decide whether a type is:

- a **record**, where structure is the interface,
- or an **abstraction**, where behavior and invariants are the interface.

Aggregates are best for the first category. Non-aggregates are usually best for the second.

3.2.6 Windows-oriented notes

All examples in this section are suitable for a Windows workflow using Visual Studio Developer Command Prompt and a recent Clang installation with `clang-cl`. A practical compilation command is:

```
clang-cl /std:c++20 /EHsc /W4 /O2 aggregate_vs_nonaggregate.cpp
```

For experimentation on Windows, a very good exercise is to:

- start from an aggregate,
- add a constructor,
- change member order,
- add default member initializers,
- test designated initialization,
- observe which call sites continue to compile and which change meaning.

This exercise quickly reveals how strongly aggregate usage is tied to the visible structure of the type.

3.2.7 Closing perspective

Aggregate and non-aggregate types represent two different and equally important directions in modern C++ design. Aggregate types expose structure and favor direct member-wise initialization. Non-aggregate types centralize control through constructors and are better suited for enforcing invariants, hiding representation, and supporting long-term abstraction stability. The newer C++20 aggregate rules make aggregates more practical and expressive than older simplified descriptions suggest, especially with designated initializers and broader structural usability. However, the core design question remains unchanged: should this type be initialized as transparent structure, or should it be constructed through a controlled abstraction boundary? A programmer who understands this distinction deeply can design classes with greater clarity. Such a programmer knows when brace-based directness is a strength, when constructor-based control is necessary, and how to avoid the maintenance traps that arise when the structure of a type silently becomes part of its public contract.

Part II

Move Semantics, Perfect Forwarding & Lifetime Design

Deep Dive into Move Semantics

4.1 Why Move Semantics?

Move semantics became one of the most important design changes in Modern C++ because they changed the cost model of value-based programming. Before their introduction, returning objects by value, storing heavy objects inside standard containers, and composing abstractions from resource-owning classes often implied extra copying work. Even when compilers could optimize some of these cases, programmers still had to think defensively about temporary objects, hidden allocations, and the cost of transferring ownership.
Modern C++ changed this landscape in two complementary ways:

1. It introduced **move operations**, allowing an object that is about to expire to transfer its internal resources instead of duplicating them.

2. It refined **copy-elision rules**, especially from C++17 onward, so that many apparent copies and moves are not merely optimized away as an optional improvement, but are instead specified by the language model in a much stronger way.

As a result, Modern C++ encourages a style in which objects are passed and returned by value much more naturally than in older code bases. This is one of the deep reasons why RAII-heavy design, standard-library containers, value types, and composable abstractions scale much better today than they did in the C++98 era.

4.1.1 Performance history

In early C++ design, resource-managing classes usually followed a simple rule: if an object owned memory, a file handle, a mutex, a socket, or any other non-trivial resource, copying that object meant creating a second owner or duplicating the resource representation. This was often expensive, and in some classes it was not even semantically meaningful.
For example, consider a class that owns a dynamically allocated buffer. Under the pre-C++11 model, if such an object was returned by value, inserted into a container, or used as a temporary result of an expression, the language primarily relied on copying. The copy constructor had to allocate new storage and duplicate the contents. For large buffers, long strings, or complex ownership graphs, this could be costly.
This historical cost pressure shaped older coding styles:

- programmers preferred output parameters over return-by-value,
- resource-owning classes were often made non-copyable,
- designs leaned more heavily on pointers and manual lifetime control,
- temporary objects were treated with suspicion.

That style was not only verbose, but also frequently less safe. It pushed programmers toward raw ownership protocols that were harder to reason about, harder to maintain, and more error-prone.
C++11 introduced `T&&` and move constructors so that an expiring object could transfer ownership of its internals. Instead of allocating a second buffer and copying bytes, a moved-from source could hand over its pointer, size, and capacity to the destination, then reset itself into a valid but unspecified state.
This made a major conceptual shift possible:

> Copy means duplication of value. Move means transfer of representation or ownership from an object that is about to expire.

That distinction transformed performance-sensitive code. Standard-library containers, strings, smart pointers, futures, and many user-defined types became significantly cheaper to return, rearrange, and store.

A minimal ownership example illustrates the historical difference clearly:

Pre-move cost pattern

```
#include <cstddef>
#include <algorithm>

class Buffer
{
    std::size_t size_;
    int* data_;

public:
    explicit Buffer(std::size_t n)
        : size_(n), data_(new int[n]{})
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }

    Buffer(const Buffer& other)
```

```
        : size_(other.size_), data_(new int[other.size_])
    {
        std::copy(other.data_, other.data_ + size_, data_);
    }
};
```

The copy constructor above performs allocation and element copying. If this type is returned by value or reallocated inside a container, the operation may be expensive.

Modern move-aware pattern

```
#include <cstddef>
#include <algorithm>
#include <utility>

class Buffer
{
    std::size_t size_;
    int* data_;

public:
    explicit Buffer(std::size_t n)
        : size_(n), data_(new int[n]{})
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }
```

```
Buffer(const Buffer& other)
    : size_(other.size_), data_(new int[other.size_])
{
    std::copy(other.data_, other.data_ + size_, data_);
}

Buffer(Buffer&& other) noexcept
    : size_(other.size_), data_(other.data_)
{
    other.size_ = 0;
    other.data_ = nullptr;
}

Buffer& operator=(Buffer&& other) noexcept
{
    if (this != &other)
    {
        delete[] data_;
        size_ = other.size_;
        data_ = other.data_;
        other.size_ = 0;
        other.data_ = nullptr;
    }
    return *this;
}

Buffer& operator=(const Buffer& other)
{
```

```
        if (this != &other)
        {
            int* new_data = new int[other.size_];
            std::copy(other.data_, other.data_ + other.size_, new_data);
            delete[] data_;
            data_ = new_data;
            size_ = other.size_;
        }
        return *this;
    }
};
```

The move constructor above does not duplicate the array. It transfers the pointer. For large owned resources, that is the central performance motivation behind move semantics.

Why the standard library benefited immediately

Move semantics were especially powerful because the standard library already used value-based interfaces. Once move support arrived, the same high-level interfaces became much more efficient without requiring programmers to abandon RAII or container-based design.
For example:

- `std::vector<T>` can move elements during reallocation when possible,
- `std::string` can often transfer internal storage instead of copying characters,
- `std::unique_ptr<T>` is fundamentally move-only because ownership transfer is its core semantics,
- function return by value became far more practical for large objects.

This is why move semantics should not be seen as a niche optimization. They are a foundational language mechanism that made Modern C++ value-oriented design scale to systems-level workloads.

4.1.2 Copy-elision evolution

Move semantics alone do not explain modern performance. Equally important is the evolution of copy elision and the object model itself.
Historically, copy elision was a permitted optimization. The language allowed implementations to omit certain object creations even if constructors or destructors had observable side effects. This was already unusual, because most optimizations are not allowed to change observable behavior in that way.
Classic cases included:

- returning a local object by value,
- initializing an object from a temporary of the same type,
- some throw and handler cases.

This gave rise to well-known terms such as:

- **RVO** — Return Value Optimization,
- **NRVO** — Named Return Value Optimization.

However, before C++17, these were still largely framed as optimization opportunities. A compiler was allowed to elide construction in certain cases, but programmers could not treat all such cases as guaranteed by the language.

Pre-C++17 intuition

Consider this function:

```
#include <vector>

std::vector<int> make_data()
{
    std::vector<int> v{1, 2, 3, 4, 5};
    return v;
}
```

In older C++ discussions, one would ask:

- Will the compiler apply NRVO and construct directly in the caller?
- If not, will the returned object at least be moved instead of copied?
- If neither happens, do we pay for a full copy?

That was already much better in C++11 than in C++98 because the fallback path could use a move constructor. Still, the programmer had to think in layers of possibility: elide if possible, otherwise move if possible, otherwise copy.

C++17 strengthened the model

C++17 made a major conceptual change. In many cases involving prvalues of class type, the language no longer models the program as creating a separate temporary object and then copying or moving from it. Instead, the prvalue initializes the final destination directly. This is one of the most important modern lessons:

> In post-C++17 C++, many expressions that look like temporary-producing expressions are better understood as direct initialization of the final result object.

That is why code such as the following is so important:

```
class TokenStream
{
public:
    TokenStream() = default;
    TokenStream(const TokenStream&) = delete;
    TokenStream(TokenStream&&) = delete;
};

TokenStream make_stream()
{
    return TokenStream{};
}
```

In modern rules, this pattern is well-formed because the returned prvalue can initialize the function result object directly. There is no requirement for an actual move operation to happen in that return expression.
This is a major evolution from older mental models.

Temporary materialization

Another essential modern concept is **temporary materialization conversion**. A prvalue is not always best imagined as a fully materialized temporary object living somewhere in memory. In the modern object model, materialization happens only when a real temporary object is actually needed.
That is why advanced modern C++ explanations distinguish carefully between:

- **producing a prvalue**,
- **materializing a temporary object**,

- **binding a reference or initializing a destination object**.

This matters because it explains why some constructions no longer require copyability or movability even though older intuition says a temporary must exist first.

Guaranteed copy elision does not eliminate the need for move semantics

A common misunderstanding is that C++17 made move semantics unimportant. That is false. Guaranteed copy elision helps when the language can initialize the final object directly from a prvalue of the same type. But many important operations still rely on move semantics:

- moving from named objects,
- container reallocation,
- swaps and transfers of ownership,
- returning named locals when optional elision is not performed,
- algorithms that rearrange elements,
- explicit ownership transfer through `std::move`.

So the correct modern view is:

> Guaranteed copy elision reduced the number of situations that require actual moves, but move semantics remain essential for the situations where object identity and ownership transfer still matter.

A practical comparison

The following examples show the difference between guaranteed direct initialization and move-based transfer.

Guaranteed direct initialization from a prvalue

```
class LargeObject
{
public:
    LargeObject() = default;
    LargeObject(const LargeObject&) = delete;
    LargeObject(LargeObject&&) = delete;
};

LargeObject build()
{
    return LargeObject{};
}
```

This works because the returned prvalue initializes the result object directly.

Move still matters for named objects

```
#include <utility>

class LargeObject
{
public:
    LargeObject() = default;
    LargeObject(const LargeObject&) = delete;
    LargeObject(LargeObject&&) noexcept = default;
};

void consume(LargeObject obj)
{
```

```
}

int main()
{
    LargeObject x;
    consume(std::move(x));
}
```

Here the source is a named object. This is not the same situation as returning a prvalue like `LargeObject{}`. The transfer relies on move semantics.

Named return optimization in current MSVC practice

In practical Windows development with MSVC, optional named return value optimization can be influenced by compiler settings. Modern MSVC supports stronger optional elision behavior with `/Zc:nrvo`, and this option is automatically enabled by several common build modes such as `/O2`, `/permissive-`, or `/std:c++20` and later. This affects optional elision cases such as returning a named local, while mandatory elision required by the language remains in force regardless.

4.1.3 Move vs Copy cost models

To design good C++ classes, one must understand that copying and moving have different semantic and performance meanings. They are not merely two spellings for the same operation.

Copy cost model

A copy operation conceptually means:

> Create a new object that owns or represents the same value independently of the source.

Depending on the type, copying may involve:

- allocating new dynamic memory,
- duplicating file or OS handles through a separate protocol,
- copying element sequences,
- incrementing reference counts,
- recursively copying subobjects.

Therefore copy cost often scales with object size or representation complexity. For many resource-owning types, copy is linear in the amount of managed data.
Typical examples:

- copying a large `std::vector<int>` duplicates all elements,
- copying a large owning string may duplicate character storage,
- copying a tree-like custom object may recursively duplicate many nodes.

Move cost model

A move operation conceptually means:

> Transfer the internals of a soon-to-expire source object into a destination, leaving the source valid but otherwise unspecified.

For resource-owning types, moving is often much cheaper because it may involve only:

- pointer transfer,
- size and capacity transfer,

- handle transfer,
- resetting the source object.

That often makes move cost approximately constant with respect to the amount of owned dynamic storage.
For example, moving a vector often transfers its internal pointer, size, and capacity rather than each element individually.

But move is not always cheaper

An advanced cost model must also recognize the limits of the slogan “move is cheap”.
Move is often cheaper, but not always.

Case 1: trivially copyable types For small plain data types, copying may be as cheap as moving, or even identical at machine level.

```
struct Point
{
    int x;
    int y;
};
```

For such types, copy and move are not fundamentally different performance events. Both are usually just register or memory transfers.

Case 2: types whose move must still process many elements Some user-defined types cannot implement move as a simple pointer steal. A move may still need to visit subobjects or preserve internal invariants. In such designs, moving may be cheaper than copying, but not constant-time.

Case 3: move-only but not free A move-only type such as `std::unique_ptr<T>` is semantically about ownership transfer, not just speed. It is very cheap to move, but the deeper reason to move it is correctness of ownership.

Case 4: noexcept and container behavior For many standard containers, whether a type's move constructor is marked `noexcept` can influence whether move is preferred during internal relocation. A throwing move constructor can restrict some optimizations and may cause a container to choose copying when that is the stronger exception-safety path.
This is why well-designed move constructors for resource-owning value types are commonly declared `noexcept`.

A class-level cost comparison

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

class Tracer
{
    std::vector<int> data_;

public:
    explicit Tracer(std::size_t n)
        : data_(n, 42)
    {
        std::cout << "construct\n";
    }

    Tracer(const Tracer& other)
```

```
        : data_(other.data_)
    {
        std::cout << "copy\n";
    }

    Tracer(Tracer&& other) noexcept
        : data_(std::move(other.data_))
    {
        std::cout << "move\n";
    }
};

Tracer make_tracer()
{
    Tracer t(1000000);
    return t;
}

int main()
{
    Tracer a = make_tracer();
}
```

In modern compilation, several outcomes are possible conceptually:

- the object may be constructed directly into `a`,
- if optional elision is not applied in some context, move may be used,
- deep copy is the most expensive fallback.

The key design lesson is that a modern type should ideally support all three layers correctly:

1. direct construction when the language permits it,
2. efficient move when transfer is needed,
3. correct copy when true duplication is semantically required.

Semantic difference matters more than micro-benchmarks

It is tempting to reduce move semantics to a benchmarking topic, but that misses the deeper design value.
Copy and move express different intentions:

- **copy** preserves the source as an independent equal-value object,
- **move** treats the source as a relinquishing object whose resources may be transferred.

This affects API design, ownership reasoning, exception safety, and container interaction. Performance is only one consequence of that semantic distinction.

When should you design for cheap move?

A type benefits strongly from move semantics when it owns external or heap-based resources, such as:

- dynamic memory,
- file handles,
- sockets,
- mutex wrappers with ownership-like semantics,
- large buffers,

- parse trees,
- AST nodes,
- command buffers,
- large container aggregates.

Such types are central in compilers, interpreters, GUI engines, networking systems, and backend services. In all of these domains, move semantics support both expressive interfaces and efficient execution.

When should you not overemphasize move?

For tiny scalar aggregates and trivially copyable structures, the most important properties are often:

- clarity,
- triviality,
- standard-layout behavior when needed,
- simple pass-by-value design.

In these cases, move semantics may exist formally, but they are not where the real optimization leverage lies.

4.1.4 Design conclusions

The question “Why move semantics?” has three complementary answers.

First, **historically**, move semantics solved a real performance and expressiveness problem in value-oriented C++ code. They made it practical to write high-level RAII abstractions without paying deep-copy cost in every transfer path.
Second, **language-evolution wise**, move semantics work together with copy-elision rules. C++11 gave programmers efficient transfer of resources. C++17 then strengthened the object model so that many temporary-looking operations no longer require an actual move or copy at all.
Third, **architecturally**, move semantics encode ownership transfer in the type system. They are not only faster than copying for many classes; they are also the correct semantic tool for representing expiring objects and one-way transfer of resources.
For Modern C++ design, the best mindset is this:

1. prefer clean value-oriented interfaces,
2. rely on guaranteed direct initialization where the language provides it,
3. implement efficient move operations for resource-owning types,
4. keep copy semantics only when independent duplication is meaningful,
5. use `noexcept` move operations when appropriate,
6. remember that moved-from objects must remain valid, even if their exact value is unspecified.

That is the real foundation on which later topics such as perfect forwarding, sink parameters, ownership APIs, and lifetime-oriented class design are built.

4.1.5 Extensive examples

Example 1: expensive copy, cheap move

```
#include <iostream>
```

```
#include <cstring>
#include <utility>

class TextBuffer
{
    char* data_;
    std::size_t size_;

public:
    explicit TextBuffer(const char* text)
    {
        size_ = std::strlen(text);
        data_ = new char[size_ + 1];
        std::memcpy(data_, text, size_ + 1);
    }

    ~TextBuffer()
    {
        delete[] data_;
    }

    TextBuffer(const TextBuffer& other)
    {
        size_ = other.size_;
        data_ = new char[size_ + 1];
        std::memcpy(data_, other.data_, size_ + 1);
        std::cout << "copy\n";
    }
```

```
    TextBuffer(TextBuffer&& other) noexcept
        : data_(other.data_), size_(other.size_)
    {
        other.data_ = nullptr;
        other.size_ = 0;
        std::cout << "move\n";
    }

    const char* c_str() const
    {
        return data_ ? data_ : "";
    }
};

TextBuffer build_text()
{
    return TextBuffer("Modern C++ move semantics");
}

int main()
{
    TextBuffer a = build_text();
    std::cout << a.c_str() << '\n';
}
```

This example demonstrates the practical reason move was introduced. Copy duplicates the owned buffer. Move transfers the pointer. In many modern builds, the return path may even be directly constructed without either operation.

Example 2: named local and optional elision

```
#include <vector>

std::vector<int> make_numbers()
{
    std::vector<int> result{1, 2, 3, 4, 5};
    return result;
}
```

This is a classic NRVO-style pattern. A compiler may construct `result` directly in the caller's storage. If optional named return elision is not performed in some build configuration, move semantics still provide an efficient fallback.

Example 3: explicit move from a named object

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

void consume(std::unique_ptr<int> p)
{
    if (p)
        std::cout << *p << '\n';
}

int main()
{
    auto ptr = std::make_unique<int>(42);

    consume(std::move(ptr));
```

```
    if (!ptr)
        std::cout << "ownership transferred\n";
}
```

This example is not about copy elision. It is about explicit ownership transfer from a named object. This remains a core use case for move semantics.

Example 4: pass-by-value becomes practical

```
#include <string>
#include <utility>

class Person
{
    std::string name_;

public:
    explicit Person(std::string name)
        : name_(std::move(name))
    {
    }
};
```

This constructor style is common in Modern C++. If the caller passes an rvalue, the parameter can be constructed efficiently and then moved into the member. If the caller passes an lvalue, one copy may occur into the parameter, followed by a move into the member. The design is simple, expressive, and often efficient enough.

4.1.6 Windows build notes

The following commands are suitable for testing these examples on Windows.

MSVC command line

```
cl /std:c++23 /EHsc /W4 /nologo example.cpp
cl /std:c++23 /EHsc /W4 /O2 /Zc:nrvo /nologo example.cpp
```

The first command builds with a modern language mode and standard exception handling. The second adds optimization and explicitly enables optional named return value optimization behavior in MSVC.

GCC on Windows via MSYS2 or MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

4.1.7 Practical rules to remember

1. Move semantics exist because copying resource-owning objects can be expensive or semantically inappropriate.
2. C++11 introduced the language machinery for ownership transfer through rvalue references and move operations.
3. C++17 strengthened the model of prvalues and guaranteed direct initialization in important cases.
4. Guaranteed copy elision does not replace move semantics; it complements them.

5. Copy cost is usually proportional to the amount of state that must be duplicated.
6. Move cost is often much smaller for resource-owning types, but not every type benefits equally.
7. A move constructor should usually leave the source valid but unspecified.
8. For many resource-owning value types, `noexcept` move operations are a best practice.

4.2 Implementing Move Constructors

Move constructors are one of the central mechanisms that made Modern C++ practical for high-performance value-oriented programming. A move constructor exists to transfer the internal state of an object that is about to expire into a newly created object, instead of duplicating that state through a deep copy.
According to the C++ language rules, a move constructor is a non-template constructor whose first parameter is of type `X&&`, `const X&&`, `volatile X&&`, or `const volatile X&&`, with no other parameters or only parameters that have default arguments. In real-world design, the overwhelmingly important form is `X(X&&)`. This is the form used by standard types, user-defined RAII wrappers, containers, handle classes, and ownership-based abstractions.
The practical purpose of a move constructor is not merely speed in the abstract. Its purpose is to preserve efficient and expressive object-oriented and value-based design while avoiding unnecessary duplication of expensive resources such as dynamic memory, file handles, sockets, synchronization objects, large buffers, and ownership graphs.

4.2.1 Designing optimal move operations

A good move constructor must satisfy both semantic and performance goals. It should transfer ownership or transferable state from the source object into the destination object, leave the source object in a valid state, avoid unnecessary work, and preserve class invariants.

What the language actually does for implicitly-defined moves

If a class does not explicitly declare a move constructor, the language may implicitly declare one as defaulted, but only under specific conditions. In particular, a class will not get an implicit move constructor if it already has a user-declared copy constructor, copy assignment operator, move assignment operator, or destructor.

This rule is extremely important in practice. Many classes accidentally lose implicit move support simply because the programmer writes a destructor or copy operation. Once that happens, code that looks movable may silently fall back to copying, or may become non-movable altogether depending on the structure of the type.

For classes that can use the default behavior safely, the best implementation is often:

```
class Widget
{
public:
    Widget() = default;
    Widget(const Widget&) = default;
    Widget(Widget&&) = default;
    Widget& operator=(const Widget&) = default;
    Widget& operator=(Widget&&) = default;
    ~Widget() = default;
};
```

This is the cleanest form when all bases and members already provide correct semantics and there is no need for a custom move protocol.

Memberwise move and invariant preservation

An implicitly-defined move constructor performs a memberwise move of bases and members. That means the quality of the generated move operation depends directly on the quality of each base class and each member type.

If all members are already good movable types, defaulting is usually ideal. If the class manages raw resources manually, then a custom move constructor is often needed.
The central design principle is this:

> Prefer `= default` when the class is already composed of correct movable parts. Write a custom move constructor only when the type must manually transfer ownership or must reestablish invariants explicitly.

The source object after move

A move operation does not destroy the source object. The source remains alive until its destructor runs. Therefore, the moved-from object must remain valid. Its exact value is typically unspecified, but it must still satisfy the fundamental requirements of the type so that destruction and reassignment remain safe.
This leads to one of the most important rules in move design:

1. transfer the owned state,
2. reset the source into a safe state,
3. preserve all class invariants in both objects.

A classical manually managed buffer demonstrates this clearly:

```
#include <cstddef>
#include <utility>
#include <algorithm>

class Buffer
{
    std::size_t size_;
```

```
    int* data_;

public:
    explicit Buffer(std::size_t n)
        : size_(n), data_(new int[n]{})
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }

    Buffer(const Buffer& other)
        : size_(other.size_), data_(new int[other.size_])
    {
        std::copy(other.data_, other.data_ + size_, data_);
    }

    Buffer(Buffer&& other) noexcept
        : size_(other.size_), data_(other.data_)
    {
        other.size_ = 0;
        other.data_ = nullptr;
    }

    Buffer& operator=(const Buffer& other)
    {
        if (this != &other)
```

```
    {
        int* new_data = new int[other.size_];
        std::copy(other.data_, other.data_ + other.size_, new_data);
        delete[] data_;
        data_ = new_data;
        size_ = other.size_;
    }
    return *this;
}

Buffer& operator=(Buffer&& other) noexcept
{
    if (this != &other)
    {
        delete[] data_;
        size_ = other.size_;
        data_ = other.data_;
        other.size_ = 0;
        other.data_ = nullptr;
    }
    return *this;
}

std::size_t size() const noexcept
{
    return size_;
}

const int* data() const noexcept
```

```
    {
        return data_;
    }
};
```

This is a classical ownership-transfer move constructor. The destination steals the pointer and size, and the source is reset so that its destructor remains safe.

Why this is optimal

The move constructor above is optimal in the sense that it avoids deep copying and performs only constant-time state transfer. It does not allocate memory. It does not duplicate the buffer. It does not perform element-by-element copying. For a resource-owning class, this is the essence of a high-quality move operation.
However, optimal does not always mean hand-written. If the data member were a `std::vector<int>` instead of a raw pointer, then defaulting would usually be better than manual code:

```
#include <vector>

class Buffer2
{
    std::vector<int> data_;

public:
    Buffer2() = default;
    explicit Buffer2(std::size_t n) : data_(n) {}

    Buffer2(const Buffer2&) = default;
    Buffer2(Buffer2&&) noexcept = default;
```

```
    Buffer2& operator=(const Buffer2&) = default;
    Buffer2& operator=(Buffer2&&) noexcept = default;
    ~Buffer2() = default;
};
```

This version is better because it delegates ownership management to standard-library types that already implement correct move semantics.

When not to write a custom move constructor

Do not write a custom move constructor just to imitate what the compiler can already generate correctly. Custom code increases maintenance cost, makes the type longer and more error-prone, and can accidentally weaken triviality, constexpr usability, or exception guarantees.
A custom move constructor is usually justified only when at least one of the following is true:

- the class owns a raw resource directly,
- the class has invariants that require repair after transfer,
- the class needs special instrumentation or ABI control,
- the class must intentionally suppress or customize part of the memberwise move behavior.

A bad move constructor pattern

The following style is logically broken:

```
class BadBuffer
{
    std::size_t size_;
```

```
    int* data_;

public:
    BadBuffer(BadBuffer&& other) noexcept
        : size_(other.size_), data_(other.data_)
    {
    }

    ~BadBuffer()
    {
        delete[] data_;
    }
};
```

This constructor steals the pointer but does not reset the source object. Both objects will later attempt to destroy the same pointer. That is a double-delete bug.

Another bad pattern: moving by copying

A move constructor that performs full deep copy defeats the point of move semantics:

```
class SlowBuffer
{
    std::size_t size_;
    int* data_;

public:
    SlowBuffer(SlowBuffer&& other)
        : size_(other.size_), data_(new int[other.size_])
    {
```

```
        std::copy(other.data_, other.data_ + size_, data_);
    }
};
```

This is not an efficient transfer operation. It is just a copy under a misleading name.

Designing around standard members

The most effective Modern C++ strategy is often to build classes from standard movable members and then default the move constructor:

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

class ProjectState
{
    std::string name_;
    std::vector<int> ids_;
    std::unique_ptr<int> primary_id_;

public:
    ProjectState() = default;
    ProjectState(std::string name, std::vector<int> ids, std::unique_ptr<int> p)
        : name_(std::move(name)),
          ids_(std::move(ids)),
          primary_id_(std::move(p))
    {
    }
```

```
    ProjectState(const ProjectState&) = delete;
    ProjectState(ProjectState&&) noexcept = default;
    ProjectState& operator=(const ProjectState&) = delete;
    ProjectState& operator=(ProjectState&&) noexcept = default;
    ~ProjectState() = default;
};
```

This design is concise, expressive, and correct. It is also typically more robust than manual pointer ownership code.

4.2.2 Exception guarantees

Exception behavior is one of the most important parts of implementing move constructors correctly. A move constructor can be throwing or non-throwing, but in most value-oriented designs, non-throwing move is strongly preferred.

Why `noexcept` matters

The C++ Core Guidelines explicitly recommend making move operations `noexcept`. The reason is both semantic and practical:

- programmers usually expect move to be safe and lightweight,
- standard-library facilities can use non-throwing move operations more aggressively and efficiently,
- exception-safe container operations often depend on whether move can throw.

This is why a well-designed move constructor for a resource-owning type is commonly written like this:

```
class FileHandle
{
    void* native_handle_;

public:
    FileHandle() noexcept : native_handle_(nullptr) {}

    FileHandle(FileHandle&& other) noexcept
        : native_handle_(other.native_handle_)
    {
        other.native_handle_ = nullptr;
    }
};
```

If the move operation simply transfers a handle or pointer, there is usually no reason for it to throw.

Defaulted move constructors and implied exception specification

A defaulted or implicitly-declared move constructor receives an implied exception specification from the operations required to move its bases and members. In practice, this means that the move constructor is non-throwing only if the move construction of the relevant subobjects is also non-throwing.
So, when you write:

```
class Session
{
    std::string name_;
    std::unique_ptr<int> token_;
```

```
public:
    Session(Session&&) = default;
};
```

the resulting exception specification depends on the move behavior of `std::string` and `std::unique_ptr<int>` as used by the implementation and language rules.
When you know your class should be non-throwing to move, it is often best to state that explicitly:

```
class Session
{
    std::string name_;
    std::unique_ptr<int> token_;

public:
    Session() = default;
    Session(const Session&) = delete;
    Session(Session&&) noexcept = default;
    Session& operator=(const Session&) = delete;
    Session& operator=(Session&&) noexcept = default;
};
```

Strong guarantee versus basic guarantee

Move constructors are constructors, so they create a new object rather than mutate an existing one. Their exception story is therefore simpler than move assignment, but it is still useful to reason about guarantees.

Non-throwing move constructor If a move constructor is truly non-throwing, then object transfer into a new destination is straightforward and highly compatible with standard-library relocation strategies.

Potentially-throwing move constructor If a move constructor might throw, then code using the type may need fallback strategies. In standard-library containers and generic components, this can influence whether copying is preferred over moving in some operations, because the library may need to preserve stronger guarantees.

A move constructor that can throw

Here is an example where move may throw because it performs allocation:

```
#include <string>
#include <stdexcept>

class ReencodedText
{
    std::string text_;

public:
    ReencodedText() = default;

    explicit ReencodedText(std::string text)
        : text_(std::move(text))
    {
    }

    ReencodedText(ReencodedText&& other)
        : text_("moved:" + other.text_)
    {
        other.text_.clear();
    }
};
```

This constructor performs string concatenation during the move, which may allocate memory and therefore may throw. It is a poor move design because it changes the moved value in a non-transfer-like manner and weakens exception behavior unnecessarily.
A better move constructor is usually the plain transfer:

```
#include <string>
#include <utility>

class PlainText
{
    std::string text_;

public:
    PlainText() = default;

    explicit PlainText(std::string text)
        : text_(std::move(text))
    {
    }

    PlainText(const PlainText&) = default;
    PlainText(PlainText&&) noexcept = default;
    PlainText& operator=(const PlainText&) = default;
    PlainText& operator=(PlainText&&) noexcept = default;
    ~PlainText() = default;
};
```

Exception-safe hand-written move constructor

If you must implement a custom move constructor, the constructor should ideally perform only operations that cannot fail. Pointer transfer, integer transfer, size transfer, handle transfer, and null-reset are the typical building blocks.

```
#include <utility>
#include <cstddef>

class Packet
{
    unsigned char* bytes_;
    std::size_t size_;

public:
    Packet() noexcept
        : bytes_(nullptr), size_(0)
    {
    }

    explicit Packet(std::size_t n)
        : bytes_(new unsigned char[n]{}), size_(n)
    {
    }

    ~Packet()
    {
        delete[] bytes_;
    }
```

```
    Packet(const Packet&) = delete;
    Packet& operator=(const Packet&) = delete;

    Packet(Packet&& other) noexcept
        : bytes_(other.bytes_), size_(other.size_)
    {
        other.bytes_ = nullptr;
        other.size_ = 0;
    }

    Packet& operator=(Packet&& other) noexcept
    {
        if (this != &other)
        {
            delete[] bytes_;
            bytes_ = other.bytes_;
            size_ = other.size_;
            other.bytes_ = nullptr;
            other.size_ = 0;
        }
        return *this;
    }
};
```

This is a strong model for custom resource-owning move support. It is direct, non-throwing, and preserves correctness.

Using type traits to reason about move guarantees

Modern generic code often needs to know whether a type is nothrow-move-constructible. This can be tested using standard type traits:

```
#include <type_traits>
#include <string>
#include <memory>

struct A
{
    std::string s;
    A(A&&) noexcept = default;
};

struct B
{
    std::unique_ptr<int> p;
    B(B&&) noexcept = default;
};

static_assert(std::is_nothrow_move_constructible_v<A>);
static_assert(std::is_nothrow_move_constructible_v<B>);
```

This is useful in template libraries, container-like abstractions, and custom frameworks.

Do not promise `noexcept` incorrectly

Marking a move constructor `noexcept` when it can actually throw is a serious design error. If such a function throws, the program may terminate rather than propagate the exception normally.

Therefore, the rule is simple:

> Use `noexcept` when the move constructor truly cannot fail according to the operations it performs.

4.2.3 Move-only types

A move-only type is a type that supports move construction and move assignment, but suppresses copying. These types are essential in Modern C++ because some forms of ownership and some resources cannot be meaningfully duplicated.

Why some types must be move-only

Not every resource has copy semantics. If a class represents unique ownership of a file descriptor, socket, process handle, mutex lock, GPU object, or heap allocation that must have exactly one owner, copying may be nonsensical or dangerous. In these cases, transfer of ownership is meaningful, but duplication is not.
The standard example is `std::unique_ptr`. It is movable but not copyable.

Canonical move-only design

The standard pattern is:

- default constructor as needed,
- deleted copy constructor,
- deleted copy assignment operator,
- defaulted or custom move constructor,
- defaulted or custom move assignment operator,

- suitable destructor.

A minimal move-only handle type looks like this:

```
class UniqueHandle
{
    int handle_;

public:
    UniqueHandle() noexcept
        : handle_(-1)
    {
    }

    explicit UniqueHandle(int h) noexcept
        : handle_(h)
    {
    }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept
        : handle_(other.handle_)
    {
        other.handle_ = -1;
    }

    UniqueHandle& operator=(UniqueHandle&& other) noexcept
    {
```

```
        if (this != &other)
        {
            release();
            handle_ = other.handle_;
            other.handle_ = -1;
        }
        return *this;
    }

    ~UniqueHandle()
    {
        release();
    }

private:
    void release() noexcept
    {
        if (handle_ != -1)
        {
            handle_ = -1;
        }
    }
};
```

Move-only with standard smart pointers

In real applications, move-only design is often as simple as holding a `std::unique_ptr` member. The type then naturally becomes move-only if copying is deleted or unavailable.

```
#include <memory>
```

```
#include <string>
#include <utility>

class Document
{
    std::string title_;
    std::unique_ptr<std::string> text_;

public:
    Document(std::string title, std::string text)
        : title_(std::move(title)),
          text_(std::make_unique<std::string>(std::move(text)))
    {
    }

    Document(const Document&) = delete;
    Document& operator=(const Document&) = delete;
    Document(Document&&) noexcept = default;
    Document& operator=(Document&&) noexcept = default;
    ~Document() = default;

    const std::string& title() const noexcept
    {
        return title_;
    }

    const std::string& text() const noexcept
    {
        return *text_;
```

```
    }
};
```

This is the preferred Modern C++ style. Ownership is explicit. Copying is intentionally disallowed. Movement is efficient and safe.

Passing move-only types

Move-only types are commonly passed in one of two ways:

1. by rvalue reference when the function needs a direct sink interface,
2. by value when transferring ownership into the function is the main goal and one extra move is acceptable.

Example using a sink by value:

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

void consume(std::unique_ptr<int> p)
{
    if (p)
        std::cout << *p << '\n';
}

int main()
{
    auto ptr = std::make_unique<int>(42);
    consume(std::move(ptr));
}
```

This is a very idiomatic Modern C++ pattern for move-only ownership transfer.

Move-only polymorphic bases

Public copy and move support is often undesirable in polymorphic base classes because of slicing concerns. A polymorphic base frequently suppresses public copy and move operations entirely and provides cloning through virtual interfaces when deep copying is needed.

```
#include <memory>

class Node
{
public:
    Node() = default;
    Node(const Node&) = delete;
    Node& operator=(const Node&) = delete;
    Node(Node&&) = delete;
    Node& operator=(Node&&) = delete;
    virtual ~Node() = default;

    virtual std::unique_ptr<Node> clone() const = 0;
};
```

This prevents accidental slicing and expresses the intended ownership model more clearly.

A move-only type with invariant-rich state

The following example shows a more realistic move-only RAII class:

```
#include <string>
#include <utility>
#include <iostream>
```

```
class Logger
{
    int fd_;
    std::string path_;

public:
    Logger() noexcept
        : fd_(-1), path_()
    {
    }

    Logger(int fd, std::string path)
        : fd_(fd), path_(std::move(path))
    {
    }

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

    Logger(Logger&& other) noexcept
        : fd_(other.fd_), path_(std::move(other.path_))
    {
        other.fd_ = -1;
    }

    Logger& operator=(Logger&& other) noexcept
    {
        if (this != &other)
        {
```

```
            close_if_needed();
            fd_ = other.fd_;
            path_ = std::move(other.path_);
            other.fd_ = -1;
        }
        return *this;
    }

    ~Logger()
    {
        close_if_needed();
    }

    bool valid() const noexcept
    {
        return fd_ != -1;
    }

private:
    void close_if_needed() noexcept
    {
        if (fd_ != -1)
        {
            fd_ = -1;
        }
    }
};
```

This class transfers both the low-level handle and the descriptive metadata. The moved-from object remains destructible and assignable.

Move-only and containers

Move-only types can be stored in standard containers as long as the required operations are available. This is one of the major strengths of Modern C++ design.

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

int main()
{
    std::vector<std::unique_ptr<int>> values;

    values.push_back(std::make_unique<int>(10));
    values.push_back(std::make_unique<int>(20));
    values.emplace_back(std::make_unique<int>(30));
}
```

This works because `std::unique_ptr` is movable. Before move semantics, such ownership-oriented container patterns were far less natural.

4.2.4 Design checklist for move constructors

When implementing or reviewing a move constructor, the following checklist is useful:

1. Does the class truly need a custom move constructor, or would `= default` be better?
2. Does the move operation transfer ownership rather than duplicate resources?
3. Is the moved-from object left valid and safe to destroy?
4. Are all class invariants preserved after the transfer?
5. Is the move constructor correctly marked `noexcept` when it truly cannot throw?

6. If the type is intended to be move-only, are copy operations explicitly deleted?
7. If the type is polymorphic, should public copy and move be suppressed to avoid slicing?
8. Are raw resources wrapped where possible in standard library types to simplify correctness?

4.2.5 Extensive examples

Example 1: defaulted move constructor is the best design

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

class BuildArtifact
{
    std::string name_;
    std::vector<unsigned char> bytes_;
    std::unique_ptr<int> version_;

public:
    BuildArtifact() = default;

    BuildArtifact(std::string n, std::vector<unsigned char> b, std::unique_ptr<int>
    ↪  v)
        : name_(std::move(n)),
          bytes_(std::move(b)),
          version_(std::move(v))
    {
    }
```

```
    BuildArtifact(const BuildArtifact&) = delete;
    BuildArtifact& operator=(const BuildArtifact&) = delete;
    BuildArtifact(BuildArtifact&&) noexcept = default;
    BuildArtifact& operator=(BuildArtifact&&) noexcept = default;
    ~BuildArtifact() = default;
};
```

Example 2: hand-written move constructor for raw ownership

```
#include <cstddef>
#include <algorithm>

class RawImage
{
    std::size_t width_;
    std::size_t height_;
    unsigned char* pixels_;

public:
    RawImage() noexcept
        : width_(0), height_(0), pixels_(nullptr)
    {
    }

    RawImage(std::size_t w, std::size_t h)
        : width_(w), height_(h), pixels_(new unsigned char[w * h]{})
    {
    }
```

```
~RawImage()
{
    delete[] pixels_;
}

RawImage(const RawImage& other)
    : width_(other.width_),
      height_(other.height_),
      pixels_(new unsigned char[other.width_ * other.height_])
{
    std::copy(other.pixels_,
              other.pixels_ + (width_ * height_),
              pixels_);
}

RawImage& operator=(const RawImage& other)
{
    if (this != &other)
    {
        unsigned char* new_pixels = new unsigned char[other.width_ *
        ↪  other.height_];
        std::copy(other.pixels_,
                  other.pixels_ + (other.width_ * other.height_),
                  new_pixels);

        delete[] pixels_;
        pixels_ = new_pixels;
        width_ = other.width_;
        height_ = other.height_;
```

```
    }
    return *this;
}

RawImage(RawImage&& other) noexcept
    : width_(other.width_),
      height_(other.height_),
      pixels_(other.pixels_)
{
    other.width_ = 0;
    other.height_ = 0;
    other.pixels_ = nullptr;
}

RawImage& operator=(RawImage&& other) noexcept
{
    if (this != &other)
    {
        delete[] pixels_;
        width_ = other.width_;
        height_ = other.height_;
        pixels_ = other.pixels_;

        other.width_ = 0;
        other.height_ = 0;
        other.pixels_ = nullptr;
    }
    return *this;
}
```

```
};
```

Example 3: move-only task object

```
#include <memory>
#include <string>
#include <utility>

class Task
{
    std::string title_;
    std::unique_ptr<int> priority_;

public:
    Task(std::string title, int priority)
        : title_(std::move(title)),
          priority_(std::make_unique<int>(priority))
    {
    }

    Task(const Task&) = delete;
    Task& operator=(const Task&) = delete;
    Task(Task&&) noexcept = default;
    Task& operator=(Task&&) noexcept = default;
    ~Task() = default;
};
```

Example 4: using a move-only type in a container

```
#include <vector>
```

```
#include <memory>
#include <string>

class Job
{
    std::string name_;
    std::unique_ptr<int> id_;

public:
    Job(std::string name, int id)
        : name_(std::move(name)),
          id_(std::make_unique<int>(id))
    {
    }

    Job(const Job&) = delete;
    Job& operator=(const Job&) = delete;
    Job(Job&&) noexcept = default;
    Job& operator=(Job&&) noexcept = default;
};

int main()
{
    std::vector<Job> jobs;
    jobs.emplace_back("compile", 1);
    jobs.emplace_back("link", 2);
    jobs.emplace_back("package", 3);
}
```

4.2.6 Windows build and test notes

The following commands are suitable for testing the examples in Windows environments.

MSVC command line

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo example.cpp
cl /std:c++23 /EHsc /W4 /O2 /permissive- /nologo example.cpp
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

GCC on Windows via MSYS2 or MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

Recommended warning mindset

For move-constructor work in production code on Windows, it is wise to compile with strong warnings enabled and test both normal and optimized builds. Debug builds sometimes make object lifetime and constructor traffic easier to observe, while optimized builds better reflect the performance intent of move-aware code.

4.2.7 Concluding design rules

The best implementation strategy for move constructors in Modern C++ can be summarized as follows:

1. Use `= default` whenever the class is composed of well-behaved movable bases and members.

2. Write a custom move constructor only when the type directly manages raw resources or needs explicit invariant repair.
3. Always leave the moved-from object valid.
4. Prefer non-throwing move operations; use `noexcept` when it is truthful.
5. For unique ownership, make the type move-only by deleting copy operations.
6. For polymorphic base classes, suppress public copy and move to avoid slicing unless a very deliberate design requires otherwise.
7. Prefer standard movable members such as `std::string`, `std::vector`, and `std::unique_ptr` instead of manual resource management whenever possible.

These rules form the practical foundation for advanced topics such as perfect forwarding, sink parameters, allocator-aware design, container relocation behavior, and large-scale lifetime-oriented architecture in Modern C++.

4.3 Perfect Forwarding Theory

Perfect forwarding is one of the most important language techniques introduced with C++11 and refined in later standards. It enables a function template to pass its arguments to another function while preserving the original value category and qualifiers of the arguments.
Without perfect forwarding, wrapper functions, adapter classes, container factories, and generic libraries would introduce unnecessary copies or incorrect overload resolution. Perfect forwarding ensures that the forwarded arguments behave exactly as if the caller had invoked the final function directly.
The mechanism of perfect forwarding relies on three essential components:

1. rvalue references

2. reference collapsing rules
3. the forwarding utility `std::forward`

These features allow templates to preserve the value category of arguments so that lvalues remain lvalues and rvalues remain rvalues when forwarded. This capability is crucial for efficient generic programming and for the implementation of container operations such as `emplace_back`.

4.3.1 Reference collapsing

Reference collapsing rules define how the compiler resolves references to references when templates and type deduction create such combinations.
C++ normally does not allow references to references in ordinary declarations. However, template substitution may temporarily produce such combinations. The language therefore specifies deterministic rules that collapse these combinations into a single valid reference type. The reference collapsing rules are:

1. `T&` `&` becomes `T&`
2. `T&` `&&` becomes `T&`
3. `T&&` `&` becomes `T&`
4. `T&&` `&&` becomes `T&&`

In practice this rule can be summarized in a simple principle:

> If either reference is an lvalue reference, the result is an lvalue reference. Only rvalue reference combined with rvalue reference remains an rvalue reference.

These rules exist primarily to make perfect forwarding possible in templates.

Example demonstrating reference collapsing

```
#include <iostream>

template<typename T>
void analyze(T&& value)
{
    std::cout << "Function called\n";
}

int main()
{
    int x = 10;

    analyze(x);       // lvalue argument
    analyze(20);      // rvalue argument
}
```

When the function is called with `x`, template deduction deduces:

```
T = int&
```

The parameter type becomes:

```
T&& -> int& &&
```

After reference collapsing:

```
int&
```

Thus the parameter becomes an lvalue reference.
When called with `20`, the deduced type is:

```
T = int
```

The parameter becomes:

```
int&&
```

No collapsing occurs.
This mechanism enables templates to correctly represent both lvalues and rvalues.

4.3.2 Universal / forwarding references

A forwarding reference (formerly called a universal reference) occurs when a function template parameter has the form:

```
T&&
```

where `T` is a template parameter deduced from the call site.
In this context the reference behaves differently from a normal rvalue reference. Instead of requiring an rvalue argument, the parameter can bind to either lvalues or rvalues.
Forwarding references preserve the value category of the argument so that it can later be forwarded correctly using `std::forward`.

Forwarding reference behavior

```
#include <iostream>
#include <utility>

void process(int& x)
{
    std::cout << "lvalue overload\n";
}
```

```
void process(int&& x)
{
    std::cout << "rvalue overload\n";
}

template<typename T>
void wrapper(T&& arg)
{
    process(std::forward<T>(arg));
}

int main()
{
    int x = 10;

    wrapper(x);      // calls lvalue overload
    wrapper(20);     // calls rvalue overload
}
```

Here:

- `wrapper(x)` forwards an lvalue
- `wrapper(20)` forwards an rvalue

The function `std::forward` restores the correct value category of the original argument.

Why std::forward is required

Even when the parameter type is `T&&`, the variable `arg` inside the function is always an lvalue expression because it has a name.

Without forwarding:

```
process(arg);
```

the lvalue overload would always be selected.
Using `std::forward` restores the original category:

```
process(std::forward<T>(arg));
```

4.3.3 Forwarding constructors

Forwarding constructors are constructors that use forwarding references to pass arguments directly to member objects or base classes.
This technique avoids unnecessary temporary objects and preserves constructor overload selection.
A classical example appears in wrapper classes.

Forwarding constructor example

```
#include <string>
#include <utility>

class Person
{
    std::string name_;

public:

    template<typename T>
    explicit Person(T&& name)
        : name_(std::forward<T>(name))
```

```
    {
    }
};
```

This constructor allows both of the following:

```
std::string s = "Alice";

Person p1(s);          // copy
Person p2("Bob");      // move
```

The correct constructor of `std::string` will be selected automatically.

Perfect forwarding with multiple parameters

Variadic templates allow forwarding of multiple arguments simultaneously.

```
#include <utility>
#include <string>

class Record
{
    std::string name_;
    int id_;

public:

    template<typename T1, typename T2>
    Record(T1&& name, T2&& id)
        : name_(std::forward<T1>(name)),
          id_(std::forward<T2>(id))
```

```
    {
    }
};
```

This constructor preserves the value category of each parameter independently.

Avoiding accidental constructor hijacking

Forwarding constructors can sometimes interfere with copy or move constructors.
A common defensive technique is to explicitly declare the special member functions:

```
Person(const Person&) = default;
Person(Person&&) = default;
```

This ensures correct behavior when copying or moving objects of the same type.

4.3.4 Factory pattern with forwarding

Perfect forwarding is heavily used in factory functions and generic object construction.
A factory function constructs objects without knowing their exact constructor signature.
Forwarding allows it to pass arbitrary arguments to the final constructor.

Generic factory function

```
#include <memory>
#include <utility>

template<typename T, typename... Args>
std::unique_ptr<T> make_object(Args&&... args)
{
    return std::unique_ptr<T>(
```

```
        new T(std::forward<Args>(args)...)
    );
}
```

Usage:

```
class Widget
{
public:
    Widget(int x, double y) {}
};

auto w = make_object<Widget>(10, 3.5);
```

This pattern preserves the exact argument types and categories when constructing `Widget`.

Relation to std::make_unique and std::make_shared

Standard library factory functions such as:

```
std::make_unique
std::make_shared
```

are implemented using this same perfect forwarding technique. They forward arguments to the constructor of the target type, eliminating redundant temporary objects and ensuring efficient initialization.

4.3.5 Perfect forwarding with variadic templates

Variadic templates make it possible to forward an arbitrary number of arguments.

```cpp
#include <vector>
#include <utility>

template<typename T, typename... Args>
void emplace_back_wrapper(std::vector<T>& v, Args&&... args)
{
    v.emplace_back(std::forward<Args>(args)...);
}
```

This mirrors how standard containers implement `emplace` operations.

4.3.6 Common pitfalls

Perfect forwarding is powerful but requires careful usage.

Forgetting std::forward

```cpp
func(arg);    // always lvalue
```

This loses the value category and prevents move semantics.

Using std::move instead of std::forward

```cpp
func(std::move(arg));
```

This forces the argument to be treated as an rvalue even if it was originally an lvalue.
Correct forwarding requires:

```cpp
func(std::forward<T>(arg));
```

Forwarding references only exist in templates

A type such as:

```
void f(int&& x);
```

is not a forwarding reference. It is a pure rvalue reference.
Forwarding references occur only when:

- the parameter is `T&&`
- `T` is deduced from the call site

4.3.7 Design guidelines

Effective usage of perfect forwarding follows several practical rules:

1. Use forwarding references primarily in template wrappers and factories.
2. Always pair forwarding references with `std::forward`.
3. Avoid unnecessary forwarding constructors in ordinary classes.
4. Prefer simple constructors when perfect forwarding is not required.
5. Ensure copy and move constructors are still declared explicitly when forwarding constructors exist.

4.3.8 Summary

Perfect forwarding enables template functions to propagate arguments without altering their value category. This mechanism relies on reference collapsing rules, forwarding references, and the forwarding utility `std::forward`. These features allow generic libraries and containers

to construct objects efficiently, eliminate redundant copies, and preserve correct overload resolution during forwarding operations.

Perfect forwarding is therefore a foundational technique for modern C++ template design, enabling the implementation of generic factories, container emplacement operations, and highly efficient abstraction layers.

Ownership Models & Lifetime Control

5.1 Raw Ownership

Raw ownership is one of the most delicate areas in Modern C++. It lies at the boundary between low-level expressiveness and lifetime risk. The language allows direct manipulation of memory addresses, object lifetimes, buffer ranges, and resource handles, but that power also makes it easy to create leaks, double deletion, dangling pointers, invalid iterators, and subtle aliasing errors.
Modern C++ design does not forbid raw pointers. Instead, it distinguishes carefully between:

1. raw pointers and references used as **non-owning access paths**,
2. explicit ownership handles such as `std::unique_ptr` and `std::shared_ptr`,
3. view types such as `std::span` and `std::string_view`,
4. range-based borrowing models where iterators may safely outlive a particular range object.

The central modern rule is simple:

> A raw pointer is usually an observer, not an owner.

This rule is aligned with current C++ Core Guidelines design guidance. A raw `T*` is treated as non-owning by default, and a raw reference `T&` is also non-owning. Ownership, when present, should be made explicit through resource handles or clearly marked legacy conventions. This separation is the foundation of safe lifetime design in large Modern C++ systems.

5.1.1 Manual ownership design

Manual ownership design refers to code in which the programmer directly manages lifetime responsibilities instead of delegating them to standard RAII handles. This appears most commonly in legacy code, low-level systems interfaces, custom allocators, embedded code, C interoperability layers, and hand-written resource wrappers.

What ownership means

An owning object is the object responsible for releasing a resource. That resource may be:

- dynamically allocated memory,
- a file handle,
- a socket,
- a mutex-like native handle,
- a graphics or OS handle,
- a buffer returned by a C API,
- memory mapped storage,
- a custom runtime resource.

If ownership is ambiguous, bugs become likely:

- nobody deletes the object and memory leaks,
- multiple parties delete it and a double free occurs,
- a borrower continues using it after destruction,
- an API silently transfers ownership without documenting it.

This is why modern guidance strongly discourages expressing transfer of ownership through plain `T*` or `T&`. Ownership should be explicit.

Raw pointers are not resource handles

A raw pointer can point to an object, but by itself it does not encode who owns that object, when it must be destroyed, or whether it may be null. A raw reference similarly does not encode ownership. It only guarantees binding to an object.
This distinction is extremely important in interface design:

- `T*` usually means optional non-owning access,
- `T&` usually means required non-owning access,
- `const T*` usually means optional read-only non-owning access,
- `const T&` usually means required read-only non-owning access.

In new code, if ownership is intended, a smart pointer or value-returning design is usually preferable.

Bad manual ownership example

```
class Widget
{
public:
    Widget() = default;
};

Widget* make_widget()
{
    Widget* p = new Widget{};
    return p;
}
```

This interface is dangerous because the caller cannot know ownership intent by reading the type alone. Is the caller supposed to delete the returned pointer? Is ownership shared? Is it cached internally? The interface is underspecified.

Better ownership designs

The first preference is often returning by value:

```
class Widget
{
public:
    Widget() = default;
};

Widget make_widget()
{
    return Widget{};
```

```
}
```

If true dynamic ownership transfer is required, use `std::unique_ptr`:

```
#include <memory>

class Widget
{
public:
    Widget() = default;
};

std::unique_ptr<Widget> make_widget()
{
    return std::make_unique<Widget>();
}
```

This version communicates exclusive ownership transfer clearly and safely.

Legacy manual ownership

There are still environments where raw ownership cannot be avoided:

- ABI-stable interfaces,
- C-compatible APIs,
- operating-system callbacks,
- constrained runtimes,
- existing code bases too large to refactor immediately.

In such environments, ownership must be documented with precision and encapsulated aggressively.

A classical manual RAII wrapper looks like this:

```
#include <utility>

class Buffer
{
    int* data_;
    std::size_t size_;

public:
    explicit Buffer(std::size_t n)
        : data_(new int[n]{}), size_(n)
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }

    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;

    Buffer(Buffer&& other) noexcept
        : data_(other.data_), size_(other.size_)
    {
        other.data_ = nullptr;
        other.size_ = 0;
```

```
}

Buffer& operator=(Buffer&& other) noexcept
{
    if (this != &other)
    {
        delete[] data_;
        data_ = other.data_;
        size_ = other.size_;
        other.data_ = nullptr;
        other.size_ = 0;
    }
    return *this;
}

int* data() noexcept
{
    return data_;
}

const int* data() const noexcept
{
    return data_;
}

std::size_t size() const noexcept
{
    return size_;
}
```

```
};
```

This is manual ownership done responsibly. The class owns the resource, releases it in the destructor, disables copying, and supports move transfer.

When a raw pointer member is suspicious

A class that contains a raw pointer member must be reviewed carefully. The key question is:

> Does this pointer own, observe, or merely cache access to another object?

Consider:

```
class LegacyImage
{
    unsigned char* pixels_;
    std::size_t width_;
    std::size_t height_;
};
```

This class is incomplete from an ownership perspective. If `pixels_` owns memory, then destructor, copy control, and move support must be defined appropriately. If it merely observes external memory, then destruction must not free it. The difference is semantic, architectural, and safety-critical.

Manual ownership checklist

When raw ownership is unavoidable, verify all of the following:

1. exactly one component owns the resource at a time,
2. the owner releases it in all normal and exceptional paths,

3. all non-owners are clearly documented as borrowers or observers,
4. copy operations are either correctly deep-copying or explicitly deleted,
5. move operations transfer ownership cleanly,
6. null state and moved-from state remain valid,
7. returned pointers and references never outlive the underlying object.

5.1.2 Borrow semantics

Borrow semantics describe temporary access to an object, resource, or range without transferring ownership. A borrower uses something that must remain alive elsewhere. This is one of the deepest conceptual distinctions in Modern C++:

> Ownership controls destruction. Borrowing only grants access.

Borrowing with pointers and references

The simplest borrow forms are raw pointers and references.

```
#include <iostream>

class Logger
{
public:
    void write(const char* text) const
    {
        std::cout << text << '\n';
    }
};
```

```
void print_message(const Logger& log)
{
    log.write("hello");
}
```

The parameter `const Logger&` is a borrowed access path. The function does not own the logger and must not extend its lifetime beyond the call.
Likewise:

```
void touch(Logger* log)
{
    if (log)
        log->write("touched");
}
```

The pointer is also borrowed and non-owning.

Borrowing means lifetime dependency

Borrowing is safe only if the owner outlives the borrower. This is the essential lifetime rule.
A classic dangling bug:

```
const char* bad_text()
{
    std::string s = "temporary";
    return s.c_str();
}
```

The returned pointer borrows storage owned by `s`. Once the function returns, `s` is destroyed, and the pointer dangles.
A better design returns ownership or value:

```
#include <string>

std::string good_text()
{
    std::string s = "safe";
    return s;
}
```

Borrowing in the ranges model

Modern C++ formalizes borrowing in the ranges library with the concept `std::ranges::borrowed_range`. The standard definition states that a type models `borrowed_range` only when iterator validity is not tied to the lifetime of the range object itself. This is a major language-library idea because it separates two cases:

1. ranges whose iterators become invalid when the range object dies,
2. ranges whose iterators are borrowed from some deeper source and remain valid independently of the wrapper object.

In practical terms, this allows generic code to return iterators from certain range parameters without immediately creating dangling iterators.

Conceptual borrow example

```
#include <ranges>
#include <span>
#include <vector>

template<std::ranges::borrowed_range R>
auto first_iterator(R&& r)
```

```
{
    return std::ranges::begin(r);
}

int main()
{
    std::vector<int> v{1, 2, 3, 4};
    auto it = first_iterator(std::span<int>{v});
}
```

This illustrates the conceptual role of borrowed ranges. A view such as `std::span` borrows access to an underlying buffer and does not own the buffer itself.

Borrowing and function parameters

A function that merely reads or mutates an object transiently should usually accept a borrowed form:

- `T&` or `const T&` for single required objects,
- `T*` or `const T*` for nullable objects,
- `std::span<T>` for contiguous sequences,
- `std::string_view` for read-only text sequences,
- iterator pairs or ranges for generic traversal.

Examples:

```
#include <span>
```

```
void scale(std::span<int> values, int factor)
{
    for (int& v : values)
        v *= factor;
}
```

```
#include <string_view>
#include <iostream>

void print_name(std::string_view name)
{
    std::cout << name << '\n';
}
```

In both cases the function borrows access. It does not own the sequence or text.

Borrowing must not silently escape

A borrower should not store a borrowed address, pointer, reference, span, or view beyond the lifetime it can guarantee.
Dangerous design:

```
#include <string_view>

class BadStore
{
    std::string_view view_;

public:
    void set(std::string_view v)
    {
```

```
        view_ = v;
    }

    std::string_view get() const
    {
        return view_;
    }
};
```

This class may be correct only if all callers ensure the original character storage outlives the `BadStore` object. That is often too fragile.

Safer design when persistence is needed:

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

class GoodStore
{
    std::string text_;

public:
    void set(std::string_view v)
    {
        text_ = v;
    }

    std::string_view get() const noexcept
    {
        return text_;
    }
```

```
};
```

Here the class owns the string data and may safely expose borrowed read-only views of its own storage.

5.1.3 Non-owning views

Non-owning views are lightweight objects that describe access to existing data without controlling that data's lifetime. They are among the most important abstractions in Modern C++ because they combine efficiency, expressiveness, and clarity.
Current standard and Microsoft documentation both characterize views as lightweight, non-owning access types. In particular:

- `std::span` is a lightweight view over a contiguous sequence,
- `std::string_view` is a read-only non-owning handle to character data,
- ranges views generally refer to elements they do not own, except for specialized owning wrappers.

std::span as a view

`std::span<T>` represents a view over a contiguous sequence of `T`. It does not own the elements. It simply stores pointer-and-extent style access in a safer and more expressive type.

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

void print_values(std::span<const int> values)
{
```

```
    for (int v : values)
        std::cout << v << ' ';
    std::cout << '\n';
}

int main()
{
    std::vector<int> data{10, 20, 30, 40};
    print_values(data);
}
```

This function does not care whether the caller provides a built-in array, `std::array`, or `std::vector`. It borrows contiguous access only.

Why span is better than pointer plus size

Traditional pointer-plus-size interfaces are easy to misuse:

```
void process(const int* p, std::size_t n);
```

This style separates the buffer and its extent, making mismatches possible.
A view-based interface is clearer:

```
#include <span>

void process(std::span<const int> values);
```

This encodes the sequence as one parameter and carries extent information with it.

std::string_view as a read-only text view

`std::string_view` is a non-owning read-only view of character data. It avoids copies and can accept many string-like sources.

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

void greet(std::string_view name)
{
    std::cout << "Hello, " << name << '\n';
}

int main()
{
    std::string s = "Ayman";
    greet(s);
    greet("World");
}
```

This interface is efficient and flexible because it borrows text rather than owning it.

The lifetime trap of string_view

Because `std::string_view` does not own characters, it becomes dangerous when bound to storage that dies too early.
Bad example:

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

```
std::string_view broken()
{
    std::string s = "temporary";
    return s;
}
```

This returns a view into destroyed storage.

Another dangerous pattern:

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

int main()
{
    std::string_view v = std::string("danger");
}
```

The temporary `std::string` dies at the end of the full expression, leaving `v` dangling.

Safer alternative:

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

int main()
{
    std::string s = "safe";
    std::string_view v = s;
}
```

Now the owner `s` remains alive while `v` is used.

Views are cheap to copy

One important design property of views is that they are lightweight and inexpensive to copy. For this reason, a view type is usually passed and returned by value, not by reference.
Good style:

```
#include <span>

std::span<const int> head(std::span<const int> values)
{
    return values.first(3);
}
```

Less useful style:

```
#include <span>

const std::span<const int>& bad_head(const std::span<const int>& values)
{
    return values;
}
```

View types are already reference-like. Adding another layer of reference often complicates lifetime reasoning without performance benefit.

Views do not justify ownership assumptions

A span or string view parameter does not mean the callee owns or may retain the data. It means the callee has borrowed access for the duration of the valid lifetime.
This distinction is essential in class design.
Bad persistent member unless lifetime is externally guaranteed:

```
#include <span>

class Reader
{
    std::span<const unsigned char> bytes_;

public:
    explicit Reader(std::span<const unsigned char> b)
        : bytes_(b)
    {
    }
};
```

This may be valid in tightly controlled code, but it requires that the underlying buffer outlive the `Reader`. If long-term storage is needed, ownership should be taken instead.
Safer owning version:

```
#include <vector>
#include <span>

class Reader
{
    std::vector<unsigned char> bytes_;

public:
    explicit Reader(std::span<const unsigned char> b)
        : bytes_(b.begin(), b.end())
    {
    }
```

```
    std::span<const unsigned char> view() const noexcept
    {
        return bytes_;
    }
};
```

This version owns the data and may safely expose a borrowed view of its own storage.

Custom non-owning views

You can build custom view-like types, but they must obey the same lifetime discipline: they may describe data, but they must not pretend to own it.

```
#include <cstddef>

template<typename T>
class ArrayView
{
    T* ptr_;
    std::size_t count_;

public:
    ArrayView() noexcept : ptr_(nullptr), count_(0) {}

    ArrayView(T* ptr, std::size_t count) noexcept
        : ptr_(ptr), count_(count)
    {
    }

    T* begin() const noexcept { return ptr_; }
```

```
    T* end() const noexcept { return ptr_ + count_; }
    std::size_t size() const noexcept { return count_; }
    T& operator[](std::size_t i) const noexcept { return ptr_[i]; }
};
```

This kind of type is useful in low-level libraries, but the programmer must still guarantee that the underlying buffer remains valid.

Owning versus viewing in one design

A common high-quality pattern is:

1. own data internally with `std::string`, `std::vector`, `std::array`, or smart pointers,
2. expose borrowed views outward with `std::string_view` or `std::span`.

Example:

```
#include <vector>
#include <span>

class PacketStorage
{
    std::vector<unsigned char> bytes_;

public:
    explicit PacketStorage(std::vector<unsigned char> bytes)
        : bytes_(std::move(bytes))
    {
    }
```

```
    std::span<const unsigned char> bytes() const noexcept
    {
        return bytes_;
    }
};
```

This design cleanly separates ownership from observation.

5.1.4 Advanced raw lifetime pitfalls

Dangling from local variables

```
int* bad_pointer()
{
    int x = 42;
    return &x;
}
```

The returned pointer refers to a dead stack object.

Dangling view from temporary

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

std::string_view bad_view()
{
    return std::string("abc");
}
```

The view outlives the temporary string.

Storing borrow beyond call boundary

```
#include <string_view>

class Config
{
    std::string_view name_;

public:
    void set_name(std::string_view s)
    {
        name_ = s;
    }
};
```

This may dangle unless callers guarantee that the storage outlives the `Config` object.

Mismatched delete forms

```
int* p = new int[10];
delete p;
```

This is incorrect because array allocation requires `delete[]`.

Double delete after shallow copy

```
class BadOwner
{
    int* p_;

public:
```

```
    explicit BadOwner(int* p) : p_(p) {}
    ~BadOwner() { delete p_; }
};
```

If copied implicitly, both copies will destroy the same pointer.

5.1.5 Practical design rules

The following rules provide a robust modern discipline for raw ownership and borrowed access:

1. Treat raw pointers as non-owning by default.
2. Treat references as non-owning by default.
3. Express exclusive ownership with `std::unique_ptr` or value types when possible.
4. Use `std::shared_ptr` only when shared lifetime is truly required.
5. Use `std::span` for contiguous borrowed sequences.
6. Use `std::string_view` for read-only borrowed text.
7. Never return pointers, references, or views to local objects.
8. Do not persist a borrowed pointer, reference, span, or string view unless lifetime is externally guaranteed.
9. Prefer internal ownership plus externally exposed views.
10. Encapsulate manual ownership inside RAII classes instead of spreading raw `new`/`delete` across the program.

5.1.6 Extensive examples

Example 1: legacy raw owner wrapper

```
#include <utility>
#include <cstddef>

class IntBlock
{
    int* ptr_;
    std::size_t count_;

public:
    explicit IntBlock(std::size_t n)
        : ptr_(new int[n]{}), count_(n)
    {
    }

    ~IntBlock()
    {
        delete[] ptr_;
    }

    IntBlock(const IntBlock&) = delete;
    IntBlock& operator=(const IntBlock&) = delete;

    IntBlock(IntBlock&& other) noexcept
        : ptr_(other.ptr_), count_(other.count_)
    {
        other.ptr_ = nullptr;
```

```
        other.count_ = 0;
    }

    IntBlock& operator=(IntBlock&& other) noexcept
    {
        if (this != &other)
        {
            delete[] ptr_;
            ptr_ = other.ptr_;
            count_ = other.count_;
            other.ptr_ = nullptr;
            other.count_ = 0;
        }
        return *this;
    }

    int* data() noexcept { return ptr_; }
    const int* data() const noexcept { return ptr_; }
    std::size_t size() const noexcept { return count_; }
};
```

Example 2: borrowed span interface

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

void increment_all(std::span<int> values)
{
    for (int& v : values)
```

```
        ++v;
}

int main()
{
    std::vector<int> numbers{1, 2, 3, 4};
    increment_all(numbers);

    for (int n : numbers)
        std::cout << n << ' ';
}
```

Example 3: read-only text borrowing

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

bool starts_with_hash(std::string_view text)
{
    return !text.empty() && text.front() == '#';
}

int main()
{
    std::string line = "#include";
    std::cout << starts_with_hash(line) << '\n';
    std::cout << starts_with_hash("plain text") << '\n';
}
```

Example 4: internal ownership with external view

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

class NameTable
{
    std::string primary_;

public:
    explicit NameTable(std::string s)
        : primary_(std::move(s))
    {
    }

    std::string_view primary() const noexcept
    {
        return primary_;
    }
};
```

Example 5: incorrect lifetime with string_view

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

std::string_view bad_name()
{
    std::string s = "temporary";
    return s;
```

```
}
```

Example 6: corrected owning return

```
#include <string>

std::string good_name()
{
    std::string s = "persistent by value";
    return s;
}
```

5.1.7 Windows build and analysis notes

On Windows with MSVC, raw ownership and borrowed-view bugs are easier to detect when code analysis is enabled. Current Microsoft diagnostics explicitly recognize lifetime risks involving raw pointers, `std::span`, and `std::string_view`, including dangling views from temporaries and references to cheap non-owning view types.

MSVC command line

```
cl /std:c++23 /EHsc /W4 /permissive- /analyze /nologo example.cpp
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

GCC on Windows via MSYS2 or MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

Recommended Windows workflow

For ownership-sensitive code on Windows:

1. compile with strong warnings,
2. enable MSVC code analysis for lifetime and guideline-based checks,
3. prefer `std::span` over pointer-plus-length APIs,
4. prefer `std::string_view` only for borrowed read-only text,
5. test in both Debug and Release configurations,
6. review any class that stores raw pointers, references, spans, or string views as members.

5.1.8 Final design conclusions

Raw ownership is not forbidden in Modern C++, but it must be treated as a specialized, high-responsibility design choice. The modern language and library ecosystem strongly encourage a clearer separation:

- ownership should be explicit,
- borrowing should be lightweight and local,
- non-owning views should not outlive their sources,
- manual lifetime control should be encapsulated behind RAII.

For high-quality Modern C++ architecture, the strongest pattern is usually:

1. own internally with RAII types,

2. expose externally through borrowed references, spans, string views, and range-based views,

3. never confuse access with ownership.

That distinction is one of the most important foundations for safe object-oriented design, lifetime control, and large-scale systems programming in Modern C++.

5.2 Modern Smart Pointer Ownership

Modern C++ ownership design is built on the principle that lifetime responsibility should be explicit, local, and safe by default. The standard smart pointers are among the most important mechanisms that make this possible. They transform raw heap allocation and ad hoc deletion patterns into structured ownership models with precise semantics.
The three primary standard smart pointers are:

1. `std::unique_ptr` for exclusive ownership

2. `std::shared_ptr` for shared ownership

3. `std::weak_ptr` for non-owning observation of shared ownership

Modern guidance strongly prefers these types over raw owning pointers. In particular, `std::unique_ptr` should be preferred unless shared ownership is truly required. This is because exclusive ownership is simpler, more predictable, and cheaper than reference-counted ownership.

5.2.1 unique_ptr architecture

The standard defines a unique pointer as an object that owns another object and manages that object through a pointer. More precisely, it stores a pointer to an object and disposes of that

object when the `unique_ptr` itself is destroyed. That is the fundamental exclusive-ownership model.

Core semantic model

A `unique_ptr<T>` has the following architectural properties:

- exactly one active owner at a time
- destruction of the owner destroys the managed object
- copying is disabled
- moving transfers ownership
- custom deletion policy is supported through the deleter type

This means `unique_ptr` is a move-only RAII handle. It models the most direct ownership relationship possible in Modern C++.

Why unique ownership matters

Exclusive ownership is ideal when one object, function, or subsystem is solely responsible for lifetime. Examples include:

- owning a dynamically allocated implementation object
- returning a newly created heap object from a factory
- storing polymorphic objects in containers
- wrapping OS handles or C API resources
- building trees where each node owns its children

This ownership form is preferred because object destruction is deterministic. When the owning `unique_ptr` dies, the resource dies.

Layout and type model

A `unique_ptr` conceptually contains:

1. a stored pointer
2. a deleter object

The deleter is part of the type. Therefore these are different types:

```
std::unique_ptr<int>
std::unique_ptr<int, MyDeleter>
```

This design allows `unique_ptr` to manage resources that are not deleted with ordinary `delete`. For example, file handles, C library objects, memory returned by a custom allocator, or objects requiring special cleanup logic.

Move-only architecture

A unique pointer cannot be copied because copying would create two owners of the same object, violating exclusive ownership. It can only be moved.

```
#include <memory>
#include <utility>

struct Widget
{
    int value{0};
};

int main()
{
```

```
    std::unique_ptr<Widget> p1 = std::make_unique<Widget>();
    p1->value = 42;

    std::unique_ptr<Widget> p2 = std::move(p1);

    if (!p1)
    {
    }

    if (p2)
    {
        int x = p2->value;
        (void)x;
    }
}
```

After the move, `p1` becomes empty and `p2` becomes the sole owner.

Array specialization

The standard library also provides array ownership support through `unique_ptr<T[]>`. This specialization uses array deletion semantics.

```
#include <memory>

int main()
{
    auto data = std::make_unique<int[]>(5);

    for (int i = 0; i < 5; ++i)
```

```
        data[i] = i * 10;
}
```

This is important because arrays require `delete[]` rather than `delete`.

Custom deleter example

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

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

int main()
{
    std::unique_ptr<std::FILE, FileCloser> file(std::fopen("test.txt", "w"));

    if (file)
    {
        std::fputs("hello\n", file.get());
    }
}
```

This demonstrates one of the most important architectural strengths of `unique_ptr`: it generalizes RAII to arbitrary resource types.

Preferred construction

In modern code, construction should usually use `std::make_unique`:

```
#include <memory>

class Engine
{
public:
    Engine(int) {}
};

int main()
{
    auto e = std::make_unique<Engine>(5);
}
```

This is clearer, safer, and avoids exposing raw `new` unnecessarily.

Polymorphic ownership

One of the strongest uses of `unique_ptr` is owning derived objects through base pointers.

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

class Shape
{
public:
    virtual ~Shape() = default;
    virtual int area() const = 0;
```

```
};

class Square : public Shape
{
    int side_;

public:
    explicit Square(int s) : side_(s) {}
    int area() const override { return side_ * side_; }
};

int main()
{
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Square>(4));
}
```

This pattern is central to modern polymorphic container design.

5.2.2 Implementing unique_ptr from scratch

A standard-conforming `unique_ptr` is complex and supports many advanced rules, including custom pointer types, array specialization, deleter traits, conversions, and partial specialization behavior. However, the core ownership model can be understood by building a simplified educational version.

Goals of a minimal implementation

A minimal educational `unique_ptr`-like type should demonstrate:

- sole ownership of a raw pointer

- destruction in the destructor
- deleted copy operations
- move transfer
- pointer access operations
- release and reset semantics

Minimal single-object implementation

```
#include <utility>
#include <cstddef>

template<typename T>
class SimpleUniquePtr
{
    T* ptr_;

public:
    SimpleUniquePtr() noexcept
        : ptr_(nullptr)
    {
    }

    explicit SimpleUniquePtr(T* p) noexcept
        : ptr_(p)
    {
    }
```

```
    ~SimpleUniquePtr()
    {
        delete ptr_;
    }

    SimpleUniquePtr(const SimpleUniquePtr&) = delete;
    SimpleUniquePtr& operator=(const SimpleUniquePtr&) = delete;

    SimpleUniquePtr(SimpleUniquePtr&& other) noexcept
        : ptr_(other.ptr_)
    {
        other.ptr_ = nullptr;
    }

    SimpleUniquePtr& operator=(SimpleUniquePtr&& other) noexcept
    {
        if (this != &other)
        {
            delete ptr_;
            ptr_ = other.ptr_;
            other.ptr_ = nullptr;
        }
        return *this;
    }

    T* get() const noexcept
    {
        return ptr_;
    }
```

```
T& operator*() const noexcept
{
    return *ptr_;
}

T* operator->() const noexcept
{
    return ptr_;
}

explicit operator bool() const noexcept
{
    return ptr_ != nullptr;
}

T* release() noexcept
{
    T* temp = ptr_;
    ptr_ = nullptr;
    return temp;
}

void reset(T* p = nullptr) noexcept
{
    if (ptr_ != p)
    {
        delete ptr_;
        ptr_ = p;
```

```
        }
    }

    void swap(SimpleUniquePtr& other) noexcept
    {
        std::swap(ptr_, other.ptr_);
    }
};
```

How the simplified version works

The logic is straightforward:

1. the constructor takes ownership of a raw pointer
2. the destructor destroys the owned object
3. copy is deleted to preserve uniqueness
4. move steals the pointer and nulls the source
5. `release()` relinquishes ownership without deleting
6. `reset()` replaces the owned pointer

Example usage

```
#include <iostream>

struct Number
{
    int value;
    explicit Number(int v) : value(v) {}
```

```
};

int main()
{
    SimpleUniquePtr<Number> p(new Number(99));

    if (p)
        std::cout << p->value << '\n';

    SimpleUniquePtr<Number> q = std::move(p);

    if (!p && q)
        std::cout << (*q).value << '\n';

    Number* raw = q.release();
    delete raw;
}
```

Why the standard version is richer

The real `std::unique_ptr` is more powerful than this teaching model because it supports:

- custom deleters as part of the type
- arrays through partial specialization
- deleter state
- pointer aliases through deleter-defined pointer types
- constrained conversions

- interoperability with standard algorithms and containers

Still, the simplified implementation captures the essence of exclusive ownership.

Educational custom-deleter version

```
#include <utility>

template<typename T, typename Deleter>
class SimpleUniquePtr2
{
    T* ptr_;
    Deleter deleter_;

public:
    SimpleUniquePtr2() noexcept
        : ptr_(nullptr), deleter_()
    {
    }

    explicit SimpleUniquePtr2(T* p, Deleter d = Deleter()) noexcept
        : ptr_(p), deleter_(d)
    {
    }

    ~SimpleUniquePtr2()
    {
        if (ptr_)
            deleter_(ptr_);
    }
```

```
    SimpleUniquePtr2(const SimpleUniquePtr2&) = delete;
    SimpleUniquePtr2& operator=(const SimpleUniquePtr2&) = delete;

    SimpleUniquePtr2(SimpleUniquePtr2&& other) noexcept
        : ptr_(other.ptr_), deleter_(std::move(other.deleter_))
    {
        other.ptr_ = nullptr;
    }

    SimpleUniquePtr2& operator=(SimpleUniquePtr2&& other) noexcept
    {
        if (this != &other)
        {
            if (ptr_)
                deleter_(ptr_);

            ptr_ = other.ptr_;
            deleter_ = std::move(other.deleter_);
            other.ptr_ = nullptr;
        }
        return *this;
    }

    T* get() const noexcept { return ptr_; }
};
```

This better reflects the two-part architecture of pointer plus deleter.

5.2.3 shared_ptr reference counting internals

A `shared_ptr` implements shared ownership. The last remaining owner destroys the managed object. Unlike `unique_ptr`, ownership is not stored in only one object. Multiple `shared_ptr` instances can cooperate through a control block.

General architecture

A `shared_ptr` conceptually consists of:

1. a stored pointer used to access the managed object
2. a pointer to a control block

The control block is the critical internal structure. Current official library documentation describes it as holding:

- the number of `shared_ptr` owners
- the number of `weak_ptr` observers
- the deleter
- the allocator for the control block, if any

This internal design is what enables shared ownership and weak observation to cooperate correctly.

Strong and weak counts

The control block usually maintains two logical counters:

- a strong count for owning `shared_ptr` instances

- a weak count for non-owning `weak_ptr` instances

The lifecycle works like this:

1. creation initializes strong ownership
2. copying a `shared_ptr` increments the strong count
3. destroying or resetting a `shared_ptr` decrements the strong count
4. when the strong count becomes zero, the managed object is destroyed
5. the control block remains alive while weak observers still exist
6. when both strong and weak participation are gone, the control block itself is released

This two-level model is essential. It allows `weak_ptr` to observe lifetime state even after the managed object has been destroyed.

Basic shared ownership example

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

struct Node
{
    int value;
    explicit Node(int v) : value(v)
    {
        std::cout << "Node constructed\n";
    }

    ~Node()
```

```
    {
        std::cout << "Node destroyed\n";
    }
};

int main()
{
    std::shared_ptr<Node> p1 = std::make_shared<Node>(10);

    {
        std::shared_ptr<Node> p2 = p1;
        std::shared_ptr<Node> p3 = p2;

        std::cout << p1.use_count() << '\n';
    }

    std::cout << p1.use_count() << '\n';
}
```

Inside the inner scope, there are three owners. After `p2` and `p3` die, only `p1` remains.

Why make_shared is important

`std::make_shared` is usually preferred to constructing from raw `new`. Official Microsoft documentation states that `make_shared` allocates and constructs both the object and a `shared_ptr` to manage it. This is important because it can reduce allocation overhead compared with separate object and control-block allocations.

```
#include <memory>
```

```
class Widget
{
public:
    Widget(int, double) {}
};

int main()
{
    auto p = std::make_shared<Widget>(5, 3.14);
}
```

Aliasing and stored pointer distinction

A subtle but important design detail is that the stored pointer used by a `shared_ptr` for access and the ownership represented by its control block are conceptually distinct. This is what makes aliasing constructors possible in the standard library. Even when advanced aliasing is not used directly, it is useful to understand that shared ownership is anchored in the control block, not merely in the visible pointer value.

Thread-safety model

Official library documentation states that different `shared_ptr` objects can be read and written concurrently even when they share ownership of the same resource. The reference counts are therefore implemented with concurrency in mind. This does not mean the pointed-to object itself becomes thread-safe. Only the ownership bookkeeping has built-in synchronization semantics.

Educational control-block sketch

A simplified teaching model of `shared_ptr` internals may look like this:

```
#include <cstddef>
#include <utility>

template<typename T>
class SimpleSharedPtr
{
    struct ControlBlock
    {
        T* ptr;
        std::size_t strong_count;
        std::size_t weak_count;

        explicit ControlBlock(T* p)
            : ptr(p), strong_count(1), weak_count(0)
        {
        }
    };

    ControlBlock* ctrl_;

public:
    SimpleSharedPtr() noexcept : ctrl_(nullptr) {}

    explicit SimpleSharedPtr(T* p)
        : ctrl_(p ? new ControlBlock(p) : nullptr)
    {
    }

    ~SimpleSharedPtr()
```

```
{
    release();
}

SimpleSharedPtr(const SimpleSharedPtr& other) noexcept
    : ctrl_(other.ctrl_)
{
    if (ctrl_)
        ++ctrl_->strong_count;
}

SimpleSharedPtr& operator=(const SimpleSharedPtr& other) noexcept
{
    if (this != &other)
    {
        release();
        ctrl_ = other.ctrl_;
        if (ctrl_)
            ++ctrl_->strong_count;
    }
    return *this;
}

SimpleSharedPtr(SimpleSharedPtr&& other) noexcept
    : ctrl_(other.ctrl_)
{
    other.ctrl_ = nullptr;
}
```

```
    SimpleSharedPtr& operator=(SimpleSharedPtr&& other) noexcept
    {
        if (this != &other)
        {
            release();
            ctrl_ = other.ctrl_;
            other.ctrl_ = nullptr;
        }
        return *this;
    }

    T* get() const noexcept
    {
        return ctrl_ ? ctrl_->ptr : nullptr;
    }

    T& operator*() const noexcept
    {
        return *ctrl_->ptr;
    }

    T* operator->() const noexcept
    {
        return ctrl_->ptr;
    }

    std::size_t use_count() const noexcept
    {
        return ctrl_ ? ctrl_->strong_count : 0;
```

```
    }

private:
    void release() noexcept
    {
        if (!ctrl_)
            return;

        --ctrl_->strong_count;

        if (ctrl_->strong_count == 0)
        {
            delete ctrl_->ptr;
            ctrl_->ptr = nullptr;

            if (ctrl_->weak_count == 0)
                delete ctrl_;
        }

        ctrl_ = nullptr;
    }
};
```

This is only an educational sketch. A real implementation must handle custom deleters, allocators, exceptions, aliasing, atomic counts, conversions, and much more. But it accurately exposes the essential reference-counting architecture.

Cost model of shared ownership

Compared with `unique_ptr`, `shared_ptr` is more expensive because it requires:

- a control block
- counter maintenance
- synchronized ownership updates
- more complex destruction behavior

That is why modern guidelines say to prefer `unique_ptr` unless shared ownership is truly necessary.

When shared_ptr is appropriate

Use shared ownership only when lifetime must be extended collaboratively by multiple independent owners. Typical examples include:

- graph nodes with shared substructures
- plugin or service objects shared across modules
- asynchronous tasks that retain shared state
- caches and registries with externally retained objects

If one owner is clearly primary and others merely observe, `weak_ptr` or raw borrowed access is often a better design.

5.2.4 weak_ptr & cycle-breaking

A `weak_ptr` points to an object managed by one or more `shared_ptr` instances, but does not participate in shared ownership. This is the core idea behind non-owning observation of a shared-lifetime object.

Why weak ownership exists

If every relationship in a shared graph were represented by `shared_ptr`, cycles could form. A cycle prevents the strong count from ever reaching zero, which means the objects would leak. For example, if object A strongly owns B and B strongly owns A, both objects keep each other alive forever even if no external owner remains.
That is exactly the problem `weak_ptr` solves.

Basic weak_ptr behavior

A `weak_ptr`:

- observes an object controlled by a shared control block
- does not increment the shared owner count
- can be converted temporarily into a `shared_ptr` via `lock()`
- can detect expiration when the object has already been destroyed

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

struct Item
{
    int value{123};
};

int main()
{
    std::weak_ptr<Item> weak;
```

```
    {
        auto strong = std::make_shared<Item>();
        weak = strong;

        if (auto locked = weak.lock())
            std::cout << locked->value << '\n';
    }

    if (auto locked = weak.lock())
    {
        std::cout << "still alive\n";
    }
    else
    {
        std::cout << "expired\n";
    }
}
```

Cycle problem with shared_ptr

```
#include <memory>

struct B;

struct A
{
    std::shared_ptr<B> b;
};

struct B
```

```
{
    std::shared_ptr<A> a;
};

int main()
{
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();

    a->b = b;
    b->a = a;
}
```

This creates a reference cycle. Even when a and b leave scope, each object is still kept alive by the other.

Breaking the cycle correctly

One side of the back-reference should be non-owning:

```
#include <memory>

struct B;

struct A
{
    std::shared_ptr<B> b;
};

struct B
```

```
{
    std::weak_ptr<A> a;
};

int main()
{
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();

    a->b = b;
    b->a = a;
}
```

Now the cycle is broken because only one direction contributes to strong ownership.

Tree and parent-pointer design

A common real-world pattern is that children are strongly owned, while parent links are weak.

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

class Node : public std::enable_shared_from_this<Node>
{
public:
    std::vector<std::shared_ptr<Node>> children;
    std::weak_ptr<Node> parent;

    void add_child(const std::shared_ptr<Node>& child)
    {
        child->parent = shared_from_this();
```

```
        children.push_back(child);
    }
};

int main()
{
    auto root = std::make_shared<Node>();
    auto child = std::make_shared<Node>();

    root->add_child(child);
}
```

This is one of the most instructive smart-pointer designs in object graphs.

enable_shared_from_this context

When an object already managed by `shared_ptr` needs to produce another `shared_ptr` that shares the same control block, `std::enable_shared_from_this` is the standard mechanism. It avoids constructing a second, unrelated control block for the same object. This is closely related to `weak_ptr` because the implementation conceptually tracks the owning shared control block for safe re-acquisition of shared ownership.

Weak observation is not borrowed raw access

A `weak_ptr` is not the same thing as a raw pointer. A raw pointer merely observes an address and says nothing about object liveness. A `weak_ptr` observes a shared ownership group and can query whether the object is still alive through `lock()`.
That is its great advantage: it can participate in safe liveness checks without extending the lifetime unnecessarily.

5.2.5 Design guidance

The modern ownership hierarchy is best understood as follows:

1. use plain values when dynamic allocation is unnecessary
2. use `std::unique_ptr` for exclusive ownership
3. use `std::shared_ptr` only when multiple owners are truly required
4. use `std::weak_ptr` for non-owning observation of shared lifetime
5. use raw pointers, references, spans, and views for borrowing, not owning

Parameter design

Smart pointers should appear in function parameters only when lifetime semantics are part of the interface.
Typical meanings:

- `T*` or `T&`: borrowed access
- `std::unique_ptr<T>`: transfer of exclusive ownership
- `std::shared_ptr<T>`: function will share ownership
- `const std::shared_ptr<T>&`: function may retain or inspect shared ownership state

Example sink:

```
#include <memory>

class Job
{
```

```
};

void submit(std::unique_ptr<Job> job)
{
    (void)job;
}
```

Example shared retention:

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

class Service
{
};

class Registry
{
    std::vector<std::shared_ptr<Service>> items_;

public:
    void add(std::shared_ptr<Service> s)
    {
        items_.push_back(std::move(s));
    }
};
```

Anti-patterns

Avoid the following:

- constructing multiple independent `shared_ptr` objects from the same raw pointer
- using `shared_ptr` by default when no sharing exists
- storing `shared_ptr` in both directions of a graph
- passing smart pointers where simple borrowing parameters would be clearer
- mixing raw owning pointers with smart ownership in the same path without strict rules

A particularly dangerous mistake is:

```
int* p = new int(5);

std::shared_ptr<int> a(p);
std::shared_ptr<int> b(p);
```

This creates two separate control blocks for the same raw pointer, which will usually lead to double deletion.
Correct design:

```
auto a = std::make_shared<int>(5);
auto b = a;
```

5.2.6 Extensive examples

Example 1: factory returning unique ownership

```
#include <memory>
#include <string>

class Report
{
```

```
    std::string name_;

public:
    explicit Report(std::string n) : name_(std::move(n)) {}
};

std::unique_ptr<Report> make_report(std::string name)
{
    return std::make_unique<Report>(std::move(name));
}
```

Example 2: move-only container of polymorphic objects

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

class Command
{
public:
    virtual ~Command() = default;
    virtual void run() = 0;
};

class ExitCommand : public Command
{
public:
    void run() override {}
};

int main()
```

```
{
    std::vector<std::unique_ptr<Command>> commands;
    commands.push_back(std::make_unique<ExitCommand>());
}
```

Example 3: shared state across tasks

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

struct State
{
    int counter{0};
};

int main()
{
    auto state = std::make_shared<State>();

    std::vector<std::shared_ptr<State>> tasks;
    tasks.push_back(state);
    tasks.push_back(state);
    tasks.push_back(state);
}
```

Example 4: weak observer cache link

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

```
class Texture
{
public:
    ~Texture()
    {
        std::cout << "Texture destroyed\n";
    }
};

int main()
{
    std::weak_ptr<Texture> cached;

    {
        auto tex = std::make_shared<Texture>();
        cached = tex;

        if (auto live = cached.lock())
        {
        }
    }

    if (!cached.lock())
    {
        std::cout << "No live texture\n";
    }
}
```

Example 5: educational weak-count sketch

```
#include <cstddef>

struct DemoControlBlock
{
    void* ptr;
    std::size_t strong_count;
    std::size_t weak_count;

    DemoControlBlock(void* p)
        : ptr(p), strong_count(1), weak_count(0)
    {
    }
};
```

This simple model highlights the essence of `shared_ptr`/`weak_ptr` cooperation: object lifetime is tied to the strong count, while control-block lifetime extends until weak observation is also gone.

5.2.7 Windows build notes

The following commands are suitable for Windows testing.

MSVC command line

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo example.cpp
cl /std:c++23 /EHsc /W4 /O2 /permissive- /nologo example.cpp
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

GCC on Windows via MSYS2 or MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

Practical Windows guidance

When testing smart-pointer ownership on Windows:

1. prefer `std::make_unique` and `std::make_shared`
2. compile with warnings enabled
3. avoid creating multiple smart pointers from the same raw pointer
4. test destruction order in debug builds with small tracing destructors
5. reserve `shared_ptr` for genuine shared ownership paths

5.2.8 Concluding rules

The modern smart-pointer ownership model can be summarized in a few decisive principles:

1. `std::unique_ptr` is the primary smart pointer for ownership because it is simple, deterministic, and efficient.
2. `std::unique_ptr` is architecturally a pointer-plus-deleter move-only RAII handle.
3. A simplified custom implementation is useful for learning, but the standard type is far richer and safer.
4. `std::shared_ptr` works through a control block that tracks strong and weak participation and stores cleanup state.
5. `std::weak_ptr` observes shared lifetime without extending it.

6. Cycles in shared ownership must be broken with `std::weak_ptr`.

7. Ownership should be chosen intentionally, not mechanically.

For high-quality Modern C++ architecture, the strongest default rule remains:

> Prefer values first, `unique_ptr` second, and `shared_ptr` only when true shared lifetime is required.

5.3 Hybrid Ownership Patterns

Hybrid ownership patterns arise when a single design uses more than one lifetime model at the same time. This is not an advanced corner case. It is the normal reality of serious Modern C++ systems. A subsystem may own objects through `std::shared_ptr`, expose temporary access through references or pointers, protect critical regions through RAII guards, and hide implementation details behind a handle-body boundary.

The most important architectural lesson is that ownership and access are not the same thing. A component may own an object, borrow an object, guard access to an object, or hide the physical representation of an object. These are different design roles, and mixing them carelessly creates leaks, dangling references, data races, or ABI fragility.

Hybrid ownership patterns are therefore about combining lifetime tools without confusing their responsibilities.

In this section, three important patterns are studied in detail:

1. shared ownership combined with borrowed access,

2. borrowed access combined with RAII guards,

3. handle-body designs that separate public interface from private representation.

5.3.1 Shared + borrowed

A shared-plus-borrowed design appears when an object has shared lifetime management, but individual functions only need temporary non-owning access to that object or to one of its subobjects.
This is one of the most practical ownership combinations in Modern C++. The system may retain a resource with `std::shared_ptr`, but internal helper functions often should not accept or store shared ownership unless they are part of the lifetime protocol.

The core distinction

A `shared_ptr<T>` expresses shared ownership. A raw pointer, reference, `std::span`, or `std::string_view` usually expresses borrowed access. Therefore the correct question is not:

> Can this function receive a `shared_ptr`?

but rather:

> Does this function need to participate in ownership, or does it only need temporary access?

If a function only reads or mutates an object during the call and does not retain it, borrowed access is typically the stronger design.

Guideline-based parameter meaning

The C++ Core Guidelines distinguish these meanings clearly:

- pass `shared_ptr<T>` by value when the function will share ownership,
- pass `const shared_ptr<T>&` when the function might retain shared ownership,

- use ordinary references or pointers when ownership transfer or retention is not part of the contract.

This is a central design rule for avoiding accidental reference-count churn and for keeping function contracts explicit.

A clean shared-plus-borrowed example

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

class Document
{
    std::string title_;

public:
    explicit Document(std::string title)
        : title_(std::move(title))
    {
    }

    const std::string& title() const noexcept
    {
        return title_;
    }
};

void print_title(const Document& doc)
{
```

```
    std::cout << doc.title() << '\n';
}

void process_document(const std::shared_ptr<Document>& maybe_keep)
{
    print_title(*maybe_keep);
}
```

In this design:

- `process_document` receives a shared owner handle,
- `print_title` receives only a borrowed reference.

This is often better than forcing every helper function to receive `shared_ptr<Document>`.

When the function should take shared ownership explicitly

If a function stores the object for later use, launches asynchronous work that retains it, or inserts it into a shared registry, then borrowing is not enough. In that case the function should take a `shared_ptr` parameter in a way that matches its semantics.
Example:

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

class Session
{
};

class SessionPool
```

```
{
    std::vector<std::shared_ptr<Session>> items_;

public:
    void retain(std::shared_ptr<Session> s)
    {
        items_.push_back(std::move(s));
    }
};
```

This function truly becomes part of the ownership graph, so shared ownership in the parameter is appropriate.

Borrowing from shared ownership safely

Borrowing from a `shared_ptr` is usually safe only while at least one owning `shared_ptr` remains alive throughout the borrowed use.
A common pattern is to create a local strong owner first, then borrow from it:

```
#include <memory>
#include <string>

class Config
{
    std::string name_;

public:
    explicit Config(std::string name)
        : name_(std::move(name))
    {
```

```
    }

    const std::string& name() const noexcept
    {
        return name_;
    }
};

void use_config(const Config& cfg)
{
    auto n = cfg.name();
    (void)n;
}

void caller(const std::shared_ptr<Config>& cfg)
{
    use_config(*cfg);
}
```

Here the borrow is valid because the shared owner remains alive during the call.

Aliasing danger

The Core Guidelines warn against passing a raw pointer or reference derived from an aliased smart pointer unless a local copy of the smart pointer is kept first. The reason is subtle but important: if other code can reseat or reset the shared owner, then the borrowed reference may outlive the actual ownership that kept the object alive.
A robust pattern is:

```
#include <memory>
```

```
class Widget
{
public:
    void ping() const noexcept {}
};

std::shared_ptr<Widget> g_widget;

void safe_use()
{
    auto keep_alive = g_widget;
    if (keep_alive)
    {
        Widget& w = *keep_alive;
        w.ping();
    }
}
```

The local copy `keep_alive` stabilizes lifetime for the duration of the borrowed use.

Shared owner plus non-owning views

Hybrid design often appears not only with references and pointers, but also with view types.

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

class Buffer
{
```

```
    std::vector<int> data_;

public:
    explicit Buffer(std::vector<int> data)
        : data_(std::move(data))
    {
    }

    std::span<const int> view() const noexcept
    {
        return data_;
    }
};

void inspect(std::span<const int> values)
{
    for (int v : values)
    {
        (void)v;
    }
}

void handle(const std::shared_ptr<Buffer>& owner)
{
    inspect(owner->view());
}
```

The `shared_ptr` owns the object, while `std::span` provides a borrowed view into the owned storage.

shared_ptr plus weak_ptr plus borrow

A more advanced hybrid arrangement uses all three roles:

- `shared_ptr` as active ownership,
- `weak_ptr` as non-owning lifetime observation,
- reference or view as temporary borrowed access after successful locking.

```
#include <memory>

class Service
{
public:
    void run() noexcept {}
};

void use_service(Service& svc)
{
    svc.run();
}

void try_use(const std::weak_ptr<Service>& weak)
{
    if (auto strong = weak.lock())
    {
        use_service(*strong);
    }
}
```

This is an excellent example of hybrid ownership discipline. The observer does not extend lifetime unless it first acquires a fresh strong owner, and once it does, ordinary borrowed access becomes safe for the current scope.

Design rules for shared + borrowed

1. Pass `shared_ptr` only when ownership semantics are part of the function contract.
2. Use references, pointers, spans, and views for transient access.
3. Create a local strong owner before borrowing from shared state that might otherwise disappear.
4. Use `weak_ptr` for observation when lifetime is shared but retention is optional.
5. Avoid spreading reference counting through every layer of helper functions.

5.3.2 Borrowed + RAII guard

Borrowed-plus-RAII-guard design appears when code borrows access to a resource but also needs a scoped object that enforces some safety condition during that access. The most common example is mutex protection through `std::lock_guard` or `std::unique_lock`. This pattern is central in concurrent and resource-sensitive code because access alone is not enough. The borrow must be accompanied by a lifetime-scoped protection mechanism.

RAII guard meaning

A guard is an object whose constructor establishes a condition and whose destructor restores or releases that condition. For mutexes, the lock is acquired in the constructor and released in the destructor.
This means the code borrows access to shared state, but the safety of that access is tied to the guard object's scope.

Basic mutex guard example

```
#include <mutex>
#include <vector>

class SafeCounter
{
    mutable std::mutex m_;
    int value_{0};

public:
    void increment()
    {
        std::lock_guard<std::mutex> guard(m_);
        ++value_;
    }

    int get() const
    {
        std::lock_guard<std::mutex> guard(m_);
        return value_;
    }
};
```

Here:

- the function borrows access to `value_`,
- the `lock_guard` protects the borrowed access,
- the destructor of the guard ensures the lock is released on all exit paths.

Why this is a hybrid ownership pattern

The guard does not own the protected data. It owns only the lock state for a scope. The protected data still belongs to its enclosing object or subsystem. Therefore the pattern is hybrid:

- the object owns the resource,
- the function borrows access to the resource,
- the guard owns a temporary protocol obligation for the scope.

That temporary protocol obligation is just as important as memory ownership.

Borrowing guarded references carefully

Sometimes code wants to expose access to data while a guard remains active. This can be safe if the lifetime relationship is explicit.
A useful pattern is a small proxy object that owns the guard and provides borrowed access:

```
#include <mutex>
#include <utility>

template<typename T>
class LockedRef
{
    std::unique_lock<std::mutex> lock_;
    T* ptr_;

public:
    LockedRef(std::mutex& m, T& obj)
        : lock_(m), ptr_(&obj)
    {
```

```
    }

    T& get() noexcept
    {
        return *ptr_;
    }

    T* operator->() noexcept
    {
        return ptr_;
    }
};

class Settings
{
    mutable std::mutex m_;
    int level_{0};

public:
    LockedRef<int> locked_level()
    {
        return LockedRef<int>(m_, level_);
    }
};
```

This pattern ties borrowed access to the lifetime of the RAII guard object.

Using the locked proxy

```
int main()
```

```
{
    Settings s;
    auto ref = s.locked_level();
    ref.get() = 5;
}
```

The critical property is that the borrowed reference must not escape beyond the guard's lifetime.

A dangerous borrowed-guard separation

The following pattern is unsafe because it separates the borrow from the scope that protects it:

```
#include <mutex>

class BadStore
{
    std::mutex m_;
    int value_{0};

public:
    int* unsafe_pointer()
    {
        std::lock_guard<std::mutex> guard(m_);
        return &value_;
    }
};
```

The returned pointer no longer has any synchronization guarantee once the function returns, because the guard has already been destroyed.

Better scoped access pattern

A safer pattern is to keep the operation inside the guarded scope:

```
#include <mutex>

class BetterStore
{
    mutable std::mutex m_;
    int value_{0};

public:
    template<typename F>
    auto with_locked_value(F&& f) -> decltype(f(value_))
    {
        std::lock_guard<std::mutex> guard(m_);
        return f(value_);
    }
};
```

Usage:

```
int main()
{
    BetterStore s;

    s.with_locked_value([](int& v)
    {
        v = 10;
    });
```

```
    int snapshot = s.with_locked_value([](int& v)
    {
        return v;
    });

    (void)snapshot;
}
```

This design preserves the crucial rule that the borrowed reference exists only while the guard is alive.

Borrowed text and buffer views under guards

The same idea applies to spans and string views. A view may be borrowed from owned storage, but if concurrent mutation is possible, the view may need a guard-scoped lifetime.

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

class NameStore
{
    mutable std::mutex m_;
    std::string name_;

public:
    explicit NameStore(std::string name)
        : name_(std::move(name))
    {
    }
```

```
    template<typename F>
    auto with_name(F&& f) const -> decltype(f(std::string_view{}))
    {
        std::lock_guard<std::mutex> guard(m_);
        return f(std::string_view{name_});
    }
};
```

The borrowed view is safe only inside the callback's scope while the guard is active.

Unnamed temporary guard bug

The Core Guidelines and current Microsoft analysis both emphasize that unnamed temporary guards are a mistake. A temporary guard may be destroyed immediately, ending the protection before the critical section actually runs.
Bad pattern:

```
#include <mutex>

std::mutex m;

void f()
{
    std::lock_guard<std::mutex>{m};
}
```

Correct pattern:

```
#include <mutex>
```

```
std::mutex m;

void f()
{
    std::lock_guard<std::mutex> guard(m);
}
```

Design rules for borrowed + RAII guard

1. Keep borrowed access inside the lifetime of the guard.
2. Do not return unprotected pointers or references to guarded internal state.
3. Prefer callback-style or proxy-style designs when access must stay synchronized.
4. Never rely on unnamed temporary guards.
5. Treat protocol ownership, such as lock ownership, as seriously as memory ownership.

5.3.3 Handle + body patterns

Handle-body patterns separate public interface from private representation. The public object acts as a handle, while the hidden implementation object is the body.
In current Microsoft documentation this is described as Pimpl, short for pointer to implementation. The C++ Core Guidelines recommend considering Pimpl for stable library ABI, and show the modern idiom with an opaque nested implementation type and a `std::unique_ptr` member.
This is one of the clearest hybrid ownership patterns because it combines:

- value-like public interface,
- internal exclusive ownership of implementation,

- strict separation between API and representation.

Why handle-body exists

The handle-body pattern solves several practical problems:

1. reducing recompilation caused by private implementation changes,
2. stabilizing public class layout,
3. hiding implementation-heavy dependencies from headers,
4. preserving clean public APIs while allowing rich private machinery.

Modern Pimpl architecture

A modern Pimpl class typically looks like this:

```
// widget.h
#include <memory>
#include <string>

class Widget
{
    class Impl;
    std::unique_ptr<Impl> pimpl_;

public:
    Widget();
    explicit Widget(std::string name);
    ~Widget();
```

```
    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;

    Widget(const Widget&) = delete;
    Widget& operator=(const Widget&) = delete;

    void draw() const;
};
```

Implementation file:

```
// widget.cpp
#include "widget.h"
#include <iostream>
#include <utility>

class Widget::Impl
{
    std::string name_;

public:
    Impl() = default;

    explicit Impl(std::string name)
        : name_(std::move(name))
    {
    }

    void draw() const
    {
```

```
        std::cout << "draw: " << name_ << '\n';
    }
};

Widget::Widget()
    : pimpl_(std::make_unique<Impl>())
{
}

Widget::Widget(std::string name)
    : pimpl_(std::make_unique<Impl>(std::move(name)))
{
}

Widget::~Widget() = default;
Widget::Widget(Widget&&) noexcept = default;
Widget& Widget::operator=(Widget&&) noexcept = default;

void Widget::draw() const
{
    pimpl_->draw();
}
```

Why unique_ptr is the natural body owner

The body is usually owned exclusively by the handle, so `std::unique_ptr` is the most natural representation:

- it expresses one-handle-per-body ownership,
- it avoids reference counting overhead,

- it supports incomplete types correctly when destruction is defined in the implementation file.

This is exactly why both the Core Guidelines example and current Microsoft guidance use `unique_ptr<impl>`.

Handle-body as a hybrid pattern

The pattern is hybrid because different layers have different responsibilities:

- the public handle owns the body through `unique_ptr`,
- clients borrow access only through public member functions,
- the body may itself contain further ownership structures such as vectors, spans, mutexes, or smart pointers.

So the public type is an ownership boundary, an encapsulation boundary, and often an ABI boundary.

Shared-body variation

Sometimes a handle-body design intentionally shares a body between multiple handles. In such a case the body may be stored in `std::shared_ptr`.

```
#include <memory>
#include <string>
#include <utility>

class SharedText
{
    struct Body
```

```
    {
        std::string value;

        explicit Body(std::string v)
            : value(std::move(v))
        {
        }
    };

    std::shared_ptr<Body> body_;

public:
    explicit SharedText(std::string v)
        : body_(std::make_shared<Body>(std::move(v)))
    {
    }

    std::string_view view() const noexcept
    {
        return body_->value;
    }
};
```

This is no longer classic exclusive Pimpl, but it is still a handle-body architecture. The handle owns access to a shared representation, while clients still receive only borrowed views.

Copyable handle with deep-copy body

Another variant is a value-semantics handle that performs deep copy of its body.

```
#include <memory>
```

```
#include <string>
#include <utility>

class TextValue
{
    struct Body
    {
        std::string value;

        explicit Body(std::string v)
            : value(std::move(v))
        {
        }
    };

    std::unique_ptr<Body> body_;

public:
    explicit TextValue(std::string v)
        : body_(std::make_unique<Body>(std::move(v)))
    {
    }

    TextValue(const TextValue& other)
        : body_(std::make_unique<Body>(*other.body_))
    {
    }

    TextValue& operator=(const TextValue& other)
```

```
    {
        if (this != &other)
        {
            body_ = std::make_unique<Body>(*other.body_);
        }
        return *this;
    }

    TextValue(TextValue&&) noexcept = default;
    TextValue& operator=(TextValue&&) noexcept = default;

    std::string_view view() const noexcept
    {
        return body_->value;
    }
};
```

This demonstrates that handle-body is not only about compile-time hiding. It is also a powerful way to centralize representation ownership and control copy and move behavior explicitly.

Borrowed access from handle-body objects

Because the body is hidden, client code usually interacts through borrowed interfaces:

- `const T&` returned from public accessors,
- `std::string_view` returned for text inspection,
- `std::span` returned for contiguous data views,
- callbacks executed against internal state without exposing representation directly.

Example:

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

class SampleSet
{
    struct Body
    {
        std::vector<int> values;
    };

    std::unique_ptr<Body> body_;

public:
    SampleSet()
        : body_(std::make_unique<Body>())
    {
    }

    void add(int v)
    {
        body_->values.push_back(v);
    }

    std::span<const int> values() const noexcept
    {
        return body_->values;
    }
```

```
};
```

The handle owns the body; the caller only borrows a view.

Handle-body plus synchronization

In larger systems, the body may also own synchronization tools, so the handle-body pattern combines naturally with borrowed-plus-guard patterns.

```
#include <memory>
#include <mutex>
#include <string>
#include <utility>

class SafeName
{
    struct Body
    {
        mutable std::mutex m;
        std::string name;
    };

    std::unique_ptr<Body> body_;

public:
    explicit SafeName(std::string name)
        : body_(std::make_unique<Body>())
    {
        body_->name = std::move(name);
    }
```

```
    std::string read_copy() const
    {
        std::lock_guard<std::mutex> guard(body_->m);
        return body_->name;
    }

    void update(std::string name)
    {
        std::lock_guard<std::mutex> guard(body_->m);
        body_->name = std::move(name);
    }
};
```

This example shows how hybrid patterns compose naturally:

- the handle owns the body,
- the body owns its internal state,
- access is borrowed inside member functions,
- synchronization is enforced by RAII guards.

Trade-offs of handle-body

The handle-body pattern is powerful, but it is not free. It introduces:

- one extra indirection,
- dynamic allocation for the body in typical implementations,
- more split code between interface and implementation files.

It should therefore be used intentionally, especially for stable APIs, large dependency-heavy types, or library boundaries.

Design rules for handle + body

1. Use `std::unique_ptr` for the body by default.
2. Define destructor and move operations in the implementation file when the body type is incomplete in the header.
3. Expose borrowed interfaces outward rather than leaking implementation structure.
4. Use shared-body designs only when representation sharing is truly part of the semantics.
5. Consider handle-body when ABI stability or compile-time insulation matters.

5.3.4 Comparative design map

The three hybrid patterns in this section can be compared as follows:

Shared + borrowed

- ownership is shared,
- access during local operations is borrowed,
- lifetime stabilization may require a local strong owner.

Borrowed + RAII guard

- the data is owned elsewhere,
- access is borrowed,
- the guard temporarily owns a protocol obligation such as a lock.

Handle + body

- the public object owns the private representation,
- clients borrow only public access paths,
- the pattern isolates representation and often ABI.

5.3.5 Extensive examples

Example 1: shared owner, borrowed helper

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

class Asset
{
    std::string id_;

public:
    explicit Asset(std::string id)
        : id_(std::move(id))
    {
    }

    const std::string& id() const noexcept
    {
        return id_;
    }
};
```

```
void print_asset(const Asset& a)
{
    std::cout << a.id() << '\n';
}

void retain_and_use(std::shared_ptr<Asset> a)
{
    print_asset(*a);
}
```

Example 2: weak observer upgraded to strong owner, then borrowed

```
#include <memory>

class Worker
{
public:
    void step() noexcept {}
};

void do_step(Worker& w)
{
    w.step();
}

void try_step(const std::weak_ptr<Worker>& weak)
{
    if (auto keep = weak.lock())
    {
```

```
        do_step(*keep);
    }
}
```

Example 3: borrowed access protected by RAII lock

```
#include <mutex>

class Counter
{
    mutable std::mutex m_;
    int value_{0};

public:
    void add_one()
    {
        std::lock_guard<std::mutex> guard(m_);
        ++value_;
    }

    int read() const
    {
        std::lock_guard<std::mutex> guard(m_);
        return value_;
    }
};
```

Example 4: callback-based guarded borrowing

```
#include <mutex>
```

```
#include <string>
#include <utility>

class TextStore
{
    mutable std::mutex m_;
    std::string text_;

public:
    explicit TextStore(std::string text)
        : text_(std::move(text))
    {
    }

    template<typename F>
    auto with_text(F&& f) -> decltype(f(text_))
    {
        std::lock_guard<std::mutex> guard(m_);
        return f(text_);
    }
};
```

Example 5: classic unique_ptr Pimpl

```
// header
#include <memory>

class Engine
{
    class Impl;
```

```
    std::unique_ptr<Impl> impl_;

public:
    Engine();
    ~Engine();
    Engine(Engine&&) noexcept;
    Engine& operator=(Engine&&) noexcept;

    Engine(const Engine&) = delete;
    Engine& operator=(const Engine&) = delete;

    void run() const;
};
```

```
// source
#include <iostream>

class Engine::Impl
{
public:
    void run() const
    {
        std::cout << "running\n";
    }
};

Engine::Engine()
    : impl_(std::make_unique<Impl>())
{
}
```

```
Engine::~Engine() = default;
Engine::Engine(Engine&&) noexcept = default;
Engine& Engine::operator=(Engine&&) noexcept = default;

void Engine::run() const
{
    impl_->run();
}
```

Example 6: handle-body returning a borrowed view

```
#include <memory>
#include <string>
#include <string_view>
#include <utility>

class UserName
{
    struct Body
    {
        std::string value;
    };

    std::unique_ptr<Body> body_;

public:
    explicit UserName(std::string s)
        : body_(std::make_unique<Body>())
    {
```

```
        body_->value = std::move(s);
    }

    std::string_view view() const noexcept
    {
        return body_->value;
    }
};
```

5.3.6 Windows build notes

The following commands are suitable for testing the examples in Windows environments.

MSVC command line

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo example.cpp
cl /std:c++23 /EHsc /W4 /O2 /permissive- /analyze /nologo example.cpp
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

GCC on Windows via MSYS2 or MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic -O2 example.cpp -o example.exe
```

Practical Windows guidance

For these hybrid lifetime patterns on Windows:

1. build with strong warnings enabled,
2. use MSVC code analysis for lifetime and locking mistakes,

3. keep lock guards named and scoped,
4. keep borrowed views local unless lifetime is guaranteed,
5. define Pimpl destructor and move operations in the implementation file.

5.3.7 Concluding design rules

Hybrid ownership patterns succeed when each tool keeps its exact role.

1. `shared_ptr` expresses shared lifetime, not ordinary access.
2. References, pointers, spans, and string views express borrowing, not owning.
3. RAII guards own temporary protocol obligations such as locks.
4. Handle-body designs centralize representation ownership and hide implementation.
5. Do not let borrowed access escape the lifetime that makes it safe.
6. Do not force ownership types into APIs that only need transient observation.
7. Compose these patterns deliberately instead of letting them emerge accidentally.

The deepest architectural lesson is that high-quality Modern C++ does not depend on one ownership model everywhere. It depends on using the right model at the right boundary, and on making every ownership and borrowing transition explicit.

Part III

RAII, Resource Wrappers & Exception-Safe Design

RAII Fundamentals & Deep Theory

6.1 Why RAII Works

Resource Acquisition Is Initialization (RAII) is one of the most fundamental design principles of Modern C++. It provides a deterministic and reliable mechanism for managing resources through object lifetime semantics. The central idea is that the acquisition of a resource is tied to the initialization of an object, and the release of that resource is tied to the destruction of the object.

In C++, object lifetime is precisely defined by the language standard. When an object is created, its constructor executes. When the object leaves scope or is otherwise destroyed, its destructor executes. RAII leverages this deterministic behavior to ensure that resources such as memory, file descriptors, sockets, mutexes, and handles are automatically released.

This design approach eliminates a large class of programming errors that historically plagued systems written in manual resource-management styles. Instead of requiring developers to remember to call cleanup functions explicitly, RAII guarantees that cleanup occurs automatically as part of normal language semantics.

Modern C++ libraries and frameworks rely heavily on RAII. Standard library components such as `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard`, `std::fstream`, and `std::vector` are all implementations of the RAII pattern.

The effectiveness of RAII can be understood by examining three fundamental properties:

1. deterministic destruction,
2. contrast with garbage-collected languages,
3. prevention of resource leaks and enforcement of program invariants.

6.1.1 Deterministic destruction

Deterministic destruction means that the moment an object goes out of scope, its destructor is immediately executed. This behavior is guaranteed by the C++ language and is independent of runtime heuristics.
Consider a simple example involving file management.

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

void write_log(const std::string& message)
{
    std::ofstream file("log.txt");

    file << message << '\n';

}
```

In this example:

- the `std::ofstream` constructor opens the file,
- the destructor automatically closes the file when the object leaves scope.

The programmer does not explicitly call `close()`. The destructor ensures that the resource is released.

This deterministic behavior becomes extremely important in large systems that manage scarce resources such as:

- operating system file descriptors,
- network sockets,
- GPU resources,
- mutex locks,
- memory regions.

RAII ensures that these resources are always released exactly when the object lifetime ends.

Scope-based lifetime control

RAII integrates naturally with block scope.

```
#include <iostream>

class Resource
{
public:
    Resource()
    {
        std::cout << "Resource acquired\n";
    }

    ~Resource()
    {
        std::cout << "Resource released\n";
```

```
    }
};

int main()
{
    {
        Resource r;
    }

}
```

Program output:

```
Resource acquired
Resource released
```

The resource is released immediately when the scope ends.

Exception-safe cleanup

One of the most powerful properties of RAII is that destructors run during stack unwinding when an exception occurs.

```
#include <iostream>
#include <stdexcept>

class Guard
{
public:
    Guard()
    {
```

```
        std::cout << "Acquire\n";
    }

    ~Guard()
    {
        std::cout << "Release\n";
    }
};

void work()
{
    Guard g;

    throw std::runtime_error("failure");
}

int main()
{
    try
    {
        work();
    }
    catch(...)
    {
        std::cout << "Exception caught\n";
    }
}
```

Output:

```
Acquire
```

```
Release
Exception caught
```

Even though an exception interrupts execution, the destructor still runs. This guarantees that resources are never leaked due to exceptional control flow.

RAII and mutex locking

Another classical RAII example is mutex locking.

```
#include <mutex>

std::mutex global_mutex;

void critical_section()
{
    std::lock_guard<std::mutex> lock(global_mutex);

}
```

The lock is acquired in the constructor of `lock_guard` and released in its destructor. This ensures that the mutex is always unlocked, even if an exception occurs inside the critical section.

Custom RAII resource wrapper

Programmers can easily create their own RAII wrappers.

```
#include <cstdio>

class FileHandle
```

```
{
    FILE* file_;

public:

    explicit FileHandle(const char* path)
    {
        file_ = std::fopen(path, "r");
    }

    ~FileHandle()
    {
        if (file_)
            std::fclose(file_);
    }

    FILE* get() const
    {
        return file_;
    }
};
```

Usage:

```
void read_data()
{
    FileHandle file("data.txt");

}
```

When the function exits, the file automatically closes.

6.1.2 Comparisons with GC languages

Languages with garbage collection typically manage memory automatically through runtime systems that periodically reclaim unreachable objects. While this approach simplifies memory management, it does not provide deterministic destruction.
Garbage collectors reclaim memory when the runtime decides to run a collection cycle. This timing is nondeterministic.
C++ RAII differs fundamentally from garbage-collected memory models.

Deterministic vs nondeterministic cleanup

In a garbage-collected environment:

- objects become eligible for collection when unreachable,
- actual destruction may occur much later,
- finalizers may run unpredictably.

In contrast, C++ provides deterministic destruction.

- objects are destroyed exactly when their lifetime ends,
- destructors execute immediately,
- resource release is predictable.

This property is especially important when resources are not memory.
Examples include:

- database transactions,
- file handles,

- system locks,
- network connections.

These resources must often be released promptly.

Example illustrating nondeterministic cleanup problems

Consider a program that opens a file and relies on a garbage collector to release it.
If many objects accumulate before a collection cycle occurs, file descriptors may remain open longer than expected, potentially exhausting system resources.
RAII avoids this problem entirely.

RAII provides stronger invariants

RAII does not merely reclaim memory. It enforces program invariants through structured lifetimes.
For example:

- a mutex must remain locked only while a guard exists,
- a transaction must remain open only while a transaction object exists,
- a file must remain open only while its wrapper exists.

These invariants become encoded in the type system.

6.1.3 Preventing leaks and maintaining invariants

Before RAII became common practice, C and early C++ programs relied heavily on manual cleanup patterns.

Manual cleanup pattern

```
#include <cstdio>

void legacy()
{
    FILE* f = std::fopen("data.txt", "r");

    if (!f)
        return;

    char buffer[256];

    if (!std::fgets(buffer, sizeof(buffer), f))
    {
        std::fclose(f);
        return;
    }

    std::fclose(f);
}
```

This style is error-prone because:

- every exit path must release the resource,
- developers may forget cleanup in some branches,
- exception paths complicate control flow.

RAII eliminates this complexity.

RAII solution

```
#include <cstdio>

class FileWrapper
{
    FILE* file_;

public:

    explicit FileWrapper(const char* path)
        : file_(std::fopen(path, "r"))
    {
    }

    ~FileWrapper()
    {
        if (file_)
            std::fclose(file_);
    }

    FILE* get() const
    {
        return file_;
    }
};

void modern()
{
    FileWrapper file("data.txt");
```

```
}
```

There is no need to manually track every return path.

RAII and strong invariants

RAII ensures that objects maintain valid states throughout their lifetime.
Consider a class that manages a transaction.

```
#include <iostream>

class Transaction
{
    bool active_;

public:

    Transaction()
        : active_(true)
    {
        std::cout << "Transaction started\n";
    }

    ~Transaction()
    {
        if (active_)
            std::cout << "Transaction rolled back\n";
    }
```

```
    void commit()
    {
        active_ = false;
        std::cout << "Transaction committed\n";
    }
};
```

Usage:

```
void process()
{
    Transaction tx;

    tx.commit();
}
```

If an exception occurs before `commit()`, the destructor automatically performs the rollback. This pattern guarantees that a transaction is never left incomplete.

RAII and container safety

The C++ Standard Library heavily relies on RAII to maintain container invariants.
For example, `std::vector` manages dynamic memory through automatic construction and destruction of its elements.

```
#include <vector>

void example()
{
    std::vector<int> v = {1,2,3,4};

}
```

When the vector goes out of scope:

- each element is destroyed,
- allocated memory is released.

This behavior is guaranteed regardless of how the function exits.

RAII and exception guarantees

RAII is central to exception-safe programming. Because destructors always run during stack unwinding, RAII objects ensure that resources are released even when errors occur.
The strong exception guarantee often relies on RAII-managed temporary objects.

```
#include <vector>

void safe_push(std::vector<int>& v)
{
    std::vector<int> temp = v;

    temp.push_back(42);

    v.swap(temp);
}
```

If any operation throws an exception, the temporary object cleans up automatically.

6.1.4 Summary

RAII is effective because it aligns resource management with the fundamental object lifetime rules of the C++ language.
Its effectiveness derives from three properties:

1. deterministic destruction guarantees immediate resource release,
2. predictable semantics outperform nondeterministic garbage collection for many system resources,
3. automatic destructors prevent leaks and enforce program invariants.

As a result, RAII forms the foundation of Modern C++ resource management. It enables safe systems programming, exception-safe design, and scalable software architectures where resource lifetimes are clearly encoded in the structure of the program itself.

6.2 Resources Eligible for RAII

RAII is not limited to heap memory. It is a general lifetime discipline for any resource that must be acquired and later released according to a protocol. This is one of the deepest reasons why RAII is so powerful in C++: the language does not treat resource management as a special runtime subsystem. Instead, it ties acquisition and release directly to object lifetime.
A resource is an excellent candidate for RAII when it has most or all of the following properties:

1. it must be explicitly acquired,
2. it must be explicitly released,
3. forgetting release causes leaks, exhaustion, or broken program state,
4. release must happen even on exceptions or early returns,
5. ownership can be represented by one object or one small group of objects.

This covers far more than memory. In real systems, the most important RAII resources include:

- dynamically allocated memory,
- operating-system handles,
- POSIX file descriptors,
- network sockets,
- GPU and graphics API handles,
- synchronization primitives and lock ownership,
- mapped memory objects,
- transactions, sessions, and other protocol states.

In this section, the focus is on the resource families that most clearly demonstrate how broad RAII really is.

6.2.1 Memory wrappers

Memory is the resource most programmers first associate with RAII, but in Modern C++ it should be seen as only one example among many. The standard library strongly encourages expressing dynamic memory ownership through wrappers rather than direct `new` and `delete`.

Why dynamic memory is eligible for RAII

Heap allocation follows the classic acquire/release pattern:

- memory is acquired through allocation,
- memory must later be released,
- failure to release causes leaks,

- exceptions and multiple control-flow exits make manual cleanup fragile.

This makes dynamic memory an ideal RAII resource.

Single-object ownership with std::unique_ptr

The clearest standard memory wrapper is `std::unique_ptr`. It expresses exclusive ownership of dynamically allocated storage.

```
#include <memory>

class Widget
{
    int value_;

public:
    explicit Widget(int v) : value_(v) {}
    int value() const noexcept { return value_; }
};

int main()
{
    auto p = std::make_unique<Widget>(42);

    int x = p->value();
    (void)x;
}
```

This is RAII because:

- the memory is acquired during object construction,
- the memory is released automatically when the owning `unique_ptr` is destroyed.

Array ownership

Modern C++ also supports array ownership through `std::unique_ptr<T[]>`.

```
#include <memory>
#include <cstddef>

int main()
{
    std::size_t n = 8;
    auto data = std::make_unique<int[]>(n);

    for (std::size_t i = 0; i < n; ++i)
        data[i] = static_cast<int>(i * 10);
}
```

This is much safer than pairing `new[]` with manual `delete[]` across many exit paths.

Shared memory ownership

When memory must be jointly owned, `std::shared_ptr` can wrap that resource with reference counting.

```
#include <memory>

struct State
{
    int counter{0};
};

int main()
```

```
{
    auto s1 = std::make_shared<State>();
    auto s2 = s1;
    auto s3 = s2;

    s1->counter = 7;
}
```

The underlying object remains alive until the last shared owner is destroyed.

Containers as RAII memory wrappers

Dynamic memory does not always need smart pointers. Standard containers are themselves RAII types.

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

int main()
{
    std::vector<std::string> lines;
    lines.push_back("alpha");
    lines.push_back("beta");
    lines.push_back("gamma");
}
```

The vector owns its dynamic storage and releases it in its destructor. This is pure RAII even though the programmer never sees an explicit smart pointer.

Views are not owners

Modern C++ also distinguishes clearly between owning memory wrappers and non-owning views.

```
#include <vector>
#include <span>

void print_values(std::span<const int> values)
{
    for (int v : values)
        (void)v;
}

int main()
{
    std::vector<int> data{1, 2, 3, 4};
    print_values(data);
}
```

Here:

- `std::vector<int>` owns the memory,
- `std::span<const int>` borrows access only.

This separation is essential to correct RAII-based design.

Custom heap wrapper

A simple educational memory RAII wrapper looks like this:

```cpp
#include <utility>

template<typename T>
class HeapBox
{
    T* ptr_;

public:
    explicit HeapBox(T* ptr = nullptr) noexcept
        : ptr_(ptr)
    {
    }

    ~HeapBox()
    {
        delete ptr_;
    }

    HeapBox(const HeapBox&) = delete;
    HeapBox& operator=(const HeapBox&) = delete;

    HeapBox(HeapBox&& other) noexcept
        : ptr_(other.ptr_)
    {
        other.ptr_ = nullptr;
    }

    HeapBox& operator=(HeapBox&& other) noexcept
    {
```

```
        if (this != &other)
        {
            delete ptr_;
            ptr_ = other.ptr_;
            other.ptr_ = nullptr;
        }
        return *this;
    }

    T* get() const noexcept { return ptr_; }
    T& operator*() const noexcept { return *ptr_; }
    T* operator->() const noexcept { return ptr_; }
};
```

This wrapper proves the essential pattern: memory ownership is just another acquire/release protocol.

6.2.2 OS handles (Windows/Linux/Mac)

Operating systems expose many resources through handle-like abstractions. These are among the strongest candidates for RAII because manual release errors cause real system-level leaks.

Why OS handles are ideal RAII resources

An operating-system handle or kernel object typically has these characteristics:

- it is created or opened by an API call,
- it represents a scarce system resource,
- it must be closed or released explicitly,

- forgetting to close it can exhaust process or system resources,
- exceptions and early returns make manual cleanup unreliable.

This makes handles a natural fit for destructor-based cleanup.

Windows HANDLE wrappers

On Windows, many APIs return a `HANDLE` and require `CloseHandle` when the object is no longer needed.
Examples include:

- files,
- events,
- mutexes,
- processes,
- threads,
- file mappings,
- tokens,
- many other kernel objects.

A minimal Windows handle wrapper can be written as follows:

```
#ifdef _WIN32
#include <windows.h>

class WinHandle
```

```
{
    HANDLE h_;

public:
    WinHandle() noexcept
        : h_(nullptr)
    {
    }

    explicit WinHandle(HANDLE h) noexcept
        : h_(h)
    {
    }

    ~WinHandle()
    {
        if (h_ && h_ != INVALID_HANDLE_VALUE)
            CloseHandle(h_);
    }

    WinHandle(const WinHandle&) = delete;
    WinHandle& operator=(const WinHandle&) = delete;

    WinHandle(WinHandle&& other) noexcept
        : h_(other.h_)
    {
        other.h_ = nullptr;
    }
```

```
    WinHandle& operator=(WinHandle&& other) noexcept
    {
        if (this != &other)
        {
            if (h_ && h_ != INVALID_HANDLE_VALUE)
                CloseHandle(h_);

            h_ = other.h_;
            other.h_ = nullptr;
        }
        return *this;
    }

    HANDLE get() const noexcept
    {
        return h_;
    }

    explicit operator bool() const noexcept
    {
        return h_ && h_ != INVALID_HANDLE_VALUE;
    }
};
#endif
```

This wrapper expresses ownership of a Windows kernel handle safely.

Windows file-open example

```
#ifdef _WIN32
```

```
#include <windows.h>

WinHandle open_readonly_file(const wchar_t* path)
{
    HANDLE h = CreateFileW(
        path,
        GENERIC_READ,
        FILE_SHARE_READ,
        nullptr,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        nullptr
    );

    return WinHandle(h);
}
#endif
```

The caller receives an owning RAII object instead of a naked `HANDLE`.

Unix-like systems: Linux and macOS

On Linux and macOS, many system resources are represented by file descriptors or descriptor-like integers, released by `close()`. This includes ordinary files, pipes, sockets, and many other kernel-managed objects.
Even though the underlying OS implementations differ, the lifetime protocol is still perfectly RAII-compatible:

- acquire through an API such as `open()`, `socket()`, or `pipe()`,
- release through `close()`.

That makes POSIX-style descriptors one of the purest forms of RAII-eligible resources.

6.2.3 File descriptors

A file descriptor is a small integer that refers to an open file description or similar kernel resource in POSIX systems. It must be explicitly closed. That alone makes it a prime candidate for RAII.

Why descriptors need wrappers

Manual file descriptor management is fragile because code often contains:

- multiple return paths,
- nested error handling,
- partial initialization,
- exception translation layers in C++ wrappers,
- transfer of ownership between functions.

A descriptor wrapper centralizes the release logic.

Minimal POSIX file descriptor wrapper

```
#if !defined(_WIN32)
#include <unistd.h>

class FileDescriptor
{
    int fd_;
```

```
public:
    FileDescriptor() noexcept
        : fd_(-1)
    {
    }

    explicit FileDescriptor(int fd) noexcept
        : fd_(fd)
    {
    }

    ~FileDescriptor()
    {
        if (fd_ != -1)
            ::close(fd_);
    }

    FileDescriptor(const FileDescriptor&) = delete;
    FileDescriptor& operator=(const FileDescriptor&) = delete;

    FileDescriptor(FileDescriptor&& other) noexcept
        : fd_(other.fd_)
    {
        other.fd_ = -1;
    }

    FileDescriptor& operator=(FileDescriptor&& other) noexcept
    {
```

```
        if (this != &other)
        {
            if (fd_ != -1)
                ::close(fd_);
            fd_ = other.fd_;
            other.fd_ = -1;
        }
        return *this;
    }

    int get() const noexcept
    {
        return fd_;
    }

    explicit operator bool() const noexcept
    {
        return fd_ != -1;
    }

    int release() noexcept
    {
        int temp = fd_;
        fd_ = -1;
        return temp;
    }
};
#endif
```

POSIX open example

```
#if !defined(_WIN32)
#include <fcntl.h>

FileDescriptor open_file_readonly(const char* path)
{
    int fd = ::open(path, O_RDONLY);
    return FileDescriptor(fd);
}
#endif
```

This design ensures the descriptor is released even if later operations throw an exception.

Descriptor wrappers generalize well

The same wrapper style can manage:

- regular files,
- pipes,
- event descriptors on Linux,
- epoll descriptors,
- timer descriptors,
- process-related descriptors,
- many other Unix-like kernel resources.

In all cases, the core rule is the same: if `close()` is required, RAII is strongly appropriate.

6.2.4 Network sockets

Sockets are one of the best examples of non-memory RAII resources. They are explicitly created, must be explicitly closed, and forgetting to close them causes leaks and network-state confusion.

Why sockets are RAII resources

Sockets satisfy every major RAII criterion:

- they are allocated through system or library calls,
- they consume kernel resources,
- they must be closed with the correct API,
- they often participate in complex error-handling paths,
- early return or exception without cleanup is dangerous.

Windows socket wrapper

On Windows, sockets are not released with `CloseHandle`. They must be released with `closesocket`. This is an important detail and a strong argument for separate wrapper types per resource family.

```
#ifdef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>

class WinSocket
{
```

```
    SOCKET s_;

public:
    WinSocket() noexcept
        : s_(INVALID_SOCKET)
    {
    }

    explicit WinSocket(SOCKET s) noexcept
        : s_(s)
    {
    }

    ~WinSocket()
    {
        if (s_ != INVALID_SOCKET)
            ::closesocket(s_);
    }

    WinSocket(const WinSocket&) = delete;
    WinSocket& operator=(const WinSocket&) = delete;

    WinSocket(WinSocket&& other) noexcept
        : s_(other.s_)
    {
        other.s_ = INVALID_SOCKET;
    }

    WinSocket& operator=(WinSocket&& other) noexcept
```

```
    {
        if (this != &other)
        {
            if (s_ != INVALID_SOCKET)
                ::closesocket(s_);
            s_ = other.s_;
            other.s_ = INVALID_SOCKET;
        }
        return *this;
    }

    SOCKET get() const noexcept
    {
        return s_;
    }

    explicit operator bool() const noexcept
    {
        return s_ != INVALID_SOCKET;
    }
};
#endif
```

Windows Winsock session guard

Windows networking also has a process-level initialization and cleanup protocol through `WSAStartup` and `WSACleanup`. This too is RAII-eligible.

```
#ifdef _WIN32
#include <winsock2.h>
```

```
class WinsockSession
{
    bool ok_;

public:
    WinsockSession() noexcept
        : ok_(false)
    {
        WSADATA data{};
        ok_ = (WSAStartup(MAKEWORD(2, 2), &data) == 0);
    }

    ~WinsockSession()
    {
        if (ok_)
            WSACleanup();
    }

    WinsockSession(const WinsockSession&) = delete;
    WinsockSession& operator=(const WinsockSession&) = delete;

    bool ok() const noexcept
    {
        return ok_;
    }
};
#endif
```

This is an excellent example of RAII applied not to a single object handle, but to a

process-scoped API state.

POSIX socket wrapper

On Unix-like systems, sockets are typically file descriptors and are closed with `close()`.

```
#if !defined(_WIN32)
#include <sys/socket.h>
#include <unistd.h>

class PosixSocket
{
    int fd_;

public:
    PosixSocket() noexcept
        : fd_(-1)
    {
    }

    explicit PosixSocket(int fd) noexcept
        : fd_(fd)
    {
    }

    ~PosixSocket()
    {
        if (fd_ != -1)
            ::close(fd_);
    }
```

```
    PosixSocket(const PosixSocket&) = delete;
    PosixSocket& operator=(const PosixSocket&) = delete;

    PosixSocket(PosixSocket&& other) noexcept
        : fd_(other.fd_)
    {
        other.fd_ = -1;
    }

    PosixSocket& operator=(PosixSocket&& other) noexcept
    {
        if (this != &other)
        {
            if (fd_ != -1)
                ::close(fd_);
            fd_ = other.fd_;
            other.fd_ = -1;
        }
        return *this;
    }

    int get() const noexcept
    {
        return fd_;
    }
};
#endif
```

Protocol-aware socket guards

Sometimes RAII should manage not only the socket descriptor itself, but also shutdown behavior or partial protocol cleanup.

```
#ifdef _WIN32
#include <winsock2.h>

class GracefulSocket
{
    SOCKET s_;

public:
    explicit GracefulSocket(SOCKET s = INVALID_SOCKET) noexcept
        : s_(s)
    {
    }

    ~GracefulSocket()
    {
        if (s_ != INVALID_SOCKET)
        {
            ::shutdown(s_, SD_BOTH);
            ::closesocket(s_);
        }
    }

    GracefulSocket(const GracefulSocket&) = delete;
    GracefulSocket& operator=(const GracefulSocket&) = delete;
};
```

```
#endif
```

This shows that RAII can encode not only raw release, but also domain-specific shutdown policy.

6.2.5 GPU/Graphics handles

Graphics and compute APIs are among the most RAII-worthy domains in modern systems programming. They expose many opaque handles that must be destroyed in the correct order with the correct API.

Why graphics handles are RAII resources

Graphics resources usually satisfy every RAII condition:

- they are created through explicit API calls,
- they often live in scarce driver or device memory,
- they must be destroyed with matching API functions,
- order matters,
- errors and partial construction are common,
- leaks may persist until application shutdown or device reset.

Examples include:

- buffers,
- images and textures,
- command pools,

- descriptor sets or heaps,
- shader modules,
- pipeline objects,
- framebuffers,
- swapchain-related objects.

Vulkan handles

Vulkan is especially clear about explicit lifetime management. Many resources are opaque handles and must be destroyed with matching destruction functions such as `vkDestroyBuffer`, `vkDestroyImage`, `vkDestroyPipeline`, and others.
This makes Vulkan an ideal RAII domain.

Minimal Vulkan buffer wrapper

```
#include <vulkan/vulkan.h>

class VulkanBuffer
{
    VkDevice device_;
    VkBuffer buffer_;

public:
    VulkanBuffer() noexcept
        : device_(VK_NULL_HANDLE), buffer_(VK_NULL_HANDLE)
    {
    }
```

```
    VulkanBuffer(VkDevice device, VkBuffer buffer) noexcept
        : device_(device), buffer_(buffer)
    {
    }

    ~VulkanBuffer()
    {
        if (buffer_ != VK_NULL_HANDLE)
            vkDestroyBuffer(device_, buffer_, nullptr);
    }

    VulkanBuffer(const VulkanBuffer&) = delete;
    VulkanBuffer& operator=(const VulkanBuffer&) = delete;

    VulkanBuffer(VulkanBuffer&& other) noexcept
        : device_(other.device_), buffer_(other.buffer_)
    {
        other.device_ = VK_NULL_HANDLE;
        other.buffer_ = VK_NULL_HANDLE;
    }

    VulkanBuffer& operator=(VulkanBuffer&& other) noexcept
    {
        if (this != &other)
        {
            if (buffer_ != VK_NULL_HANDLE)
                vkDestroyBuffer(device_, buffer_, nullptr);

            device_ = other.device_;
```

```
            buffer_ = other.buffer_;
            other.device_ = VK_NULL_HANDLE;
            other.buffer_ = VK_NULL_HANDLE;
        }
        return *this;
    }

    VkBuffer get() const noexcept
    {
        return buffer_;
    }
};
```

This is textbook RAII: the handle is released automatically in the destructor.

Direct3D and COM-based graphics resources

On Windows, Direct3D interfaces are COM objects. COM object lifetime is managed through `AddRef()` and `Release()`. That means COM pointers are also naturally RAII-eligible.
A minimal educational COM-style wrapper looks like this:

```
template<typename T>
class ComOwner
{
    T* ptr_;

public:
    ComOwner() noexcept
        : ptr_(nullptr)
    {
```

```
    }

    explicit ComOwner(T* ptr) noexcept
        : ptr_(ptr)
    {
    }

    ~ComOwner()
    {
        if (ptr_)
            ptr_->Release();
    }

    ComOwner(const ComOwner&) = delete;
    ComOwner& operator=(const ComOwner&) = delete;

    ComOwner(ComOwner&& other) noexcept
        : ptr_(other.ptr_)
    {
        other.ptr_ = nullptr;
    }

    ComOwner& operator=(ComOwner&& other) noexcept
    {
        if (this != &other)
        {
            if (ptr_)
                ptr_->Release();
            ptr_ = other.ptr_;
```

```
            other.ptr_ = nullptr;
        }
        return *this;
    }

    T* get() const noexcept { return ptr_; }
    T** put() noexcept { return &ptr_; }
    T* operator->() const noexcept { return ptr_; }
};
```

This style is the conceptual basis of common Windows COM RAII wrappers.

RAII for multi-step graphics construction

Graphics code often builds several resources in sequence. RAII is crucial because partial failure is common.

```
class RendererResources
{
    VulkanBuffer vertex_buffer_;
    VulkanBuffer index_buffer_;

public:
    RendererResources(VulkanBuffer vb, VulkanBuffer ib) noexcept
        : vertex_buffer_(std::move(vb)),
          index_buffer_(std::move(ib))
    {
    }
};
```

If construction of one later stage fails, already-acquired earlier resources are still released correctly by their destructors.

6.2.6 Cross-platform RAII patterns

The same design ideas recur across memory, handles, descriptors, sockets, and graphics resources.

Exclusive ownership wrapper pattern

Most RAII wrappers for low-level resources have the same shape:

1. store the raw handle,
2. acquire it in construction or factory code,
3. release it in the destructor,
4. delete copy operations,
5. support move transfer,
6. provide `get()`, `release()`, and `reset()` when needed.

Nullable invalid state

Most low-level wrappers also define an invalid state:

- `nullptr` for pointers,
- `INVALID_HANDLE_VALUE` or `nullptr` for Windows handles,
- `-1` for POSIX descriptors,
- `INVALID_SOCKET` for Winsock sockets,
- `VK_NULL_HANDLE` for Vulkan handles.

This invalid state supports safe destruction and safe moved-from objects.

API-specific release functions matter

A crucial lesson is that RAII should never erase the real release protocol. The wrapper must call the correct function for the exact resource:

- `delete` for single-object heap storage,
- `delete[]` for dynamic arrays,
- `CloseHandle` for Windows kernel handles,
- `close()` for POSIX descriptors,
- `closesocket()` for Winsock sockets,
- `vkDestroy*` for Vulkan resources,
- `Release()` for COM interfaces.

This is why generic ownership is useful, but resource-specific wrappers are often even better.

6.2.7 What should not be modeled as owning RAII blindly

RAII is powerful, but not every access path is an owner.

Borrowed views

The following are usually non-owning and therefore should not be mistaken for owning wrappers:

- raw pointers used for observation,
- references,

- `std::span`,
- `std::string_view`,
- iterators,
- raw handles passed transiently without transfer of ownership.

Owner versus borrower example

```
#include <vector>
#include <span>

class PacketStore
{
    std::vector<unsigned char> bytes_;

public:
    explicit PacketStore(std::vector<unsigned char> bytes)
        : bytes_(std::move(bytes))
    {
    }

    std::span<const unsigned char> bytes() const noexcept
    {
        return bytes_;
    }
};
```

Here the vector owns; the span borrows.

6.2.8 Design checklist

When deciding whether some resource should be wrapped with RAII, ask the following questions:

1. Is there a matching acquire/release API?
2. Can forgetting release leak memory, kernel objects, descriptors, or driver resources?
3. Can an exception or early return bypass cleanup?
4. Is there a natural single owner or explicit ownership group?
5. Can cleanup be made unconditional and idempotent through destructor logic?

If the answer is mostly yes, RAII is very likely the correct design.

6.2.9 Extensive examples

Example 1: memory wrapper

```
#include <memory>
#include <string>

class UserProfile
{
    std::unique_ptr<std::string> name_;

public:
    explicit UserProfile(std::string name)
        : name_(std::make_unique<std::string>(std::move(name)))
    {
```

```
    }

    const std::string& name() const noexcept
    {
        return *name_;
    }
};
```

Example 2: Windows HANDLE wrapper usage

```
#ifdef _WIN32
#include <windows.h>

int main()
{
    WinHandle file(CreateFileW(
        L"data.bin",
        GENERIC_READ,
        FILE_SHARE_READ,
        nullptr,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        nullptr));

    if (!file)
        return 1;

    return 0;
}
#endif
```

Example 3: POSIX descriptor wrapper usage

```
#if !defined(_WIN32)
#include <fcntl.h>

int main()
{
    FileDescriptor fd(::open("data.txt", O_RDONLY));
    if (!fd)
        return 1;

    return 0;
}
#endif
```

Example 4: Windows socket session plus socket wrapper

```
#ifdef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>

int main()
{
    WinsockSession ws;
    if (!ws.ok())
        return 1;

    WinSocket sock(::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP));
    if (!sock)
        return 1;
```

```
    return 0;
}
#endif
```

Example 5: Vulkan buffer owner skeleton

```
#include <vulkan/vulkan.h>

class MeshBuffers
{
    VulkanBuffer vertices_;
    VulkanBuffer indices_;

public:
    MeshBuffers(VulkanBuffer v, VulkanBuffer i) noexcept
        : vertices_(std::move(v)),
          indices_(std::move(i))
    {
    }
};
```

6.2.10 Windows build notes

The following commands are suitable for testing the examples on Windows.

MSVC for ordinary RAII and standard library examples

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo example.cpp
cl /std:c++23 /EHsc /W4 /O2 /permissive- /analyze /nologo example.cpp
```

MSVC for Winsock examples

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo winsock_example.cpp ws2_32.lib
```

MSVC for Win32 HANDLE examples

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo winhandle_example.cpp
```

Practical Windows guidance

For Windows-oriented RAII code:

1. use `std::make_unique` for heap ownership,
2. create dedicated wrappers for `HANDLE`, `SOCKET`, and COM interfaces instead of mixing release APIs,
3. remember that `SOCKET` uses `closesocket`, not `CloseHandle`,
4. compile with warnings and code analysis enabled,
5. keep wrappers move-only unless shared ownership is explicitly required.

6.2.11 Concluding rules

The real scope of RAII is much broader than memory. Any resource with explicit acquisition and explicit release is a candidate for RAII, and the best modern designs make this ownership visible in types.

1. Memory is RAII-eligible through smart pointers and containers.
2. Windows kernel handles are RAII-eligible through wrappers that call `CloseHandle`.
3. POSIX file descriptors are RAII-eligible through wrappers that call `close()`.

4. Network sockets are RAII-eligible, but their release API is platform-specific.
5. GPU and graphics handles are RAII-eligible because their APIs use explicit creation and destruction functions.
6. Non-owning views such as `std::span` are complementary to RAII, not replacements for ownership.

The most important design lesson is simple:

> If a resource must be released, that responsibility should usually belong to a destructor-managed owner object rather than to scattered manual cleanup logic.

Exception Safety in Resource Management

7.1 Levels of Exception Safety

Exception safety is a fundamental concept in Modern C++ resource management and library design. It defines the guarantees a program provides when an exception interrupts normal execution. Because C++ supports stack unwinding and deterministic destruction, resource management mechanisms such as RAII interact directly with exception propagation.
Exception safety describes how a program behaves when operations fail. Instead of attempting to eliminate exceptions entirely, C++ encourages designing operations that maintain program correctness even when exceptions occur.
In Modern C++ design, exception guarantees are typically categorized into three well-known levels:

1. the Basic Guarantee,
2. the Strong Guarantee,
3. the No-throw Guarantee.

These guarantees define increasingly stronger assurances about program state and resource safety.
Exception-safe design relies heavily on RAII, deterministic destruction, move semantics, and careful separation between resource acquisition and state mutation.

7.1.1 Basic guarantee

The basic exception safety guarantee ensures that when an exception occurs:

- no resources are leaked,
- all program invariants remain valid,
- objects remain in a usable but possibly modified state.

This means the operation may fail partially, but the program remains stable and safe.
The basic guarantee is the minimum acceptable level of exception safety for most systems code.

Characteristics of the basic guarantee

When a function provides the basic guarantee:

1. memory and other resources remain correctly managed,
2. class invariants remain valid,
3. objects may have changed state.

This is commonly used in operations where rollback would be expensive or unnecessary.

Example: basic guarantee in a container-like operation

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

void append_value(std::vector<int>& v, int value)
{
    v.push_back(value);
}
```

If push_back throws (for example due to allocation failure):

- the vector remains valid,
- previously stored elements remain intact,
- the operation simply fails.

The container state may be partially changed internally during the attempt, but invariants remain preserved.

Example: maintaining invariants with RAII

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

class Storage
{
    std::vector<std::unique_ptr<int>> items_;

public:

    void add(int value)
    {
        items_.push_back(std::make_unique<int>(value));
    }
};
```

Even if allocation fails:

- no memory leaks occur,

- already inserted elements remain valid,
- the container remains usable.

RAII ensures that partially constructed objects are destroyed automatically.

Manual cleanup without RAII

Before RAII became standard practice, programs relied on manual cleanup logic.

```
#include <cstdlib>

void legacy()
{
    int* p = (int*)std::malloc(sizeof(int));

    if (!p)
        return;

    *p = 42;

    std::free(p);
}
```

If an exception occurred between allocation and free, the memory could leak.
RAII eliminates this problem by binding resource lifetime to object scope.

7.1.2 Strong guarantee

The strong exception guarantee provides a stronger promise than the basic guarantee. It ensures that operations are transactional.
If an operation fails:

- the program state remains exactly as it was before the operation began,
- no partial modifications are visible.

This guarantee is often described as the *commit-or-rollback* model.
Either the operation succeeds completely, or the system behaves as though it never started.

Typical implementation technique

The strong guarantee is commonly implemented through temporary objects and swap operations.
The general strategy is:

1. perform work on temporary objects,
2. if all operations succeed, commit the result,
3. if an exception occurs, discard the temporary state.

Example: strong guarantee with copy-and-swap

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

void replace_contents(std::vector<int>& target,
                      const std::vector<int>& source)
{
    std::vector<int> temp = source;

    target.swap(temp);
}
```

If copying `source` fails, the original `target` remains unchanged.
Only after successful copying does the swap commit the new state.

Strong guarantee in assignment operators

The copy-and-swap idiom is commonly used in assignment operators.

```
#include <algorithm>

class Buffer
{
    int* data_;
    std::size_t size_;

public:

    Buffer(std::size_t n)
        : data_(new int[n]), size_(n)
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }

    Buffer(const Buffer& other)
        : data_(new int[other.size_]),
          size_(other.size_)
    {
        std::copy(other.data_, other.data_ + size_, data_);
    }
```

```
    void swap(Buffer& other) noexcept
    {
        std::swap(data_, other.data_);
        std::swap(size_, other.size_);
    }

    Buffer& operator=(Buffer other)
    {
        swap(other);
        return *this;
    }
};
```

Here the assignment operator receives its argument by value, creating a temporary copy. If copying fails, the assignment never begins. If copying succeeds, the swap commits the new state.

Strong guarantee with containers

Many operations in the standard library aim to provide the strong guarantee when possible. For example:

- vector reallocation strategies,
- container insertion operations,
- string modification operations.

Move semantics significantly improve the ability of containers to provide the strong guarantee efficiently.

Example using move semantics

```
#include <string>
#include <utility>

void safe_update(std::string& target,
                 std::string new_value)
{
    target.swap(new_value);
}
```

The new value is constructed first. If construction fails, the original string remains unchanged.

7.1.3 No-throw guarantee

The no-throw guarantee is the strongest exception safety level. It promises that an operation will never throw an exception.
Functions that provide this guarantee must complete successfully under all circumstances.
This guarantee is particularly important for:

- destructors,
- swap operations,
- move constructors,
- move assignment operators.

The C++ language supports the no-throw guarantee through the `noexcept` specifier.

Destructors and exception safety

Destructors must never allow exceptions to escape.
If a destructor throws during stack unwinding, the program terminates.
Therefore destructors are expected to behave as no-throw operations.

```
class Resource
{
public:

    ~Resource() noexcept
    {
    }
};
```

No-throw swap operations

Swap functions are typically required to be no-throw.

```
#include <utility>

class Point
{
    int x_;
    int y_;

public:

    Point(int x, int y) : x_(x), y_(y) {}

    void swap(Point& other) noexcept
```

```
    {
        std::swap(x_, other.x_);
        std::swap(y_, other.y_);
    }
};
```

No-throw swap enables strong exception guarantees in algorithms.

No-throw move constructors

Move constructors should usually be declared `noexcept`. This allows containers such as `std::vector` to move elements safely during reallocation.

```
#include <memory>

class Holder
{
    std::unique_ptr<int> ptr_;

public:

    Holder(std::unique_ptr<int> p)
        : ptr_(std::move(p))
    {
    }

    Holder(Holder&& other) noexcept
        : ptr_(std::move(other.ptr_))
    {
    }
};
```

If move operations are guaranteed not to throw, containers can rely on them to maintain strong guarantees during structural changes.

No-throw functions in system code

Low-level system code often requires no-throw guarantees to maintain stability during cleanup or shutdown.
Examples include:

- resource release functions,
- mutex unlock operations,
- memory deallocation,
- swap operations used during commit.

7.1.4 Relationship between RAII and exception guarantees

RAII is the foundation of exception-safe programming in C++.
Because destructors run automatically during stack unwinding:

- resources are released safely,
- partially constructed objects are cleaned up,
- invariants remain valid.

This enables the construction of reliable exception guarantees.
For example:

- RAII provides the basic guarantee automatically for many resource types,
- copy-and-swap combined with RAII enables the strong guarantee,
- noexcept operations enable no-throw guarantees.

7.1.5 Combining guarantees in real systems

Different functions within a program may provide different levels of guarantees depending on performance and design constraints.
Typical patterns include:

- destructors provide no-throw guarantees,
- container insertions aim for strong guarantees,
- many algorithms provide the basic guarantee.

This layered approach balances safety with efficiency.

7.1.6 Example: layered guarantees

```
#include <vector>

class DataSet
{
    std::vector<int> values_;

public:

    void add(int v)
    {
        values_.push_back(v);
    }

    void replace_all(std::vector<int> new_values)
    {
```

```
        values_.swap(new_values);
    }

    void clear() noexcept
    {
        values_.clear();
    }
};
```

In this example:

- `add()` provides the basic guarantee,
- `replace_all()` provides the strong guarantee,
- `clear()` is a no-throw operation.

7.1.7 Windows compilation guide

Compile the examples using the following commands.

MSVC

```
cl /std:c++23 /EHsc /W4 example.cpp
```

Clang

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp
```

GCC

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp
```

7.1.8 Summary

Exception safety guarantees define how programs behave when operations fail. The three levels of guarantees establish increasing degrees of reliability.

- The basic guarantee ensures no resource leaks and preserved invariants.
- The strong guarantee ensures transactional behavior with rollback.
- The no-throw guarantee ensures operations never emit exceptions.

Together with RAII and deterministic destruction, these guarantees form the foundation of safe resource management in Modern C++ systems programming.

7.2 Designing Exception-Safe Classes

Exception-safe class design is a central responsibility in Modern C++ systems programming. When a class manages resources or modifies internal state, operations must remain correct even if exceptions interrupt execution. A well-designed class preserves invariants, prevents resource leaks, and leaves objects in a valid state regardless of failure conditions.
The C++ language provides powerful mechanisms for writing exception-safe code:

- deterministic destruction,
- RAII resource management,
- move semantics,
- strong ownership models,
- scope-bound guards.

Exception-safe class design typically relies on three important implementation patterns:

1. the copy-and-swap idiom,
2. the commit/rollback pattern,
3. scoped guard objects.

Each pattern allows a class to provide strong guarantees about object integrity during operations that may throw.

7.2.1 Copy-and-swap

The copy-and-swap idiom is a classical technique used to implement exception-safe assignment operators. The central idea is to perform potentially failing work on a temporary object before modifying the original object.
If the temporary construction succeeds, the program commits the change by swapping internal state. If an exception occurs during temporary construction, the original object remains unchanged.
This pattern naturally provides the strong exception guarantee.

Basic structure of copy-and-swap

The typical implementation involves:

1. copying the source object into a temporary,
2. swapping the internal state of the temporary with the current object,
3. letting the temporary destroy the old state.

Because the swap operation should not throw, the final commit step is safe.

Example: buffer class with copy-and-swap

```
#include <algorithm>
#include <cstddef>

class Buffer
{
    int* data_;
    std::size_t size_;

public:

    explicit Buffer(std::size_t size)
        : data_(new int[size]), size_(size)
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }

    Buffer(const Buffer& other)
        : data_(new int[other.size_]),
          size_(other.size_)
    {
        std::copy(other.data_, other.data_ + size_, data_);
    }

    void swap(Buffer& other) noexcept
```

```
    {
        std::swap(data_, other.data_);
        std::swap(size_, other.size_);
    }

    Buffer& operator=(Buffer other)
    {
        swap(other);
        return *this;
    }
};
```

In this design:

- the parameter is passed by value, creating a temporary copy,
- the swap commits the new state,
- the temporary object destroys the old state automatically.

If copying fails, the assignment never begins and the original object remains unchanged.

Advantages of copy-and-swap

The copy-and-swap idiom provides several benefits:

- strong exception safety,
- simplified assignment logic,
- automatic cleanup through RAII,
- reduced code duplication.

It also integrates naturally with move semantics.

Example using move semantics

```
#include <utility>
#include <string>

class Text
{
    std::string value_;

public:

    Text(std::string value)
        : value_(std::move(value))
    {
    }

    void swap(Text& other) noexcept
    {
        value_.swap(other.value_);
    }

    Text& operator=(Text other)
    {
        swap(other);
        return *this;
    }
};
```

Here, the temporary object may be created through copy or move construction depending on the argument type.

7.2.2 Commit/rollback pattern

The commit/rollback pattern generalizes the strong exception guarantee to more complex operations. Instead of modifying an object directly, the operation performs its work on temporary state and commits the changes only when all steps succeed.
If an exception occurs before the commit step, the temporary state is discarded and the original object remains unchanged.
This pattern is widely used in container implementations and database-like operations.

Conceptual steps

The commit/rollback strategy follows a clear sequence:

1. create temporary state,
2. perform operations on the temporary state,
3. commit changes to the original object,
4. discard temporary resources.

Example: replacing container contents

```
#include <vector>

class DataSet
{
    std::vector<int> data_;

public:

    void replace(std::vector<int> new_data)
```

```
    {
        data_.swap(new_data);
    }
};
```

In this example:

- the new data is prepared before modifying the object,
- the swap operation commits the new state,
- the temporary object cleans up automatically.

Example: transactional update

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

class Ledger
{
    std::vector<int> entries_;

public:

    void add_entries(const std::vector<int>& new_entries)
    {
        std::vector<int> temp = entries_;

        for (int value : new_entries)
        {
            if (value < 0)
```

```
                throw std::runtime_error("invalid entry");

            temp.push_back(value);
        }

        entries_.swap(temp);
    }
};
```

If an exception occurs while processing the entries, the temporary vector is destroyed and the original ledger remains unchanged.

Commit step must not throw

A critical rule of this pattern is that the commit operation must not throw exceptions. Swap operations are commonly used because they are typically noexcept.

Commit/rollback in resource management

The pattern also applies to resource acquisition scenarios.

```
#include <memory>

class ResourceOwner
{
    std::unique_ptr<int> resource_;

public:

    void reset_resource(int value)
    {
```

```
        auto new_resource = std::make_unique<int>(value);

        resource_.swap(new_resource);
    }
};
```

Here the new resource is allocated first. Only after successful allocation does the object replace the previous resource.

7.2.3 Scoped guards

Scoped guards are RAII objects that enforce cleanup actions when a scope ends. They are particularly useful for maintaining invariants and ensuring rollback when partial operations fail. A scoped guard executes a cleanup operation automatically in its destructor unless the operation is explicitly cancelled.

Simple scoped guard implementation

```
#include <functional>

class ScopeGuard
{
    std::function<void()> action_;
    bool active_;

public:

    explicit ScopeGuard(std::function<void()> action)
        : action_(std::move(action)), active_(true)
    {
```

```
    }

    ~ScopeGuard()
    {
        if (active_)
            action_();
    }

    void dismiss() noexcept
    {
        active_ = false;
    }
};
```

This guard executes a stored cleanup function when it leaves scope.

Example: rollback guard

```
#include <vector>

void safe_insert(std::vector<int>& v, int value)
{
    std::size_t old_size = v.size();

    ScopeGuard rollback([&]()
    {
        v.resize(old_size);
    });

    v.push_back(value);
```

```
    rollback.dismiss();
}
```

If push_back throws an exception, the guard restores the original vector size.

Resource guard example

Scoped guards can also manage resource cleanup.

```
#include <cstdio>

void example()
{
    FILE* file = std::fopen("data.txt", "w");

    if (!file)
        return;

    ScopeGuard cleanup([&]()
    {
        std::fclose(file);
    });

}
```

The guard ensures that the file is closed even if an exception occurs.

Mutex guard example

The standard library provides ready-made scoped guards for synchronization.

```
#include <mutex>

std::mutex global_mutex;

void critical_section()
{
    std::lock_guard<std::mutex> lock(global_mutex);

}
```

The `lock_guard` class automatically unlocks the mutex when leaving scope.

Advanced guard with conditional commit

```
#include <iostream>

class TransactionGuard
{
    bool committed_;

public:

    TransactionGuard() : committed_(false)
    {
        std::cout << "Transaction started\n";
    }

    ~TransactionGuard()
    {
        if (!committed_)
```

```
            std::cout << "Transaction rolled back\n";
    }

    void commit()
    {
        committed_ = true;
        std::cout << "Transaction committed\n";
    }
};
```

Usage:

```
void process()
{
    TransactionGuard tx;

    tx.commit();
}
```

If an exception occurs before `commit()`, the destructor performs the rollback automatically.

7.2.4 Design guidelines

When designing exception-safe classes, several best practices should be followed:

1. Acquire resources using RAII wrappers.
2. Separate resource acquisition from state mutation.
3. Use copy-and-swap for assignment operators.
4. Prefer commit/rollback strategies for complex state updates.

5. Use scoped guards to maintain invariants during operations.

6. Mark destructors and swap functions as noexcept.

7. Design move operations to avoid throwing exceptions when possible.

7.2.5 Example: complete exception-safe class

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

class SafeVector
{
    std::vector<int> data_;

public:

    void add(int value)
    {
        data_.push_back(value);
    }

    void replace_all(const std::vector<int>& values)
    {
        std::vector<int> temp = values;

        data_.swap(temp);
    }

    void clear() noexcept
```

```
    {
        data_.clear();
    }
};
```

This class demonstrates multiple exception-safety strategies:

- push_back provides the basic guarantee,
- replace_all provides the strong guarantee,
- clear is a no-throw operation.

7.2.6 Windows compilation guide

Compile the examples using the following commands.

MSVC

```
cl /std:c++23 /EHsc /W4 example.cpp
```

Clang

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp
```

GCC

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp
```

7.2.7 Summary

Exception-safe class design relies on combining RAII with carefully structured operations that preserve invariants during failures.

- Copy-and-swap simplifies assignment and provides strong guarantees.
- Commit/rollback separates temporary state from final state changes.
- Scoped guards ensure cleanup and invariant restoration.

Together, these techniques enable robust and reliable class implementations that remain correct even in the presence of exceptions.

Advanced RAII Patterns

8.1 Monadic Resource Patterns

Monadic resource patterns are an advanced style of Modern C++ design in which resource-oriented operations are composed as pipelines rather than written as deeply nested control-flow trees. The idea is not to replace RAII, but to combine RAII-managed objects with composable result types such as `std::optional` and `std::expected`.
This style became much more practical in Modern C++ because the standard library now provides monadic operations for vocabulary types. In current standard wording, `std::optional` supports `and_then`, `transform`, and `or_else`; `std::expected` also provides monadic operations that allow success paths and error paths to be composed without writing repetitive branching logic.
For resource management, this is important because many operations follow the same shape:

1. acquire or open a resource,
2. transform or refine that resource,
3. perform more work only if the previous step succeeded,
4. propagate errors without manually repeating cleanup logic,
5. rely on RAII destructors for rollback and release.

This leads to three related advanced patterns:

1. resource pipelines,
2. deferred operations,
3. RAII-based transactions.

8.1.1 Resource pipelines

A resource pipeline is a chain of operations where each step receives either a valid value or a failure state and either continues the computation or propagates the failure unchanged. In C++23 and later standard-library style, this is most naturally expressed with `std::expected` or, when no error payload is needed, with `std::optional`.

Why pipelines matter in resource code

Low-level and systems code often has this repetitive form:

```
auto r1 = acquire();
if (!r1) return error;

auto r2 = step2(*r1);
if (!r2) return error;

auto r3 = step3(*r2);
if (!r3) return error;

return finish(*r3);
```

This works, but it can become noisy and repetitive. Monadic operations let the code express the same semantics more directly:

```
return acquire()
    .and_then(step2)
    .and_then(step3)
    .transform(finish);
```

The major advantage is not merely brevity. It is that success flow and failure propagation become explicit in the type structure, while RAII continues to handle cleanup automatically for every temporary resource that leaves scope.

Pipeline semantics with std::expected

For `std::expected<T, E>`:

- `and_then` continues only if a value is present and expects the next function to return another `expected`,
- `transform` maps the success value into another value,
- `or_else` handles the error path and returns another `expected`.

This makes `expected` particularly suitable for resource-oriented APIs where failures need explanatory payloads such as error codes, messages, or domain-specific status objects.

A small RAII file wrapper used in pipelines

The following wrapper is intentionally simple. It demonstrates a move-only RAII resource that can participate in pipeline-based APIs.

```
#include <cstdio>
#include <string>
#include <utility>
```

```
class FileHandle
{
    std::FILE* file_;

public:
    FileHandle() noexcept
        : file_(nullptr)
    {
    }

    explicit FileHandle(std::FILE* f) noexcept
        : file_(f)
    {
    }

    ~FileHandle()
    {
        if (file_)
            std::fclose(file_);
    }

    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    FileHandle(FileHandle&& other) noexcept
        : file_(other.file_)
    {
        other.file_ = nullptr;
    }
```

```
    FileHandle& operator=(FileHandle&& other) noexcept
    {
        if (this != &other)
        {
            if (file_)
                std::fclose(file_);
            file_ = other.file_;
            other.file_ = nullptr;
        }
        return *this;
    }

    std::FILE* get() const noexcept
    {
        return file_;
    }

    explicit operator bool() const noexcept
    {
        return file_ != nullptr;
    }
};
```

Pipeline-ready API with std::expected

```
#include <expected>
#include <string>
#include <vector>
#include <cstdio>
```

```
enum class IOError
{
    OpenFailed,
    ReadFailed,
    ParseFailed
};

std::expected<FileHandle, IOError> open_text_file(const char* path)
{
    if (std::FILE* f = std::fopen(path, "r"))
        return FileHandle(f);

    return std::unexpected(IOError::OpenFailed);
}

std::expected<std::string, IOError> read_all(FileHandle& file)
{
    std::string text;
    char buffer[256];

    while (std::fgets(buffer, sizeof(buffer), file.get()))
        text += buffer;

    if (std::ferror(file.get()))
        return std::unexpected(IOError::ReadFailed);

    return text;
}
```

```
std::expected<std::vector<std::string>, IOError> split_lines(std::string text)
{
    std::vector<std::string> lines;
    std::string current;

    for (char ch : text)
    {
        if (ch == '\n')
        {
            lines.push_back(current);
            current.clear();
        }
        else
        {
            current.push_back(ch);
        }
    }

    if (!current.empty())
        lines.push_back(current);

    return lines;
}
```

Classic nested control flow

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

```
std::expected<std::vector<std::string>, IOError>
load_lines_classic(const char* path)
{
    auto file = open_text_file(path);
    if (!file)
        return std::unexpected(file.error());

    auto text = read_all(*file);
    if (!text)
        return std::unexpected(text.error());

    auto lines = split_lines(std::move(*text));
    if (!lines)
        return std::unexpected(lines.error());

    return lines;
}
```

Monadic pipeline form

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

std::expected<std::vector<std::string>, IOError>
load_lines_pipeline(const char* path)
{
    return open_text_file(path)
        .and_then([](FileHandle& file)
```

```
        {
            return read_all(file);
        })
        .and_then([](std::string text)
        {
            return split_lines(std::move(text));
        });
}
```

This is the core monadic resource pattern. Notice what happens to the `FileHandle`:

- if opening fails, the pipeline stops immediately,
- if reading fails, the pipeline stops immediately,
- when the `FileHandle` leaves the current stage, its destructor still closes the file automatically.

The monadic style changes control flow, not ownership discipline. RAII remains the mechanism that actually releases the resource.

Using transform for success-value mapping

```
#include <expected>
#include <string>
#include <vector>
#include <cstddef>

std::expected<std::size_t, IOError> count_lines(const char* path)
{
    return load_lines_pipeline(path)
```

```
        .transform([](const std::vector<std::string>& lines)
        {
            return lines.size();
        });
}
```

Here the pipeline first produces a vector of lines, then `transform` maps that successful value into the final count.

Using or_else for recovery or translation

```
#include <expected>

std::expected<std::vector<std::string>, IOError>
load_lines_or_empty(const char* path)
{
    return load_lines_pipeline(path)
        .or_else([](IOError)
        {
            return std::expected<std::vector<std::string>, IOError>(
                std::vector<std::string>{}
            );
        });
}
```

This example shows how an error path can be transformed into an alternate result. In real production code, one would usually preserve or translate the error more carefully, but the structural idea is important: monadic APIs let resource pipelines describe both success flow and recovery flow declaratively.

Optional-based pipelines

When the failure path does not need an explanatory error value, `std::optional` can be used in the same style.

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

std::optional<std::string> trim(std::string s)
{
    while (!s.empty() && std::isspace(static_cast<unsigned char>(s.front())))
        s.erase(s.begin());

    while (!s.empty() && std::isspace(static_cast<unsigned char>(s.back())))
        s.pop_back();

    if (s.empty())
        return std::nullopt;

    return s;
}

std::optional<int> to_int(const std::string& s)
{
    try
    {
        return std::stoi(s);
    }
    catch (...)
```

```
    {
        return std::nullopt;
    }
}

std::optional<int> parse_trimmed_int(std::string text)
{
    return trim(std::move(text))
        .and_then([](std::string s)
        {
            return to_int(s);
        })
        .transform([](int x)
        {
            return x * 2;
        });
}
```

This demonstrates the same monadic structure without an explicit error object.

Design rules for resource pipelines

1. Use `std::expected` when failure information matters.
2. Use `std::optional` when the absence of a value is enough.
3. Let RAII wrappers continue to own acquisition and release.
4. Treat monadic chaining as a control-flow layer above ownership, not a replacement for ownership.
5. Keep each pipeline stage small and domain-specific.

8.1.2 Deferred operations

Deferred operations are actions scheduled to happen later, usually at scope exit, unless they are explicitly cancelled. This is one of the most elegant advanced RAII patterns because it allows cleanup, rollback, logging, restoration, and protocol completion code to be written close to the corresponding acquisition point.
In conceptual terms, a deferred operation is:

> an action object whose destructor performs work if the action remains active.

This pattern is often described informally as a scope guard or a deferred cleanup object.

Why deferred operations matter

Many real operations require a temporary obligation:

- restore a flag when leaving scope,
- unlock a subsystem after a multi-step operation,
- erase a partially inserted entry unless commit succeeds,
- close a raw resource acquired from a C API,
- record failure unless success is explicitly confirmed.

These are not quite the same as ordinary object ownership. Instead, they are scoped protocol obligations, and RAII is a perfect fit for them.

A move-only deferred guard

```
#include <utility>
#include <type_traits>

template<typename F>
class ScopeGuard
{
    F func_;
    bool active_;

public:
    explicit ScopeGuard(F f) noexcept(std::is_nothrow_move_constructible_v<F>)
        : func_(std::move(f)), active_(true)
    {
    }

    ~ScopeGuard() noexcept
    {
        if (active_)
            func_();
    }

    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;

    ScopeGuard(ScopeGuard&& other)
    ↪   noexcept(std::is_nothrow_move_constructible_v<F>)
        : func_(std::move(other.func_)), active_(other.active_)
    {
```

```
        other.active_ = false;
    }

    ScopeGuard& operator=(ScopeGuard&&) = delete;

    void release() noexcept
    {
        active_ = false;
    }
};

template<typename F>
ScopeGuard<F> make_scope_guard(F f)
{
    return ScopeGuard<F>(std::move(f));
}
```

This class provides the essential semantics:

- it is active when created,
- its destructor runs the stored function if still active,
- `release()` cancels the deferred action,
- move transfer preserves one active guard, not two.

Example: restoring a flag

```
#include <iostream>

class Engine
```

```
{
    bool busy_{false};

public:
    void perform_step()
    {
        busy_ = true;
        auto restore = make_scope_guard([this]() noexcept
        {
            busy_ = false;
        });

        std::cout << "working\n";

        restore.release();
        busy_ = false;
    }

    bool busy() const noexcept
    {
        return busy_;
    }
};
```

A more realistic implementation would leave the guard active until the end unless early dismissal is truly needed, but this example shows how state restoration is attached directly to scope lifetime.

Example: raw resource cleanup on all paths

```
#include <cstdio>

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

    auto cleanup = make_scope_guard([&]() noexcept
    {
        std::fclose(f);
    });

    std::fputs(msg, f);
}
```

Even though ordinary RAII wrappers are usually preferable, this example shows the basic deferred-action idea clearly: cleanup happens on every path unless explicitly released.

Example: rollback of a partial insertion

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

void push_checked(std::vector<int>& values, int x)
{
    values.push_back(x);

    auto rollback = make_scope_guard([&]() noexcept
```

```
    {
        values.pop_back();
    });

    if (x < 0)
        throw std::runtime_error("negative value not allowed");

    rollback.release();
}
```

This example illustrates an important advanced point: deferred guards are excellent for local rollback steps inside larger algorithms.

Deferred operations and destructor rules

Because destructors run during stack unwinding, deferred guards naturally participate in exception handling. However, their cleanup actions should themselves not throw. A throwing cleanup during unwinding would risk program termination. Therefore the best practice is:

1. make deferred cleanup actions noexcept when possible,
2. if cleanup can fail, convert failure to logging or error-state recording instead of throwing from the destructor path.

Deferred actions as protocol objects

A deferred operation is not only for cleanup. It can also encode protocol completion.

```
#include <iostream>

void process_request()
```

```
{
    std::cout << "begin request\n";

    auto finish = make_scope_guard([]() noexcept
    {
        std::cout << "end request\n";
    });

    std::cout << "processing...\n";
}
```

This pattern keeps the paired begin/end operations structurally close, which is often clearer than writing the second action at the bottom of the function and hoping every path reaches it.

8.1.3 RAII-based transactions

RAII-based transactions extend the deferred-operation idea into a larger commit-or-rollback design. The core principle is:

> the object begins a transaction when constructed, and unless commit is explicitly recorded, its destructor rolls the transaction back.

This is one of the strongest advanced RAII patterns because it combines:

- deterministic destruction,
- exception safety,
- explicit commit semantics,
- automatic rollback on all non-commit paths.

The shape of a transaction guard

A transaction guard usually holds:

1. a reference or pointer to the controlled resource or state,
2. the pre-transaction state or a rollback recipe,
3. a committed flag.

Simple transactional state object

```
#include <string>
#include <utility>

class Config
{
    std::string value_;

public:
    explicit Config(std::string v = {})
        : value_(std::move(v))
    {
    }

    const std::string& value() const noexcept
    {
        return value_;
    }

    void set_value(std::string v)
```

```
    {
        value_ = std::move(v);
    }
};
```

RAII transaction wrapper with rollback

```
#include <string>
#include <utility>

class ConfigTransaction
{
    Config& target_;
    std::string old_value_;
    bool committed_;

public:
    explicit ConfigTransaction(Config& cfg)
        : target_(cfg),
          old_value_(cfg.value()),
          committed_(false)
    {
    }

    ~ConfigTransaction() noexcept
    {
        if (!committed_)
            target_.set_value(std::move(old_value_));
    }
```

```
    ConfigTransaction(const ConfigTransaction&) = delete;
    ConfigTransaction& operator=(const ConfigTransaction&) = delete;

    void assign(std::string v)
    {
        target_.set_value(std::move(v));
    }

    void commit() noexcept
    {
        committed_ = true;
    }
};
```

Using the transaction

```
#include <stdexcept>

void update_config(Config& cfg, std::string text)
{
    ConfigTransaction tx(cfg);

    tx.assign(std::move(text));

    if (cfg.value() == "bad")
        throw std::runtime_error("invalid configuration");

    tx.commit();
}
```

If an exception occurs before `commit()`, the destructor restores the previous value automatically.

Why this works

This pattern works because the destructor is guaranteed to run during normal scope exit and during exception-driven stack unwinding. That means the rollback path is encoded in object lifetime itself.

Transaction with staged commit

A stronger version avoids mutating the target until all work is complete. This is often a better design because it naturally provides the strong exception guarantee.

```
#include <string>
#include <utility>

class ConfigBuilderTransaction
{
    Config& target_;
    std::string staged_;
    bool committed_;

public:
    explicit ConfigBuilderTransaction(Config& cfg)
        : target_(cfg),
          staged_(cfg.value()),
          committed_(false)
    {
    }
```

```
    ~ConfigBuilderTransaction() noexcept = default;

    ConfigBuilderTransaction(const ConfigBuilderTransaction&) = delete;
    ConfigBuilderTransaction& operator=(const ConfigBuilderTransaction&) = delete;

    void assign(std::string v)
    {
        staged_ = std::move(v);
    }

    void append_suffix(std::string suffix)
    {
        staged_ += suffix;
    }

    void commit()
    {
        target_.set_value(std::move(staged_));
        committed_ = true;
    }

    bool committed() const noexcept
    {
        return committed_;
    }
};
```

Usage:

```
#include <stdexcept>
```

```
void build_and_commit(Config& cfg)
{
    ConfigBuilderTransaction tx(cfg);

    tx.assign("base");
    tx.append_suffix("_final");

    if (false)
        throw std::runtime_error("simulated failure");

    tx.commit();
}
```

In this version the original object remains unchanged until the explicit commit step.

RAII transactions for multiple resources

Transactional RAII becomes especially valuable when several state changes must succeed together.

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

class Ledger
{
    std::vector<int> entries_;

public:
    void push(int v)
```

```
    {
        entries_.push_back(v);
    }

    void pop() noexcept
    {
        entries_.pop_back();
    }

    const std::vector<int>& entries() const noexcept
    {
        return entries_;
    }
};

void append_pair(Ledger& ledger, int a, int b)
{
    ledger.push(a);
    auto undo_a = make_scope_guard([&]() noexcept
    {
        ledger.pop();
    });

    ledger.push(b);
    auto undo_b = make_scope_guard([&]() noexcept
    {
        ledger.pop();
    });
```

```
    if (b < 0)
        throw std::runtime_error("second entry invalid");

    undo_b.release();
    undo_a.release();
}
```

This is a transaction built from local rollback guards. It shows that RAII transactions can be implemented either as one dedicated transaction class or as a carefully layered set of scoped guards.

Monadic transactions with expected

Transactions also combine naturally with `std::expected`. A function can stage work through a transaction object and return either success or an error, while the destructor guarantees rollback if the function exits early with an error.

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

enum class ConfigError
{
    EmptyValue,
    ForbiddenValue
};

std::expected<void, ConfigError> validate_and_commit(Config& cfg, std::string
↪   value)
{
    ConfigBuilderTransaction tx(cfg);
```

```
    if (value.empty())
        return std::unexpected(ConfigError::EmptyValue);

    if (value == "forbidden")
        return std::unexpected(ConfigError::ForbiddenValue);

    tx.assign(std::move(value));
    tx.commit();

    return {};
}
```

The transaction object does not need manual rollback code in each failure path. Early returns and exceptions both remain safe because rollback is still tied to destruction.

Design rules for RAII transactions

1. Begin the transaction in construction.
2. Make commit explicit and easy to see.
3. Ensure rollback is safe in the destructor path.
4. Prefer staging changes first when strong exception safety is required.
5. Keep rollback logic noexcept whenever possible.

8.1.4 Comparing the three patterns

These three advanced patterns are closely related but solve slightly different problems.

Resource pipelines

- best for chaining success/failure operations,
- natural with `std::expected` and `std::optional`,
- excellent for multi-step acquisition and transformation flows.

Deferred operations

- best for local cleanup and rollback obligations,
- natural with small move-only guard objects,
- excellent for restoring state, closing raw resources, and undoing local side effects.

RAII-based transactions

- best for commit-or-rollback semantics,
- natural when a larger unit of work must either fully succeed or revert,
- excellent for multi-step state changes and domain-specific consistency rules.

8.1.5 Windows build notes

The following commands are suitable for testing the examples on Windows.

MSVC

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo example.cpp
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

GCC on Windows via MSYS2 or MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

Practical Windows guidance

For current Windows-oriented Modern C++ work:

1. compile monadic `optional`/`expected` examples in a C++23 mode,
2. keep destructor-based guards non-throwing,
3. prefer dedicated RAII wrappers over raw handles,
4. use staged commit when transaction failure must leave the original state unchanged.

8.1.6 Concluding design rules

Monadic resource patterns do not replace RAII. They elevate it.

1. RAII still owns the real cleanup logic.
2. Monadic result types organize multi-step success/failure flow.
3. Deferred guards encode local obligations directly in scope.
4. Transaction wrappers encode rollback semantics directly in lifetime.
5. The most robust advanced code uses these techniques together rather than in isolation.

The deepest lesson is this:

> Resource safety in Modern C++ becomes especially powerful when ownership is managed by RAII, failure is represented in types, and multi-step work is composed as explicit pipelines and transactions rather than scattered manual cleanup code.

8.2 Multi-Resource Structures

In serious systems software, resources rarely appear in isolation. A network subsystem may own sockets, buffers, timers, and mutexes. A graphics subsystem may own a device handle, command resources, descriptor resources, synchronization primitives, and mapped memory. A compiler or interpreter may own source buffers, file handles, caches, synchronization objects, and temporary arenas. For this reason, advanced RAII design must scale beyond single-handle wrappers and address structures that manage multiple related resources at once.
A multi-resource structure is a class, graph, or aggregate that coordinates the lifetime of more than one resource while preserving invariants, release ordering, exception safety, and ownership clarity. The central design question is no longer merely:

> Who owns this one resource?

but rather:

> How should several resources with different lifetime roles be composed into one stable design?

This leads to three important advanced patterns:

1. RAII graphs,
2. resource slicing,
3. resource bundles.

These patterns are not alternatives to RAII. They are the natural extension of RAII into larger ownership topologies.

8.2.1 RAII graphs

A RAII graph is a set of objects connected by ownership and observation relationships, where every owned edge is represented by an owning RAII type and every observing edge is represented by a non-owning type. The word graph is important because real systems often contain trees, directed acyclic graphs, parent-child relationships, peer references, caches, back-references, and auxiliary views.
The first rule of a RAII graph is simple:

> Every edge in the graph must express the correct lifetime meaning.

That means:

- exclusive ownership should usually be expressed with value members or `std::unique_ptr`,
- shared ownership should be expressed only when truly required, typically with `std::shared_ptr`,
- observing links should usually be expressed with references, raw pointers, `std::weak_ptr`, `std::span`, or `std::string_view`, depending on the domain.

Tree-shaped RAII graphs

The most straightforward RAII graph is a tree. Each parent owns its children, and no child owns its parent. This maps naturally to exclusive ownership.

```
#include <memory>
#include <string>
#include <vector>
#include <utility>
```

```
class Node
{
    std::string name_;
    std::vector<std::unique_ptr<Node>> children_;

public:
    explicit Node(std::string name)
        : name_(std::move(name))
    {
    }

    Node& add_child(std::string name)
    {
        children_.push_back(std::make_unique<Node>(std::move(name)));
        return *children_.back();
    }

    const std::string& name() const noexcept
    {
        return name_;
    }

    const std::vector<std::unique_ptr<Node>>& children() const noexcept
    {
        return children_;
    }
};
```

This structure is naturally exception-safe:

- if child creation fails, the parent remains valid,
- already-owned children remain owned,
- destruction recursively releases the entire tree.

Why tree ownership is often ideal

Exclusive tree ownership is ideal when:

- one logical owner controls each subobject,
- destruction should cascade deterministically,
- subobjects should not outlive the root,
- aliasing of ownership is undesirable.

This is common for:

- abstract syntax trees,
- GUI widget hierarchies,
- command trees,
- ownership-oriented scene graphs,
- interpreter scopes and nested environments.

Parent links in a RAII graph

Many graphs are not pure trees because child objects often need access back to a parent. In such cases, the back-reference should usually be non-owning.

```
#include <memory>
#include <string>
#include <vector>
#include <utility>

class Node
{
    std::string name_;
    Node* parent_;
    std::vector<std::unique_ptr<Node>> children_;

public:
    explicit Node(std::string name, Node* parent = nullptr)
        : name_(std::move(name)), parent_(parent)
    {
    }

    Node& add_child(std::string name)
    {
        children_.push_back(std::make_unique<Node>(std::move(name), this));
        return *children_.back();
    }

    Node* parent() noexcept
    {
```

```
        return parent_;
    }

    const Node* parent() const noexcept
    {
        return parent_;
    }
};
```

Here:

- ownership flows downward through `std::unique_ptr`,
- the upward link is only borrowed,
- destruction remains deterministic and cycle-free.

Shared RAII graphs

Some domains require objects to be owned by multiple participants. In that case, `std::shared_ptr` may be appropriate, but only if shared lifetime is genuinely part of the model.

```
#include <memory>
#include <string>
#include <vector>
#include <utility>

class Asset
{
    std::string name_;
```

```
public:
    explicit Asset(std::string name)
        : name_(std::move(name))
    {
    }

    const std::string& name() const noexcept
    {
        return name_;
    }
};

class SceneObject
{
    std::shared_ptr<Asset> mesh_;
    std::shared_ptr<Asset> material_;

public:
    SceneObject(std::shared_ptr<Asset> mesh,
                std::shared_ptr<Asset> material)
        : mesh_(std::move(mesh)),
          material_(std::move(material))
    {
    }
};
```

This is a graph of shared owners. It is appropriate when the mesh and material must remain alive as long as any scene object still uses them.

Avoiding ownership cycles

The most important structural hazard in shared RAII graphs is the ownership cycle. If two objects hold `std::shared_ptr` to each other, neither can be destroyed.

```
#include <memory>

struct B;

struct A
{
    std::shared_ptr<B> b;
};

struct B
{
    std::shared_ptr<A> a;
};
```

This design leaks because the strong counts never reach zero.
The correct pattern is to make at least one edge non-owning.

```
#include <memory>

struct B;

struct A
{
    std::shared_ptr<B> b;
};
```

```
struct B
{
    std::weak_ptr<A> a;
};
```

This is the standard cycle-breaking pattern.

Layered RAII graphs

Larger systems often contain multiple ownership layers at once. A root object may exclusively own subsystems, while those subsystems may share some reusable resources, and helper code may borrow temporary views into both.

```
#include <memory>
#include <string>
#include <vector>
#include <span>
#include <utility>

class SharedBuffer
{
    std::vector<unsigned char> bytes_;

public:
    explicit SharedBuffer(std::vector<unsigned char> bytes)
        : bytes_(std::move(bytes))
    {
    }

    std::span<const unsigned char> view() const noexcept
```

```
    {
        return bytes_;
    }
};

class Subsystem
{
    std::shared_ptr<SharedBuffer> buffer_;

public:
    explicit Subsystem(std::shared_ptr<SharedBuffer> buffer)
        : buffer_(std::move(buffer))
    {
    }

    std::span<const unsigned char> bytes() const noexcept
    {
        return buffer_->view();
    }
};

class Engine
{
    std::vector<std::unique_ptr<Subsystem>> systems_;

public:
    void add_system(std::shared_ptr<SharedBuffer> buffer)
    {
        systems_.push_back(std::make_unique<Subsystem>(std::move(buffer)));
```

```
    }
};
```

This example combines:

- exclusive ownership of subsystems,
- shared ownership of reusable buffer resources,
- borrowed span views for temporary access.

This is a true RAII graph rather than a single ownership pattern.

Graph design rules

A good RAII graph design usually follows these rules:

1. Use the simplest ownership edge that correctly models lifetime.
2. Prefer exclusive ownership unless sharing is truly needed.
3. Break cycles explicitly with non-owning links.
4. Keep borrowed edges obviously non-owning.
5. Ensure destruction order follows dependency order.

8.2.2 Resource slicing

Resource slicing is the design practice of exposing only part of a larger owned resource as a non-owning subresource view. In advanced RAII systems, a large structure often owns the complete resource, while clients operate only on slices, windows, subranges, subviews, or lightweight projections of that resource.
This is especially important because high-quality RAII design separates:

- ownership of the whole resource,
- borrowed access to part of the resource.

The term slicing here does not mean object slicing through base-by-value conversion. It means extracting a logically smaller view from a larger owned resource while preserving ownership in the original object.

Why slicing matters

Many large owned resources are naturally partitioned:

- a byte buffer into packet headers and payloads,
- a text document into line views,
- a mapped file into sections,
- a large array into windows or tiles,
- a GPU upload buffer into suballocations,
- a resource bundle into smaller borrowed facets.

The owner should remain responsible for release, while the slice should remain cheap and non-owning.

Span-based slicing

`std::span` is one of the clearest tools for resource slicing because it represents a borrowed view over contiguous data.

```
#include <vector>
#include <span>
#include <cstddef>
#include <utility>

class PacketBuffer
{
    std::vector<unsigned char> bytes_;

public:
    explicit PacketBuffer(std::vector<unsigned char> bytes)
        : bytes_(std::move(bytes))
    {
    }

    std::span<const unsigned char> all() const noexcept
    {
        return bytes_;
    }

    std::span<const unsigned char> header() const noexcept
    {
        return std::span<const unsigned char>(bytes_).first(8);
    }

    std::span<const unsigned char> payload() const noexcept
    {
        auto full = std::span<const unsigned char>(bytes_);
        return full.subspan(8);
```

```
    }
};
```

In this design:

- `PacketBuffer` owns the bytes,
- `header()` and `payload()` return borrowed slices,
- no additional ownership is created.

String-view slicing

The same principle applies to text resources.

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

class SourceText
{
    std::string text_;

public:
    explicit SourceText(std::string text)
        : text_(std::move(text))
    {
    }

    std::string_view all() const noexcept
    {
```

```
        return text_;
    }

    std::string_view prefix(std::size_t count) const noexcept
    {
        return std::string_view(text_).substr(0, count);
    }

    std::string_view suffix(std::size_t pos) const noexcept
    {
        return std::string_view(text_).substr(pos);
    }
};
```

Here the owned text remains inside `SourceText`, while string views provide slices into it.

Lifetime discipline for slices

The essential rule for resource slicing is:

> A slice never outlives the owned resource from which it was derived.

This means span and string-view slices are excellent return values for transient access, but dangerous as long-lived stored members unless the owner lifetime is externally guaranteed. Dangerous pattern:

```
#include <string_view>

class BadCache
{
    std::string_view view_;
```

```
public:
    void set(std::string_view v)
    {
        view_ = v;
    }
};
```

This may dangle if the caller provided a temporary or short-lived source.

Safer owning-plus-slicing pattern:

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

class GoodCache
{
    std::string text_;

public:
    void set(std::string text)
    {
        text_ = std::move(text);
    }

    std::string_view view() const noexcept
    {
        return text_;
    }
};
```

Slicing composite resource owners

A multi-resource owner may expose slices of several internally owned resources without transferring ownership.

```
#include <vector>
#include <string>
#include <span>
#include <string_view>
#include <utility>

class CompilationUnit
{
    std::string source_name_;
    std::vector<unsigned char> source_bytes_;

public:
    CompilationUnit(std::string name, std::vector<unsigned char> bytes)
        : source_name_(std::move(name)),
          source_bytes_(std::move(bytes))
    {
    }

    std::string_view name() const noexcept
    {
        return source_name_;
    }

    std::span<const unsigned char> bytes() const noexcept
    {
```

```
        return source_bytes_;
    }

    std::span<const unsigned char> prefix(std::size_t n) const noexcept
    {
        return std::span<const unsigned char>(source_bytes_).first(n);
    }
};
```

This is a sliced view over a composite RAII owner.

Subresource handles versus slices

Sometimes a slice is not just a view but a structured subresource descriptor. A common example is a suballocation in a larger buffer.

```
#include <vector>
#include <span>
#include <cstddef>
#include <utility>

class ArenaBuffer
{
    std::vector<std::byte> storage_;

public:
    explicit ArenaBuffer(std::size_t size)
        : storage_(size)
    {
    }
```

```
    std::span<std::byte> slice(std::size_t offset, std::size_t count) noexcept
    {
        return std::span<std::byte>(storage_).subspan(offset, count);
    }

    std::span<const std::byte> slice(std::size_t offset, std::size_t count) const
    ↪   noexcept
    {
        return std::span<const std::byte>(storage_).subspan(offset, count);
    }
};
```

The arena owns the backing memory; slices describe subregions.

Resource slicing rules

1. Let the full owner remain clearly identifiable.
2. Expose slices as non-owning values such as `std::span` and `std::string_view`.
3. Never pretend that a slice owns the underlying resource.
4. Avoid storing slices beyond the guaranteed lifetime of the owner.
5. Prefer slice-returning member functions over exposing raw internals.

8.2.3 Resource bundles

A resource bundle is a single object that owns several resources together and presents them as one coherent lifetime unit. This pattern is essential when resources must be acquired, moved, released, or rolled back together.

A bundle may contain:

- several homogeneous resources,
- several heterogeneous resources,
- a mixture of memory, handles, locks, buffers, and views,
- subordinate bundles,
- transaction guards and synchronization guards.

Why bundling matters

Resource bundles are valuable when:

- resources are logically inseparable,
- partial release would violate invariants,
- acquisition order and destruction order must be coordinated,
- a subsystem should expose one stable RAII owner instead of many loose handles.

Examples include:

- a network connection object owning a socket, send buffer, receive buffer, and statistics state,
- a renderer context owning multiple device-related handles,
- a parser context owning input, tables, diagnostics sink, and temporary arenas,
- a database session owning a connection, prepared statements, transaction state, and synchronization objects.

Simple heterogeneous resource bundle

```
#include <memory>
#include <string>
#include <vector>
#include <utility>

class SessionResources
{
    std::unique_ptr<int> token_;
    std::string name_;
    std::vector<unsigned char> buffer_;

public:
    SessionResources(std::unique_ptr<int> token,
                     std::string name,
                     std::vector<unsigned char> buffer)
        : token_(std::move(token)),
          name_(std::move(name)),
          buffer_(std::move(buffer))
    {
    }

    const std::string& name() const noexcept
    {
        return name_;
    }

    const std::vector<unsigned char>& buffer() const noexcept
    {
```

```
        return buffer_;
    }

    const int* token() const noexcept
    {
        return token_.get();
    }
};
```

This object owns three different resources as one lifetime unit.

Tuple-style bundling versus named bundling

A bundle can be represented structurally with `std::tuple`, but named fields are often better for clarity in long-lived types.
Tuple example:

```
#include <tuple>
#include <memory>
#include <string>
#include <vector>

using Bundle = std::tuple<
    std::unique_ptr<int>,
    std::string,
    std::vector<unsigned char>
>;
```

Named class example:

```
#include <memory>
```

```
#include <string>
#include <vector>
#include <utility>

class Bundle2
{
    std::unique_ptr<int> id_;
    std::string label_;
    std::vector<unsigned char> bytes_;

public:
    Bundle2(std::unique_ptr<int> id,
            std::string label,
            std::vector<unsigned char> bytes)
        : id_(std::move(id)),
          label_(std::move(label)),
          bytes_(std::move(bytes))
    {
    }
};
```

For educational and generic programming, tuple bundling is useful. For application architecture, named bundling is usually easier to maintain.

Exception-safe bundle construction

A key advantage of RAII bundles is that construction is naturally exception-safe. If one member finishes construction and a later member throws, the already-constructed members are destroyed automatically in reverse order.

```
#include <memory>
```

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

class BuildContext
{
    std::string module_name_;
    std::vector<int> token_ids_;
    std::unique_ptr<int> status_;

public:
    BuildContext(std::string module_name,
                 std::vector<int> token_ids,
                 std::unique_ptr<int> status)
        : module_name_(std::move(module_name)),
          token_ids_(std::move(token_ids)),
          status_(std::move(status))
    {
    }
};
```

The language itself provides the cleanup discipline. This is one of the strongest reasons to prefer bundles over scattered raw handles.

Bundle plus view facade

A resource bundle often owns heavy state internally while exposing only lightweight borrowed access.

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

```cpp
#include <string_view>
#include <span>
#include <utility>

class FileBundle
{
    std::string path_;
    std::vector<unsigned char> contents_;

public:
    FileBundle(std::string path, std::vector<unsigned char> contents)
        : path_(std::move(path)),
          contents_(std::move(contents))
    {
    }

    std::string_view path() const noexcept
    {
        return path_;
    }

    std::span<const unsigned char> contents() const noexcept
    {
        return contents_;
    }
};
```

This is often the best interface style for multi-resource RAII objects:

- own internally,

- expose externally through views.

Synchronization bundles and scoped locking

A bundle may also contain several synchronization resources. Current standard-library design supports coordinated locking through `std::scoped_lock` over multiple mutexes, making it natural to treat a set of protected resources as one synchronized unit.

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

class SharedStateBundle
{
    mutable std::mutex m1_;
    mutable std::mutex m2_;
    std::string name_;
    std::vector<int> values_;

public:
    void update(std::string name, int value)
    {
        std::scoped_lock lock(m1_, m2_);
        name_ = std::move(name);
        values_.push_back(value);
    }
};
```

This is a bundle of protected resources whose mutation is coordinated as one scoped operation.

Nested bundles

Large systems often compose bundles out of smaller bundles.

```
#include <memory>
#include <string>
#include <vector>
#include <utility>

class InputResources
{
    std::string file_name_;
    std::vector<unsigned char> bytes_;

public:
    InputResources(std::string file_name, std::vector<unsigned char> bytes)
        : file_name_(std::move(file_name)),
          bytes_(std::move(bytes))
    {
    }
};

class ParseResources
{
    std::unique_ptr<int> state_;
    std::vector<int> tokens_;

public:
    ParseResources(std::unique_ptr<int> state, std::vector<int> tokens)
        : state_(std::move(state)),
```

```
          tokens_(std::move(tokens))
    {
    }
};

class CompilerBundle
{
    InputResources input_;
    ParseResources parse_;

public:
    CompilerBundle(InputResources input, ParseResources parse)
        : input_(std::move(input)),
          parse_(std::move(parse))
    {
    }
};
```

Nested bundling is often cleaner than a single oversized class with unrelated data members.

Transactional bundles

A bundle may also represent a set of resources whose state changes should commit together.

```
#include <string>
#include <vector>
#include <utility>

class StateBundle
{
    std::string name_;
```

```
    std::vector<int> values_;

public:
    StateBundle(std::string name, std::vector<int> values)
        : name_(std::move(name)),
          values_(std::move(values))
    {
    }

    void swap(StateBundle& other) noexcept
    {
        name_.swap(other.name_);
        values_.swap(other.values_);
    }

    const std::string& name() const noexcept
    {
        return name_;
    }

    const std::vector<int>& values() const noexcept
    {
        return values_;
    }
};

class Owner
{
    StateBundle state_;
```

```
public:
    Owner(std::string name, std::vector<int> values)
        : state_(std::move(name), std::move(values))
    {
    }

    void replace(StateBundle next)
    {
        state_.swap(next);
    }
};
```

This is a bundled commit pattern: the new aggregate state is prepared first, then swapped in atomically at the logical level.

Bundle design rules

1. Bundle resources when they form one logical lifetime unit.
2. Prefer named bundles for architectural clarity.
3. Use member initialization to exploit automatic rollback during partial construction.
4. Expose views or references instead of leaking ownership outward.
5. Consider nested bundles when the structure becomes large.

8.2.4 Multi-resource failure handling

One of the most important concerns in multi-resource structures is failure during partial acquisition. The RAII answer is consistent:

- each subresource should be wrapped in its own correct owner,
- the aggregate should rely on member destruction order,
- commit steps should occur only after all prerequisite acquisitions succeed.

Two-phase acquisition through local owners

```
#include <memory>
#include <string>
#include <vector>
#include <utility>

class ComplexBundle
{
    std::unique_ptr<int> handle_;
    std::string text_;
    std::vector<int> values_;

public:
    ComplexBundle(std::unique_ptr<int> handle,
                  std::string text,
                  std::vector<int> values)
        : handle_(std::move(handle)),
          text_(std::move(text)),
          values_(std::move(values))
    {
    }
};

ComplexBundle make_bundle()
```

```
{
    auto h = std::make_unique<int>(7);
    std::string text = "ready";
    std::vector<int> values{1, 2, 3, 4};

    return ComplexBundle(std::move(h), std::move(text), std::move(values));
}
```

This construction style is robust because all temporaries are RAII-managed before the final aggregate is built.

8.2.5 Comparative design map

The three patterns in this section solve related but different structural problems.

RAII graphs

- describe ownership topology,
- focus on edges between objects,
- emphasize cycle prevention and lifetime direction.

Resource slicing

- describe partial access to larger owned resources,
- focus on view semantics,
- emphasize owner-versus-slice lifetime boundaries.

Resource bundles

- describe aggregate lifetime units,
- focus on coordinated acquisition and release,
- emphasize coherent construction, rollback, and interface clarity.

8.2.6 Extensive examples

Example 1: RAII graph with parent borrow and child ownership

```
#include <memory>
#include <string>
#include <vector>
#include <utility>

class Widget
{
    std::string id_;
    Widget* parent_;
    std::vector<std::unique_ptr<Widget>> children_;

public:
    explicit Widget(std::string id, Widget* parent = nullptr)
        : id_(std::move(id)), parent_(parent)
    {
    }

    Widget& emplace_child(std::string id)
    {
```

```
        children_.push_back(std::make_unique<Widget>(std::move(id), this));
        return *children_.back();
    }

    const std::string& id() const noexcept
    {
        return id_;
    }

    const Widget* parent() const noexcept
    {
        return parent_;
    }
};
```

Example 2: shared RAII graph with weak back-reference

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

class Node : public std::enable_shared_from_this<Node>
{
    std::vector<std::shared_ptr<Node>> children_;
    std::weak_ptr<Node> parent_;

public:
    void add_child(const std::shared_ptr<Node>& child)
    {
        child->parent_ = shared_from_this();
        children_.push_back(child);
```

```
    }

    std::shared_ptr<Node> parent() const
    {
        return parent_.lock();
    }
};
```

Example 3: owned buffer with sliced packet views

```
#include <vector>
#include <span>
#include <cstddef>
#include <utility>

class Packet
{
    std::vector<unsigned char> bytes_;

public:
    explicit Packet(std::vector<unsigned char> bytes)
        : bytes_(std::move(bytes))
    {
    }

    std::span<const unsigned char> version_field() const noexcept
    {
        return std::span<const unsigned char>(bytes_).subspan(0, 2);
    }
```

```
    std::span<const unsigned char> flags_field() const noexcept
    {
        return std::span<const unsigned char>(bytes_).subspan(2, 2);
    }

    std::span<const unsigned char> payload() const noexcept
    {
        return std::span<const unsigned char>(bytes_).subspan(4);
    }
};
```

Example 4: text owner with lexical slices

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

class ScriptText
{
    std::string source_;

public:
    explicit ScriptText(std::string source)
        : source_(std::move(source))
    {
    }

    std::string_view all() const noexcept
    {
        return source_;
```

```
    }

    std::string_view first_line() const noexcept
    {
        auto pos = source_.find('\n');
        if (pos == std::string::npos)
            return source_;
        return std::string_view(source_).substr(0, pos);
    }
};
```

Example 5: heterogeneous resource bundle

```
#include <memory>
#include <string>
#include <vector>
#include <utility>

class RuntimeBundle
{
    std::unique_ptr<int> session_id_;
    std::string module_name_;
    std::vector<unsigned char> bytecode_;

public:
    RuntimeBundle(std::unique_ptr<int> session_id,
                  std::string module_name,
                  std::vector<unsigned char> bytecode)
        : session_id_(std::move(session_id)),
          module_name_(std::move(module_name)),
```

```
          bytecode_(std::move(bytecode))
    {
    }

    std::string_view module_name() const noexcept
    {
        return module_name_;
    }

    std::span<const unsigned char> bytecode() const noexcept
    {
        return bytecode_;
    }
};
```

Example 6: synchronized resource bundle

```
#include <mutex>
#include <string>
#include <vector>
#include <span>

class ProtectedBundle
{
    mutable std::mutex name_mutex_;
    mutable std::mutex data_mutex_;
    std::string name_;
    std::vector<int> data_;

public:
```

```
    void replace(std::string name, std::vector<int> data)
    {
        std::scoped_lock lock(name_mutex_, data_mutex_);
        name_ = std::move(name);
        data_ = std::move(data);
    }

    std::string snapshot_name() const
    {
        std::scoped_lock lock(name_mutex_);
        return name_;
    }

    std::vector<int> snapshot_data() const
    {
        std::scoped_lock lock(data_mutex_);
        return data_;
    }
};
```

8.2.7 Windows build notes

MSVC

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo example.cpp
cl /std:c++23 /EHsc /W4 /O2 /permissive- /analyze /nologo example.cpp
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

GCC on Windows via MSYS2 or MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

Practical Windows guidance

For multi-resource RAII structures on Windows:

1. prefer named aggregate classes over loose raw handles,
2. use `std::scoped_lock` when multiple mutexes must be acquired together,
3. expose slices as `std::span` and `std::string_view` only when owner lifetime is stable,
4. keep shared ownership explicit and minimize it where possible,
5. model parent-child graphs with clear owning and non-owning edges.

8.2.8 Concluding design rules

Multi-resource RAII design succeeds when ownership topology, access topology, and aggregate lifetime are all made explicit.

1. Use RAII graphs to model who owns whom.
2. Use resource slicing to expose partial access without transferring ownership.
3. Use resource bundles to group related lifetimes into one coherent object.
4. Avoid cycles unless they are intentionally broken with non-owning links.
5. Prefer internal ownership and external views.
6. Let aggregate construction and destruction do the rollback work for you.

The deepest lesson is that RAII scales from one handle to entire systems only when structure is designed deliberately. A single correct wrapper is useful, but a correct ownership graph, a safe slicing discipline, and a coherent resource bundle are what make large Modern C++ systems stable, comprehensible, and exception-safe.

Part IV

Dynamic Polymorphism, Vtables & Runtime Architecture

Dynamic Polymorphism Basics

9.1 Why OOP Exists in C++

Object-Oriented Programming in C++ was introduced to solve several structural problems that arise in large-scale software systems. While the C programming language provides powerful tools for low-level programming and procedural design, large programs often require mechanisms for abstraction, modularity, extensibility, and runtime polymorphism. C++ extends the procedural model by introducing classes, encapsulation, inheritance, and dynamic dispatch.

The motivation for object-oriented mechanisms in C++ can be understood through several core design goals.

9.1.1 Encapsulation and abstraction

Encapsulation allows data and behavior to be grouped together in a single abstraction. This reduces accidental coupling and improves maintainability. In C++, a class defines both the internal representation and the operations that manipulate that representation.

```
#include <string>


class Account
```

```
{
private:
    double balance_;

public:
    explicit Account(double initial)
        : balance_(initial)
    {
    }

    void deposit(double amount)
    {
        balance_ += amount;
    }

    double balance() const
    {
        return balance_;
    }
};
```

The internal representation is hidden from the outside world, allowing the implementation to evolve without affecting client code.

9.1.2 Interface-based design

Large software systems often operate through abstract interfaces rather than concrete implementations. This allows code to depend on behavior rather than specific types.

```
#include <memory>
```

```
class Shape
{
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;
};

class Circle : public Shape
{
    double radius_;

public:
    explicit Circle(double r)
        : radius_(r)
    {
    }

    double area() const override
    {
        return 3.141592653589793 * radius_ * radius_;
    }
};
```

The user of the interface interacts only with the abstract behavior rather than the concrete implementation.

9.1.3 Extensibility

Object-oriented design allows systems to grow by introducing new derived classes without modifying existing code. This supports open-ended architectures such as plugin systems, rendering pipelines, serialization frameworks, and interpreters.

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

class Renderer
{
public:
    virtual ~Renderer() = default;
    virtual void render() = 0;
};

class OpenGLRenderer : public Renderer
{
public:
    void render() override
    {
    }
};

class VulkanRenderer : public Renderer
{
public:
    void render() override
    {
    }
```

```
};
```

The rest of the program can interact with the abstract interface regardless of which concrete renderer is used.

9.2 Virtual Methods & Late Binding

Dynamic polymorphism in C++ relies on virtual functions. A virtual function allows the program to decide which implementation should execute at runtime rather than compile time. This mechanism is known as late binding or dynamic dispatch.
When a virtual function is called through a base-class pointer or reference, the actual function invoked corresponds to the dynamic type of the object rather than the static type of the pointer.

9.2.1 Basic virtual dispatch

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

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

    virtual void speak() const
    {
        std::cout << "Animal sound\n";
    }
};
```

```
class Dog : public Animal
{
public:
    void speak() const override
    {
        std::cout << "Bark\n";
    }
};

class Cat : public Animal
{
public:
    void speak() const override
    {
        std::cout << "Meow\n";
    }
};
```

Using the interface:

```
#include <vector>

int main()
{
    std::vector<std::unique_ptr<Animal>> animals;

    animals.push_back(std::make_unique<Dog>());
    animals.push_back(std::make_unique<Cat>());

    for (const auto& a : animals)
```

```
        a->speak();
}
```

Even though the container stores pointers of type `Animal`, the correct derived implementation is invoked.

9.2.2 Runtime polymorphism in containers

Dynamic polymorphism allows heterogeneous collections of objects to be handled uniformly.

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

class Command
{
public:
    virtual ~Command() = default;
    virtual void execute() = 0;
};

class SaveCommand : public Command
{
public:
    void execute() override
    {
    }
};

class LoadCommand : public Command
{
```

```
public:
    void execute() override
    {
    }
};

int main()
{
    std::vector<std::unique_ptr<Command>> commands;

    commands.push_back(std::make_unique<SaveCommand>());
    commands.push_back(std::make_unique<LoadCommand>());

    for (auto& cmd : commands)
        cmd->execute();
}
```

This design pattern is widely used in interpreters, compilers, GUI frameworks, and plugin systems.

9.2.3 Virtual destructors

Any class intended for polymorphic use should provide a virtual destructor. This ensures that deleting an object through a base-class pointer invokes the correct derived destructor.

```
class Base
{
public:
    virtual ~Base() = default;
};
```

Without a virtual destructor, deleting through a base pointer leads to undefined behavior.

9.3 Abstract Classes & Interfaces

An abstract class is a class that cannot be instantiated directly and exists only as a base class for derived implementations. In C++, abstract classes are created by declaring at least one pure virtual function.

9.3.1 Pure virtual functions

A pure virtual function is declared by assigning `=0` to a virtual member function.

```
class Device
{
public:
    virtual ~Device() = default;
    virtual void initialize() = 0;
    virtual void shutdown() = 0;
};
```

This class defines an interface rather than a concrete type.

9.3.2 Concrete implementations

```
#include <iostream>

class Keyboard : public Device
{
public:
    void initialize() override
```

```
    {
        std::cout << "Keyboard initialized\n";
    }

    void shutdown() override
    {
        std::cout << "Keyboard shutdown\n";
    }
};

class Mouse : public Device
{
public:
    void initialize() override
    {
        std::cout << "Mouse initialized\n";
    }

    void shutdown() override
    {
        std::cout << "Mouse shutdown\n";
    }
};
```

Usage:

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

int main()
```

```
{
    std::vector<std::unique_ptr<Device>> devices;

    devices.push_back(std::make_unique<Keyboard>());
    devices.push_back(std::make_unique<Mouse>());

    for (auto& d : devices)
    {
        d->initialize();
        d->shutdown();
    }
}
```

Abstract interfaces are extremely common in large frameworks and system-level APIs.

9.3.3 Interface design principles

Effective abstract interface design usually follows these principles:

- keep the interface minimal,
- avoid exposing implementation details,
- prefer stable semantic contracts,
- provide virtual destructors for polymorphic bases.

9.4 Overriding, final, override

Modern C++ provides language keywords that improve safety when working with inheritance and virtual functions.

9.4.1 The override specifier

The `override` keyword indicates that a function is intended to override a virtual function from a base class. If the function does not correctly override a base-class function, the compiler produces an error.

```
class Base
{
public:
    virtual void process(int x)
    {
    }
};

class Derived : public Base
{
public:
    void process(int x) override
    {
    }
};
```

If the function signature is incorrect, the compiler detects the error.

```
class Broken : public Base
{
public:
    void process(double x) override
    {
    }
};
```

This fails because the signature does not match the base function.

9.4.2 The final specifier

The `final` specifier prevents further overriding of a virtual function or further inheritance of a class.

```
class Logger
{
public:
    virtual void write() final
    {
    }
};
```

Derived classes cannot override the function.

```
class DerivedLogger : public Logger
{
public:
    void write() override
    {
    }
};
```

This results in a compilation error.
Classes themselves can also be declared final.

```
class Immutable final
{
};
```

No class can derive from a final class.

9.4.3 Combining override and final

A method can be declared both override and final, meaning it overrides a base method and cannot be overridden again.

```
class BaseProcessor
{
public:
    virtual void run()
    {
    }
};

class OptimizedProcessor : public BaseProcessor
{
public:
    void run() override final
    {
    }
};
```

This ensures that the override occurs at this level and cannot be further modified.

9.4.4 Best practices

When writing polymorphic classes in Modern C++:

- always mark overriding functions with `override`,
- declare destructors virtual in polymorphic base classes,
- use `final` when a class hierarchy should not be extended,

- prefer interface-style abstract bases for extensibility.

9.4.5 Compilation guide for Windows

MSVC

```
cl /std:c++23 /EHsc /W4 example.cpp
```

Clang

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp
```

GCC

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp
```

Vtable Internals & Layout (Deep)

10.1 Vtable Structure

The implementation of dynamic polymorphism in C++ is one of the most important bridges between the language model and runtime architecture. At the source level, the programmer writes virtual functions, base-class pointers, abstract interfaces, and overrides. At the machine level, the implementation must provide a mechanism that enables a call made through a base-class interface to reach the correct derived-class function at runtime.

The C++ language standard specifies the semantic behavior of virtual dispatch, but it does not prescribe one exact object layout or one exact table representation for all compilers and platforms. In practice, mainstream implementations use virtual tables, commonly called vtables, together with one or more hidden pointers inside polymorphic objects, commonly called vptrs.

A deep understanding of vtable structure is valuable because it explains:

1. how runtime dispatch works,
2. why polymorphic objects often have larger size than non-polymorphic objects,
3. why multiple inheritance changes object layout,
4. why ABI compatibility matters for shared libraries, plugins, and cross-module interfaces,

5. why different compiler families cannot automatically be assumed to share a binary object model.

This section studies three major aspects:

1. how vtables are generated,
2. memory layout and offsets,
3. important ABI differences between GCC/Clang and MSVC.

10.1.1 What a vtable conceptually is

A vtable is an implementation-defined table associated with a polymorphic class. It usually contains enough information for the runtime to locate the correct function implementation for a virtual call. Depending on the compiler ABI, the table may also contain RTTI-related pointers, offset adjustments, destructors, thunk entries, and base-related metadata.
At a high conceptual level, a polymorphic object usually contains at least one hidden pointer:

- the `vptr`, stored inside the object,
- pointing to a compiler-generated table for the object's dynamic type.

When a virtual function is called through a base pointer or reference, the implementation loads the vptr, indexes into the vtable, and jumps to the target function or thunk.
This means the runtime behavior of a virtual call often resembles the following conceptual sequence:

1. load object address,
2. load hidden vptr from the object,

3. read function address from a fixed slot in the table,

4. branch to that function.

This is a conceptual description. The exact instruction sequence depends on the target architecture, optimization level, inheritance model, and ABI.

10.1.2 How vtables are generated

A compiler typically generates vtable-related data when it encounters a class that is polymorphic, meaning a class that declares or inherits at least one virtual function. Such a class needs runtime dispatch support.

Minimal example

```
class Base
{
public:
    virtual ~Base() = default;
    virtual void f();
};

class Derived : public Base
{
public:
    void f() override;
};
```

This source-level code implies several runtime-level needs:

- `Base` must support calls through a base interface,

- `Derived` must supply an override for the slot used by `f()`,
- deleting through `Base*` must call the correct destructor path.

Therefore the implementation will generally emit:

- one vtable representation for `Base`,
- one vtable representation for `Derived`,
- RTTI data for the polymorphic types,
- one or more hidden initialization steps in constructors that install the correct vptr value into the object.

When the vptr is installed

A common implementation strategy is that each constructor writes the appropriate vptr value for the subobject currently being constructed. Likewise, destructors may temporarily change the effective vptr state as construction and destruction move through base and derived layers. This is deeply important because during construction and destruction, the dynamic type behavior used for virtual dispatch is not simply "the most-derived type at all times" in the naive sense. The implementation must preserve the standard's rules for virtual dispatch during these phases, and this usually leads to constructor and destructor code that installs different vptr values at different times.

Simple conceptual model

For a simple single-inheritance class, one can imagine the compiler doing something conceptually like this:

```
struct Base_runtime_layout
{
    void** vptr;
};

struct Derived_runtime_layout
{
    void** vptr;
};
```

This is only a teaching model. Real implementations are more complex, but it illustrates the basic idea that the object's first machine word often stores a hidden pointer to dispatch metadata.

Slots for destructors and virtual functions

Compilers usually place virtual destructor entries and ordinary virtual-function entries into fixed slots. The exact slot layout depends on the ABI.
A conceptual example for a simple class might look like this:

```
vtable for Base:
  [metadata / type information / offset information]
  [destructor entry]
  [deleting destructor entry or destructor variant]
  [Base::f]
```

For the derived class:

```
vtable for Derived:
  [metadata / type information / offset information]
  [destructor entry]
```

```
  [deleting destructor entry or destructor variant]
  [Derived::f]
```

Again, the exact shape differs by ABI, but this illustrates the replacement idea: when a function is overridden, the derived table uses the same logical dispatch slot, now pointing to the overriding function or an adjusting thunk.

Key function and emission model

In real C++ ABIs, especially on common Unix-like targets using GCC and Clang, vtable emission is usually tied to an out-of-line virtual function definition known historically as the key function concept. This matters for ODR and object-file emission rules because the compiler and linker must ensure one canonical emitted definition of the vtable-related data where appropriate.

For practical development, this means a class with virtual functions often requires at least one corresponding out-of-line definition in a source file to ensure that the compiler emits the necessary vtable and typeinfo in a predictable translation unit.

A classical beginner error is declaring virtual functions but forgetting to define one that is required, leading to linker errors mentioning vtables or typeinfo.

Linker symptom example

```
class Base
{
public:
    virtual ~Base();
    virtual void f();
};
```

If `Base:: Base()` or `Base::f()` is declared but never defined where needed, a linker may

report missing symbols associated with the vtable or RTTI support because the implementation expected those virtual members to anchor related emission.

10.1.3 Single inheritance layout

Single inheritance is the easiest case to study because the object often contains just one vptr and one linear sequence of data members.

Example

```
class A
{
public:
    virtual ~A() = default;
    virtual void f();
    int x;
};

class B : public A
{
public:
    void f() override;
    int y;
};
```

A simple conceptual runtime layout might look like this:

```
A object:
  [vptr]
  [x]
```

```
B object:
  [A subobject:
      vptr
      x]
  [y]
```

In many implementations, the vptr is at offset zero for the primary base or most-derived object in the simple single-inheritance case. This is important because it makes dispatch efficient and allows a base pointer to coincide with the object address itself.

Illustrative size example

```cpp
#include <iostream>

class A
{
public:
    virtual ~A() = default;
    int x;
};

class B : public A
{
public:
    int y;
};

int main()
{
    std::cout << sizeof(A) << '\n';
```

```
    std::cout << sizeof(B) << '\n';
}
```

On a 64-bit target, one often sees that `A` is larger than an `int`-only structure would otherwise be, because the hidden vptr typically consumes one pointer-sized machine word, and padding may be inserted to satisfy alignment.

Virtual dispatch path

```
void call_f(A& a)
{
    a.f();
}
```

The generated code usually resembles this conceptual pattern:

1. load address of `a`,
2. load vptr from offset zero or the ABI-defined location,
3. fetch function address from the appropriate table slot,
4. call that function with `this` adjusted as needed.

In the simple case, no `this` adjustment is necessary because the base subobject begins at offset zero.

10.1.4 Memory layout and offsets

Vtables become more interesting when object layout is no longer a simple single-inheritance chain. Multiple inheritance, virtual inheritance, and secondary base subobjects force the implementation to track offsets carefully.

Multiple inheritance example

```
class Left
{
public:
    virtual ~Left() = default;
    virtual void lf();
    int lx;
};

class Right
{
public:
    virtual ~Right() = default;
    virtual void rf();
    int rx;
};

class Derived : public Left, public Right
{
public:
    void lf() override;
    void rf() override;
    int dz;
};
```

A conceptual object layout may look like this:

```
Derived object:
  [Left subobject:
      vptr for Left-view dispatch
```

```
    lx]
  [Right subobject:
    vptr for Right-view dispatch
    rx]
  [dz]
```

This means the full object may contain more than one vptr. Each polymorphic base subobject may need its own dispatch entry point for calls made through pointers or references to that base.

Why offsets matter

Suppose a `Derived` object is viewed through a `Right*`. The address of the `Right` base subobject is usually not equal to the address of the complete `Derived` object. Therefore:

- converting `Derived*` to `Right*` may add a fixed offset,
- a virtual call through `Right*` may require adjustment back to the correct complete-object or overriding-function context,
- destructor dispatch and cross-casts depend on these offsets being represented consistently.

Pointer adjustment example

```cpp
#include <iostream>

class Left
{
public:
    virtual ~Left() = default;
};
```

```
class Right
{
public:
    virtual ~Right() = default;
};

class Derived : public Left, public Right
{
};

int main()
{
    Derived d;

    Left* lp = &d;
    Right* rp = &d;

    std::cout << static_cast<void*>(lp) << '\n';
    std::cout << static_cast<void*>(rp) << '\n';
}
```

On typical implementations, these printed addresses differ because `Right` is a secondary base subobject located at a nonzero offset within the complete object.

Thunks

When a derived override is called through a base subobject whose `this` pointer is not already in the correct form for the target function, the compiler may generate a thunk. A thunk is a small helper entry that adjusts `this` and then jumps to the real function body.

Conceptually:

```
Right-view virtual call slot:
  -> thunk adjusts this from Right-subobject address
  -> jumps to Derived::rf
```

This is one of the key reasons ABI documentation discusses vtable slots and offset-to-top values in such detail.

Virtual inheritance

Virtual inheritance is even more complex because the shared virtual base may not sit at a fixed offset independent of the most-derived type in the same simple way as non-virtual bases. Implementations therefore often require additional metadata and more complicated adjustment logic.
Example:

```
class VBase
{
public:
    virtual ~VBase() = default;
    virtual void vf();
    int vb;
};

class Left : virtual public VBase
{
public:
    int lx;
};
```

```
class Right : virtual public VBase
{
public:
    int rx;
};

class Final : public Left, public Right
{
public:
    void vf() override;
    int fz;
};
```

Now the complete object contains one shared `VBase` subobject, but access to it requires ABI-specific mechanisms that may involve virtual base offsets stored in metadata.

Offset-to-top concept

In ABIs derived from the Itanium C++ ABI, a vtable is not just a simple array of function pointers. The address point used by the object's vptr usually points into a richer structure that includes entries such as:

- offset-to-top,
- typeinfo pointer,
- virtual function slots,
- sometimes virtual base offset entries or related metadata in surrounding table regions.

The offset-to-top helps the implementation recover the complete-object address from a base-subobject address. This is especially important for `dynamic_cast`, RTTI, and destructor paths.

Primary versus secondary vtables

In complex inheritance structures, the most-derived type may conceptually have:

- a primary vtable used for the primary base or main object view,
- secondary vtables or secondary address points for other polymorphic base subobjects.

This explains why the phrase “the vtable” is often an oversimplification. A complex class may participate in several related virtual-table views.

10.1.5 How overrides map into slots

One of the most important implementation principles is slot stability within a given ABI. If a base declares virtual functions in a certain order, the derived type’s dispatch layout preserves the corresponding logical slot positions, replacing entries with overrides where needed. Example:

```
class Base
{
public:
    virtual ~Base() = default;
    virtual void f();
    virtual void g();
};

class Derived : public Base
{
public:
    void f() override;
    void g() override;
};
```

A conceptual view is:

```
Base dispatch slots:
  slot for destructor variant(s)
  slot for f
  slot for g

Derived dispatch slots:
  slot for destructor variant(s) -> Derived destructor path
  slot for f -> Derived::f
  slot for g -> Derived::g
```

This slot preservation is what makes polymorphic calls through a `Base*` work correctly when the dynamic object is actually `Derived`.

10.1.6 RTTI relationship

Polymorphic classes usually participate in runtime type identification support. This is why many implementations associate typeinfo pointers with vtable-related structures.
Operations such as:

- `dynamic_cast`,
- `typeid` applied to polymorphic glvalues,
- safe cross-casting in multiple inheritance,

typically depend on metadata reachable through the vptr and table structures.

Example

```
#include <typeinfo>
#include <iostream>

class Base
{
public:
    virtual ~Base() = default;
};

class Derived : public Base
{
};

void inspect(Base& b)
{
    std::cout << typeid(b).name() << '\n';
}
```

The implementation of `typeid(b)` for a polymorphic reference uses runtime type information associated with the object's dynamic type, and this is typically connected to vtable-related metadata.

10.1.7 ABI differences between GCC/Clang and MSVC

One of the most important practical facts in low-level C++ is that the standard does not define one universal binary object model for classes, vtables, or RTTI. Different compiler families use different ABIs.

This matters tremendously for:

- shared libraries,
- plugins,
- cross-compiler linking,
- binary serialization of object layouts,
- reverse engineering,
- debugging and crash forensics.

GCC and Clang on mainstream Unix-like targets

On common Linux and many Unix-like targets, GCC and Clang generally follow the Itanium C++ ABI family for object layout, name mangling, RTTI structure, and vtable organization. Despite the name, this ABI is not limited to the historical Itanium processor. It became the dominant C++ ABI specification for many non-Windows targets.
Important characteristics commonly associated with this ABI family include:

- vtable address points embedded in richer table structures,
- offset-to-top entries,
- explicit typeinfo pointers in the vtable structure,
- detailed rules for primary bases, secondary bases, and virtual bases,
- thunk and adjustment conventions defined in ABI terms.

For practical purposes, GCC and Clang are often binary-compatible with each other on the same platform when they target the same underlying ABI and standard library family, though details such as standard library versioning and flags still matter.

MSVC on Windows

MSVC uses a distinct C++ ABI model on Windows. While it also uses vftables and hidden pointers, the organization, naming, RTTI structures, and multiple-inheritance details differ from the Itanium-style ABI used by GCC and Clang on many Unix-like systems.
Important practical consequences include:

- class binary layout is not assumed identical to Itanium-style ABI implementations,
- RTTI data structures are different,
- vtable-related metadata and adjustment mechanisms are represented differently,
- name mangling is different,
- direct binary interoperability between MSVC-built polymorphic class hierarchies and GCC/Clang Unix-style ABI layouts must not be assumed.

This is why cross-compiler or cross-ABI boundaries should generally avoid exposing C++ class layouts directly and instead prefer C interfaces, COM-style interfaces on Windows, abstract C-compatible handles, or carefully controlled plugin boundaries.

Why GCC and Clang are often grouped together

GCC and Clang are often discussed together because on major non-Windows targets they usually follow the same ABI specification family. However, that does not mean every internal code-generation detail is identical. It means they generally agree on externally visible ABI contracts required for linking and object interoperability on those targets.

Why MSVC must be treated separately

MSVC must be treated separately because the Windows C++ ABI ecosystem is historically distinct. Even if source-level semantics are the same, low-level layout details such as:

- class layout in multiple inheritance,
- RTTI metadata structures,
- virtual base handling,
- mangled symbol names,
- destructor entry conventions,

are different enough that one must not assume drop-in binary compatibility with GCC/Clang Itanium-style layouts.

10.1.8 Illustrative comparison mindset

The safest mental model is:

- source-level C++ semantics are standardized,
- binary layout details are ABI-governed,
- GCC/Clang commonly share an ABI family on Unix-like targets,
- MSVC uses its own ABI family on Windows.

Therefore, when your code depends on exact vtable shape or class layout, you are no longer working only at the language level. You are working at the compiler-ABI level.

10.1.9 Practical implications for Windows developers

For Windows-focused work, especially when using MSVC, the following rules are important:

1. Do not expose polymorphic C++ class layouts across compiler-family boundaries unless the ABI contract is explicitly controlled.

2. Prefer abstract interfaces with well-defined build environments when distributing binary libraries.

3. Use the same compiler family, version family, and runtime configuration across producer and consumer when binary compatibility matters.

4. Avoid assuming GCC/Clang Unix ABI documentation applies directly to MSVC layout details.

5. Remember that the standard guarantees virtual semantics, not one universal vtable layout.

10.1.10 Pedagogical object-layout examples

The following examples are useful for experimenting with object sizes and pointer adjustments on Windows.

Example 1: single inheritance size observation

```
#include <iostream>

class Plain
{
public:
    int x;
};

class Poly
{
public:
    virtual ~Poly() = default;
```

```
    int x;
};

int main()
{
    std::cout << "sizeof(Plain) = " << sizeof(Plain) << '\n';
    std::cout << "sizeof(Poly)  = " << sizeof(Poly) << '\n';
}
```

This demonstrates the hidden space cost typically introduced by polymorphism.

Example 2: multiple inheritance pointer adjustment

```
#include <iostream>

class Left
{
public:
    virtual ~Left() = default;
    int lx{1};
};

class Right
{
public:
    virtual ~Right() = default;
    int rx{2};
};

class Derived : public Left, public Right
```

```
{
public:
    int dz{3};
};

int main()
{
    Derived d;

    Left* lp = &d;
    Right* rp = &d;
    Derived* dp = &d;

    std::cout << "Derived* = " << static_cast<void*>(dp) << '\n';
    std::cout << "Left*    = " << static_cast<void*>(lp) << '\n';
    std::cout << "Right*   = " << static_cast<void*>(rp) << '\n';
}
```

This helps visualize how secondary base pointers can point to interior offsets of the complete object.

Example 3: virtual dispatch through different views

```
#include <iostream>

class Left
{
public:
    virtual ~Left() = default;
    virtual void id()
```

```
    {
        std::cout << "Left\n";
    }
};

class Right
{
public:
    virtual ~Right() = default;
    virtual void id()
    {
        std::cout << "Right\n";
    }
};

class Derived : public Left, public Right
{
public:
    void id() override
    {
        std::cout << "Derived\n";
    }
};

int main()
{
    Derived d;

    Left* lp = &d;
```

```
    Right* rp = &d;

    lp->id();
    rp->id();
}
```

Depending on the ABI and implementation, one or both call paths may involve thunk-based adjustment before reaching the final overriding function body.

Example 4: virtual inheritance experiment

```
#include <iostream>

class VBase
{
public:
    virtual ~VBase() = default;
    int vb{0};
};

class Left : virtual public VBase
{
public:
    int lx{1};
};

class Right : virtual public VBase
{
public:
    int rx{2};
```

```
};

class Final : public Left, public Right
{
public:
    int fz{3};
};

int main()
{
    std::cout << "sizeof(VBase) = " << sizeof(VBase) << '\n';
    std::cout << "sizeof(Left)  = " << sizeof(Left) << '\n';
    std::cout << "sizeof(Right) = " << sizeof(Right) << '\n';
    std::cout << "sizeof(Final) = " << sizeof(Final) << '\n';
}
```

This shows how virtual inheritance often increases layout complexity and object size.

10.1.11 Deep design lessons

A deep understanding of vtables leads to several important architectural lessons.

Lesson 1: dynamic polymorphism is not free

A polymorphic class often carries:

- additional object size due to one or more vptrs,
- indirect-call cost,
- potential cache effects,

- more complicated ABI and layout behavior under inheritance.

This does not make virtual dispatch bad. It means it should be used deliberately.

Lesson 2: ABI matters for library boundaries

The source code:

```
struct Base
{
    virtual ~Base() = default;
    virtual void f() = 0;
};
```

looks simple, but the binary representation depends on the ABI. Therefore such types are safe abstractions at the source level, but binary-level compatibility must still be controlled carefully.

Lesson 3: vtables are an implementation technique, not a language promise

The standard specifies virtual semantics, not one mandated table structure. Vtables are the dominant implementation strategy in mainstream compilers, but one should always remember that the deeper guarantee is semantic behavior, not a specific table shape.

Lesson 4: multiple inheritance and virtual inheritance are layout features, not merely syntax

Once multiple inheritance enters the design, hidden pointer adjustments and more complex dispatch metadata become part of runtime behavior. This has consequences for performance, object size, and ABI stability.

10.1.12 Windows compilation guide

MSVC

```
cl /std:c++23 /EHsc /W4 /d1reportSingleClassLayoutBase example.cpp
cl /std:c++23 /EHsc /W4 /d1reportSingleClassLayoutDerived example.cpp
```

The `/d1reportSingleClassLayout<class>` form is a useful MSVC diagnostic aid for studying implementation layout details during learning and debugging.

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

GCC via MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

Recommended study workflow on Windows

For studying vtable internals on Windows:

1. compile small inheritance examples with MSVC,
2. inspect `sizeof` results and pointer-adjustment addresses,
3. use MSVC class-layout diagnostics,
4. compare behavior with Clang or GCC builds where practical,
5. treat ABI observations as compiler-specific unless intentionally generalized.

10.1.13 Summary

Vtable structure is the dominant practical mechanism used by mainstream C++ implementations to realize dynamic polymorphism. A compiler generates virtual-dispatch support for polymorphic classes, installs hidden vptr values in objects, and arranges table entries so overridden functions can be reached correctly at runtime.

In simple single inheritance, the layout is relatively straightforward. In multiple inheritance and virtual inheritance, offsets, subobject boundaries, adjustment thunks, and ABI metadata become central. GCC and Clang commonly follow the Itanium-style ABI family on major Unix-like targets, while MSVC uses a distinct Windows C++ ABI model.

For Modern C++ developers, the most important conclusion is this:

> Virtual dispatch semantics are portable at the language level, but vtable shape, object layout, and RTTI representation are ABI-level properties that must be understood carefully whenever binary compatibility or deep runtime architecture matters.

10.2 Multiple/Virtual Inheritance

Multiple inheritance and virtual inheritance are among the most technically rich parts of the C++ object model. They extend ordinary single inheritance so that a class can inherit from more than one base class, and they also provide a mechanism to share a common base subobject across several inheritance paths. These features are powerful, but they also introduce deeper layout rules, more complicated dispatch machinery, pointer adjustments, and important performance consequences.

At the language level, multiple inheritance means that a class may have more than one direct base class. Virtual inheritance means that a base class is inherited as a virtual base, so that the most-derived object contains only one shared instance of that base even if it is reached through more than one path.

For advanced runtime understanding, three topics are essential:

1. diamond problems,
2. adjustor thunks,
3. performance implications.

10.2.1 Basic multiple inheritance

A class with more than one direct base class is a multiple-inheritance class.

```
class Left
{
public:
    virtual ~Left() = default;
    virtual void lf();
    int lx;
};

class Right
{
public:
    virtual ~Right() = default;
    virtual void rf();
    int rx;
};

class Derived : public Left, public Right
{
public:
```

```
    void lf() override;
    void rf() override;
    int dz;
};
```

At the source level, `Derived` inherits both interfaces and both data members. At the implementation level, this usually means that the complete object contains multiple base subobjects, and each polymorphic base subobject may need its own virtual-dispatch entry point.
A simplified conceptual layout often looks like this:

```
Derived object:
  [Left subobject]
  [Right subobject]
  [Derived members]
```

If the base classes are polymorphic, each relevant base subobject may carry its own hidden virtual-table pointer view.

10.2.2 Virtual inheritance

Virtual inheritance changes the rules for base-subobject placement. When a base is inherited virtually, the language guarantees that in the complete most-derived object there is only one shared instance of that virtual base, even if it is reached through multiple inheritance paths.

```
class Base
{
public:
    virtual ~Base() = default;
    int value;
```

```
};

class Left : virtual public Base
{
public:
    int lx;
};

class Right : virtual public Base
{
public:
    int rx;
};

class Final : public Left, public Right
{
public:
    int fz;
};
```

Here, `Final` contains one shared `Base` subobject, not two separate copies. This is the central semantic purpose of virtual inheritance.

10.2.3 Diamond problems

The diamond problem is the most famous structural issue solved by virtual inheritance.

The classic diamond

Consider this non-virtual inheritance hierarchy:

```
class Base
{
public:
    int id;
};

class Left : public Base
{
};

class Right : public Base
{
};

class Final : public Left, public Right
{
};
```

The inheritance graph is diamond-shaped:

```
      Base
     /    \
  Left    Right
     \    /
      Final
```

Without virtual inheritance, `Final` contains two separate `Base` subobjects:

- one through `Left`,
- one through `Right`.

This causes ambiguity.

Ambiguous member access

```
int main()
{
    Final obj;
    obj.id = 10;
}
```

This is ill-formed because the compiler cannot know whether `id` refers to the `Base` subobject inherited through `Left` or the one inherited through `Right`.
The programmer would have to qualify the path explicitly:

```
int main()
{
    Final obj;
    obj.Left::id = 10;
    obj.Right::id = 20;
}
```

This shows that two distinct base subobjects really exist.

Why this is a structural problem

The issue is not only syntactic ambiguity. It is also semantic duplication:

- duplicated state,
- duplicated base identity,
- larger object size,
- more complicated conversions,

- potentially surprising behavior in shared-base designs.

If the common base represents one logical shared capability or one logical identity, two independent copies are often wrong.

Virtual inheritance solves the diamond duplication

Now consider the same hierarchy with virtual inheritance:

```
class Base
{
public:
    int id;
};

class Left : virtual public Base
{
};

class Right : virtual public Base
{
};

class Final : public Left, public Right
{
};
```

Now the complete `Final` object contains only one shared `Base` subobject.
Member access is no longer ambiguous in the same way:

```
int main()
```

```
{
    Final obj;
    obj.id = 42;
}
```

This now refers to the unique shared `Base` subobject.

Construction responsibilities

When virtual inheritance is used, construction of the virtual base is the responsibility of the most-derived class, not the intermediate classes in the ordinary way.

```
#include <iostream>

class Base
{
public:
    explicit Base(int x)
    {
        std::cout << "Base(" << x << ")\n";
    }
};

class Left : virtual public Base
{
public:
    Left() : Base(1)
    {
        std::cout << "Left\n";
    }
```

```
};

class Right : virtual public Base
{
public:
    Right() : Base(2)
    {
        std::cout << "Right\n";
    }
};

class Final : public Left, public Right
{
public:
    Final() : Base(99), Left(), Right()
    {
        std::cout << "Final\n";
    }
};
```

When constructing a `Final` object, the virtual base `Base` is constructed under control of `Final`, the most-derived class. The `Base(1)` and `Base(2)` mentions in the intermediate classes do not determine the final virtual-base construction for the most-derived object.

Polymorphic diamond

The diamond problem also interacts with virtual dispatch.

```
#include <iostream>
```

```
class Base
{
public:
    virtual ~Base() = default;
    virtual void f()
    {
        std::cout << "Base\n";
    }
};

class Left : virtual public Base
{
public:
    void f() override
    {
        std::cout << "Left\n";
    }
};

class Right : virtual public Base
{
public:
    void f() override
    {
        std::cout << "Right\n";
    }
};

class Final : public Left, public Right
```

```
{
public:
    void f() override
    {
        std::cout << "Final\n";
    }
};
```

The implementation must now support:

- one shared `Base` subobject,
- correct dispatch through `Base*`,
- correct dispatch through `Left*`,
- correct dispatch through `Right*`,
- correct pointer adjustments between these views.

This is one reason why virtual inheritance requires richer table metadata and more complex object-layout machinery than ordinary single inheritance.

10.2.4 Adjustor thunks

An adjustor thunk is a small compiler-generated helper used when a virtual call or related operation needs to adjust the `this` pointer before entering the final implementation. Thunks are a major implementation technique in multiple inheritance and virtual inheritance.

Why `this` adjustment is needed

In a single-inheritance case, the base subobject often begins at offset zero, so a base pointer and the complete-object pointer may coincide. In multiple inheritance, this is often no longer true.

```
#include <iostream>

class Left
{
public:
    virtual ~Left() = default;
    int lx{1};
};

class Right
{
public:
    virtual ~Right() = default;
    int rx{2};
};

class Derived : public Left, public Right
{
public:
    int dz{3};
};

int main()
{
    Derived d;

    Left* lp = &d;
    Right* rp = &d;
    Derived* dp = &d;
```

```
    std::cout << static_cast<void*>(dp) << '\n';
    std::cout << static_cast<void*>(lp) << '\n';
    std::cout << static_cast<void*>(rp) << '\n';
}
```

On typical implementations, the `Right*` address differs from the `Derived*` address because the `Right` subobject begins at a nonzero offset inside the complete object.
If a virtual call is made through `Right*`, but the overriding function body expects the `this` pointer in another view, the implementation may need an adjustment step.

Conceptual thunk model

Suppose `Derived` overrides a virtual function originally declared in `Right`. A call through `Right*` might conceptually look like this:

1. load the vptr from the `Right` subobject,
2. load the virtual-call target from the corresponding slot,
3. jump to an adjustor thunk,
4. thunk adjusts `this`,
5. thunk jumps to the actual overriding function body.

This adjustment is necessary because the override may be implemented in terms of the complete-object layout or another subobject layout.

Example illustrating secondary-base dispatch

```
#include <iostream>

class Left
{
public:
    virtual ~Left() = default;
    virtual void f()
    {
        std::cout << "Left::f\n";
    }
};

class Right
{
public:
    virtual ~Right() = default;
    virtual void f()
    {
        std::cout << "Right::f\n";
    }
};

class Derived : public Left, public Right
{
public:
    void f() override
    {
        std::cout << "Derived::f\n";
```

```
    }
};

int main()
{
    Derived d;

    Left* lp = &d;
    Right* rp = &d;

    lp->f();
    rp->f();
}
```

Both calls must ultimately reach `Derived::f()`, but the two starting pointer values may differ. The implementation must therefore ensure that the correct `this` reaches the final body, often through a thunk or ABI-specific equivalent entry point.

Virtual-base adjustments

Virtual inheritance makes adjustment even more complicated because the offset to the shared virtual base may depend on the most-derived layout. The runtime can no longer always rely on a single fixed compile-time offset in the same simple way as for non-virtual secondary bases. This is one reason ABIs for virtual inheritance include richer metadata such as virtual-base offsets and related virtual-call adjustment information.

Thunk role in destructors

Adjustor thunks are not only about ordinary virtual methods. They also matter for destructor dispatch and deleting through base pointers in complex hierarchies.

```
class Base1
{
public:
    virtual ~Base1() = default;
};

class Base2
{
public:
    virtual ~Base2() = default;
};

class Derived : public Base1, public Base2
{
public:
    ~Derived() override = default;
};
```

If a `Base2*` points into the interior of a `Derived` object and is deleted, the implementation must recover the correct complete-object context for destruction and deallocation. ABI-specific destructor entries and adjustment mechanisms make this possible.

Thunks and covariant returns

Although this section focuses on inheritance layout, it is important to note that thunks also arise in related situations such as covariant return adjustments. In such cases, the implementation may need to adjust either the incoming `this` pointer, the outgoing result pointer, or both, depending on the ABI and the exact override relationship.

Why thunks matter conceptually

Even if the programmer never sees a thunk in source code, understanding them explains:

- why multiple inheritance is more complex at runtime,
- why some virtual calls involve more than a simple table lookup,
- why ABI documentation distinguishes direct entry points from adjusting entry points,
- why binary layout and dispatch conventions are compiler-ABI properties, not purely language-level concepts.

10.2.5 Performance implications

Multiple inheritance and virtual inheritance are correct language features, but they are not free. Their performance implications should be understood carefully, especially in systems code, graphics code, game engines, plugin architectures, and low-level frameworks.

Object size

The first cost is often larger object size. A class with multiple polymorphic bases may contain more than one hidden virtual-table pointer. Virtual inheritance may require additional layout support and can increase the space needed for metadata and alignment.

```
#include <iostream>

class A
{
public:
    virtual ~A() = default;
    int a;
```

```
};

class B
{
public:
    virtual ~B() = default;
    int b;
};

class C : public A, public B
{
public:
    int c;
};

int main()
{
    std::cout << sizeof(A) << '\n';
    std::cout << sizeof(B) << '\n';
    std::cout << sizeof(C) << '\n';
}
```

On typical implementations, `C` is noticeably larger than a naive sum of plain integers would suggest, because the object must carry polymorphic base structure and alignment padding.

Pointer adjustment cost

A virtual call in a single-inheritance hierarchy is often relatively direct: load one vptr, fetch one function entry, and call it. In multiple inheritance, additional pointer adjustment may be required before the final function executes.

That means the cost model may include:

- extra address adjustment,
- thunk indirection,
- more complicated code generation around dispatch.

This does not always mean huge overhead, but it does mean the dispatch path is less trivial than in the simplest single-inheritance case.

Virtual inheritance cost

Virtual inheritance typically costs more than ordinary multiple inheritance because the location of a virtual base may need runtime-supported metadata rather than being treated as a simple fixed direct subobject layout in all contexts.
Common consequences include:

- more complicated object layout,
- more complex constructor and destructor logic,
- more metadata in virtual-table-related structures,
- more work for upcasts, cross-casts, and some virtual calls,
- potentially larger code size.

Constructor and destructor complexity

During construction and destruction, virtual inheritance may require special virtual-table states or ABI-specific construction support. This makes constructor and destructor code larger and more complicated than in simple class hierarchies.

The reason is that the runtime must preserve correct semantics while different base layers are being built or torn down, and in the presence of virtual bases the final offsets and dispatch views are more intricate.

Cache and locality effects

Because multiple-inheritance and virtual-inheritance objects can be larger and have more scattered subobject views, they may also influence cache behavior.
Potential effects include:

- fewer objects fitting in the same cache lines,
- more pointer-chasing in polymorphic structures,
- more instruction-cache pressure due to thunks and complex dispatch paths,
- less predictable memory-layout intuition for hot loops.

These effects matter most in performance-critical object-heavy code.

Impact on pointer-to-member representations

On MSVC, pointer-to-member representations differ depending on whether the class uses no inheritance, single inheritance, multiple inheritance, or virtual inheritance. This is a strong practical signal that more complex inheritance models require richer representations and more interpretation code.
For programmers, this means that the inheritance model affects not only object layout, but also the low-level representation of related language features.

Cost of `dynamic_cast`

Cross-casting and checked casting are typically more expensive in multiple and virtual inheritance because the implementation must navigate more complex runtime type information and base-subobject relationships.
Example:

```
#include <iostream>

class Base
{
public:
    virtual ~Base() = default;
};

class Left : virtual public Base
{
};

class Right : virtual public Base
{
};

class Final : public Left, public Right
{
};

int main()
{
    Final f;
```

```
    Left* lp = &f;

    if (Right* rp = dynamic_cast<Right*>(lp))
    {
        std::cout << "cross-cast succeeded\n";
    }
}
```

This kind of checked cross-cast depends on RTTI and the ABI's representation of the full inheritance graph.

When the cost is acceptable

The existence of these costs does not mean multiple or virtual inheritance should never be used. The correct design question is whether the semantic model justifies the runtime and layout complexity.
The costs are often acceptable when:

- the class hierarchy models real orthogonal interfaces,
- shared-base semantics are genuinely required,
- object count is moderate,
- the code is not in the hottest dispatch path,
- interface quality and architectural clarity matter more than the last few nanoseconds of dispatch.

When to be cautious

A programmer should be more cautious when:

- designing high-frequency hot-path polymorphic objects,
- building data-oriented systems where object size is critical,
- exposing binary interfaces across compiler or ABI boundaries,
- mixing complex inheritance with performance-sensitive virtual dispatch loops,
- using virtual inheritance only to mimic a design style rather than to solve a real shared-base problem.

10.2.6 Design patterns and guidance

Use multiple inheritance primarily for interfaces or true orthogonal roles

One strong design pattern is to use multiple inheritance when a type genuinely combines orthogonal interface roles.

```
class Serializable
{
public:
    virtual ~Serializable() = default;
    virtual void save() const = 0;
};

class Drawable
{
public:
    virtual ~Drawable() = default;
    virtual void draw() const = 0;
};
```

```
class Icon : public Serializable, public Drawable
{
public:
    void save() const override
    {
    }

    void draw() const override
    {
    }
};
```

This is often clearer than forcing unrelated capabilities into one artificial base class.

Use virtual inheritance only when one shared base is semantically required

Virtual inheritance should usually be reserved for cases where one shared base subobject is the correct semantic model.
A classic example is a shared identity or shared state base in a diamond hierarchy.

```
class Entity
{
public:
    int id;
};

class Networked : virtual public Entity
{
};
```

```
class Renderable : virtual public Entity
{
};

class Player : public Networked, public Renderable
{
};
```

Here, having one shared `Entity` base is often the intended design.

Prefer composition when inheritance complexity is not semantically justified

If shared-base semantics are not essential, composition is often easier to maintain and more predictable at runtime.

```
class Position
{
public:
    int x{0};
    int y{0};
};

class Health
{
public:
    int hp{100};
};

class Player
{
```

```
    Position pos_;
    Health health_;
};
```

Composition avoids many of the ABI and layout complexities discussed above.

10.2.7 Extensive examples

Example 1: non-virtual diamond ambiguity

```
class Base
{
public:
    int v{0};
};

class Left : public Base
{
};

class Right : public Base
{
};

class Final : public Left, public Right
{
};

int main()
{
```

```
    Final f;

    f.Left::v = 1;
    f.Right::v = 2;
}
```

This program demonstrates that two distinct base subobjects exist.

Example 2: virtual diamond with shared base

```
#include <iostream>

class Base
{
public:
    int v{0};
};

class Left : virtual public Base
{
};

class Right : virtual public Base
{
};

class Final : public Left, public Right
{
};
```

```
int main()
{
    Final f;
    f.v = 42;
    std::cout << f.v << '\n';
}
```

Now only one shared base subobject exists.

Example 3: pointer adjustment in multiple inheritance

```
#include <iostream>

class A
{
public:
    virtual ~A() = default;
};

class B
{
public:
    virtual ~B() = default;
};

class C : public A, public B
{
};

int main()
```

```
{
    C obj;

    C* cp = &obj;
    A* ap = &obj;
    B* bp = &obj;

    std::cout << static_cast<void*>(cp) << '\n';
    std::cout << static_cast<void*>(ap) << '\n';
    std::cout << static_cast<void*>(bp) << '\n';
}
```

This helps visualize that different base views can point to different offsets inside the same object.

Example 4: virtual dispatch through multiple bases

```
#include <iostream>

class A
{
public:
    virtual ~A() = default;
    virtual void run()
    {
        std::cout << "A::run\n";
    }
};

class B
```

```
{
public:
    virtual ~B() = default;
    virtual void run()
    {
        std::cout << "B::run\n";
    }
};

class C : public A, public B
{
public:
    void run() override
    {
        std::cout << "C::run\n";
    }
};

int main()
{
    C obj;
    A* ap = &obj;
    B* bp = &obj;

    ap->run();
    bp->run();
}
```

The implementation must ensure that both calls resolve correctly to `C::run()`.

Example 5: virtual inheritance constructor behavior

```
#include <iostream>

class Base
{
public:
    explicit Base(int x)
    {
        std::cout << "Base(" << x << ")\n";
    }
};

class Left : virtual public Base
{
public:
    Left() : Base(1)
    {
        std::cout << "Left\n";
    }
};

class Right : virtual public Base
{
public:
    Right() : Base(2)
    {
        std::cout << "Right\n";
    }
};
```

```
class Final : public Left, public Right
{
public:
    Final() : Base(99), Left(), Right()
    {
        std::cout << "Final\n";
    }
};

int main()
{
    Final f;
}
```

This shows the most-derived-class responsibility for the virtual base.

Example 6: size comparison

```
#include <iostream>

class Plain
{
public:
    int x;
};

class SI
{
public:
```

```
    virtual ~SI() = default;
    int x;
};

class MI1
{
public:
    virtual ~MI1() = default;
    int a;
};

class MI2
{
public:
    virtual ~MI2() = default;
    int b;
};

class Multi : public MI1, public MI2
{
public:
    int c;
};

class VB
{
public:
    virtual ~VB() = default;
    int v;
```

```
};

class VL : virtual public VB
{
public:
    int l;
};

class VR : virtual public VB
{
public:
    int r;
};

class Diamond : public VL, public VR
{
public:
    int d;
};

int main()
{
    std::cout << "sizeof(Plain)   = " << sizeof(Plain) << '\n';
    std::cout << "sizeof(SI)      = " << sizeof(SI) << '\n';
    std::cout << "sizeof(Multi)   = " << sizeof(Multi) << '\n';
    std::cout << "sizeof(Diamond) = " << sizeof(Diamond) << '\n';
}
```

This example helps study the space cost of different inheritance models.

10.2.8 Windows build notes

MSVC

```
cl /std:c++23 /EHsc /W4 /permissive- /nologo example.cpp
cl /std:c++23 /EHsc /W4 /d1reportSingleClassLayoutBase
↪  /d1reportSingleClassLayoutFinal example.cpp
```

Clang on Windows

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

GCC on Windows via MinGW-w64

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

Recommended Windows study workflow

For studying multiple and virtual inheritance on Windows:

1. compile small controlled examples,
2. compare pointer addresses for different base views,
3. inspect `sizeof` results,
4. use MSVC class-layout diagnostics for deeper layout inspection,
5. avoid assuming that GCC/Clang Unix ABI layout rules exactly match MSVC internals.

10.2.9 Concluding design rules

Multiple inheritance and virtual inheritance are not merely syntax-level language features. They are runtime-layout features with real ABI, dispatch, and performance implications.

1. Use multiple inheritance when a type truly combines distinct interface roles.
2. Use virtual inheritance when one shared base subobject is semantically necessary.
3. Understand that the diamond problem is fundamentally about duplicated base subobjects and ambiguous identity.
4. Remember that adjustor thunks exist because different base views may require different `this` adjustments.
5. Expect larger object layouts and more complex dispatch paths in multiple and especially virtual inheritance.
6. Prefer composition when inheritance complexity does not buy real semantic clarity.

The deepest lesson is that advanced inheritance in C++ is powerful because it models real structural relationships, but that power is implemented through richer runtime machinery. Good design therefore requires both semantic discipline at the source level and respect for the object-model costs at the ABI and performance level.

Dynamic Polymorphism Best Practices

11.1 When to use virtual

Dynamic polymorphism in C++ enables runtime selection of behavior through base-class interfaces. This mechanism is implemented using virtual functions and is a cornerstone of many extensible architectures such as plugin systems, framework APIs, device abstraction layers, and runtime component models.
However, virtual dispatch introduces both conceptual and runtime costs. Therefore, it should be used deliberately rather than automatically.

11.1.1 The fundamental purpose of virtual functions

A virtual function allows a program to call a function through a base-class interface while executing the correct implementation belonging to the dynamic type of the object.

```
#include <iostream>

class Shape
{
public:
    virtual ~Shape() = default;
```

```
    virtual double area() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle(double r) : radius_(r) {}

    double area() const override
    {
        return 3.141592653589793 * radius_ * radius_;
    }

private:
    double radius_;
};

class Rectangle : public Shape
{
public:
    Rectangle(double w, double h) : width_(w), height_(h) {}

    double area() const override
    {
        return width_ * height_;
    }

private:
    double width_;
```

```
    double height_;
};
```

Now the caller can operate on a heterogeneous collection of shapes.

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

int main()
{
    std::vector<std::unique_ptr<Shape>> shapes;

    shapes.push_back(std::make_unique<Circle>(3.0));
    shapes.push_back(std::make_unique<Rectangle>(4.0, 5.0));

    for (const auto& s : shapes)
    {
        std::cout << s->area() << '\n';
    }
}
```

The correct implementation is selected at runtime through virtual dispatch.

11.1.2 Use virtual when behavior varies by derived type

The most appropriate use of virtual functions is when different derived classes provide different implementations of a shared conceptual operation.
Examples include:

- graphical objects with different drawing logic,

- device drivers for different hardware,
- serialization logic for different data structures,
- plugin-based extensibility systems.

11.1.3 Avoid virtual when behavior is fixed

If a function will never vary across derived types, it should not be virtual.

```
class Logger
{
public:
    void write_message(const std::string& msg)
    {
        buffer_.push_back(msg);
    }

private:
    std::vector<std::string> buffer_;
};
```

Here the behavior is uniform. Making such functions virtual introduces unnecessary runtime cost and complexity.

11.1.4 Prefer non-virtual interfaces when polymorphism is unnecessary

Many designs can use composition or templates instead of dynamic polymorphism. Compile-time polymorphism using templates often produces simpler and faster code.

```
template<typename T>
```

```
double compute_area(const T& obj)
{
    return obj.area();
}
```

This uses static polymorphism instead of dynamic dispatch.

11.1.5 Guideline for deciding

Virtual functions should be used when:

- behavior depends on runtime object type,
- the interface must remain stable while implementations vary,
- objects are manipulated through base pointers or references,
- runtime extensibility is required.

Virtual functions should be avoided when:

- behavior is known at compile time,
- performance-critical loops dominate execution,
- templates or composition provide a clearer solution.

11.2 Avoiding OOP anti-patterns

Object-oriented programming provides powerful modeling tools, but misuse can lead to brittle designs and inefficient systems. Several anti-patterns appear frequently when dynamic polymorphism is used without careful design.

11.2.1 Deep inheritance hierarchies

Large inheritance trees often make systems harder to maintain and reason about.

```
class Widget
{
};

class VisualWidget : public Widget
{
};

class InteractiveWidget : public VisualWidget
{
};

class ButtonWidget : public InteractiveWidget
{
};

class ColoredButtonWidget : public ButtonWidget
{
};
```

Such hierarchies become fragile because:

- each layer introduces coupling,
- small changes propagate widely,
- object layout becomes more complex.

Prefer shallow hierarchies combined with composition.

11.2.2 The fragile base class problem

When derived classes depend heavily on the internal behavior of base classes, small modifications in the base class can break derived classes.
Example scenario:

```
class Base
{
public:
    virtual void process()
    {
        step1();
        step2();
    }

protected:
    virtual void step1() {}
    virtual void step2() {}
};
```

Derived classes may override `step1` or `step2`. If the base implementation changes order or semantics, derived classes may behave incorrectly.
Design recommendation:

- keep base-class invariants simple,
- avoid exposing too many override points,
- document expectations clearly.

11.2.3 Inheritance used only for code reuse

Inheritance should represent an "is-a" relationship, not merely reuse of implementation.
Incorrect design:

```
class Stack : public std::vector<int>
{
};
```

This exposes vector operations that violate stack semantics.
Better design:

```
class Stack
{
public:
    void push(int v)
    {
        data_.push_back(v);
    }

    void pop()
    {
        data_.pop_back();
    }

private:
    std::vector<int> data_;
};
```

This uses composition rather than inheritance.

11.2.4 Virtual functions used unnecessarily

Excessive virtual methods increase object size and dispatch overhead.
Bad example:

```
class Counter
{
public:
    virtual int value() const
    {
        return count_;
    }

private:
    int count_{0};
};
```

If no derived behavior exists, the virtual keyword should not be used.

11.2.5 Ignoring the rule of zero

Classes participating in polymorphism often interact with resource management. A poor design mixes manual memory management with inheritance.
Prefer RAII wrappers and smart pointers.

11.3 Virtual destructor rules

One of the most important rules of polymorphic design in C++ concerns destructors.
If a class is intended to be used polymorphically and objects may be deleted through a pointer to the base class, the base class must have a virtual destructor.

11.3.1 Why virtual destructors are required

Consider the following incorrect example.

```
class Base
{
public:
    ~Base() = default;
};

class Derived : public Base
{
public:
    ~Derived()
    {
        std::cout << "Derived destroyed\n";
    }
};
```

Now consider deleting through a base pointer.

```
int main()
{
    Base* ptr = new Derived;
    delete ptr;
}
```

Without a virtual destructor in `Base`, only the base destructor runs. The derived destructor is skipped, which leads to undefined behavior and possible resource leaks.

11.3.2 Correct polymorphic destructor design

```
class Base
{
public:
    virtual ~Base() = default;
};

class Derived : public Base
{
public:
    ~Derived() override
    {
        std::cout << "Derived destroyed\n";
    }
};
```

Now deletion through a base pointer invokes the correct destructor chain.

11.3.3 Protected destructors for non-polymorphic deletion

If a base class should not be deleted through a base pointer, its destructor can be protected.

```
class Interface
{
protected:
    ~Interface() = default;
};
```

This prevents accidental deletion via base pointers.

11.3.4 Pure virtual destructors

A destructor can be declared pure virtual to make the class abstract, but it still requires a definition.

```
class Abstract
{
public:
    virtual ~Abstract() = 0;
};

Abstract::~Abstract() = default;
```

Even though the destructor is pure virtual, the base class must provide an implementation.

11.4 Polymorphism + RAII interactions

Dynamic polymorphism and RAII (Resource Acquisition Is Initialization) interact in important ways when managing resources in object hierarchies.

11.4.1 Polymorphic objects with RAII-managed resources

Derived classes often manage resources such as memory, files, or network handles. RAII ensures that these resources are released when the object is destroyed.

```
#include <fstream>
#include <memory>

class Logger
{
```

```
public:
    virtual ~Logger() = default;
    virtual void log(const std::string& msg) = 0;
};

class FileLogger : public Logger
{
public:
    explicit FileLogger(const std::string& file)
        : file_(file)
    {
    }

    void log(const std::string& msg) override
    {
        file_ << msg << '\n';
    }

private:
    std::ofstream file_;
};
```

The file resource is automatically closed when `FileLogger` is destroyed.

11.4.2 RAII with smart pointers and polymorphism

Modern C++ encourages storing polymorphic objects in smart pointers.

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

```
int main()
{
    std::vector<std::unique_ptr<Logger>> loggers;

    loggers.push_back(std::make_unique<FileLogger>("log.txt"));

    for (const auto& l : loggers)
    {
        l->log("hello");
    }
}
```

Here:

- `unique_ptr` manages ownership,
- virtual destructors ensure correct cleanup,
- RAII guarantees deterministic destruction.

11.4.3 Polymorphic RAII wrappers

Sometimes RAII objects themselves participate in polymorphism.

```
class Lock
{
public:
    virtual ~Lock() = default;
    virtual void acquire() = 0;
    virtual void release() = 0;
};
```

Derived implementations may wrap different synchronization primitives.

```
#include <mutex>

class MutexLock : public Lock
{
public:
    void acquire() override
    {
        m_.lock();
    }

    void release() override
    {
        m_.unlock();
    }

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

A higher-level RAII guard may use such polymorphic locks.

```
class LockGuard
{
public:
    explicit LockGuard(Lock& l)
        : lock_(l)
    {
        lock_.acquire();
```

```
    }

    ~LockGuard()
    {
        lock_.release();
    }

private:
    Lock& lock_;
};
```

This demonstrates how polymorphism and RAII cooperate to enforce safe resource management.

11.4.4 RAII and polymorphic destruction chains

Because RAII depends on deterministic destruction, correct destructor dispatch is critical.

```
#include <memory>

class Resource
{
public:
    virtual ~Resource() = default;
};

class FileResource : public Resource
{
public:
    ~FileResource()
```

```
    {
        // cleanup file
    }
};

int main()
{
    std::unique_ptr<Resource> r = std::make_unique<FileResource>();
}
```

When `r` goes out of scope, the correct derived destructor runs, ensuring proper resource cleanup.

11.4.5 Design guidance

Combining RAII and dynamic polymorphism safely requires several design rules.

- Always provide a virtual destructor in polymorphic base classes.
- Prefer smart pointers to manage polymorphic object lifetimes.
- Keep base classes lightweight and focused on interface definition.
- Ensure derived classes maintain strong exception safety.
- Avoid manual resource management inside polymorphic hierarchies.

11.4.6 Compilation guide for Windows

Compile using MSVC:

```
cl /std:c++23 /EHsc /W4 example.cpp
```

Compile using Clang:

```
clang++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

Compile using GCC (MinGW-w64):

```
g++ -std=c++23 -Wall -Wextra -pedantic example.cpp -o example.exe
```

11.4.7 Summary

Dynamic polymorphism enables runtime flexibility and extensible architecture. When combined with RAII and modern resource management, it allows safe and maintainable object-oriented systems.
Best practices include:

- use virtual functions only when runtime polymorphism is required,
- keep inheritance hierarchies simple,
- always define virtual destructors for polymorphic bases,
- combine polymorphism with RAII and smart pointers for safe resource management,
- avoid anti-patterns such as deep inheritance chains and misuse of inheritance for code reuse.

Careful application of these principles ensures that dynamic polymorphism remains a powerful and safe design tool in Modern C++ systems.

Part V

Static Polymorphism, Templates & Type Erasure

CRTP Deep Dive

12.1 Static polymorphism mechanics

Static polymorphism is a form of polymorphic behavior resolved entirely during compilation rather than at runtime. Instead of storing a vptr, consulting a vtable, and dispatching through an indirect call, the compiler selects concrete operations directly from the type information available during template instantiation. In Modern C++, the most widely recognized idiom for expressing static polymorphism in class hierarchies is the Curiously Recurring Template Pattern (CRTP), where a derived type passes itself as a template argument to its base.
The essential CRTP structure is conceptually simple:

```
template<typename Derived>
class Base
{
public:
    Derived& derived()
    {
        return static_cast<Derived&>(*this);
    }

    const Derived& derived() const
    {
```

```
        return static_cast<const Derived&>(*this);
    }

    void interface()
    {
        derived().implementation();
    }
};

class Widget : public Base<Widget>
{
public:
    void implementation()
    {
        // concrete behavior
    }
};
```

Here, `Base<Widget>` knows the exact concrete type at compile time. When `interface()` calls `derived().implementation()`, the compiler does not need dynamic dispatch. It sees the exact target function and can inline aggressively when optimization is enabled.

12.1.1 Why CRTP works

CRTP works because template instantiation turns the generic base into a concrete base specialized for the derived type. After substitution, the compiler effectively reasons about code that looks as though it were handwritten specifically for the concrete class.
For example, after instantiating `Base<Widget>`, the compiler knows that:

```
Derived& derived();
```

becomes logically equivalent to:

```
Widget& derived();
```

and therefore:

```
derived().implementation();
```

becomes a direct call on `Widget`. This gives the compiler much more room for inlining, dead-code elimination, constant propagation, and devirtualization-like optimization without requiring a virtual function at all.

12.1.2 The role of `static_cast`

The key operation in CRTP is the cast from the base subobject to the actual derived object:

```
static_cast<Derived&>(*this)
```

This cast is valid only when the current object is truly a base subobject of `Derived`. That is why the CRTP base must only be used through a correctly derived class. A misuse such as constructing the base independently is logically invalid and should be prevented by design. A common defensive technique is to keep the base constructor protected:

```
template<typename Derived>
class Base
{
protected:
    Base() = default;
    ~Base() = default;

public:
```

```
    Derived& derived()
    {
        return static_cast<Derived&>(*this);
    }

    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};
```

This does not make the pattern magically safe against all misuse, but it prevents the most obvious incorrect standalone instantiation of the base.

12.1.3 A full static polymorphism example

The following example shows a reusable CRTP base that exposes a public algorithm while requiring the derived class to provide a single customization point.

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

template<typename Derived>
class LoggerBase
{
public:
    void log(std::string_view message)
    {
        before_log();
```

```
        derived().write_log(message);
        after_log();
    }

protected:
    void before_log()
    {
        std::cout << "[BEGIN] ";
    }

    void after_log()
    {
        std::cout << " [END]\n";
    }

private:
    Derived& derived()
    {
        return static_cast<Derived&>(*this);
    }
};

class ConsoleLogger : public LoggerBase<ConsoleLogger>
{
public:
    void write_log(std::string_view message)
    {
        std::cout << message;
    }
```

```
};

int main()
{
    ConsoleLogger logger;
    logger.log("CRTP static dispatch");
}
```

In this design, `LoggerBase` owns the reusable algorithmic skeleton, while `ConsoleLogger` supplies only the specific behavior. This is one of the strongest practical uses of CRTP: implementing a template method pattern without runtime polymorphism.

12.1.4 CRTP versus virtual dispatch

A virtual-function design would look similar at a high level, but the mechanism differs fundamentally:

```
class LoggerBase
{
public:
    virtual ~LoggerBase() = default;

    void log(std::string_view message)
    {
        before_log();
        write_log(message);
        after_log();
    }

protected:
```

```
    virtual void write_log(std::string_view message) = 0;

    void before_log()
    {
        std::cout << "[BEGIN] ";
    }

    void after_log()
    {
        std::cout << " [END]\n";
    }
};
```

The virtual form provides substitutability through a base pointer or reference. The CRTP form provides compile-time reuse and compile-time dispatch. The choice depends on whether runtime substitution is required. If objects must be stored heterogeneously behind one abstract interface, CRTP alone is insufficient. If the goal is zero-overhead reuse for statically known types, CRTP is often a strong fit.

12.1.5 A modern note: explicit object parameters

Recent Modern C++ evolution introduced explicit object parameters, often referred to as `deducing this`. This feature can reduce boilerplate in some cases where CRTP was historically used to recover cv/ref correctness or to forward behavior across related types. However, CRTP remains highly relevant for mixins, interface layering, policy assembly, and reusable compile-time behavior across families of types. Thus, explicit object parameters complement CRTP; they do not eliminate it.

12.2 Compile-time interface design

One of the most important uses of CRTP is compile-time interface design. In this style, a base template defines the shape of an interface, while the derived type is expected to provide certain operations. The compiler validates the arrangement during template instantiation.

12.2.1 Unconstrained interface design

The traditional CRTP style predates concepts and uses documentation and compiler errors to define the required interface:

```
template<typename Derived>
class Shape
{
public:
    double area() const
    {
        return derived().area_impl();
    }

    void draw() const
    {
        derived().draw_impl();
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
```

```
};

class Circle : public Shape<Circle>
{
public:
    explicit Circle(double r) : radius(r) {}

    double area_impl() const
    {
        return 3.14159265358979323846 * radius * radius;
    }

    void draw_impl() const
    {
        std::cout << "Drawing Circle with radius = " << radius << '\n';
    }

private:
    double radius{};
};
```

This works well, but if a derived type forgets to define `draw_impl()` or `area_impl()`, the resulting template diagnostics may be longer than desired.

12.2.2 Constrained interface design with concepts

C++20 concepts greatly improve compile-time interface design by allowing explicit declaration of requirements. This makes CRTP more readable, more maintainable, and easier to diagnose.

```
#include <concepts>
```

```
#include <iostream>

template<typename T>
concept ShapeRequirements =
    requires(const T& t)
    {
        { t.area_impl() } -> std::convertible_to<double>;
        { t.draw_impl() } -> std::same_as<void>;
    };

template<typename Derived>
    requires ShapeRequirements<Derived>
class Shape
{
public:
    double area() const
    {
        return derived().area_impl();
    }

    void draw() const
    {
        derived().draw_impl();
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
```

```
    }
};

class Rectangle : public Shape<Rectangle>
{
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area_impl() const
    {
        return width * height;
    }

    void draw_impl() const
    {
        std::cout << "Rectangle(" << width << ", " << height << ")\n";
    }

private:
    double width{};
    double height{};
};
```

This design is dramatically clearer than unconstrained CRTP. The interface contract is now encoded directly in the language rather than being implicit in documentation alone.

12.2.3 Separating public interface from customization points

A strong CRTP interface is usually divided into two layers:

1. A stable public interface exposed by the CRTP base.

2. A narrow set of customization points implemented by the derived type.

This pattern reduces duplication and centralizes common algorithmic logic.

```
#include <iostream>
#include <string_view>

template<typename Derived>
class Formatter
{
public:
    void print(std::string_view text) const
    {
        auto formatted = derived().format_impl(text);
        std::cout << formatted << '\n';
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};

class BracketFormatter : public Formatter<BracketFormatter>
{
public:
    std::string format_impl(std::string_view text) const
    {
        return "[" + std::string(text) + "]";
```

```
    }
};

class AngleFormatter : public Formatter<AngleFormatter>
{
public:
    std::string format_impl(std::string_view text) const
    {
        return "<" + std::string(text) + ">";
    }
};
```

The public interface `print()` exists only once, while each derived class implements only `format_impl()`.

12.2.4 Compile-time interface validation with `static_assert`

In addition to concepts, `static_assert` can be used to enforce assumptions:

```
#include <type_traits>

template<typename Derived>
class NumericMixin
{
    static_assert(std::is_class_v<Derived>,
                  "NumericMixin requires a class type");

public:
    int double_value() const
    {
```

```
        return derived().value() * 2;
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};
```

In modern code, concepts are typically preferable for interface requirements, while `static_assert` remains useful for expressing invariants and internal assumptions.

12.2.5 A standard-library-adjacent example mindset

Modern C++ standard library design demonstrates that compile-time interfaces can be layered effectively. The ranges ecosystem, for example, uses concepts to express semantic requirements and a helper base (`std::ranges::view_interface`) to provide reusable interface functionality for derived view types. That style is extremely instructive for CRTP authors: the public interface can be selectively synthesized based on what the derived type already provides, while constraints keep invalid instantiations from producing chaotic errors.

12.2.6 When compile-time interface design is appropriate

Compile-time interface design is especially appropriate when:

1. The concrete type is known statically.
2. Maximum inlining opportunity is desirable.
3. The interface should be enforced by concepts rather than runtime abstract bases.

4. The design benefits from code reuse across many small value-like or view-like types.
5. No heterogeneous runtime container of base pointers is required.

It is less appropriate when:

1. Runtime substitution is required.
2. ABI stability across separately compiled binaries is important.
3. Template-instantiation cost or code size is a stronger concern than dispatch cost.
4. Users require non-template binary interfaces.

12.3 CRTP mixins & policy classes

CRTP becomes especially powerful when used to create mixins and policy-based designs. A mixin adds reusable behavior to a type. A policy class parameterizes a design decision such as storage strategy, locking strategy, formatting style, or arithmetic mode.

12.3.1 CRTP mixins

A mixin is a class that contributes behavior without necessarily representing an is-a relationship in the domain model. CRTP mixins are ideal for adding opt-in functionality.

```
#include <iostream>
#include <string>
#include <utility>

template<typename Derived>
class Printable
```

```
{
public:
    void print() const
    {
        std::cout << derived().to_string() << '\n';
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};

template<typename Derived>
class Comparable
{
public:
    bool operator==(const Comparable& other) const
    {
        return derived().compare_key() ==
               static_cast<const Derived&>(other).compare_key();
    }

    bool operator!=(const Comparable& other) const
    {
        return !(*this == other);
    }
```

```
private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};

class Person
    : public Printable<Person>,
      public Comparable<Person>
{
public:
    Person(std::string n, int a) : name(std::move(n)), age(a) {}

    std::string to_string() const
    {
        return "Person{name=" + name + ", age=" + std::to_string(age) + "}";
    }

    auto compare_key() const
    {
        return std::pair{name, age};
    }

private:
    std::string name;
    int age{};
};
```

```
int main()
{
    Person p1{"Ayman", 40};
    Person p2{"Ayman", 40};

    p1.print();
    std::cout << std::boolalpha << (p1 == p2) << '\n';
}
```

This example shows two independent mixins layered onto one concrete type. Each mixin adds a capability while remaining statically bound.

12.3.2 Design advice for mixins

When writing CRTP mixins:

1. Keep each mixin small and single-purpose.
2. Avoid hidden data members unless they are truly part of the abstraction.
3. Avoid assuming unrelated capabilities unless they are documented as requirements or constrained with concepts.
4. Prefer stateless mixins when possible.
5. Keep the public interface stable and the derived customization surface narrow.

12.3.3 Policy classes

A policy class is not necessarily a CRTP base. It is typically a template parameter that supplies a strategy. CRTP and policy classes are often combined to build highly reusable designs.
The next example uses policy classes for behavior selection:

```
#include <iostream>
#include <mutex>
#include <string>
#include <string_view>

struct NoLockPolicy
{
    void lock()   {}
    void unlock() {}
};

struct MutexLockPolicy
{
    void lock()
    {
        m.lock();
    }

    void unlock()
    {
        m.unlock();
    }

private:
    std::mutex m;
};

struct ConsoleWritePolicy
{
```

```
    void write(std::string_view text)
    {
        std::cout << text << '\n';
    }
};

template<typename LockPolicy, typename WritePolicy>
class Logger : private LockPolicy, private WritePolicy
{
public:
    void log(std::string_view text)
    {
        LockGuard guard(*this);
        this->write(text);
    }

private:
    class LockGuard
    {
    public:
        explicit LockGuard(LockPolicy& policy) : p(policy)
        {
            p.lock();
        }

        ~LockGuard()
        {
            p.unlock();
        }
```

```
    private:
        LockPolicy& p;
    };
};

int main()
{
    Logger<NoLockPolicy, ConsoleWritePolicy> logger1;
    logger1.log("single-threaded logger");

    Logger<MutexLockPolicy, ConsoleWritePolicy> logger2;
    logger2.log("thread-safe logger");
}
```

Here the type is assembled at compile time from orthogonal policies. This is one of the cleanest examples of zero-overhead configurability in C++.

12.3.4 Combining CRTP with policy classes

CRTP can provide interface reuse while policy classes provide behavior variation:

```
#include <iostream>
#include <string_view>

struct UpperCasePolicy
{
    static void transform_and_print(std::string_view text)
    {
        for (char ch : text)
```

```
        {
            if (ch >= 'a' && ch <= 'z')
                std::cout << static_cast<char>(ch - 'a' + 'A');
            else
                std::cout << ch;
        }
        std::cout << '\n';
    }
};

struct IdentityPolicy
{
    static void transform_and_print(std::string_view text)
    {
        std::cout << text << '\n';
    }
};

template<typename Derived, typename Policy>
class MessageBase
{
public:
    void show() const
    {
        Policy::transform_and_print(derived().message());
    }

private:
    const Derived& derived() const
```

```
    {
        return static_cast<const Derived&>(*this);
    }
};

class WarningMessage
    : public MessageBase<WarningMessage, UpperCasePolicy>
{
public:
    std::string_view message() const
    {
        return "warning: static polymorphism in action";
    }
};

class PlainMessage
    : public MessageBase<PlainMessage, IdentityPolicy>
{
public:
    std::string_view message() const
    {
        return "plain output";
    }
};
```

This pattern is expressive and highly efficient. The structure of the interface comes from the CRTP base, while the operational variation comes from the policy.

12.3.5 Mixin pitfalls

Despite its power, CRTP mixin design has pitfalls:

1. Name lookup can become subtle in multiple-base designs.
2. Error messages can become long when requirements are implicit.
3. The wrong granularity of mixins can produce fragmented, difficult-to-read APIs.
4. Stateful mixins can complicate object layout and construction.
5. Cross-mixin coupling can quietly reintroduce inheritance complexity.

The best mitigation is disciplined design: narrow responsibilities, explicit constraints, and careful attention to object semantics.

12.3.6 Using `std::enable_shared_from_this` as a familiar pattern

A very well-known standard-library facility, `std::enable_shared_from_this<T>`, is often taught as a familiar example of a self-referential base template. Although its purpose is ownership management rather than general static polymorphism, it demonstrates how a class template parameterized on the derived type can inject useful behavior into a concrete class. This makes it pedagogically useful when introducing CRTP-like structures.

12.4 CRTP performance advantages

The performance advantages of CRTP come from what it allows the compiler to know. When the target type is fixed at compile time, the compiler can reason about the whole call chain much more aggressively than in runtime-polymorphic code.

12.4.1 Direct call opportunities

With CRTP, calls typically become direct calls on concrete types. Under optimization, these direct calls are often inlined.

```
template<typename Derived>
class AccumulatorBase
{
public:
    int run(int x) const
    {
        return derived().step1(x) + derived().step2(x);
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};

class FastAccumulator : public AccumulatorBase<FastAccumulator>
{
public:
    int step1(int x) const { return x + 1; }
    int step2(int x) const { return x * 2; }
};
```

When optimization is enabled, `run()`, `step1()`, and `step2()` may all inline into the caller. That enables further simplification. For example, constant arguments can lead to constant folding. Temporary results can disappear entirely.

12.4.2 No vptr, no vtable lookup, no indirect branch

A classic runtime-polymorphic object usually contains a vptr and dispatches through an indirect call sequence. CRTP-based objects normally do not require that machinery merely to provide interface reuse. This can reduce:

1. Indirection at the call site.
2. Per-object metadata overhead associated with dynamic polymorphism.
3. Branch-prediction uncertainty caused by indirect calls.

This matters most in very hot code paths, tight loops, small value-like types, and template-heavy generic libraries where abstraction cost must remain negligible.

12.4.3 Example: static versus dynamic dispatch

```
#include <chrono>
#include <iostream>
#include <memory>
#include <vector>

class DynamicBase
{
public:
    virtual ~DynamicBase() = default;
    virtual int compute(int x) const = 0;
};

class DynamicDerived : public DynamicBase
{
```

```
public:
    int compute(int x) const override
    {
        return x * 3 + 1;
    }
};

template<typename Derived>
class StaticBase
{
public:
    int compute(int x) const
    {
        return derived().compute_impl(x);
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};

class StaticDerived : public StaticBase<StaticDerived>
{
public:
    int compute_impl(int x) const
    {
        return x * 3 + 1;
```

```
    }
};

int main()
{
    constexpr int iterations = 50'000'000;

    {
        std::unique_ptr<DynamicBase> obj = std::make_unique<DynamicDerived>();
        volatile int sum = 0;

        auto start = std::chrono::steady_clock::now();
        for (int i = 0; i < iterations; ++i)
            sum += obj->compute(i);
        auto end = std::chrono::steady_clock::now();

        std::cout << "Dynamic: "
                  << std::chrono::duration_cast<std::chrono::milliseconds>(end -
                  ↪  start).count()
                  << " ms, sum = " << sum << '\n';
    }

    {
        StaticDerived obj;
        volatile int sum = 0;

        auto start = std::chrono::steady_clock::now();
        for (int i = 0; i < iterations; ++i)
            sum += obj.compute(i);
```

```
        auto end = std::chrono::steady_clock::now();

        std::cout << "Static:  "
                  << std::chrono::duration_cast<std::chrono::milliseconds>(end -
                  ↪  start).count()
                  << " ms, sum = " << sum << '\n';
    }
}
```

This benchmark is pedagogical. Actual results depend on compiler, optimization level, processor, branch prediction, and surrounding code. In real systems, dynamic dispatch may be negligible or dominant depending on context. The correct lesson is not that CRTP is always faster, but that CRTP removes one category of runtime cost and usually increases optimization freedom.

12.4.4 Better optimization visibility

The real strength of CRTP is not only avoiding virtual dispatch; it is increasing the optimizer's visibility. Once concrete operations become visible in the same instantiation context, the compiler may:

1. Inline small functions.
2. Propagate constants.
3. Eliminate dead branches.
4. Remove temporary objects.
5. Specialize code paths for the exact type.

This is one reason generic programming in Modern C++ can achieve near hand-written performance.

12.4.5 Data layout benefits

CRTP mixins are often stateless. Empty stateless bases may benefit from empty-base optimization. This can allow behavior to be added without increasing object size in many implementations.

```
#include <iostream>

template<typename Derived>
class Taggable
{
public:
    const char* tag() const
    {
        return derived().tag_impl();
    }

private:
    const Derived& derived() const
    {
        return static_cast<const Derived&>(*this);
    }
};

class Packet : public Taggable<Packet>
{
public:
    int id{42};

    const char* tag_impl() const
```

```
    {
        return "network-packet";
    }
};

int main()
{
    std::cout << sizeof(Packet) << '\n';
}
```

If the mixin remains stateless, the object may carry the behavior with little or no layout penalty beyond the actual state members of the concrete type.

12.4.6 The cost side of CRTP

A rigorous treatment must also state the costs. CRTP can improve runtime performance but may increase:

1. Compile time due to template instantiation.
2. Code size due to many specializations.
3. Debugging complexity in heavily layered generic designs.
4. Error verbosity when constraints are weak or absent.

This trade-off is extremely important in large code bases. Modern tooling can help identify expensive template instantiations, and disciplined design can keep CRTP-based abstractions lightweight.

12.4.7 When CRTP performance advantages are most meaningful

CRTP is especially attractive in the following situations:

1. Numeric kernels and hot loops.
2. Small adapters and views.
3. Embedded or systems code where object layout matters.
4. Compile-time configurable frameworks.
5. Library internals where static type information is naturally available.

Its advantages are less compelling when:

1. Dispatch is cold or infrequent.
2. Runtime substitution is architecturally required.
3. Shared binary interfaces matter more than marginal call overhead.
4. The dominant cost lies elsewhere, such as I/O, allocation, or cache misses from larger data structures.

12.4.8 Practical benchmark guidance on Windows

For meaningful CRTP measurements on Windows, prefer the following process:

1. Build in `Release`.
2. Enable high optimization.
3. Benchmark outside the debugger.

4. Use large enough iteration counts to reduce timer noise.
5. Prevent the compiler from optimizing away the full computation.
6. Compare not only wall-clock time but also generated assembly when appropriate.

12.4.9 Windows build guidance for the examples

For Visual Studio 2022:

```
Project Properties
  C/C++ -> Language -> C++ Language Standard -> ISO C++20 Standard (/std:c++20)
  C/C++ -> Optimization -> Optimization -> Maximize Speed (/O2)
  C/C++ -> Code Generation -> Enable C++ Exceptions -> Yes (/EHsc)
```

For command-line compilation with MSVC:

```
cl /std:c++20 /EHsc /O2 /W4 crtp_example.cpp
```

If a specific example uses concepts or ranges, the same configuration is suitable. If experimenting with newer explicit object parameter syntax, use the latest language mode supported by the compiler:

```
cl /std:c++latest /EHsc /O2 /W4 crtp_example.cpp
```

For MinGW-w64 g++ on Windows:

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic crtp_example.cpp -o crtp_example.exe
```

12.5 Best practices for CRTP in large systems

12.5.1 Prefer CRTP for behavior reuse, not domain modeling

CRTP is strongest when used as an implementation technique rather than as the primary way to express business-domain inheritance. In other words, use it to provide reusable capabilities such as formatting, comparison, counting, validation, visitation support, lightweight interfaces, and view-like helpers.

12.5.2 Constrain everything important

In Modern C++20 and later, unconstrained CRTP should increasingly be seen as a legacy style. Concepts make contracts explicit and diagnostics better.

12.5.3 Keep customization points minimal

A CRTP base should require as few derived operations as possible. The smaller the required surface, the easier the pattern is to reason about and document.

12.5.4 Avoid hidden semantic traps

A CRTP base should not quietly assume ownership rules, exception behavior, thread-safety guarantees, or ordering semantics unless those are part of a clearly documented contract.

12.5.5 Do not force CRTP where virtual dispatch is the right model

If the system fundamentally requires runtime polymorphism, plugin boundaries, heterogeneous collections, or ABI-oriented interface stability, then virtual functions or type-erasure techniques are often the correct design.

12.6 CRTP and type erasure in the bigger picture

This chapter belongs to a part focused on static polymorphism, templates, and type erasure. These techniques are related but solve different problems.

12.6.1 CRTP

CRTP provides compile-time reuse and static dispatch. It is excellent when concrete types are known at compile time.

12.6.2 Virtual functions

Virtual dispatch provides classic runtime polymorphism through inheritance and base pointers or references.

12.6.3 Type erasure

Type erasure provides runtime polymorphism without requiring a common inheritance hierarchy in user types. It is often useful when heterogeneous runtime storage is required but inheritance would be too restrictive or intrusive.

12.6.4 Choosing correctly

A useful summary is:

1. Choose CRTP when the type is statically known and abstraction overhead should disappear.
2. Choose virtual dispatch when substitutability through a stable abstract base is the core requirement.

3. Choose type erasure when runtime heterogeneity is needed but user-facing inheritance is undesirable.

12.7 Conclusion

CRTP remains one of the most important static polymorphism techniques in Modern C++. Its enduring value comes from the combination of three properties:

1. It enables strong compile-time interface layering.
2. It enables mixin-based code reuse and policy-driven composition.
3. It enables highly optimizable abstractions that often compile down to direct, inlined code.

In modern practice, CRTP is most effective when combined with concepts, narrow customization points, stateless mixins where possible, and careful awareness of the trade-off between runtime efficiency and compile-time complexity. It should be treated neither as a universal replacement for virtual functions nor as an outdated idiom. Instead, it is a precise and powerful tool that continues to occupy a central place in the architecture of high-performance Modern C++ libraries and systems.

Variant-Based Polymorphism

13.1 std::variant as a type-safe union

`std::variant` is a discriminated union introduced in C++17 that provides a type-safe alternative to traditional unions and many inheritance-based designs. Unlike a raw union, it guarantees that only one active value exists at a time and enforces type correctness at compile time.

Conceptually, `std::variant<Ts...>` represents a closed set of types. At runtime, it holds exactly one of those types and tracks the active alternative via an index.

13.1.1 Basic usage

```
#include <variant>
#include <iostream>
#include <string>

int main()
{
    std::variant<int, double, std::string> v;

    v = 42;
```

```
    std::cout << std::get<int>(v) << '\n';

    v = 3.14;
    std::cout << std::get<double>(v) << '\n';

    v = "Hello Variant";
    std::cout << std::get<std::string>(v) << '\n';
}
```

The type safety is enforced at compile time. Attempting to access the wrong type results in a well-defined exception or compile-time error depending on usage.

13.1.2 Safe access mechanisms

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, double> v = 10;

    if (auto p = std::get_if<int>(&v))
        std::cout << "int: " << *p << '\n';
    else
        std::cout << "not int\n";
}
```

`std::get_if` avoids exceptions and allows safe inspection of the current alternative.

13.1.3 Variant as algebraic data type

`std::variant` effectively models algebraic data types (ADTs), where a value can be one of several alternatives. This aligns Modern C++ with functional programming constructs, enabling expressive and type-safe modeling of domain states.

```
#include <variant>
#include <string>

struct FileError { std::string msg; };
struct NetworkError { int code; };

using Result = std::variant<int, FileError, NetworkError>;
```

This replaces many traditional uses of base classes and error hierarchies.

13.1.4 Valueless state

A variant may enter a `valueless_by_exception()` state if an exception occurs during assignment. This is rare but must be considered in robust systems.

```
if (v.valueless_by_exception())
{
    // handle exceptional state
}
```

13.2 Visiting patterns

The primary mechanism for working with `std::variant` is visitation. The function `std::visit` applies a callable to the currently active alternative.

13.2.1 Basic visitation

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, double> v = 5;

    std::visit([](auto&& value)
    {
        std::cout << value << '\n';
    }, v);
}
```

The callable must be valid for all possible alternative types.

13.2.2 Overloaded visitor pattern

A common idiom is to combine multiple lambdas into a single visitor.

```
#include <variant>
#include <iostream>
#include <string>

template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
```

```
int main()
{
    std::variant<int, double, std::string> v = "hello";

    std::visit(overloaded{
        [](int i) { std::cout << "int: " << i << '\n'; },
        [](double d) { std::cout << "double: " << d << '\n'; },
        [](const std::string& s) { std::cout << "string: " << s << '\n'; }
    }, v);
}
```

This pattern is widely used because it enables clean separation of behavior per type.

13.2.3 Returning values from visitors

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, double> v = 10;

    auto result = std::visit([](auto x) -> double
    {
        return x * 2;
    }, v);

    std::cout << result << '\n';
}
```

The return type must be consistent across all branches.

13.2.4 Multiple variant visitation

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, double> v1 = 2;
    std::variant<int, double> v2 = 3.5;

    std::visit([](auto a, auto b)
    {
        std::cout << a + b << '\n';
    }, v1, v2);
}
```

The complexity grows combinatorially with the number of variants, but implementations typically optimize using dispatch tables or switch-based strategies.

13.2.5 C++26 member visit

C++26 introduces a member `visit()` function:

```
#include <variant>
#include <iostream>

int main()
{
```

```
    std::variant<int, double> v = 5;

    v.visit([](auto x)
    {
        std::cout << x << '\n';
    });
}
```

This simplifies syntax and integrates visitation more naturally into object-oriented style.

13.3 Pattern matching (C++23/26)

Modern C++ increasingly moves toward pattern matching semantics built on top of `std::variant`.

13.3.1 Variant-based pattern matching today

Currently, pattern matching is expressed via `std::visit`:

```
std::visit(overloaded{
    [](int i) { /* handle int */ },
    [](double d) { /* handle double */ }
}, v);
```

This is considered a form of manual pattern matching.

13.3.2 Limitations of current approach

- Verbose syntax
- No exhaustiveness checking

- Harder to read for complex nested patterns

13.3.3 Emerging pattern matching proposals

Recent standard proposals introduce a formal pattern matching model based on a *variant protocol*.
Key ideas:

1. Pattern matching operates on variant-like types.
2. Structured matching similar to functional languages.
3. Potential exhaustiveness checking.

13.3.4 Conceptual future syntax

Although not finalized, proposals introduce constructs similar to:

```
// conceptual future syntax
match (v)
{
    case int i:
        // handle int
    case double d:
        // handle double
}
```

These designs aim to provide clearer and safer branching compared to `std::visit`.

13.3.5 Relationship with variant

Pattern matching is fundamentally built on the idea of variant-like types. The standardization effort explicitly defines a protocol so that types such as `std::variant` and `std::expected` can participate uniformly.

13.3.6 Current best practice

Until native pattern matching is fully standardized:

1. Use `std::visit` with overloaded lambdas
2. Keep visitors small and focused
3. Prefer returning values over side effects when possible

13.4 Eliminating inheritance hierarchies

One of the most significant uses of `std::variant` is replacing classical inheritance hierarchies.

13.4.1 Traditional OOP approach

```
class Shape
{
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;
};
```

```
class Circle : public Shape
{
public:
    double area() const override { return 3.14 * r * r; }
private:
    double r{1.0};
};
```

This design relies on:

- virtual dispatch
- heap allocation in many cases
- pointer indirection

13.4.2 Variant-based approach

```
#include <variant>
#include <iostream>

struct Circle
{
    double r;
};

struct Rectangle
{
    double w, h;
};
```

```cpp
using Shape = std::variant<Circle, Rectangle>;

double area(const Shape& s)
{
    return std::visit([](auto&& shape) -> double
    {
        using T = std::decay_t<decltype(shape)>;

        if constexpr (std::is_same_v<T, Circle>)
            return 3.141592653589793 * shape.r * shape.r;
        else if constexpr (std::is_same_v<T, Rectangle>)
            return shape.w * shape.h;
    }, s);
}
```

13.4.3 Advantages

1. No virtual functions
2. No heap allocation required
3. Better cache locality
4. Compile-time exhaustiveness (with care)
5. Easier reasoning about all possible types

13.4.4 Trade-offs

1. Closed set of types (cannot extend externally)
2. Code size may increase due to template instantiation

3. Visitors must handle all alternatives

13.4.5 Solving the expression problem

Variant-based polymorphism is particularly strong when:

- The set of types is fixed
- New operations are frequently added

This is the opposite of traditional OOP, where adding new types is easy but adding new operations is difficult.

13.4.6 Real-world design pattern

A modern system often combines:

- `std::variant` for internal representation
- free functions for operations
- visitors for dispatch

```
struct Add { int a, b; };
struct Multiply { int a, b; };

using Expr = std::variant<Add, Multiply>;

int evaluate(const Expr& e)
{
    return std::visit([](auto&& op)
    {
```

```
        using T = std::decay_t<decltype(op)>;

        if constexpr (std::is_same_v<T, Add>)
            return op.a + op.b;
        else if constexpr (std::is_same_v<T, Multiply>)
            return op.a * op.b;
    }, e);
}
```

13.5 Variant vs CRTP vs Virtual dispatch

13.5.1 Variant-based polymorphism

- Static type set
- Runtime selection
- No inheritance required

13.5.2 CRTP

- Compile-time polymorphism
- No runtime branching
- Best for reusable behavior

13.5.3 Virtual dispatch

- Runtime extensibility
- Supports heterogeneous containers

- Stable ABI boundaries

13.5.4 Decision guideline

1. Use `std::variant` when the type set is closed and operations vary
2. Use CRTP when behavior reuse is primary and types are static
3. Use virtual functions when runtime substitution is required

13.6 Conclusion

Variant-based polymorphism represents a major shift in Modern C++ design philosophy. It enables:

1. Type-safe unions replacing unsafe constructs
2. Functional-style pattern matching via visitation
3. Elimination of many inheritance hierarchies
4. Strong compile-time guarantees with minimal runtime overhead

With ongoing evolution toward native pattern matching, `std::variant` is becoming a foundational building block for expressive, safe, and high-performance polymorphic designs in Modern C++.

Type Erasure Techniques

14.1 Function types

Type erasure in Modern C++ is easiest to understand when it is built on a solid understanding of function types, callable objects, invocation rules, and the distinction between a function itself and an object that can be called like a function.

14.1.1 What a function type is

A function type describes a callable entity by its return type, parameter list, and in modern standard rules also relevant qualifiers such as `noexcept` where applicable. A function type is not the same thing as a function pointer type. For example:

```
int add(int, int);              // function declaration
using Fn = int(int, int);       // function type
using FnPtr = int(*)(int, int); // pointer-to-function type
```

In the example above:

```
Fn
```

is the pure function type, while

```
FnPtr
```

is a pointer type that can point to functions of type `int(int, int)`.
This distinction matters throughout type erasure because wrappers such as `std::function<R(Args...)>` are parameterized by a function signature, not by a function pointer type.

14.1.2 Function types versus callable objects

Modern C++ uses the broader concept of a *callable object*. A callable may be:

1. A free function
2. A function pointer
3. A lambda closure object
4. A functor type with `operator()`
5. A member function pointer combined with an object
6. A reference wrapper to a callable

This broader callable model is what makes type erasure powerful. A wrapper need not care whether the underlying target is a plain function, a stateless lambda, a stateful lambda, or a large function object. It only needs the target to be invocable with the required signature.

14.1.3 Examples of function types

```
#include <iostream>


int square(int x)
```

```
{
    return x * x;
}

double scale(double x, double factor)
{
    return x * factor;
}

int main()
{
    using UnaryInt = int(int);
    using BinaryDouble = double(double, double);

    UnaryInt* p1 = &square;
    BinaryDouble* p2 = &scale;

    std::cout << p1(5) << '\n';
    std::cout << p2(3.0, 2.5) << '\n';
}
```

14.1.4 noexcept and function types

In standard-conforming C++17 and later, `noexcept` participates in function types in relevant contexts. This is important for generic code and for callable wrappers that preserve exception specifications in their signatures.

```
void f() noexcept;
void g();
```

```
using NoThrowFn = void() noexcept;
using ThrowingFn = void();

NoThrowFn* p1 = &f;
// NoThrowFn* p2 = &g; // invalid
ThrowingFn* p3 = &g;
ThrowingFn* p4 = &f;   // valid conversion to less restrictive type
```

This point becomes especially important in modern wrapper families such as `std::move_only_function`, `std::copyable_function`, and `std::function_ref`, which are specified in terms of signatures that may carry cv/ref and `noexcept` qualifiers.

14.1.5 Lambdas as function objects

Lambdas are not function types. A lambda expression creates a unique closure type, which is a class type with a callable `operator()`. That means lambdas participate in callable systems through object semantics, not through being raw functions.

```
#include <iostream>

int main()
{
    auto twice = [](int x) { return x * 2; };

    std::cout << twice(21) << '\n';
}
```

A non-capturing lambda may also be convertible to a function pointer of compatible type:

```
#include <iostream>
```

```
int main()
{
    auto plus_one = [](int x) { return x + 1; };

    int(*fp)(int) = plus_one;

    std::cout << fp(10) << '\n';
}
```

A capturing lambda cannot convert to a plain function pointer because it carries state.

14.1.6 Why this matters for type erasure

Type erasure becomes necessary when code wants to accept or store arbitrary callable objects behind one uniform interface. Without type erasure, the exact concrete closure type of a lambda or the exact functor type becomes part of the surrounding type system. That is often undesirable at interface boundaries, across translation units, or in APIs that want uniformity.

14.2 std::function internals

`std::function` is the classic standard-library owning type-erased callable wrapper. It stores any compatible copyable callable target and exposes a fixed call signature.

14.2.1 Observable contract of std::function

At the language-user level, the most important facts are:

1. It is a wrapper for callable targets with one fixed signature.
2. It may be empty.

3. Invoking an empty wrapper throws `std::bad_function_call`.
4. It supports `target_type()` and `target<T>()` for limited target inspection.
5. It is copyable, which means the stored target must meet the semantic requirements necessary for that ownership model.

```cpp
#include <functional>
#include <iostream>

int add_one(int x)
{
    return x + 1;
}

int main()
{
    std::function<int(int)> f = add_one;

    if (f)
        std::cout << f(10) << '\n';

    f = nullptr;

    if (!f)
        std::cout << "empty\n";
}
```

14.2.2 The standard specifies behavior, not layout

A very important rule for professional C++ programmers is that the standard specifies the *observable behavior* of `std::function`, but it does not require one exact internal layout. Therefore, any discussion of internals must be carefully divided into:

1. What the standard guarantees
2. What implementations commonly do

The standard guarantees invocation behavior, emptiness semantics, destruction of the stored target, and access via `target_type()` and `target<T>()`. It does not mandate a specific object representation such as a virtual base table, a specific manager function signature, or a specific small-buffer size.

14.2.3 Typical implementation model

Although not standardized as a required layout, most implementations conceptually use a structure similar to the following:

1. Storage for the erased target
2. An invoker function pointer used to call the target through erased storage
3. One or more manager operations for copy, move, destroy, and target access

A simplified pedagogical model looks like this:

```
struct FunctionStorage
{
    alignas(std::max_align_t) unsigned char buffer[32];
};
```

```
using InvokeFn = int(*)(const void*, int);
using DestroyFn = void(*)(void*);
using CopyFn = void(*)(const void*, void*);

struct ErasedCallable
{
    FunctionStorage storage;
    InvokeFn invoke{};
    DestroyFn destroy{};
    CopyFn copy{};
};
```

This is not the standards mandated structure. It is only an educational mental model.

14.2.4 Common small-object optimization

Many implementations avoid heap allocation for sufficiently small callable targets. This is widely known as small-object optimization. For `std::function` specifically, the standard does not prescribe a mandatory small-buffer strategy. However, the standards newer callable wrappers explicitly include recommended practice encouraging implementations to avoid dynamic allocation for small contained values.
This means a programmer must not write code that depends on a specific small-buffer size, but it is reasonable to understand that many real implementations optimize small callables.

14.2.5 Invocation path

When `std::function<R(Args...)>` is invoked, the wrapper typically:

1. Checks whether it is empty

2. Dispatches to an erased invoker
3. The invoker reinterprets the stored target as its true concrete type
4. The target is invoked with forwarded arguments

A conceptual example:

```
template<typename T>
int invoke_impl(const void* storage, int x)
{
    auto callable = static_cast<const T*>(storage);
    return (*callable)(x);
}
```

In actual libraries, the machinery is more sophisticated because it must handle construction, assignment, destruction, emptiness, type queries, and allocator-related historical design concerns.

14.2.6 Target access

The standard provides limited reflective access to the stored target:

```
#include <functional>
#include <iostream>
#include <typeinfo>

int triple(int x)
{
    return x * 3;
}
```

```
int main()
{
    std::function<int(int)> f = triple;

    std::cout << f.target_type().name() << '\n';

    if (auto p = f.target<int(*)(int)>())
        std::cout << (*p)(7) << '\n';
}
```

This access is intentionally narrow. Type erasure hides the concrete target behind a fixed interface, so target inspection is available only in a controlled and limited way.

14.2.7 Empty-call behavior

```
#include <functional>
#include <iostream>

int main()
{
    std::function<void()> f;

    try
    {
        f();
    }
    catch (const std::bad_function_call&)
    {
        std::cout << "empty function call\n";
```

```
    }
}
```

This is a major semantic distinction from non-owning wrappers such as `std::function_ref`, whose contracts and preconditions are designed differently.

14.2.8 A practical example

```
#include <functional>
#include <iostream>
#include <string>
#include <vector>

struct PrefixPrinter
{
    std::string prefix;

    void operator()(const std::string& text) const
    {
        std::cout << prefix << text << '\n';
    }
};

void emit(const std::vector<std::string>& items,
          const std::function<void(const std::string&)>& sink)
{
    for (const auto& item : items)
        sink(item);
}
```

```
int main()
{
    std::vector<std::string> items{"one", "two", "three"};

    PrefixPrinter printer{"item: "};
    emit(items, printer);
}
```

This is a typical API style when the convenience of accepting heterogeneous copyable callables matters more than avoiding the overhead of ownership and erasure.

14.2.9 What std::function is not ideal for

`std::function` is not always the right tool. It may be suboptimal when:

1. The target is move-only
2. The wrapper should not own the target
3. The call path is in a very hot inner loop
4. You need exact propagation of cv/ref/`noexcept` properties
5. You want zero allocation and minimal erased state

These pressures are part of why the standard library has expanded its wrapper family with `std::move_only_function`, `std::copyable_function`, and `std::function_ref`.

14.3 Custom type-erased wrappers

Type erasure is not limited to `std::function`. It is a general design technique: store an unknown concrete type behind a fixed interface, and use erased operations to interact with it.

14.3.1 The essence of type erasure

A type-erased wrapper usually contains:

1. Opaque storage or an opaque pointer
2. One or more erased operations
3. Lifetime management logic
4. Optional type recovery hooks

The essential idea is simple: compile the concrete type information into helper functions, then store only erased state plus those helpers at runtime.

14.3.2 A minimal owning callable wrapper

The following example builds a small educational callable wrapper. It is intentionally simplified and is not a production replacement for `std::function`.

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

template<typename Signature>
class SimpleFunction;

template<typename R, typename... Args>
class SimpleFunction<R(Args...)>
{
public:
    SimpleFunction() = default;
```

```
template<typename F>
SimpleFunction(F f)
    : self(std::make_unique<Model<F>>(std::move(f)))
{
}

SimpleFunction(const SimpleFunction& other)
    : self(other.self ? other.self->clone() : nullptr)
{
}

SimpleFunction& operator=(const SimpleFunction& other)
{
    if (this != &other)
        self = other.self ? other.self->clone() : nullptr;
    return *this;
}

SimpleFunction(SimpleFunction&&) noexcept = default;
SimpleFunction& operator=(SimpleFunction&&) noexcept = default;

explicit operator bool() const noexcept
{
    return static_cast<bool>(self);
}

R operator()(Args... args) const
{
```

```
        return self->invoke(std::forward<Args>(args)...);
    }

private:
    struct Concept
    {
        virtual ~Concept() = default;
        virtual R invoke(Args&&... args) const = 0;
        virtual std::unique_ptr<Concept> clone() const = 0;
    };

    template<typename F>
    struct Model final : Concept
    {
        explicit Model(F f_) : f(std::move(f_)) {}

        R invoke(Args&&... args) const override
        {
            return f(std::forward<Args>(args)...);
        }

        std::unique_ptr<Concept> clone() const override
        {
            return std::make_unique<Model<F>>(f);
        }

        F f;
    };
```

```
    std::unique_ptr<Concept> self;
};

int add(int a, int b)
{
    return a + b;
}

int main()
{
    SimpleFunction<int(int, int)> f = add;
    std::cout << f(10, 20) << '\n';

    SimpleFunction<int(int, int)> g =
        [](int a, int b) { return a * b; };

    std::cout << g(6, 7) << '\n';
}
```

14.3.3 What this example teaches

This wrapper demonstrates the core type-erasure mechanics:

1. `Concept` defines the erased interface
2. `Model<F>` binds one concrete type `F` to that interface
3. The wrapper stores a pointer to the erased base
4. Invocation happens through the erased virtual interface

This is already type erasure, even though it uses virtual functions internally. Type erasure is a broader technique than non-virtual polymorphism.

14.3.4 A non-owning callable view

Sometimes ownership is the wrong design. If the caller guarantees that the referenced callable outlives the wrapper, a non-owning erased reference can be far smaller and faster. The new standard `std::function_ref` formalizes this style.
A minimal educational version:

```
#include <iostream>
#include <utility>

template<typename Signature>
class FunctionRef;

template<typename R, typename... Args>
class FunctionRef<R(Args...)>
{
public:
    FunctionRef() = delete;

    template<typename F>
    FunctionRef(F& f) noexcept
        : object(static_cast<void*>(&f))
    {
        thunk = [](void* obj, Args... args) -> R
        {
            return (*static_cast<F*>(obj))(std::forward<Args>(args)...);
        };
```

```
    }

    R operator()(Args... args) const
    {
        return thunk(object, std::forward<Args>(args)...);
    }

private:
    void* object{};
    R (*thunk)(void*, Args...){};
};

int main()
{
    auto lambda = [](int x) { return x * 5; };

    FunctionRef<int(int)> view = lambda;

    std::cout << view(8) << '\n';
}
```

This example mirrors the central idea behind `std::function_ref`: one erased pointer to the bound entity plus one erased thunk used for invocation.

14.3.5 Important lifetime warning

A non-owning erased wrapper does not extend the lifetime of the callable. This is its main advantage and its main risk.

```
FunctionRef<int(int)> make_bad_ref()
```

```
{
    auto local = [](int x) { return x + 100; };
    return FunctionRef<int(int)>(local); // dangling after return
}
```

This is invalid design because the referenced callable dies when the function returns.

14.3.6 Policy-based erased wrapper

Type erasure can be customized by policy. For example, a wrapper can be:

1. Owning or non-owning
2. Copyable or move-only
3. Heap-based or small-buffer-based
4. Exception-throwing on empty call or contract-based with preconditions

A modern understanding of type erasure should view `std::function`, `std::move_only_function`, `std::copyable_function`, and `std::function_ref` as points in this design space rather than as unrelated facilities.

14.3.7 A custom type-erased drawable wrapper

Type erasure is not limited to callables. The next example erases a drawing interface without requiring inheritance in user types.

```
#include <iostream>
#include <memory>
#include <string>
#include <utility>
```

```
class Drawable
{
public:
    template<typename T>
    Drawable(T obj)
        : self(std::make_unique<Model<T>>(std::move(obj)))
    {
    }

    Drawable(const Drawable& other)
        : self(other.self ? other.self->clone() : nullptr)
    {
    }

    Drawable& operator=(const Drawable& other)
    {
        if (this != &other)
            self = other.self ? other.self->clone() : nullptr;
        return *this;
    }

    Drawable(Drawable&&) noexcept = default;
    Drawable& operator=(Drawable&&) noexcept = default;

    void draw() const
    {
        self->draw();
    }
```

```
private:
    struct Concept
    {
        virtual ~Concept() = default;
        virtual void draw() const = 0;
        virtual std::unique_ptr<Concept> clone() const = 0;
    };

    template<typename T>
    struct Model final : Concept
    {
        explicit Model(T value) : object(std::move(value)) {}

        void draw() const override
        {
            object.draw();
        }

        std::unique_ptr<Concept> clone() const override
        {
            return std::make_unique<Model<T>>(object);
        }

        T object;
    };

    std::unique_ptr<Concept> self;
};
```

```
struct Circle
{
    void draw() const
    {
        std::cout << "draw circle\n";
    }
};

struct Rectangle
{
    void draw() const
    {
        std::cout << "draw rectangle\n";
    }
};

int main()
{
    Drawable a = Circle{};
    Drawable b = Rectangle{};

    a.draw();
    b.draw();
}
```

This is an excellent teaching example because it shows type erasure as a general interface technique, not merely as a function-wrapper trick.

14.4 The modern callable-wrapper family

The standard library now presents multiple type-erased callable wrappers, each representing a different design choice.

14.4.1 std::function

`std::function<R(Args...)>` is:

1. Owning
2. Copyable
3. Signature-based
4. Potentially allocation-using
5. Historically the most widely available erased callable wrapper

14.4.2 std::move_only_function

`std::move_only_function` is designed for owning erased callables that need not be copyable. This is important for move-only closures and for modern systems where copying a callback is not always semantically correct or efficient.
Its signature family preserves cv/ref/`noexcept` dimensions, which makes it more expressive than the classic `std::function` model in some advanced designs.

14.4.3 std::copyable_function

`std::copyable_function` generalizes the notion of a callable object with a signature while explicitly preserving richer call qualifiers than the classic `std::function` interface family.

The working draft also includes recommended practice encouraging implementations to avoid dynamic allocation for small contained values.

14.4.4 std::function_ref

`std::function_ref` is non-owning. It behaves more like a view than like an owning box. Its specified model is particularly educational: it stores a bound entity plus a thunk pointer. That makes it conceptually lightweight and excellent for parameter passing when the lifetime is guaranteed externally.

14.4.5 Practical selection guide

A practical mental rule is:

1. Use `std::function` when convenient owning copyable callbacks are desired
2. Use `std::move_only_function` when ownership is needed but copying is not
3. Use `std::copyable_function` when richer call-qualification semantics matter
4. Use `std::function_ref` when you only need a non-owning callable view

14.5 Type erasure vs inheritance vs variant

These three techniques all solve polymorphism problems, but they solve different problems and have different trade-offs.

14.5.1 Type erasure

Type erasure hides the concrete type behind a fixed interface while allowing objects of unrelated types to participate, as long as they satisfy the required operations.

Strengths:

1. Decouples user types from inheritance requirements
2. Supports runtime polymorphism
3. Useful at API boundaries
4. Can be designed as owning or non-owning
5. Good for value-like wrappers

Weaknesses:

1. May allocate
2. May add an indirect call
3. Often hides the exact type from optimization
4. Can increase implementation complexity

14.5.2 Inheritance-based polymorphism

Inheritance-based runtime polymorphism uses a common base class and virtual dispatch.
Strengths:

1. Familiar classical object-oriented model
2. Good when domain objects naturally share a stable abstract base
3. Supports heterogeneous collections of base pointers
4. Works well with plugins and extensible hierarchies

Weaknesses:

1. Intrusive: types must inherit from the base
2. Ties interface to class hierarchy
3. Usually requires heap allocation in many practical designs
4. Can encourage rigid inheritance trees

14.5.3 Variant-based polymorphism

`std::variant` represents a closed set of alternative types and dispatches through visitation.
Strengths:

1. Type-safe closed union
2. No virtual functions required
3. Often better locality than pointer-heavy hierarchies
4. Exhaustive handling can be expressed cleanly

Weaknesses:

1. Closed world: adding a new alternative changes the variant type
2. Can lead to many visitor branches
3. Less suitable when the set of participating types must remain open

14.5.4 A side-by-side conceptual comparison

Suppose a system must support drawing different shapes.
Inheritance version:

```
struct Shape
{
    virtual ~Shape() = default;
    virtual void draw() const = 0;
};
```

Variant version:

```
using Shape = std::variant<Circle, Rectangle, Triangle>;
```

Type-erasure version:

```
class Drawable
{
    // erased draw() interface
};
```

The choice depends on what is fixed and what is open:

1. If the set of concrete types is closed, `std::variant` is often excellent.
2. If the interface must be open to unrelated user types without inheritance coupling, type erasure is attractive.
3. If a stable abstract base is central to the design, inheritance is natural.

14.5.5 The expression-problem perspective

A useful high-level way to compare these models is to ask which axis changes more often:

1. New concrete types
2. New operations

Inheritance often makes adding new types easier and adding new operations harder. `std::variant` often makes adding new operations easier and adding new types harder. Type erasure sits in the middle as a boundary technique: it is often chosen not because it perfectly solves the expression problem, but because it produces a clean runtime-polymorphic interface without requiring inheritance in user types.

14.5.6 Performance perspective

A professional comparison should be realistic:

1. Inheritance usually costs one indirection through virtual dispatch and often pointer-based object management.
2. Type erasure often costs one erased indirection and possibly allocation depending on the wrapper design and target size.
3. Variant often enables stronger optimization when the active set is small and known, but visitation may still branch at runtime.

There is no universal winner. The correct choice is architectural, not ideological.

14.5.7 A decision table

```
Need open set of user-defined types without forcing inheritance:
    Prefer type erasure

Need closed set of alternatives with exhaustive handling:
    Prefer std::variant

Need classic abstract base and virtual substitution:
    Prefer inheritance

Need lightweight non-owning callable parameter:
    Prefer std::function_ref

Need owning callback that may be move-only:
    Prefer std::move_only_function

Need convenient widely supported owning callback:
    Prefer std::function
```

14.6 Advanced design guidance

14.6.1 Do not overuse std::function in hot paths

`std::function` is wonderfully convenient, but convenience is not free. In very hot loops, template parameters, CRTP, direct lambdas, or non-owning callable views may be better choices.

14.6.2 Use non-owning wrappers for parameters when ownership is not needed

A callback parameter that is only used during the function call often should not force ownership. A non-owning wrapper or a forwarding template may be more precise.

14.6.3 Use owning wrappers for storage

If a callable must be stored beyond the call site, ownership becomes essential. That is the natural domain of `std::function`, `std::move_only_function`, and similar wrappers.

14.6.4 Keep erased interfaces narrow

The narrower the erased interface, the easier it is to optimize, document, test, and reason about.

14.6.5 Prefer semantic clarity over cleverness

Type erasure is powerful enough to build elaborate frameworks, but the best production designs often keep it simple: one clear erased interface, one clear ownership model, and explicit lifetime rules.

14.7 Windows build guidance

All examples in this chapter can be compiled in a Windows environment with modern compilers.

14.7.1 Visual Studio 2022

Recommended project settings:

```
Project Properties
  C/C++ -> Language -> C++ Language Standard -> ISO C++20 Standard (/std:c++20)
  C/C++ -> Code Generation -> Enable C++ Exceptions -> Yes (/EHsc)
  C/C++ -> Optimization -> Optimization -> Maximize Speed (/O2) for Release
  C/C++ -> Warning Level -> Level4 (/W4)
```

For examples that use the newest standard-library callable wrappers available in your installed toolset, use the latest language mode if needed:

```
Project Properties
  C/C++ -> Language -> C++ Language Standard -> Preview - Features from the Latest
  ↪  C++ Working Draft (/std:c++latest)
```

14.7.2 MSVC command line

```
cl /std:c++20 /EHsc /W4 chapter14_example.cpp
```

For the newest available working-draft features in your compiler:

```
cl /std:c++latest /EHsc /W4 chapter14_example.cpp
```

14.7.3 MinGW-w64 g++ on Windows

```
g++ -std=c++20 -Wall -Wextra -pedantic chapter14_example.cpp -o
↪  chapter14_example.exe
```

If your installed standard library does not yet provide a newer callable wrapper such as `std::function_ref` or `std::copyable_function`, keep the conceptual chapter text and compile only the custom-wrapper examples or the facilities already available in your toolchain.

14.8 Conclusion

Type erasure is one of the most important architectural techniques in Modern C++. It allows programmers to separate interface from concrete type while preserving runtime flexibility and avoiding intrusive inheritance requirements.
The deepest lessons of this chapter are the following:

1. Function types define the signature layer on which erased callable wrappers are built.
2. `std::function` is the classic owning copyable wrapper, but it is only one point in a much larger design space.
3. Custom wrappers reveal the true mechanics of type erasure: erased storage, erased operations, and controlled lifetime management.
4. Type erasure, inheritance, and `std::variant` are complementary tools, each best suited to different forms of polymorphism.

A mature Modern C++ programmer should be comfortable moving among all of these techniques and selecting the one whose trade-offs most accurately match the architecture, performance goals, ownership model, and extensibility needs of the system being built.

Part VI

GOF Patterns Modernized

GOF Patterns Modernized

15.1 Strategy Pattern

The classical Strategy Pattern encapsulates a family of algorithms behind a common interface and makes them interchangeable. In traditional object-oriented design, this is often implemented through a polymorphic base class with virtual dispatch. In Modern C++, the pattern has become broader and more expressive. It can now be implemented with dynamic polymorphism, static polymorphism, lambdas, function objects, type-erased wrappers, or constrained templates depending on the ownership model, performance goals, and extensibility requirements.

15.1.1 Classical dynamic strategy

The traditional object-oriented form is still valid when runtime substitution is required.

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

class SortStrategy
{
public:
```

```
    virtual ~SortStrategy() = default;
    virtual void sort(std::vector<int>& data) const = 0;
};

class AscendingSort : public SortStrategy
{
public:
    void sort(std::vector<int>& data) const override
    {
        std::sort(data.begin(), data.end());
    }
};

class DescendingSort : public SortStrategy
{
public:
    void sort(std::vector<int>& data) const override
    {
        std::sort(data.begin(), data.end(), std::greater<>{});
    }
};

class SortContext
{
public:
    explicit SortContext(std::unique_ptr<SortStrategy> s)
        : strategy(std::move(s))
    {
    }
```

```
    void set_strategy(std::unique_ptr<SortStrategy> s)
    {
        strategy = std::move(s);
    }

    void execute(std::vector<int>& data) const
    {
        strategy->sort(data);
    }

private:
    std::unique_ptr<SortStrategy> strategy;
};

int main()
{
    std::vector<int> values{4, 1, 9, 2, 7};

    SortContext ctx(std::make_unique<AscendingSort>());
    ctx.execute(values);

    for (int v : values)
        std::cout << v << ' ';
    std::cout << '\n';

    ctx.set_strategy(std::make_unique<DescendingSort>());
    ctx.execute(values);
```

```
    for (int v : values)
        std::cout << v << ' ';
    std::cout << '\n';
}
```

This form is appropriate when the strategy must be stored polymorphically and changed at runtime.

15.1.2 Modern functional strategy with lambdas

Modern C++ often replaces an inheritance-based strategy hierarchy with a callable object.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

class SortContext
{
public:
    using Strategy = std::function<void(std::vector<int>&)>;

    explicit SortContext(Strategy s)
        : strategy(std::move(s))
    {
    }

    void set_strategy(Strategy s)
    {
        strategy = std::move(s);
```

```
    }

    void execute(std::vector<int>& data) const
    {
        strategy(data);
    }

private:
    Strategy strategy;
};

int main()
{
    std::vector<int> values{4, 1, 9, 2, 7};

    SortContext ctx([](std::vector<int>& data)
    {
        std::sort(data.begin(), data.end());
    });

    ctx.execute(values);

    for (int v : values)
        std::cout << v << ' ';
    std::cout << '\n';

    ctx.set_strategy([](std::vector<int>& data)
    {
        std::sort(data.begin(), data.end(), std::greater<>{});
```

```
    });

    ctx.execute(values);

    for (int v : values)
        std::cout << v << ' ';
    std::cout << '\n';
}
```

This version is often simpler and eliminates the need for a dedicated class hierarchy.

15.1.3 Static strategy with templates

If the strategy is known at compile time and performance is critical, templates or concepts are often superior.

```
#include <algorithm>
#include <concepts>
#include <iostream>
#include <vector>

template<typename S>
concept Sorter =
    requires(S s, std::vector<int>& data)
    {
        { s(data) } -> std::same_as<void>;
    };

template<Sorter Strategy>
class SortContext
```

```
{
public:
    explicit SortContext(Strategy s)
        : strategy(std::move(s))
    {
    }

    void execute(std::vector<int>& data) const
    {
        strategy(data);
    }

private:
    Strategy strategy;
};

int main()
{
    auto ascending = [](std::vector<int>& data)
    {
        std::sort(data.begin(), data.end());
    };

    std::vector<int> values{5, 3, 8, 1};

    SortContext ctx(ascending);
    ctx.execute(values);

    for (int v : values)
```

```
        std::cout << v << ' ';
    std::cout << '\n';
}
```

This design eliminates virtual dispatch and allows inlining of the strategy.

15.1.4 Modern guidance

In Modern C++, Strategy is no longer tied to inheritance. A strategy may be:

1. A virtual interface object when runtime openness matters.
2. A lambda or function object when lightweight runtime replacement is enough.
3. A template parameter when the strategy is fixed at compile time.
4. A type-erased callable wrapper when uniform storage is needed.

15.2 Observer Pattern

The classical Observer Pattern defines a one-to-many dependency so that when one object changes state, all dependents are notified. In older C++ code, this pattern often relied on raw pointers and ad hoc lifetime assumptions. Modern C++ improves the design significantly through RAII, smart pointers, weak ownership, lambdas, and safer callback registration models.

15.2.1 Classical interface-based observer

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

```
#include <vector>

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void update(const std::string& message) = 0;
};

class Subject
{
public:
    void add_observer(std::shared_ptr<Observer> obs)
    {
        observers.push_back(obs);
    }

    void notify(const std::string& message)
    {
        auto it = observers.begin();

        while (it != observers.end())
        {
            if (auto obs = it->lock())
            {
                obs->update(message);
                ++it;
            }
            else
```

```
            {
                it = observers.erase(it);
            }
        }
    }

private:
    std::vector<std::weak_ptr<Observer>> observers;
};

class ConsoleObserver : public Observer
{
public:
    explicit ConsoleObserver(std::string n)
        : name(std::move(n))
    {
    }

    void update(const std::string& message) override
    {
        std::cout << name << " received: " << message << '\n';
    }

private:
    std::string name;
};

int main()
{
```

```
    Subject subject;

    auto a = std::make_shared<ConsoleObserver>("A");
    auto b = std::make_shared<ConsoleObserver>("B");

    subject.add_observer(a);
    subject.add_observer(b);

    subject.notify("initial event");

    b.reset();

    subject.notify("second event");
}
```

The use of `std::weak_ptr` is a major modern improvement. It prevents the subject from forcing observer lifetime and avoids reference cycles.

15.2.2 Observer with callback registration

Many modern systems do not require an observer class hierarchy. A callback-based approach is often more direct.

```
#include <functional>
#include <iostream>
#include <string>
#include <vector>

class Subject
{
```

```
public:
    using Callback = std::function<void(const std::string&)>;

    void subscribe(Callback cb)
    {
        callbacks.push_back(std::move(cb));
    }

    void notify(const std::string& message) const
    {
        for (const auto& cb : callbacks)
            cb(message);
    }

private:
    std::vector<Callback> callbacks;
};

int main()
{
    Subject subject;

    subject.subscribe([](const std::string& msg)
    {
        std::cout << "Observer 1: " << msg << '\n';
    });

    subject.subscribe([](const std::string& msg)
    {
```

```
        std::cout << "Observer 2: " << msg << '\n';
    });

    subject.notify("event fired");
}
```

This form is frequently more expressive for GUI systems, task pipelines, and application-level event dispatch.

15.2.3 Modern guidance

Modern Observer design should address:

1. Lifetime management
2. Unsubscription policy
3. Reentrancy
4. Thread-safety when needed
5. Clear ownership boundaries

The biggest modernization is that Observer is no longer merely a virtual interface pattern. It is now commonly expressed through safe callback mechanisms and weak ownership.

15.3 Adapter, Decorator, Bridge

These patterns remain highly relevant, but Modern C++ changes how they are expressed.

15.3.1 Adapter Pattern

Adapter converts one interface into another expected by clients. The pattern is especially useful when integrating legacy code or third-party APIs.

```
#include <iostream>
#include <string>

class LegacyPrinter
{
public:
    void old_print_api(const char* text)
    {
        std::cout << "LegacyPrinter: " << text << '\n';
    }
};

class IPrinter
{
public:
    virtual ~IPrinter() = default;
    virtual void print(const std::string& text) = 0;
};

class PrinterAdapter : public IPrinter
{
public:
    explicit PrinterAdapter(LegacyPrinter& p)
        : printer(p)
    {
```

```
    }

    void print(const std::string& text) override
    {
        printer.old_print_api(text.c_str());
    }

private:
    LegacyPrinter& printer;
};

int main()
{
    LegacyPrinter legacy;
    PrinterAdapter adapter(legacy);

    adapter.print("Hello through adapter");
}
```

A modern adaptation may also use composition plus a lambda or a small wrapper object instead of formal inheritance if no stable abstract base is needed.

15.3.2 Decorator Pattern

Decorator adds responsibilities to an object dynamically without modifying the underlying class.

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

```
class Text
{
public:
    virtual ~Text() = default;
    virtual std::string render() const = 0;
};

class PlainText : public Text
{
public:
    explicit PlainText(std::string t)
        : text(std::move(t))
    {
    }

    std::string render() const override
    {
        return text;
    }

private:
    std::string text;
};

class BoldDecorator : public Text
{
public:
    explicit BoldDecorator(std::unique_ptr<Text> inner)
```

```
        : wrapped(std::move(inner))
    {
    }

    std::string render() const override
    {
        return "<b>" + wrapped->render() + "</b>";
    }

private:
    std::unique_ptr<Text> wrapped;
};

class ItalicDecorator : public Text
{
public:
    explicit ItalicDecorator(std::unique_ptr<Text> inner)
        : wrapped(std::move(inner))
    {
    }

    std::string render() const override
    {
        return "<i>" + wrapped->render() + "</i>";
    }

private:
    std::unique_ptr<Text> wrapped;
};
```

```
int main()
{
    std::unique_ptr<Text> text =
        std::make_unique<ItalicDecorator>(
            std::make_unique<BoldDecorator>(
                std::make_unique<PlainText>("Modern C++")));

    std::cout << text->render() << '\n';
}
```

The modern improvement is strong ownership with `std::unique_ptr`, rather than manual memory handling.

15.3.3 Functional decorator

In some modern systems, decoration is better modeled as function composition.

```
#include <functional>
#include <iostream>
#include <string>

int main()
{
    std::function<std::string(std::string)> bold =
        [](std::string s) { return "<b>" + s + "</b>"; };

    std::function<std::string(std::string)> italic =
        [](std::string s) { return "<i>" + s + "</i>"; };
```

```
    std::string text = "Modern C++";

    std::cout << italic(bold(text)) << '\n';
}
```

This is often clearer than class-based decoration when only transformation behavior is needed.

15.3.4 Bridge Pattern

Bridge separates abstraction from implementation so the two can vary independently.

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

class Device
{
public:
    virtual ~Device() = default;
    virtual void turn_on() = 0;
    virtual void turn_off() = 0;
};

class TV : public Device
{
public:
    void turn_on() override
    {
        std::cout << "TV on\n";
    }
```

```
    void turn_off() override
    {
        std::cout << "TV off\n";
    }
};

class Radio : public Device
{
public:
    void turn_on() override
    {
        std::cout << "Radio on\n";
    }

    void turn_off() override
    {
        std::cout << "Radio off\n";
    }
};

class RemoteControl
{
public:
    explicit RemoteControl(std::unique_ptr<Device> d)
        : device(std::move(d))
    {
    }
```

```
    void on()
    {
        device->turn_on();
    }

    void off()
    {
        device->turn_off();
    }

private:
    std::unique_ptr<Device> device;
};

int main()
{
    RemoteControl tv_remote(std::make_unique<TV>());
    tv_remote.on();
    tv_remote.off();

    RemoteControl radio_remote(std::make_unique<Radio>());
    radio_remote.on();
    radio_remote.off();
}
```

The abstraction and implementation axes are cleanly separated. Modern C++ improves this pattern mainly through stronger ownership and cleaner interface boundaries.

15.4 Command Pattern

The classical Command Pattern encapsulates a request as an object. It is useful for delayed execution, undo systems, macro recording, queues, and job dispatching.

15.4.1 Classical command hierarchy

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

class Command
{
public:
    virtual ~Command() = default;
    virtual void execute() = 0;
};

class Light
{
public:
    void on()
    {
        std::cout << "Light on\n";
    }

    void off()
    {
        std::cout << "Light off\n";
```

```
    }
};

class LightOnCommand : public Command
{
public:
    explicit LightOnCommand(Light& l)
        : light(l)
    {
    }

    void execute() override
    {
        light.on();
    }

private:
    Light& light;
};

class LightOffCommand : public Command
{
public:
    explicit LightOffCommand(Light& l)
        : light(l)
    {
    }

    void execute() override
```

```
    {
        light.off();
    }

private:
    Light& light;
};

int main()
{
    Light light;

    std::vector<std::unique_ptr<Command>> queue;
    queue.push_back(std::make_unique<LightOnCommand>(light));
    queue.push_back(std::make_unique<LightOffCommand>(light));

    for (auto& cmd : queue)
        cmd->execute();
}
```

15.4.2 Modern command with callables

In many modern code bases, Command is expressed far more simply using lambdas.

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

class Light
{
```

```
public:
    void on()
    {
        std::cout << "Light on\n";
    }

    void off()
    {
        std::cout << "Light off\n";
    }
};

int main()
{
    Light light;

    std::vector<std::function<void()>> queue;

    queue.push_back([&light] { light.on(); });
    queue.push_back([&light] { light.off(); });

    for (auto& cmd : queue)
        cmd();
}
```

This is one of the clearest examples of how lambdas modernize a classical GOF pattern.

15.4.3 Undo-capable command

A modern command system often pairs execution with undo.

```
#include <functional>
#include <iostream>
#include <stack>
#include <string>

struct Command
{
    std::function<void()> execute;
    std::function<void()> undo;
};

class TextBuffer
{
public:
    void append(const std::string& s)
    {
        data += s;
    }

    void erase_last(std::size_t count)
    {
        data.erase(data.size() - count);
    }

    void print() const
    {
        std::cout << data << '\n';
    }
```

```
private:
    std::string data;
};

int main()
{
    TextBuffer buffer;
    std::stack<Command> history;

    std::string fragment = "Hello";

    Command cmd{
        [&buffer, fragment] { buffer.append(fragment); },
        [&buffer, size = fragment.size()] { buffer.erase_last(size); }
    };

    cmd.execute();
    history.push(cmd);
    buffer.print();

    history.top().undo();
    history.pop();
    buffer.print();
}
```

15.5 Prototype Pattern (modern take)

The classical Prototype Pattern creates new objects by cloning existing ones. In older object-oriented systems, this usually meant a virtual `clone()` function returning a raw pointer.

Modern C++ strongly improves this design through smart pointers and value semantics.

15.5.1 Classical modernized prototype

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

class Shape
{
public:
    virtual ~Shape() = default;
    virtual std::unique_ptr<Shape> clone() const = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle(int r)
        : radius(r)
    {
    }

    std::unique_ptr<Shape> clone() const override
    {
        return std::make_unique<Circle>(*this);
    }

    void draw() const override
```

```
    {
        std::cout << "Circle radius = " << radius << '\n';
    }

private:
    int radius{};
};

int main()
{
    std::unique_ptr<Shape> original = std::make_unique<Circle>(42);
    std::unique_ptr<Shape> copy = original->clone();

    original->draw();
    copy->draw();
}
```

Returning `std::unique_ptr` is the key modernization. It makes ownership explicit and exception-safe.

15.5.2 Prototype with value types

In many modern systems, Prototype is less necessary because value types already support copy and move naturally.

```
#include <iostream>
#include <string>

struct ReportConfig
{
```

```
    std::string title;
    int width{};
    int height{};
};

int main()
{
    ReportConfig base{"Annual Report", 800, 600};

    ReportConfig customized = base;
    customized.title = "Custom Report";

    std::cout << base.title << '\n';
    std::cout << customized.title << '\n';
}
```

This is the modern lesson: when value semantics already express the idea clearly, a formal Prototype pattern may be unnecessary.

15.5.3 Modern guidance

Prototype remains useful when:

1. Objects are polymorphic and copying must preserve dynamic type.
2. Construction is expensive or configuration-heavy.
3. A registry of preconfigured objects is desired.

It is often unnecessary when plain value copying is sufficient.

15.6 Visitor Pattern (dynamic, static, variant-based)

Visitor is one of the most heavily modernized GOF patterns in C++. Historically, it was expressed as a double-dispatch object-oriented pattern. Today, C++ supports multiple visitor styles: classical dynamic visitor, static visitation through templates, and variant-based visitation through `std::visit`.

15.6.1 Classical dynamic visitor

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

class Circle;
class Rectangle;

class Visitor
{
public:
    virtual ~Visitor() = default;
    virtual void visit(const Circle& c) = 0;
    virtual void visit(const Rectangle& r) = 0;
};

class Shape
{
public:
    virtual ~Shape() = default;
    virtual void accept(Visitor& v) const = 0;
```

```
};

class Circle : public Shape
{
public:
    explicit Circle(double r) : radius(r) {}

    double get_radius() const
    {
        return radius;
    }

    void accept(Visitor& v) const override
    {
        v.visit(*this);
    }

private:
    double radius{};
};

class Rectangle : public Shape
{
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double get_width() const
    {
        return width;
```

```
    }

    double get_height() const
    {
        return height;
    }

    void accept(Visitor& v) const override
    {
        v.visit(*this);
    }

private:
    double width{};
    double height{};
};

class AreaPrinter : public Visitor
{
public:
    void visit(const Circle& c) override
    {
        std::cout << "Circle area = "
                  << 3.141592653589793 * c.get_radius() * c.get_radius()
                  << '\n';
    }

    void visit(const Rectangle& r) override
    {
```

```
        std::cout << "Rectangle area = "
                  << r.get_width() * r.get_height()
                  << '\n';
    }
};

int main()
{
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(2.0));
    shapes.push_back(std::make_unique<Rectangle>(3.0, 4.0));

    AreaPrinter printer;

    for (const auto& shape : shapes)
        shape->accept(printer);
}
```

This pattern remains useful when the hierarchy is open and runtime polymorphism is required.

15.6.2 Static visitor with templates

If the set of types is known statically, templates often provide a cleaner and faster alternative.

```
#include <iostream>

struct Circle
{
    double radius{};
};
```

```
struct Rectangle
{
    double width{};
    double height{};
};

struct AreaVisitor
{
    void operator()(const Circle& c) const
    {
        std::cout << "Circle area = "
                  << 3.141592653589793 * c.radius * c.radius
                  << '\n';
    }

    void operator()(const Rectangle& r) const
    {
        std::cout << "Rectangle area = "
                  << r.width * r.height
                  << '\n';
    }
};

int main()
{
    AreaVisitor visitor;
    visitor(Circle{2.0});
    visitor(Rectangle{3.0, 4.0});
```

```
}
```

This is no longer the classical GOF Visitor structure, but it expresses visitation behavior statically.

15.6.3 Variant-based visitor

Modern C++17 and later provide `std::variant` and `std::visit`, which often replace classical visitor hierarchies for closed sets of types.

```
#include <iostream>
#include <variant>

struct Circle
{
    double radius{};
};

struct Rectangle
{
    double width{};
    double height{};
};

using Shape = std::variant<Circle, Rectangle>;

template<class... Ts>
struct Overloaded : Ts...
{
    using Ts::operator()...;
```

```
};

template<class... Ts>
Overloaded(Ts...) -> Overloaded<Ts...>;

int main()
{
    Shape a = Circle{2.0};
    Shape b = Rectangle{3.0, 4.0};

    auto area_visitor = Overloaded{
        [](const Circle& c)
        {
            std::cout << "Circle area = "
                      << 3.141592653589793 * c.radius * c.radius
                      << '\n';
        },
        [](const Rectangle& r)
        {
            std::cout << "Rectangle area = "
                      << r.width * r.height
                      << '\n';
        }
    };

    std::visit(area_visitor, a);
    std::visit(area_visitor, b);
}
```

This style is often clearer and avoids intrusive inheritance when the alternative set is fixed.

15.6.4 Comparing the three forms

Dynamic visitor is appropriate when:

1. The hierarchy must be open to new runtime-derived types.
2. Objects are already organized through virtual interfaces.

Static visitor is appropriate when:

1. The types are known at compile time.
2. Performance and simplicity matter more than runtime openness.

Variant-based visitor is appropriate when:

1. The set of alternatives is closed.
2. New operations are added more often than new types.
3. A type-safe union is more appropriate than inheritance.

15.7 Pattern composition in Modern C++

One of the most important modern lessons is that design patterns are no longer rigid blueprints. Modern C++ allows them to be mixed, simplified, or partially replaced by language and library facilities.
Examples include:

1. Strategy expressed as lambdas instead of classes.
2. Observer expressed as callbacks and weak ownership instead of raw pointer lists.

3. Command expressed as stored callables rather than dedicated command classes.
4. Visitor expressed through `std::variant` and `std::visit` instead of double dispatch.
5. Prototype simplified into ordinary value copying when polymorphism is not required.

This is the central modernization principle: preserve the design intent, but use the strongest modern language tools.

15.8 Design guidance for modernized patterns

15.8.1 Prefer value semantics where possible

Many classical patterns were shaped by older constraints around copying, memory management, and object identity. Modern C++ often favors plain value types, moves, and explicit ownership.

15.8.2 Use smart pointers intentionally

`std::unique_ptr` is usually the correct choice for exclusive ownership.
`std::shared_ptr` should be used only when shared lifetime is genuinely needed.
`std::weak_ptr` is often the correct supporting tool in observer-like relationships.

15.8.3 Prefer lambdas and small function objects for local behavior

When a behavior is small and localized, a lambda is often more readable than a dedicated class hierarchy.

15.8.4 Prefer templates or concepts for static policy variation

If the variation is compile-time and performance-sensitive, templates are often the cleanest modernization.

15.8.5 Use runtime polymorphism only when runtime openness is truly required

Virtual dispatch remains essential in many designs, but it should be chosen for architectural reasons rather than habit.

15.9 Windows build guidance

All examples in this chapter can be compiled on Windows using Visual Studio 2022 or a recent MinGW-w64 toolchain.

15.9.1 Visual Studio 2022 project settings

Use these recommended settings:

```
Project Properties
  C/C++ -> Language -> C++ Language Standard -> ISO C++20 Standard (/std:c++20)
  C/C++ -> Code Generation -> Enable C++ Exceptions -> Yes (/EHsc)
  C/C++ -> Optimization -> Optimization -> Maximize Speed (/O2) for Release
  C/C++ -> Warning Level -> Level4 (/W4)
```

For command-line compilation with MSVC:

```
cl /std:c++20 /EHsc /W4 chapter15_patterns.cpp
```

For MinGW-w64 g++ on Windows:

```
g++ -std=c++20 -Wall -Wextra -pedantic chapter15_patterns.cpp -o
↪  chapter15_patterns.exe
```

15.10 Conclusion

The GOF patterns remain valuable, but Modern C++ changes how they should be written. The modernized view is not to discard the patterns, but to reinterpret them through:

1. RAII and explicit ownership
2. smart pointers and weak references
3. lambdas and function objects
4. templates and concepts
5. `std::variant` and visitation
6. simpler value-oriented designs where full pattern ceremony is unnecessary

A strong Modern C++ programmer should understand both the original design intent of the classical patterns and the modern language facilities that make those patterns safer, lighter, and more expressive.

C++-Specific Design Patterns

16.1 Pimpl

The Pointer-to-Implementation (Pimpl) idiom is a core C++-specific design pattern used to achieve compilation firewalling, ABI stability, and reduced header dependencies. It separates interface from implementation by moving private data and implementation details into a separately defined structure.

16.1.1 Motivation

The main goals of Pimpl are:

1. Reduce compile-time dependencies
2. Improve build times
3. Preserve ABI stability across binary boundaries
4. Hide implementation details

In large-scale systems, especially those with many translation units, Pimpl is essential for maintaining manageable build times and stable interfaces.

16.1.2 Basic Pimpl structure

```
// widget.h
#pragma once
#include <memory>

class Widget
{
public:
    Widget();
    ~Widget();

    void do_work();

private:
    struct Impl;
    std::unique_ptr<Impl> impl;
};
```

```
// widget.cpp
#include "widget.h"
#include <iostream>

struct Widget::Impl
{
    int data{42};

    void do_work_impl()
    {
        std::cout << "Working with data = " << data << '\n';
```

```
    }
};

Widget::Widget()
    : impl(std::make_unique<Impl>())
{
}

Widget::~Widget() = default;

void Widget::do_work()
{
    impl->do_work_impl();
}
```

16.1.3 Important rule: destructor must be defined out-of-line

Because `std::unique_ptr<Impl>` requires the complete type at destruction time, the destructor must be defined in the source file where `Impl` is complete.

16.1.4 Copy and move semantics

By default, Pimpl disables copying due to `std::unique_ptr`. To support copying:

```
Widget::Widget(const Widget& other)
    : impl(std::make_unique<Impl>(*other.impl))
{
}

Widget& Widget::operator=(const Widget& other)
```

```
{
    if (this != &other)
        impl = std::make_unique<Impl>(*other.impl);
    return *this;
}
```

Move operations are typically defaulted:

```
Widget(Widget&&) noexcept = default;
Widget& operator=(Widget&&) noexcept = default;
```

16.1.5 Pimpl trade-offs

Advantages:

1. Reduced recompilation
2. Encapsulation of implementation details
3. Stable binary interface

Costs:

1. Heap allocation
2. Indirection
3. Slight runtime overhead

16.2 RAII patterns

Resource Acquisition Is Initialization (RAII) is the foundational pattern of Modern C++. It binds resource lifetime to object lifetime, ensuring deterministic release.

16.2.1 Basic RAII example

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

void write_file(const std::string& path)
{
    std::ofstream file(path);
    file << "RAII example\n";
} // file closed automatically
```

16.2.2 Custom RAII wrapper

```
#include <iostream>

class FileHandle
{
public:
    explicit FileHandle(const char* name)
    {
        std::cout << "Opening file: " << name << '\n';
    }

    ~FileHandle()
    {
        std::cout << "Closing file\n";
    }
};

int main()
```

```
{
    FileHandle f("data.txt");
}
```

16.2.3 Scoped lock RAII

```
#include <mutex>

std::mutex m;

void safe_increment()
{
    std::lock_guard<std::mutex> lock(m);
    // critical section
}
```

16.2.4 RAII for exception safety

RAII guarantees cleanup even in the presence of exceptions.

```
#include <memory>

void process()
{
    auto ptr = std::make_unique<int>(42);
    throw std::runtime_error("error");
} // memory released automatically
```

16.2.5 Scope guard pattern

```
#include <functional>
```

```
class ScopeGuard
{
public:
    explicit ScopeGuard(std::function<void()> f)
        : func(std::move(f)), active(true)
    {
    }

    ~ScopeGuard()
    {
        if (active)
            func();
    }

    void dismiss()
    {
        active = false;
    }

private:
    std::function<void()> func;
    bool active;
};
```

16.3 Resource acquisition filters

Resource acquisition filters extend RAII by adding validation or transformation logic during acquisition. They ensure that only valid resources are constructed and propagated.

16.3.1 Validated resource wrapper

```
#include <stdexcept>
#include <string>

class ValidFile
{
public:
    explicit ValidFile(const std::string& path)
    {
        if (path.empty())
            throw std::invalid_argument("Invalid path");
        // simulate open
    }

    ~ValidFile()
    {
        // close resource
    }
};
```

16.3.2 Factory with validation

```
#include <memory>

class Resource
{
public:
    static std::unique_ptr<Resource> create(int id)
    {
```

```
        if (id < 0)
            return nullptr;
        return std::unique_ptr<Resource>(new Resource(id));
    }

private:
    explicit Resource(int id_) : id(id_) {}
    int id;
};
```

16.3.3 RAII + filtering pipeline

```
#include <iostream>
#include <optional>

std::optional<int> acquire(int value)
{
    if (value > 0)
        return value;
    return std::nullopt;
}

int main()
{
    auto res = acquire(10);
    if (res)
        std::cout << *res << '\n';
}
```

16.3.4 Design intent

Resource acquisition filters enforce invariants at construction time, ensuring:

1. No invalid resource exists
2. Errors are handled early
3. Objects are always in valid states

16.4 Policy-based design

Policy-based design uses template parameters to inject behavior into a class. It allows flexible composition of behavior at compile time.

16.4.1 Basic policy pattern

```
#include <iostream>

struct AddPolicy
{
    static int apply(int a, int b)
    {
        return a + b;
    }
};

struct MultiplyPolicy
{
    static int apply(int a, int b)
```

```
    {
        return a * b;
    }
};

template<typename Policy>
class Calculator
{
public:
    int compute(int a, int b)
    {
        return Policy::apply(a, b);
    }
};

int main()
{
    Calculator<AddPolicy> add_calc;
    Calculator<MultiplyPolicy> mul_calc;

    std::cout << add_calc.compute(2, 3) << '\n';
    std::cout << mul_calc.compute(2, 3) << '\n';
}
```

16.4.2 Policy composition

```
#include <iostream>

struct LoggingPolicy
```

```
{
    static void log(int value)
    {
        std::cout << "Result: " << value << '\n';
    }
};

template<typename ComputePolicy, typename LogPolicy>
class Engine
{
public:
    int run(int a, int b)
    {
        int result = ComputePolicy::apply(a, b);
        LogPolicy::log(result);
        return result;
    }
};
```

16.4.3 Benefits

1. Zero runtime overhead
2. High flexibility
3. Strong compile-time guarantees

16.4.4 Trade-offs

1. Increased compile-time complexity

2. Potential code bloat

16.5 Compile-time polymorphism patterns

Compile-time polymorphism is achieved using templates, concepts, and CRTP. It avoids runtime dispatch entirely.

16.5.1 CRTP pattern

```
template<typename Derived>
class Base
{
public:
    void interface()
    {
        static_cast<Derived*>(this)->implementation();
    }
};

class Concrete : public Base<Concrete>
{
public:
    void implementation()
    {
        // specific behavior
    }
};
```

16.5.2 Concept-based polymorphism

```
#include <concepts>
#include <iostream>

template<typename T>
concept Drawable =
    requires(T t)
    {
        { t.draw() } -> std::same_as<void>;
    };

template<Drawable T>
void render(const T& obj)
{
    obj.draw();
}

struct Circle
{
    void draw() const
    {
        std::cout << "Circle\n";
    }
};

int main()
{
    Circle c;
```

```
    render(c);
}
```

16.5.3 Tag dispatch

```
#include <iostream>

struct FastTag {};
struct SafeTag {};

void process_impl(int x, FastTag)
{
    std::cout << "Fast: " << x << '\n';
}

void process_impl(int x, SafeTag)
{
    std::cout << "Safe: " << x << '\n';
}

template<typename Tag>
void process(int x, Tag tag)
{
    process_impl(x, tag);
}
```

16.5.4 If constexpr dispatch

```
#include <iostream>
#include <type_traits>
```

```
template<typename T>
void print(T value)
{
    if constexpr (std::is_integral_v<T>)
        std::cout << "Integral: " << value << '\n';
    else
        std::cout << "Other: " << value << '\n';
}
```

16.5.5 Modern insights

Compile-time polymorphism:

1. Eliminates virtual dispatch
2. Enables inlining and optimization
3. Requires types known at compile time

16.6 Conclusion

C++-specific design patterns leverage unique language features such as RAII, templates, and strong type systems. These patterns differ from classical GOF patterns in that they are often zero-cost abstractions and tightly integrated with the language semantics.
Key insights:

1. Pimpl improves compilation and ABI stability.
2. RAII ensures deterministic resource management.

3. Resource acquisition filters enforce correctness at construction.

4. Policy-based design enables flexible compile-time composition.

5. Compile-time polymorphism provides high-performance abstractions.

Modern C++ encourages combining these patterns to build systems that are efficient, safe, and maintainable while minimizing runtime overhead.

Architecture & Framework-Level Patterns

17.1 Dependency Injection

Dependency Injection is an architectural pattern in which an object receives its collaborators from the outside instead of constructing them internally. In Modern C++, this pattern is strongly shaped by explicit ownership, RAII, smart pointers, interfaces, templates, and value semantics.
The fundamental goal of Dependency Injection is to reduce coupling between a class and the concrete services it depends on. This improves testability, configurability, substitution, and architectural clarity.

17.1.1 Why Dependency Injection matters in C++

In unmanaged or loosely disciplined code, classes often create their own dependencies directly:

```
class Logger
{
public:
    void write(const std::string& msg)
    {
        std::cout << msg << '\n';
    }
```

```
};

class Service
{
public:
    void run()
    {
        Logger logger;
        logger.write("running");
    }
};
```

This style is simple, but it tightly couples `Service` to a specific `Logger` implementation and makes testing harder. Modern architecture favors injecting the dependency from the outside.

17.1.2 Constructor injection

Constructor injection is the most common and often the strongest form of Dependency Injection in C++. It makes dependencies explicit and guarantees that the object is fully initialized with valid collaborators.

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

class ILogger
{
public:
    virtual ~ILogger() = default;
    virtual void write(const std::string& msg) = 0;
```

```
};

class ConsoleLogger : public ILogger
{
public:
    void write(const std::string& msg) override
    {
        std::cout << "[console] " << msg << '\n';
    }
};

class Service
{
public:
    explicit Service(ILogger& logger_)
        : logger(logger_)
    {
    }

    void run()
    {
        logger.write("service started");
    }

private:
    ILogger& logger;
};

int main()
```

```
{
    ConsoleLogger logger;
    Service service(logger);
    service.run();
}
```

This design communicates the dependency clearly and avoids hidden construction logic.

17.1.3 Injection by reference versus pointer

In Modern C++, choosing the injected type expresses important semantics:

1. `T&` means the dependency must exist and must not be null.
2. `T*` means the dependency may be absent or optional.
3. `std::unique_ptr<T>` means ownership is transferred.
4. `std::shared_ptr<T>` means shared lifetime is intended.

Example with ownership transfer:

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

class ILogger
{
public:
    virtual ~ILogger() = default;
    virtual void write(const std::string& msg) = 0;
};
```

```
class FileLogger : public ILogger
{
public:
    void write(const std::string& msg) override
    {
        std::cout << "[file] " << msg << '\n';
    }
};

class App
{
public:
    explicit App(std::unique_ptr<ILogger> logger_)
        : logger(std::move(logger_))
    {
    }

    void execute()
    {
        logger->write("app executing");
    }

private:
    std::unique_ptr<ILogger> logger;
};

int main()
{
```

```
    auto logger = std::make_unique<FileLogger>();
    App app(std::move(logger));
    app.execute();
}
```

17.1.4 Setter injection

Setter injection is useful when the dependency is optional or reconfigurable after construction, but it is usually weaker than constructor injection because it allows temporarily incomplete states.

```
#include <iostream>
#include <string>

class ILogger
{
public:
    virtual ~ILogger() = default;
    virtual void write(const std::string& msg) = 0;
};

class ConsoleLogger : public ILogger
{
public:
    void write(const std::string& msg) override
    {
        std::cout << msg << '\n';
    }
};
```

```
class Worker
{
public:
    void set_logger(ILogger* logger_)
    {
        logger = logger_;
    }

    void process()
    {
        if (logger)
            logger->write("processing");
    }

private:
    ILogger* logger{};
};
```

17.1.5 Template-based injection

C++ also supports compile-time injection through templates. This avoids runtime polymorphism and is often ideal for performance-sensitive systems or header-only frameworks.

```
#include <iostream>
#include <string>

struct ConsoleLogger
{
    void write(const std::string& msg) const
    {
```

```
        std::cout << "[console] " << msg << '\n';
    }
};

template<typename Logger>
class Service
{
public:
    explicit Service(Logger logger_)
        : logger(std::move(logger_))
    {
    }

    void run()
    {
        logger.write("template-injected service");
    }

private:
    Logger logger;
};

int main()
{
    Service<ConsoleLogger> service(ConsoleLogger{});
    service.run();
}
```

This is often called static dependency injection because the collaborator type is fixed at compile time.

17.1.6 Interface injection versus value injection

Not all dependencies need to be interfaces. In Modern C++, injecting value objects is often simpler and more expressive than forcing a virtual interface.

```
#include <iostream>
#include <string>

struct Config
{
    std::string host;
    int port{};
};

class Client
{
public:
    explicit Client(Config cfg)
        : config(std::move(cfg))
    {
    }

    void connect() const
    {
        std::cout << "Connecting to " << config.host
                  << ':' << config.port << '\n';
    }

private:
    Config config;
```

```
};
```

This is still Dependency Injection. The dependency is data rather than a behavior interface.

17.1.7 Modern guidance for Dependency Injection

In Modern C++, good Dependency Injection usually follows these principles:

1. Prefer constructor injection for required dependencies.
2. Express ownership clearly with references, pointers, or smart pointers.
3. Prefer values when polymorphism is unnecessary.
4. Prefer templates when the dependency is fixed at compile time and performance matters.
5. Avoid overengineering with service locators when direct injection is sufficient.

17.2 Composition over inheritance

Composition over inheritance is one of the most important architectural principles in Modern C++. It recommends building larger behavior from smaller collaborating parts rather than relying excessively on deep class hierarchies.
Inheritance remains useful, especially for runtime polymorphism, but composition is often safer, simpler, and more flexible.

17.2.1 Why composition is preferred

Inheritance creates strong coupling between base and derived types. Changes in the base may affect all derived classes. In contrast, composition builds behavior by owning or referencing independent parts.
Instead of this:

```
class EngineVehicle
{
public:
    void start_engine() {}
};

class Car : public EngineVehicle
{
};
```

Modern design often prefers this:

```
#include <iostream>

class Engine
{
public:
    void start() const
    {
        std::cout << "engine started\n";
    }
};

class Car
{
public:
    explicit Car(Engine engine_)
        : engine(std::move(engine_))
    {
    }
```

```
    void start()
    {
        engine.start();
    }

private:
    Engine engine;
};

int main()
{
    Car car(Engine{});
    car.start();
}
```

Here, `Car` is built from an `Engine`; it is not an `Engine`.

17.2.2 Composition and testability

Composition improves testing because parts can be replaced independently.

```
#include <iostream>
#include <string>

class PaymentGateway
{
public:
    virtual ~PaymentGateway() = default;
    virtual bool charge(double amount) = 0;
```

```
};

class RealGateway : public PaymentGateway
{
public:
    bool charge(double amount) override
    {
        std::cout << "Charging " << amount << '\n';
        return true;
    }
};

class FakeGateway : public PaymentGateway
{
public:
    bool charge(double) override
    {
        return true;
    }
};

class CheckoutService
{
public:
    explicit CheckoutService(PaymentGateway& gateway_)
        : gateway(gateway_)
    {
    }
```

```
    bool checkout(double amount)
    {
        return gateway.charge(amount);
    }

private:
    PaymentGateway& gateway;
};
```

This is a composition-based design with a replaceable collaborator.

17.2.3 Mixing composition and polymorphism

Composition does not mean rejecting all inheritance. A strong Modern C++ architecture often uses inheritance only at well-defined boundaries and uses composition almost everywhere else. For example:

1. Small interface hierarchy for plugins or device drivers.
2. Composition of those interfaces inside application services.
3. Value objects for configuration and state.

17.2.4 Composition with policies

Compile-time composition is also common. Instead of runtime inheritance, a class may be assembled from policies:

```
#include <iostream>

struct NoLock
```

```
{
    void lock() {}
    void unlock() {}
};

struct StdoutSink
{
    void write(const char* msg) const
    {
        std::cout << msg << '\n';
    }
};

template<typename LockPolicy, typename SinkPolicy>
class Logger : private LockPolicy, private SinkPolicy
{
public:
    void log(const char* msg)
    {
        this->lock();
        this->write(msg);
        this->unlock();
    }
};

int main()
{
    Logger<NoLock, StdoutSink> logger;
    logger.log("policy-composed logger");
```

```
}
```

This is composition expressed through templates rather than object members.

17.2.5 Modern guidance for composition

Composition is often the better choice when:

1. The relationship is not a true is-a relationship.
2. Behavior should be assembled from parts.
3. Reuse is needed without exposing base-class fragility.
4. State and ownership boundaries should remain explicit.

Inheritance remains appropriate when:

1. Runtime substitutability is the primary requirement.
2. A stable abstract interface is needed.
3. The semantic relation truly is polymorphic and interface-driven.

17.3 Plugin architectures (dynamic & static)

Plugin architectures allow an application to be extended with independently developed modules. In C++, plugin systems are especially important in IDEs, media systems, game engines, scientific tools, GUI frameworks, and enterprise software.
There are two major architectural forms:

1. Dynamic plugins, loaded at runtime from shared libraries.
2. Static plugins, linked into the final executable and registered at build time.

17.3.1 Core goals of a plugin architecture

A robust plugin architecture should define:

1. A stable contract between host and plugin.
2. Versioning rules.
3. Discovery and loading rules.
4. Lifetime and ownership rules.
5. Error handling and unload policy.

17.3.2 Dynamic plugin architecture

Dynamic plugins are typically built as DLLs on Windows and loaded with platform APIs such as `LoadLibrary` and `GetProcAddress`. The host application does not need to link directly against the plugin implementation at build time.
A common pattern is to expose a narrow `extern "C"` factory function from the plugin and return an abstract interface pointer.
Host-side interface:

```
// plugin_api.h
#pragma once
#include <string>

class IPlugin
{
public:
    virtual ~IPlugin() = default;
    virtual std::string name() const = 0;
```

```
    virtual void run() = 0;
};

using CreatePluginFn = IPlugin* (*)();
using DestroyPluginFn = void (*)(IPlugin*);
```

Plugin implementation:

```
// sample_plugin.cpp
#include "plugin_api.h"
#include <iostream>

class SamplePlugin : public IPlugin
{
public:
    std::string name() const override
    {
        return "SamplePlugin";
    }

    void run() override
    {
        std::cout << "SamplePlugin running\n";
    }
};

extern "C" __declspec(dllexport) IPlugin* create_plugin()
{
    return new SamplePlugin();
}
```

```
extern "C" __declspec(dllexport) void destroy_plugin(IPlugin* p)
{
    delete p;
}
```

Host-side loading on Windows:

```
#include "plugin_api.h"
#include <windows.h>
#include <iostream>
#include <memory>

int main()
{
    HMODULE mod = LoadLibraryA("sample_plugin.dll");
    if (!mod)
    {
        std::cerr << "Failed to load plugin\n";
        return 1;
    }

    auto create =
        reinterpret_cast<CreatePluginFn>(
            GetProcAddress(mod, "create_plugin"));

    auto destroy =
        reinterpret_cast<DestroyPluginFn>(
            GetProcAddress(mod, "destroy_plugin"));
```

```
    if (!create || !destroy)
    {
        std::cerr << "Failed to find exports\n";
        FreeLibrary(mod);
        return 1;
    }

    std::unique_ptr<IPlugin, DestroyPluginFn> plugin(create(), destroy);

    std::cout << plugin->name() << '\n';
    plugin->run();

    plugin.reset();
    FreeLibrary(mod);
}
```

17.3.3 Dynamic plugin design cautions

Dynamic plugin systems must be designed carefully:

1. Avoid exposing unstable C++ ABI details across compiler or CRT boundaries.
2. Prefer narrow exported C factories.
3. Define explicit ownership and destruction rules.
4. Version the plugin API.
5. Handle plugin load failure gracefully.

17.3.4 Static plugin architecture

Static plugins are linked into the application at build time. They are not discovered through DLL loading. Instead, they are registered through compile-time mechanisms such as registries, factories, or explicit initialization calls.

Example using a static registry:

```
#include <functional>
#include <iostream>
#include <memory>
#include <string>
#include <unordered_map>

class IPlugin
{
public:
    virtual ~IPlugin() = default;
    virtual std::string name() const = 0;
    virtual void run() = 0;
};

class PluginRegistry
{
public:
    using Factory = std::function<std::unique_ptr<IPlugin>()>;

    static PluginRegistry& instance()
    {
        static PluginRegistry registry;
        return registry;
```

```
    }

    void add(std::string id, Factory factory)
    {
        factories.emplace(std::move(id), std::move(factory));
    }

    std::unique_ptr<IPlugin> create(const std::string& id) const
    {
        auto it = factories.find(id);
        if (it == factories.end())
            return nullptr;
        return it->second();
    }

private:
    std::unordered_map<std::string, Factory> factories;
};

class EchoPlugin : public IPlugin
{
public:
    std::string name() const override
    {
        return "EchoPlugin";
    }

    void run() override
    {
```

```
        std::cout << "EchoPlugin running\n";
    }
};

struct EchoPluginRegistration
{
    EchoPluginRegistration()
    {
        PluginRegistry::instance().add("echo", []()
        {
            return std::make_unique<EchoPlugin>();
        });
    }
};

static EchoPluginRegistration echo_registration;

int main()
{
    auto plugin = PluginRegistry::instance().create("echo");
    if (plugin)
        plugin->run();
}
```

17.3.5 Dynamic versus static plugins

Dynamic plugins are best when:

1. New modules must be added without rebuilding the host.
2. Third parties deliver extensions independently.

3. Runtime discovery is required.

Static plugins are best when:

1. Deployment simplicity matters.
2. All modules are known at build time.
3. Maximum ABI control is desired.
4. The system should avoid runtime loading complexity.

17.3.6 A practical hybrid design

Many modern frameworks support both models:

1. Static plugins for core built-in modules.
2. Dynamic plugins for optional or third-party extensions.

This hybrid architecture is often ideal for large products.

17.3.7 Windows build guidance for dynamic plugins

Plugin interface header:

```
// plugin_api.h is shared between host and plugin
```

Build DLL with MSVC:

```
cl /std:c++20 /EHsc /LD sample_plugin.cpp /Fe:sample_plugin.dll
```

Build host:

```
cl /std:c++20 /EHsc host.cpp
```

If using CMake for a loadable module:

```
add_library(sample_plugin MODULE sample_plugin.cpp)
target_include_directories(sample_plugin PRIVATE ${CMAKE_CURRENT_SOURCE_DIR})
```

17.4 Event-driven architecture

Event-driven architecture organizes program flow around the production, distribution, and handling of events. Instead of direct linear calls between all components, producers emit events and subscribers react to them.
This style is central to GUI frameworks, networking systems, asynchronous systems, engine architectures, and decoupled business workflows.

17.4.1 Basic event-driven model

An event-driven system usually contains:

1. Event producers
2. Event objects or event payloads
3. Dispatchers or buses
4. Subscribers or handlers
5. Often an event loop

17.4.2 Simple event bus example

```
#include <functional>
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

class EventBus
{
public:
    using Handler = std::function<void(const std::string&)>;

    void subscribe(const std::string& topic, Handler handler)
    {
        handlers[topic].push_back(std::move(handler));
    }

    void publish(const std::string& topic, const std::string& payload) const
    {
        auto it = handlers.find(topic);
        if (it == handlers.end())
            return;

        for (const auto& handler : it->second)
            handler(payload);
    }

private:
```

```
    std::unordered_map<std::string, std::vector<Handler>> handlers;
};

int main()
{
    EventBus bus;

    bus.subscribe("order.created", [](const std::string& msg)
    {
        std::cout << "Handler A: " << msg << '\n';
    });

    bus.subscribe("order.created", [](const std::string& msg)
    {
        std::cout << "Handler B: " << msg << '\n';
    });

    bus.publish("order.created", "order #42");
}
```

17.4.3 Typed event dispatch

A more C++-specific design often uses event types rather than string topics.

```
#include <functional>
#include <iostream>
#include <typeindex>
#include <unordered_map>
#include <vector>
#include <memory>
```

```
class TypedEventBus
{
public:
    template<typename Event, typename Handler>
    void subscribe(Handler&& handler)
    {
        auto& vec = table[std::type_index(typeid(Event))];
        vec.push_back(
            [fn = std::forward<Handler>(handler)](const void* e)
            {
                fn(*static_cast<const Event*>(e));
            });
    }

    template<typename Event>
    void publish(const Event& event) const
    {
        auto it = table.find(std::type_index(typeid(Event)));
        if (it == table.end())
            return;

        for (const auto& handler : it->second)
            handler(&event);
    }

private:
    using ErasedHandler = std::function<void(const void*)>;
    std::unordered_map<std::type_index, std::vector<ErasedHandler>> table;
```

```
};

struct UserCreated
{
    int id{};
};

int main()
{
    TypedEventBus bus;

    bus.subscribe<UserCreated>([](const UserCreated& e)
    {
        std::cout << "User created: " << e.id << '\n';
    });

    bus.publish(UserCreated{7});
}
```

17.4.4 Signal-slot style

A signal-slot system is a specialized form of event-driven architecture. It is popular because it decouples publishers from subscribers while allowing managed connections.
A minimal custom signal:

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

template<typename... Args>
```

```
class Signal
{
public:
    using Slot = std::function<void(Args...)>;

    void connect(Slot slot)
    {
        slots.push_back(std::move(slot));
    }

    void emit(Args... args) const
    {
        for (const auto& slot : slots)
            slot(args...);
    }

private:
    std::vector<Slot> slots;
};

int main()
{
    Signal<int> on_value;

    on_value.connect([](int value)
    {
        std::cout << "slot A: " << value << '\n';
    });
```

```
    on_value.connect([](int value)
    {
        std::cout << "slot B: " << value * 2 << '\n';
    });

    on_value.emit(21);
}
```

17.4.5 Event loops

Many event-driven systems revolve around an event loop. GUI frameworks and asynchronous runtimes typically receive external events, queue them, and dispatch them to handlers.
A tiny illustrative loop:

```
#include <functional>
#include <iostream>
#include <queue>

class EventLoop
{
public:
    using Task = std::function<void()>;

    void post(Task task)
    {
        queue.push(std::move(task));
    }

    void run()
    {
```

```
        while (!queue.empty())
        {
            auto task = std::move(queue.front());
            queue.pop();
            task();
        }
    }

private:
    std::queue<Task> queue;
};

int main()
{
    EventLoop loop;

    loop.post([] { std::cout << "task 1\n"; });
    loop.post([] { std::cout << "task 2\n"; });
    loop.post([] { std::cout << "task 3\n"; });

    loop.run();
}
```

17.4.6 Architectural benefits of event-driven systems

Event-driven architecture provides:

1. Decoupling between producers and consumers
2. Asynchronous composition opportunities

3. Clear extension points
4. Good fit for GUI, network, and reactive systems

17.4.7 Architectural risks

It also introduces risks:

1. Harder control-flow tracing
2. Reentrancy pitfalls
3. Thread-safety challenges
4. Lifetime issues for subscribers
5. Event storms or uncontrolled fan-out

A disciplined design should clearly define ownership, connection lifetime, delivery order, and synchronization rules.

17.5 Modern ECS (Entity Component System) overview

Entity Component System is a data-oriented architectural pattern widely used in game engines, simulations, editors, and high-performance real-time systems. Modern ECS emphasizes separation of identity, data, and behavior.

17.5.1 Core ideas of ECS

An ECS typically consists of:

1. Entities: lightweight identifiers

2. Components: plain data attached to entities
3. Systems: logic operating over sets of components

Instead of deep inheritance hierarchies such as:

```
class GameObject {};
class Character : public GameObject {};
class Enemy : public Character {};
class BossEnemy : public Enemy {};
```

ECS favors data composition:

```
struct Position { float x{}, y{}; };
struct Velocity { float dx{}, dy{}; };
struct Health   { int hp{}; };
```

An entity may have any combination of these components.

17.5.2 Why ECS exists

Modern ECS addresses common limitations of inheritance-heavy object models:

1. Poor cache locality
2. Deep fragile hierarchies
3. Hard-to-recombine behavior
4. Excessive virtual dispatch

ECS enables flexible composition and often better performance through data-oriented layout and batched processing.

17.5.3 A minimal ECS sketch

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

using Entity = std::uint32_t;

struct Position
{
    float x{}, y{};
};

struct Velocity
{
    float dx{}, dy{};
};

class World
{
public:
    Entity create()
    {
        return next_entity++;
    }

    void add_position(Entity e, Position p)
    {
        positions[e] = p;
```

```
    }

    void add_velocity(Entity e, Velocity v)
    {
        velocities[e] = v;
    }

    void update()
    {
        for (const auto& [entity, vel] : velocities)
        {
            auto it = positions.find(entity);
            if (it != positions.end())
            {
                it->second.x += vel.dx;
                it->second.y += vel.dy;
            }
        }
    }

    void print(Entity e) const
    {
        auto it = positions.find(e);
        if (it != positions.end())
            std::cout << '(' << it->second.x << ", "
                      << it->second.y << ")\n";
    }

private:
```

```
    Entity next_entity{1};
    std::unordered_map<Entity, Position> positions;
    std::unordered_map<Entity, Velocity> velocities;
};

int main()
{
    World world;

    Entity e = world.create();
    world.add_position(e, Position{0.0f, 0.0f});
    world.add_velocity(e, Velocity{1.0f, 0.5f});

    world.update();
    world.print(e);
}
```

This example is intentionally simple. Real ECS frameworks optimize storage, iteration, and views much more aggressively.

17.5.4 Modern ECS storage models

Modern ECS implementations usually use specialized storage strategies rather than naive maps:

1. Dense arrays for component pools
2. Sparse sets for fast membership checks
3. Views and groups for efficient iteration over matching entities
4. Optional sorting for locality-sensitive systems

This is why modern ECS is often associated with data-oriented design rather than only with an architectural diagram.

17.5.5 Entities are identifiers, not objects

A major conceptual shift in ECS is that an entity is usually just an ID. The data and behavior live elsewhere.

```
using Entity = std::uint32_t;
```

This is very different from classical object-oriented thinking, where an object bundles identity, state, and methods in one unit.

17.5.6 Components should usually be simple data

Components are often kept as lightweight structures with minimal or no behavior:

```
struct Transform
{
    float x{}, y{}, z{};
};

struct Renderable
{
    int mesh_id{};
};

struct Health
{
    int current{};
    int maximum{};
```

```
};
```

This allows systems to process them efficiently and uniformly.

17.5.7 Systems operate over views

A system typically iterates over entities having a required component set.
Conceptually:

```
for each entity with Position and Velocity:
    Position.x += Velocity.dx
    Position.y += Velocity.dy
```

In a modern ECS framework, this is often expressed through a view-like abstraction.

17.5.8 ECS versus classical OOP

ECS is especially strong when:

1. Large numbers of similar objects are updated each frame.
2. Data locality and batch processing matter.
3. Behavior must be recombined freely from small data pieces.

Classical OOP may remain stronger when:

1. The domain is not update-loop driven.
2. The entity count is small.
3. Rich object identity and encapsulation dominate the design.

17.5.9 ECS and event-driven systems

Many real systems combine ECS with events:

1. ECS stores world state.
2. Events describe changes, commands, or reactions.
3. Systems read components and consume or emit events.

This hybrid design is common in engines and simulations.

17.5.10 A practical modern ECS mindset

A mature C++ view of ECS is not merely entities plus components. It includes:

1. Cache-conscious storage
2. Explicit update pipelines
3. Decoupling data from behavior
4. Composition instead of hierarchy
5. Often integration with schedulers and event buses

17.6 Architectural comparison and selection

17.6.1 Dependency Injection versus direct construction

Use Dependency Injection when you need:

1. Testability

2. Configurable collaborators
3. Clear architectural boundaries

Use direct construction when:

1. The dependency is trivial and fixed.
2. The class is local and not part of a reusable boundary.

17.6.2 Composition versus inheritance

Use composition when behavior should be assembled from parts.
Use inheritance when stable runtime substitutability is the central requirement.

17.6.3 Dynamic versus static plugins

Use dynamic plugins for third-party extensibility and runtime discovery.
Use static plugins when deployment simplicity and binary control matter more than runtime loading flexibility.

17.6.4 Event-driven architecture versus direct calls

Use events when producers and consumers should remain decoupled or asynchronous.
Use direct calls when control flow should remain simple and synchronous.

17.6.5 ECS versus object hierarchies

Use ECS when the problem is data-oriented, large-scale, and update-loop heavy.
Use traditional object models when object identity, encapsulation, and low object counts dominate.

17.7 Windows build guidance

All code in this chapter can be developed on Windows using Visual Studio 2022.
Recommended settings:

```
Project Properties
  C/C++ -> Language -> C++ Language Standard -> ISO C++20 Standard (/std:c++20)
  C/C++ -> Code Generation -> Enable C++ Exceptions -> Yes (/EHsc)
  C/C++ -> Warning Level -> Level4 (/W4)
  C/C++ -> Optimization -> Maximize Speed (/O2) for Release
```

Example host build:

```
cl /std:c++20 /EHsc /W4 host.cpp
```

Example DLL build:

```
cl /std:c++20 /EHsc /W4 /LD sample_plugin.cpp /Fe:sample_plugin.dll
```

For multi-file CMake builds:

```
cmake_minimum_required(VERSION 3.21)
project(ArchitecturePatterns LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(host host.cpp)
add_library(sample_plugin MODULE sample_plugin.cpp)
```

17.8 Conclusion

Framework-level architecture in Modern C++ is no longer dominated by inheritance-heavy models alone. Modern systems increasingly rely on:

1. Dependency Injection for explicit collaboration boundaries
2. Composition for flexible behavior assembly
3. Plugin architectures for extensibility
4. Event-driven models for decoupled communication
5. ECS for high-performance composition of large dynamic worlds

These patterns are not mutually exclusive. In strong real-world architectures, they are often combined:

1. Dependencies are injected into services.
2. Services are built by composition.
3. Features are extended through plugins.
4. Components communicate through events.
5. State-heavy real-time domains are modeled with ECS.

A professional Modern C++ architect should therefore not treat these patterns as isolated formulas, but as interoperating tools for building systems that are modular, testable, extensible, and performant.

Part VII

Full Project: Config System & Object-Oriented Architecture

Designing a Production-Grade Config Library

18.1 Architecture

A production-grade configuration library in Modern C++ must balance flexibility, performance, safety, and clarity. The architecture should reflect strong separation of concerns, clear ownership semantics, and extensibility without sacrificing compile-time guarantees.

18.1.1 Core architectural layers

A robust configuration system typically consists of the following layers:

1. **Input Layer**: Responsible for reading raw data from files, memory buffers, or streams.
2. **Parsing Layer**: Converts raw text into structured intermediate representations.
3. **Value Layer**: Represents configuration values in a type-safe manner.
4. **Access Layer**: Provides APIs for querying configuration values.
5. **Validation Layer**: Ensures correctness and constraints.

18.1.2 High-level architecture

```
ConfigSystem
   InputReader
   Parser (JSON, INI, YAML...)
   ConfigValue (variant-based)
   ConfigNode (tree structure)
   Access API
```

18.1.3 Design principles

1. Prefer value semantics for configuration data.
2. Use `std::variant` for heterogeneous value representation.
3. Separate parsing from storage.
4. Ensure strong exception safety via RAII.
5. Avoid global state.

18.2 Value hierarchy using variant

The value system is the foundation of the configuration library. It must support multiple types while preserving type safety and performance.

18.2.1 Basic value type

```
#include <variant>
#include <string>
#include <vector>
```

```
#include <unordered_map>

struct ConfigValue;

using ConfigArray = std::vector<ConfigValue>;
using ConfigObject = std::unordered_map<std::string, ConfigValue>;

struct ConfigValue
{
    using Variant = std::variant<
        std::nullptr_t,
        bool,
        int64_t,
        double,
        std::string,
        ConfigArray,
        ConfigObject
    >;

    Variant value;

    ConfigValue() : value(nullptr) {}

    template<typename T>
    ConfigValue(T v) : value(std::move(v)) {}
};
```

18.2.2 Tree structure

Configuration data is naturally hierarchical.

```
ConfigObject root;

root["database"] = ConfigObject{
    {"host", "localhost"},
    {"port", 5432}
};
```

18.2.3 Safe access helpers

```
#include <optional>

template<typename T>
std::optional<T> get(const ConfigValue& v)
{
    if (auto ptr = std::get_if<T>(&v.value))
        return *ptr;
    return std::nullopt;
}
```

18.2.4 Visitor-based operations

```
#include <iostream>

void print(const ConfigValue& v)
{
    std::visit([](const auto& val)
    {
        using T = std::decay_t<decltype(val)>;

        if constexpr (std::is_same_v<T, std::string>)
```

```
            std::cout << val << '\n';
        else if constexpr (std::is_arithmetic_v<T>)
            std::cout << val << '\n';
        else
            std::cout << "[complex type]\n";
    }, v.value);
}
```

18.2.5 Advantages of variant-based hierarchy

1. Type safety without dynamic casting
2. No virtual dispatch
3. Compact representation
4. Clear closed set of supported types

18.3 Polymorphic parsers

A production system must support multiple configuration formats. This requires a flexible parsing architecture.

18.3.1 Abstract parser interface

```
#include <string>

class IParser
{
public:
```

```
    virtual ~IParser() = default;
    virtual ConfigValue parse(const std::string& text) = 0;
};
```

18.3.2 JSON-like parser example

```
#include <sstream>

class SimpleParser : public IParser
{
public:
    ConfigValue parse(const std::string& text) override
    {
        if (text == "true")
            return ConfigValue(true);
        if (text == "false")
            return ConfigValue(false);

        std::istringstream iss(text);
        int64_t number;
        if (iss >> number)
            return ConfigValue(number);

        return ConfigValue(text);
    }
};
```

18.3.3 Parser selection via factory

```
#include <memory>
```

```
#include <string>

std::unique_ptr<IParser> create_parser(const std::string& format)
{
    if (format == "simple")
        return std::make_unique<SimpleParser>();

    return nullptr;
}
```

18.3.4 Template-based parser (static polymorphism)

```
template<typename Parser>
class ConfigLoader
{
public:
    explicit ConfigLoader(Parser parser_)
        : parser(std::move(parser_))
    {
    }

    ConfigValue load(const std::string& text)
    {
        return parser.parse(text);
    }

private:
    Parser parser;
};
```

18.3.5 Combining runtime and compile-time polymorphism

Modern C++ systems often mix both:

1. Runtime polymorphism for plugin-based parsers
2. Compile-time polymorphism for high-performance internal pipelines

18.4 RAII wrappers

RAII is critical for managing file handles, memory buffers, and parser resources safely.

18.4.1 File reader RAII wrapper

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

class FileReader
{
public:
    explicit FileReader(const std::string& path)
        : file(path)
    {
        if (!file)
            throw std::runtime_error("Failed to open file");
    }

    std::string read_all()
    {
```

```
        return std::string(
            (std::istreambuf_iterator<char>(file)),
            std::istreambuf_iterator<char>()
        );
    }

private:
    std::ifstream file;
};
```

18.4.2 Scoped parsing context

```
#include <iostream>

class ParseContext
{
public:
    ParseContext()
    {
        std::cout << "Parsing started\n";
    }

    ~ParseContext()
    {
        std::cout << "Parsing finished\n";
    }
};
```

18.4.3 Combining RAII with parsing

```
ConfigValue load_config(const std::string& path, IParser& parser)
{
    ParseContext ctx;

    FileReader reader(path);
    auto text = reader.read_all();

    return parser.parse(text);
}
```

18.4.4 Error-safe resource management

RAII ensures:

1. Files are always closed
2. Memory is always released
3. Exceptions do not leak resources

18.4.5 Custom deleter example

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

struct FileCloser
{
    void operator()(FILE* f) const
    {
```

```
        if (f)
            std::fclose(f);
    }
};

using FilePtr = std::unique_ptr<FILE, FileCloser>;

FilePtr open_file(const char* path)
{
    return FilePtr(std::fopen(path, "r"));
}
```

18.5 Integration example

```
#include <iostream>

int main()
{
    try
    {
        auto parser = create_parser("simple");

        if (!parser)
        {
            std::cerr << "No parser available\n";
            return 1;
        }

        ConfigValue config = load_config("config.txt", *parser);
```

```
        print(config);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "Error: " << ex.what() << '\n';
    }
}
```

18.6 Production considerations

18.6.1 Thread safety

A production config system should:

1. Avoid mutable global state
2. Use immutable configuration snapshots
3. Synchronize updates explicitly

18.6.2 Error handling

1. Prefer exceptions for unrecoverable parse errors
2. Use `std::optional` or `std::expected`-like patterns for recoverable cases

18.6.3 Extensibility

1. Support plugin-based parsers
2. Allow custom value converters

3. Provide hooks for validation

18.6.4 Performance

1. Minimize allocations
2. Use move semantics aggressively
3. Prefer contiguous containers

18.7 Conclusion

A production-grade configuration library in Modern C++ combines multiple advanced design techniques:

1. Variant-based value hierarchies for type-safe data representation
2. Polymorphic parsers for format extensibility
3. RAII wrappers for safe resource management
4. Clean architectural separation between layers

This design enables systems that are robust, extensible, efficient, and maintainable while fully leveraging Modern C++ capabilities.

Implementation

19.1 JSON parser

A production-grade configuration library needs a parser that is structurally correct, easy to test, exception-safe, and cleanly separated from the storage and access layers. For a JSON-based configuration system, the parser should follow the core JSON model precisely: a JSON value is one of object, array, number, string, `true`, `false`, or `null`. In a practical C++ implementation, the parser is usually written as a recursive-descent parser over a lexical stream, because this style is readable, deterministic, and maps naturally to the JSON grammar.

19.1.1 Design goals

A practical JSON parser for a configuration library should aim for the following:

1. Correct recognition of the full JSON value set.
2. Clear separation between lexing and parsing.
3. Precise error messages with source position.
4. Minimal ownership ambiguity.
5. Strong exception safety.

6. Easy integration with the internal `ConfigValue` tree.

19.1.2 Core value model

A common internal representation is a variant-based value tree. This matches JSON naturally because JSON itself is a closed algebraic set of value alternatives.

```
#include <cstdint>
#include <map>
#include <string>
#include <variant>
#include <vector>

struct ConfigValue;

using ConfigArray  = std::vector<ConfigValue>;
using ConfigObject = std::map<std::string, ConfigValue>;

struct ConfigValue
{
    using storage_type = std::variant<
        std::nullptr_t,
        bool,
        std::int64_t,
        double,
        std::string,
        ConfigArray,
        ConfigObject
    >;
```

```
    storage_type value;

    ConfigValue() : value(nullptr) {}

    template<typename T>
    ConfigValue(T v) : value(std::move(v)) {}
};
```

This design is especially suitable for configuration systems because:

1. The set of supported value kinds is explicit.
2. No base-class hierarchy is required.
3. Access can be implemented with `std::get_if` and visitation.
4. Objects and arrays can nest recursively.

19.1.3 Lexer design

Although a parser can operate directly on characters, a lexer simplifies implementation by converting the raw text into meaningful tokens.

```
#include <cctype>
#include <stdexcept>
#include <string>
#include <string_view>
#include <vector>

enum class TokenType
{
```

```
    LBrace,
    RBrace,
    LBracket,
    RBracket,
    Colon,
    Comma,
    String,
    Integer,
    Floating,
    TrueLit,
    FalseLit,
    NullLit,
    End
};

struct Token
{
    TokenType type{};
    std::string lexeme;
    std::size_t line{};
    std::size_t column{};
};

class JsonLexer
{
public:
    explicit JsonLexer(std::string_view src)
        : source(src)
    {
```

```
}

std::vector<Token> tokenize()
{
    std::vector<Token> out;

    while (!at_end())
    {
        skip_whitespace();

        if (at_end())
            break;

        const char ch = peek();

        switch (ch)
        {
        case '{': out.push_back(simple(TokenType::LBrace));   break;
        case '}': out.push_back(simple(TokenType::RBrace));   break;
        case '[': out.push_back(simple(TokenType::LBracket)); break;
        case ']': out.push_back(simple(TokenType::RBracket)); break;
        case ':': out.push_back(simple(TokenType::Colon));    break;
        case ',': out.push_back(simple(TokenType::Comma));    break;
        case '"': out.push_back(read_string());               break;
        default:
            if (ch == '-' || std::isdigit(static_cast<unsigned char>(ch)))
                out.push_back(read_number());
            else if (starts_with("true"))
                out.push_back(keyword(TokenType::TrueLit, "true"));
```

```
                else if (starts_with("false"))
                    out.push_back(keyword(TokenType::FalseLit, "false"));
                else if (starts_with("null"))
                    out.push_back(keyword(TokenType::NullLit, "null"));
                else
                    error("Unexpected character");
                break;
            }
        }

        out.push_back(Token{TokenType::End, "", line, column});
        return out;
    }

private:
    std::string_view source;
    std::size_t pos{0};
    std::size_t line{1};
    std::size_t column{1};

    bool at_end() const
    {
        return pos >= source.size();
    }

    char peek() const
    {
        return source[pos];
    }
```

```
char advance()
{
    char ch = source[pos++];
    if (ch == '\n')
    {
        ++line;
        column = 1;
    }
    else
    {
        ++column;
    }
    return ch;
}

void skip_whitespace()
{
    while (!at_end())
    {
        const char ch = peek();
        if (ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n')
            advance();
        else
            break;
    }
}

Token simple(TokenType type)
```

```
    {
        const std::size_t start_line = line;
        const std::size_t start_col  = column;
        std::string lex(1, advance());
        return Token{type, std::move(lex), start_line, start_col};
    }

    bool starts_with(std::string_view s) const
    {
        return source.substr(pos, s.size()) == s;
    }

    Token keyword(TokenType type, std::string_view text)
    {
        const std::size_t start_line = line;
        const std::size_t start_col  = column;

        for (char ignored : text)
            (void)ignored, advance();

        return Token{type, std::string(text), start_line, start_col};
    }

    Token read_string()
    {
        const std::size_t start_line = line;
        const std::size_t start_col  = column;

        advance(); // opening quote
```

```
        std::string out;

        while (!at_end())
        {
            char ch = advance();

            if (ch == '"')
                return Token{TokenType::String, std::move(out), start_line,
                ↪  start_col};

            if (ch == '\\')
            {
                if (at_end())
                    error("Unterminated escape sequence");

                char esc = advance();

                switch (esc)
                {
                case '"':  out.push_back('"');  break;
                case '\\': out.push_back('\\'); break;
                case '/':  out.push_back('/');  break;
                case 'b':  out.push_back('\b'); break;
                case 'f':  out.push_back('\f'); break;
                case 'n':  out.push_back('\n'); break;
                case 'r':  out.push_back('\r'); break;
                case 't':  out.push_back('\t'); break;
                default:
                    error("Unsupported escape sequence");
```

```
            }
        }
        else
        {
            out.push_back(ch);
        }
    }

    error("Unterminated string literal");
    return {};
}

Token read_number()
{
    const std::size_t start_pos  = pos;
    const std::size_t start_line = line;
    const std::size_t start_col  = column;

    if (peek() == '-')
        advance();

    while (!at_end() && std::isdigit(static_cast<unsigned char>(peek())))
        advance();

    bool is_floating = false;

    if (!at_end() && peek() == '.')
    {
        is_floating = true;
```

```
            advance();

            while (!at_end() && std::isdigit(static_cast<unsigned char>(peek())))
                advance();
        }

        if (!at_end() && (peek() == 'e' || peek() == 'E'))
        {
            is_floating = true;
            advance();

            if (!at_end() && (peek() == '+' || peek() == '-'))
                advance();

            while (!at_end() && std::isdigit(static_cast<unsigned char>(peek())))
                advance();
        }

        std::string lex(source.substr(start_pos, pos - start_pos));

        return Token{
            is_floating ? TokenType::Floating : TokenType::Integer,
            std::move(lex),
            start_line,
            start_col
        };
    }

    [[noreturn]] void error(const std::string& message) const
```

```
    {
        throw std::runtime_error(
            "JsonLexer error at line " + std::to_string(line) +
            ", column " + std::to_string(column) +
            ": " + message);
    }
};
```

19.1.4 Parser structure

The parser consumes tokens and constructs the configuration tree. A recursive-descent parser is a natural choice because JSON values nest recursively.

```
#include <cstdlib>
#include <stdexcept>

class JsonParser
{
public:
    explicit JsonParser(std::vector<Token> toks)
        : tokens(std::move(toks))
    {
    }

    ConfigValue parse()
    {
        ConfigValue root = parse_value();
        expect(TokenType::End, "Expected end of input");
        return root;
    }
```

```
private:
    std::vector<Token> tokens;
    std::size_t current{0};

    const Token& peek() const
    {
        return tokens[current];
    }

    const Token& previous() const
    {
        return tokens[current - 1];
    }

    bool check(TokenType t) const
    {
        return peek().type == t;
    }

    bool match(TokenType t)
    {
        if (check(t))
        {
            ++current;
            return true;
        }
        return false;
    }
```

```
const Token& expect(TokenType t, const std::string& msg)
{
    if (!check(t))
        error(peek(), msg);

    ++current;
    return previous();
}

ConfigValue parse_value()
{
    if (match(TokenType::NullLit))
        return ConfigValue(nullptr);

    if (match(TokenType::TrueLit))
        return ConfigValue(true);

    if (match(TokenType::FalseLit))
        return ConfigValue(false);

    if (match(TokenType::Integer))
        return ConfigValue(static_cast<std::int64_t>(
            std::stoll(previous().lexeme)));

    if (match(TokenType::Floating))
        return ConfigValue(std::stod(previous().lexeme));

    if (match(TokenType::String))
```

```
        return ConfigValue(previous().lexeme);

    if (match(TokenType::LBracket))
        return parse_array();

    if (match(TokenType::LBrace))
        return parse_object();

    error(peek(), "Expected JSON value");
    return {};
}

ConfigValue parse_array()
{
    ConfigArray arr;

    if (match(TokenType::RBracket))
        return ConfigValue(std::move(arr));

    do
    {
        arr.push_back(parse_value());
    }
    while (match(TokenType::Comma));

    expect(TokenType::RBracket, "Expected ']'");
    return ConfigValue(std::move(arr));
}
```

```
    ConfigValue parse_object()
    {
        ConfigObject obj;

        if (match(TokenType::RBrace))
            return ConfigValue(std::move(obj));

        do
        {
            const Token& key = expect(TokenType::String, "Expected object key");
            expect(TokenType::Colon, "Expected ':' after object key");
            obj.emplace(key.lexeme, parse_value());
        }
        while (match(TokenType::Comma));

        expect(TokenType::RBrace, "Expected '}'");
        return ConfigValue(std::move(obj));
    }

    [[noreturn]] void error(const Token& tok, const std::string& msg) const
    {
        throw std::runtime_error(
            "JsonParser error at line " + std::to_string(tok.line) +
            ", column " + std::to_string(tok.column) +
            ": " + msg);
    }
};
```

19.1.5 Front-end convenience API

A production library usually provides a single top-level parse function that hides the lexer-parser split.

```
ConfigValue parse_json(std::string_view text)
{
    JsonLexer lexer(text);
    JsonParser parser(lexer.tokenize());
    return parser.parse();
}
```

19.1.6 Example usage

```
#include <iostream>

int main()
{
    const std::string text = R"json(
    {
        "app": "demo",
        "debug": true,
        "port": 8080,
        "pi": 3.14159,
        "paths": ["bin", "cfg", "logs"]
    }
    )json";

    ConfigValue root = parse_json(text);
```

```
    if (const auto* obj = std::get_if<ConfigObject>(&root.value))
    {
        std::cout << "root object entries = " << obj->size() << '\n';
    }
}
```

19.1.7 Production notes

A full production parser usually extends this design with:

1. Full Unicode escape handling.
2. Better numeric policy for integer versus floating conversion.
3. Maximum nesting limits.
4. Optional duplicate-key policy.
5. Structured diagnostics objects rather than only exceptions.

19.2 File watcher RAII

A configuration library often needs hot-reload support. On Windows, directory monitoring is commonly implemented by opening a directory handle and using `ReadDirectoryChangesW`. Because this API depends on system handles and cleanup discipline, it is an excellent example of RAII in systems programming.

19.2.1 Design goals

A file watcher for configuration reload should:

1. Own the OS handle safely.

2. Release the handle deterministically.
3. Expose a narrow, testable interface.
4. Hide raw Win32 details from higher layers.
5. Filter only the events relevant to configuration files.

19.2.2 RAII wrapper for `HANDLE`

The first step is to wrap a Windows `HANDLE` so that cleanup happens automatically.

```
#ifdef _WIN32
#include <windows.h>
#endif

#include <stdexcept>
#include <string>
#include <utility>

class UniqueHandle
{
public:
#ifdef _WIN32
    UniqueHandle() noexcept = default;

    explicit UniqueHandle(HANDLE h) noexcept
        : handle(h)
    {
    }
```

```
    ~UniqueHandle()
    {
        reset();
    }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept
        : handle(other.handle)
    {
        other.handle = INVALID_HANDLE_VALUE;
    }

    UniqueHandle& operator=(UniqueHandle&& other) noexcept
    {
        if (this != &other)
        {
            reset();
            handle = other.handle;
            other.handle = INVALID_HANDLE_VALUE;
        }
        return *this;
    }

    HANDLE get() const noexcept
    {
        return handle;
    }
```

```
    bool valid() const noexcept
    {
        return handle != nullptr && handle != INVALID_HANDLE_VALUE;
    }

    void reset(HANDLE h = INVALID_HANDLE_VALUE) noexcept
    {
        if (valid())
            CloseHandle(handle);
        handle = h;
    }

private:
    HANDLE handle{INVALID_HANDLE_VALUE};
#endif
};
```

19.2.3 Directory watcher interface

A clean watcher should not expose raw Win32 buffers to the higher config layer.

```
#include <filesystem>
#include <string>
#include <vector>

enum class FileChangeKind
{
    Added,
    Removed,
```

```
    Modified,
    RenamedOldName,
    RenamedNewName
};

struct FileChangeEvent
{
    FileChangeKind kind{};
    std::filesystem::path path;
};

class DirectoryWatcher
{
public:
    explicit DirectoryWatcher(const std::filesystem::path& dir);

    std::vector<FileChangeEvent> poll();

private:
#ifdef _WIN32
    UniqueHandle directory;
    std::filesystem::path watched_dir;
#endif
};
```

19.2.4 Windows implementation

To monitor a directory, the code opens the directory itself and not just a normal file. The wrapper below shows the common shape of a synchronous poll-based watcher.

```
#ifdef _WIN32
#include <windows.h>
#endif

#include <array>
#include <filesystem>
#include <stdexcept>
#include <vector>

DirectoryWatcher::DirectoryWatcher(const std::filesystem::path& dir)
    : watched_dir(dir)
{
#ifdef _WIN32
    HANDLE h = CreateFileW(
        dir.c_str(),
        FILE_LIST_DIRECTORY,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        nullptr,
        OPEN_EXISTING,
        FILE_FLAG_BACKUP_SEMANTICS,
        nullptr);

    if (h == INVALID_HANDLE_VALUE)
        throw std::runtime_error("Failed to open directory watcher handle");

    directory = UniqueHandle(h);
#else
    throw
    ↪  std::runtime_error("DirectoryWatcher is implemented here only for Windows");
```

```
#endif
}

std::vector<FileChangeEvent> DirectoryWatcher::poll()
{
    std::vector<FileChangeEvent> events;

#ifdef _WIN32
    std::array<std::byte, 16 * 1024> buffer{};
    DWORD bytes_returned = 0;

    const BOOL ok = ReadDirectoryChangesW(
        directory.get(),
        buffer.data(),
        static_cast<DWORD>(buffer.size()),
        FALSE,
        FILE_NOTIFY_CHANGE_FILE_NAME |
        FILE_NOTIFY_CHANGE_LAST_WRITE |
        FILE_NOTIFY_CHANGE_SIZE,
        &bytes_returned,
        nullptr,
        nullptr);

    if (!ok)
        throw std::runtime_error("ReadDirectoryChangesW failed");

    const char* base = reinterpret_cast<const char*>(buffer.data());
    const FILE_NOTIFY_INFORMATION* info =
        reinterpret_cast<const FILE_NOTIFY_INFORMATION*>(base);
```

```
while (true)
{
    std::wstring_view wide_name(info->FileName,
                                info->FileNameLength / sizeof(WCHAR));

    FileChangeEvent ev;
    ev.path = watched_dir / std::filesystem::path(wide_name);

    switch (info->Action)
    {
    case FILE_ACTION_ADDED:
        ev.kind = FileChangeKind::Added;
        break;
    case FILE_ACTION_REMOVED:
        ev.kind = FileChangeKind::Removed;
        break;
    case FILE_ACTION_MODIFIED:
        ev.kind = FileChangeKind::Modified;
        break;
    case FILE_ACTION_RENAMED_OLD_NAME:
        ev.kind = FileChangeKind::RenamedOldName;
        break;
    case FILE_ACTION_RENAMED_NEW_NAME:
        ev.kind = FileChangeKind::RenamedNewName;
        break;
    default:
        ev.kind = FileChangeKind::Modified;
        break;
```

```
        }

        events.push_back(std::move(ev));

        if (info->NextEntryOffset == 0)
            break;

        info = reinterpret_cast<const FILE_NOTIFY_INFORMATION*>(
            reinterpret_cast<const char*>(info) + info->NextEntryOffset);
    }
#endif

    return events;
}
```

19.2.5 Config-specific filtering

A configuration system usually does not want every event. It typically wants only the relevant configuration file extensions.

```
#include <algorithm>

bool is_config_file(const std::filesystem::path& p)
{
    const auto ext = p.extension().string();
    return ext == ".json" || ext == ".cfg" || ext == ".conf";
}

std::vector<FileChangeEvent>
filter_config_events(const std::vector<FileChangeEvent>& in)
```

```
{
    std::vector<FileChangeEvent> out;

    for (const auto& ev : in)
    {
        if (is_config_file(ev.path))
            out.push_back(ev);
    }

    return out;
}
```

19.2.6 Hot-reload loop

```
#include <iostream>

int main()
{
#ifdef _WIN32
    DirectoryWatcher watcher(LR"(C:\configs)");

    for (;;)
    {
        auto raw_events = watcher.poll();
        auto events = filter_config_events(raw_events);

        for (const auto& ev : events)
        {
            std::cout << "Changed: " << ev.path.string() << '\n';
```

```
        }
    }
#endif
}
```

19.2.7 Why RAII matters here

Without RAII, every early return, exception path, or partial initialization path risks leaking the directory handle. With the wrapper-based approach, the handle is closed automatically when the watcher or the owning handle wrapper leaves scope.

19.3 Merge engine

A production configuration system rarely loads only one file. It often combines:

1. Built-in defaults.
2. System-wide configuration.
3. User configuration.
4. Environment or deployment overlays.
5. Runtime overrides.

That means the implementation needs a merge engine with explicit semantics.

19.3.1 Core merge strategies

A good merge engine should define how each kind of value is combined. A practical policy is:

1. Object + object: deep merge by key.

2. Scalar + scalar: override old with new.
3. Scalar + object or array: replace.
4. Array + array: policy-based, usually replace or concatenate.
5. Missing value: insert.

19.3.2 Merge policy model

```
enum class ArrayMergePolicy
{
    Replace,
    Append
};

struct MergeOptions
{
    ArrayMergePolicy arrays{ArrayMergePolicy::Replace};
    bool overwrite_scalars{true};
};
```

19.3.3 Helper predicates

```
bool is_object(const ConfigValue& v)
{
    return std::holds_alternative<ConfigObject>(v.value);
}

bool is_array(const ConfigValue& v)
{
```

```
    return std::holds_alternative<ConfigArray>(v.value);
}

ConfigObject& as_object(ConfigValue& v)
{
    return std::get<ConfigObject>(v.value);
}

const ConfigObject& as_object(const ConfigValue& v)
{
    return std::get<ConfigObject>(v.value);
}

ConfigArray& as_array(ConfigValue& v)
{
    return std::get<ConfigArray>(v.value);
}

const ConfigArray& as_array(const ConfigValue& v)
{
    return std::get<ConfigArray>(v.value);
}
```

19.3.4 Recursive deep merge

```
void merge_into(ConfigValue& base,
                const ConfigValue& incoming,
                const MergeOptions& options)
{
```

```
    if (is_object(base) && is_object(incoming))
    {
        auto& dst = as_object(base);
        const auto& src = as_object(incoming);

        for (const auto& [key, src_value] : src)
        {
            auto it = dst.find(key);

            if (it == dst.end())
            {
                dst.emplace(key, src_value);
            }
            else
            {
                merge_into(it->second, src_value, options);
            }
        }
        return;
    }

    if (is_array(base) && is_array(incoming))
    {
        if (options.arrays == ArrayMergePolicy::Replace)
        {
            base = incoming;
        }
        else
        {
```

```
            auto& dst = as_array(base);
            const auto& src = as_array(incoming);
            dst.insert(dst.end(), src.begin(), src.end());
        }
        return;
    }

    if (options.overwrite_scalars)
        base = incoming;
}
```

19.3.5 Example

```
#include <iostream>

int main()
{
    ConfigValue defaults = ConfigObject{
        {"server", ConfigObject{
            {"host", std::string("127.0.0.1")},
            {"port", std::int64_t(8080)}
        }},
        {"features", ConfigArray{
            ConfigValue("logging"),
            ConfigValue("metrics")
        }}
    };

    ConfigValue user = ConfigObject{
```

```
        {"server", ConfigObject{
            {"port", std::int64_t(9090)}
        }},
        {"features", ConfigArray{
            ConfigValue("debug-ui")
        }}
    };

    MergeOptions options;
    options.arrays = ArrayMergePolicy::Append;

    merge_into(defaults, user, options);

    std::cout << "merge complete\n";
}
```

19.3.6 Layered configuration stack

A typical real sequence looks like this:

```
ConfigValue final_config = parse_json(default_text);

merge_into(final_config, parse_json(system_text), MergeOptions{});
merge_into(final_config, parse_json(user_text),   MergeOptions{});
merge_into(final_config, parse_json(local_text),  MergeOptions{});
```

19.3.7 Design considerations

A production merge engine must answer several architectural questions explicitly:

1. Are arrays replaced or appended?

2. Are duplicate object keys allowed during parse?
3. Should type mismatches replace or raise an error?
4. Are null values deletion markers or ordinary values?
5. Should the merge engine track source provenance?

19.3.8 Optional provenance tracking

For diagnostics, it is often useful to know where each value came from.

```
struct SourceInfo
{
    std::string source_name;
    std::size_t line{};
    std::size_t column{};
};
```

A full production system may store `SourceInfo` alongside each parsed node or in a parallel metadata tree so that merge conflicts and validation errors can point to the correct origin file.

19.4 Visitor-based evaluation

Once the configuration tree exists, the library needs a way to evaluate, render, validate, and convert values without writing repetitive type-switch code everywhere. This is where visitor-based evaluation becomes central.
Because the value layer is based on `std::variant`, the natural evaluation mechanism is `std::visit` with overloaded visitors.

19.4.1 Utility visitor helper

```
template<class... Ts>
struct Overloaded : Ts...
{
    using Ts::operator()...;
};

template<class... Ts>
Overloaded(Ts...) -> Overloaded<Ts...>;
```

19.4.2 Pretty-print evaluation

One common evaluation task is producing readable output for diagnostics, logging, or export.

```
#include <iostream>

void print_value(const ConfigValue& v, int indent = 0);

void print_indent(int n)
{
    for (int i = 0; i < n; ++i)
        std::cout << ' ';
}

void print_value(const ConfigValue& v, int indent)
{
    std::visit(Overloaded{
        [&](std::nullptr_t)
        {
```

```
            std::cout << "null";
        },
        [&](bool b)
        {
            std::cout << (b ? "true" : "false");
        },
        [&](std::int64_t i)
        {
            std::cout << i;
        },
        [&](double d)
        {
            std::cout << d;
        },
        [&](const std::string& s)
        {
            std::cout << '"' << s << '"';
        },
        [&](const ConfigArray& arr)
        {
            std::cout << "[\n";
            for (std::size_t i = 0; i < arr.size(); ++i)
            {
                print_indent(indent + 2);
                print_value(arr[i], indent + 2);
                if (i + 1 != arr.size())
                    std::cout << ',';
                std::cout << '\n';
            }
```

```
            print_indent(indent);
            std::cout << ']';
        },
        [&](const ConfigObject& obj)
        {
            std::cout << "{\n";
            std::size_t count = 0;

            for (const auto& [k, val] : obj)
            {
                print_indent(indent + 2);
                std::cout << '"' << k << "\": ";
                print_value(val, indent + 2);
                if (++count != obj.size())
                    std::cout << ',';
                std::cout << '\n';
            }

            print_indent(indent);
            std::cout << '}';
        }
    }, v.value);
}
```

19.4.3 Path-based lookup evaluation

A configuration system often needs to evaluate a path such as `server.port` into a concrete node.

```
#include <optional>
```

```
#include <sstream>
#include <vector>

std::vector<std::string> split_path(const std::string& path)
{
    std::vector<std::string> parts;
    std::stringstream ss(path);
    std::string item;

    while (std::getline(ss, item, '.'))
        parts.push_back(item);

    return parts;
}

const ConfigValue* find_path(const ConfigValue& root, const std::string& path)
{
    const ConfigValue* current = &root;

    for (const auto& part : split_path(path))
    {
        const auto* obj = std::get_if<ConfigObject>(&current->value);
        if (!obj)
            return nullptr;

        auto it = obj->find(part);
        if (it == obj->end())
            return nullptr;
```

```
        current = &it->second;
    }

    return current;
}
```

19.4.4 Typed conversion evaluation

Once a node is found, a visitor can convert it safely into the requested application type.

```
template<typename T>
std::optional<T> evaluate_as(const ConfigValue& v);

template<>
std::optional<int> evaluate_as<int>(const ConfigValue& v)
{
    return std::visit(Overloaded{
        [](std::int64_t i) -> std::optional<int>
        {
            return static_cast<int>(i);
        },
        [](double d) -> std::optional<int>
        {
            return static_cast<int>(d);
        },
        [](const auto&) -> std::optional<int>
        {
            return std::nullopt;
        }
    }, v.value);
```

```
}

template<>
std::optional<std::string> evaluate_as<std::string>(const ConfigValue& v)
{
    return std::visit(Overloaded{
        [](const std::string& s) -> std::optional<std::string>
        {
            return s;
        },
        [](bool b) -> std::optional<std::string>
        {
            return b ? "true" : "false";
        },
        [](std::int64_t i) -> std::optional<std::string>
        {
            return std::to_string(i);
        },
        [](double d) -> std::optional<std::string>
        {
            return std::to_string(d);
        },
        [](const auto&) -> std::optional<std::string>
        {
            return std::nullopt;
        }
    }, v.value);
}
```

19.4.5 Example query API

```
template<typename T>
std::optional<T> query_as(const ConfigValue& root, const std::string& path)
{
    if (const ConfigValue* node = find_path(root, path))
        return evaluate_as<T>(*node);

    return std::nullopt;
}

int main()
{
    const std::string text = R"json(
    {
        "server": {
            "host": "localhost",
            "port": 8080
        }
    }
    )json";

    ConfigValue root = parse_json(text);

    auto port = query_as<int>(root, "server.port");
    auto host = query_as<std::string>(root, "server.host");

    if (port) std::cout << "port = " << *port << '\n';
    if (host) std::cout << "host = " << *host << '\n';
```

```
}
```

19.4.6 Validation as a visitor

Validation is another strong fit for visitation. A schema-like validation layer can walk the tree and enforce expected types.

```
bool is_scalar(const ConfigValue& v)
{
    return std::visit(Overloaded{
        [](std::nullptr_t)        { return true;  },
        [](bool)                  { return true;  },
        [](std::int64_t)          { return true;  },
        [](double)                { return true;  },
        [](const std::string&)    { return true;  },
        [](const ConfigArray&)    { return false; },
        [](const ConfigObject&)   { return false; }
    }, v.value);
}
```

19.4.7 Why visitor-based evaluation is a strong fit

Visitor-based evaluation works especially well here because:

1. The value set is closed and known.
2. Dispatch remains type-safe.
3. New operations can be added without changing the value representation.
4. The implementation avoids unsafe casts and inheritance-heavy hierarchies.

19.5 End-to-end implementation sketch

The following example shows how the major pieces fit together.

```
#include <fstream>
#include <iterator>
#include <stdexcept>
#include <string>

std::string read_all_text_file(const std::string& path)
{
    std::ifstream file(path);
    if (!file)
        throw std::runtime_error("Failed to open file: " + path);

    return std::string(
        (std::istreambuf_iterator<char>(file)),
         std::istreambuf_iterator<char>());
}

ConfigValue load_config_file(const std::string& path)
{
    return parse_json(read_all_text_file(path));
}

ConfigValue load_merged_config(const std::vector<std::string>& files)
{
    ConfigValue merged = ConfigObject{};
    MergeOptions options;
```

```
    for (const auto& path : files)
    {
        ConfigValue current = load_config_file(path);
        merge_into(merged, current, options);
    }

    return merged;
}
```

19.5.1 Using the config system

```
#include <iostream>

int main()
{
    try
    {
        ConfigValue cfg = load_merged_config({
            "default.json",
            "site.json",
            "user.json"
        });

        print_value(cfg);
        std::cout << '\n';

        auto port = query_as<int>(cfg, "server.port");
        if (port)
            std::cout << "Resolved server.port = " << *port << '\n';
```

```
    }
    catch (const std::exception& ex)
    {
        std::cerr << "Configuration error: " << ex.what() << '\n';
        return 1;
    }
}
```

19.6 Windows build guidance

For Visual Studio 2022, a practical configuration for this chapter is:

```
Project Properties
  C/C++ -> Language -> C++ Language Standard -> ISO C++20 Standard (/std:c++20)
  C/C++ -> Code Generation -> Enable C++ Exceptions -> Yes (/EHsc)
  C/C++ -> Optimization -> Optimization -> Maximize Speed (/O2) for Release
  C/C++ -> Warning Level -> Level4 (/W4)
```

Command-line MSVC build:

```
cl /std:c++20 /EHsc /W4 /O2 chapter19.cpp
```

If compiling the Windows watcher code, keep the Windows-specific parts behind `#ifdef _WIN32` and ensure that the source includes:

```
#include <windows.h>
```

19.7 Conclusion

The implementation of a production-grade configuration library demonstrates how several Modern C++ design directions fit together naturally:

1. A recursive-descent JSON parser maps cleanly onto a closed variant-based value model.
2. RAII makes low-level file and directory watching safe and maintainable.
3. A merge engine turns separate configuration layers into one resolved runtime view.
4. Visitor-based evaluation provides type-safe querying, rendering, and validation without inheritance-heavy designs.

Taken together, these techniques form a realistic architecture for a modern configuration subsystem that is extensible, testable, exception-safe, and suitable for real production use on Windows and other platforms with small platform-specific adaptation layers.

Testing, Documentation & Extensions

20.1 Unit tests

A production-grade configuration library must include a comprehensive unit testing strategy that validates correctness, stability, and edge-case behavior. Modern C++ encourages writing tests that are deterministic, isolated, and expressive.

20.1.1 Testing principles

1. Tests must be independent and repeatable.
2. Each test should validate one behavior.
3. Use value-based comparisons for deterministic results.
4. Avoid reliance on global state.
5. Prefer small focused test cases over large monolithic ones.

20.1.2 Minimal test framework approach

For portability and simplicity, a lightweight assertion mechanism can be implemented without external dependencies.

```
#include <iostream>
#include <stdexcept>

#define ASSERT_TRUE(cond) \
    do { if (!(cond)) throw std::runtime_error("Assertion failed: " #cond); } while
    ↪  (0)

#define ASSERT_EQ(a, b) \
    do { if (!((a) == (b))) throw std::runtime_error("Assertion failed: " #a " == "
    ↪  #b); } while (0)
```

20.1.3 Testing JSON parsing

```
void test_parse_simple_object()
{
    const std::string text = R"json(
    { "key": 42 }
    )json";

    ConfigValue root = parse_json(text);

    const auto* obj = std::get_if<ConfigObject>(&root.value);
    ASSERT_TRUE(obj != nullptr);

    auto it = obj->find("key");
    ASSERT_TRUE(it != obj->end());

    const auto* val = std::get_if<std::int64_t>(&it->second.value);
    ASSERT_TRUE(val != nullptr);
    ASSERT_EQ(*val, 42);
```

```
}
```

20.1.4 Testing merge engine

```
void test_merge_override()
{
    ConfigValue base = ConfigObject{
        {"a", std::int64_t(1)}
    };

    ConfigValue incoming = ConfigObject{
        {"a", std::int64_t(2)}
    };

    MergeOptions opts;
    merge_into(base, incoming, opts);

    const auto& obj = std::get<ConfigObject>(base.value);
    ASSERT_EQ(std::get<std::int64_t>(obj.at("a").value), 2);
}
```

20.1.5 Testing path queries

```
void test_query_path()
{
    ConfigValue root = ConfigObject{
        {"server", ConfigObject{
            {"port", std::int64_t(8080)}
        }}
    };
```

```
    auto port = query_as<int>(root, "server.port");
    ASSERT_TRUE(port.has_value());
    ASSERT_EQ(*port, 8080);
}
```

20.1.6 Test runner

```
int main()
{
    try
    {
        test_parse_simple_object();
        test_merge_override();
        test_query_path();

        std::cout << "All tests passed\n";
    }
    catch (const std::exception& ex)
    {
        std::cerr << "Test failure: " << ex.what() << '\n';
        return 1;
    }
}
```

20.1.7 Modern testing extensions

In a full production environment, testing may be extended with:

1. Property-based testing for random input validation.

2. Fuzz testing for parser robustness.
3. Golden file testing for stable output formats.
4. Performance benchmarks for parsing and merging.

20.2 Error handling

Error handling is a critical component of a configuration system. It must provide clear diagnostics while preserving strong guarantees.

20.2.1 Error categories

1. Lexical errors (invalid tokens)
2. Syntax errors (invalid structure)
3. Semantic errors (invalid configuration meaning)
4. Runtime access errors (missing keys, wrong types)

20.2.2 Exception-based handling

Modern C++ favors exceptions for unrecoverable errors such as parsing failures.

```
class ParseError : public std::runtime_error
{
public:
    ParseError(const std::string& msg,
               std::size_t line,
               std::size_t column)
        : std::runtime_error(msg),
```

```
          line_(line),
          column_(column)
    {
    }

    std::size_t line() const noexcept { return line_; }
    std::size_t column() const noexcept { return column_; }

private:
    std::size_t line_;
    std::size_t column_;
};
```

20.2.3 Throwing structured errors

```
[[noreturn]] void parser_error(std::size_t line,
                               std::size_t column,
                               const std::string& msg)
{
    throw ParseError(msg, line, column);
}
```

20.2.4 Recoverable errors using optional

```
#include <optional>

std::optional<int> safe_get_int(const ConfigValue& v)
{
    if (auto p = std::get_if<std::int64_t>(&v.value))
        return static_cast<int>(*p);
```

```
    return std::nullopt;
}
```

20.2.5 Error propagation design

1. Parsing functions throw exceptions on invalid input.
2. Query functions return `std::optional` for missing values.
3. High-level APIs convert errors into user-visible diagnostics.

20.2.6 Error reporting example

```
try
{
    ConfigValue cfg = parse_json("invalid json");
}
catch (const ParseError& e)
{
    std::cerr << "Parse error at "
              << e.line() << ":" << e.column()
              << " -> " << e.what() << '\n';
}
```

20.3 Thread safety

Thread safety is essential for modern applications where configuration data may be accessed concurrently.

20.3.1 Immutable configuration model

A recommended design is to treat configuration data as immutable after construction.

```
class ConfigSnapshot
{
public:
    explicit ConfigSnapshot(ConfigValue root_)
        : root(std::move(root_))
    {
    }

    const ConfigValue& get_root() const noexcept
    {
        return root;
    }

private:
    ConfigValue root;
};
```

20.3.2 Atomic snapshot swapping

```
#include <atomic>
#include <memory>

class ConfigManager
{
public:
    void update(ConfigValue new_root)
```

```
    {
        auto new_snapshot =
            std::make_shared<ConfigSnapshot>(std::move(new_root));

        std::atomic_store(&current, new_snapshot);
    }

    std::shared_ptr<const ConfigSnapshot> get() const
    {
        return std::atomic_load(&current);
    }

private:
    std::shared_ptr<ConfigSnapshot> current;
};
```

20.3.3 Reader threads

```
void reader_thread(const ConfigManager& mgr)
{
    auto snapshot = mgr.get();
    if (!snapshot)
        return;

    auto port = query_as<int>(snapshot->get_root(), "server.port");
}
```

20.3.4 Thread safety guarantees

1. Readers never block.

2. Writers replace snapshots atomically.

3. No shared mutable state inside the tree.

20.3.5 Alternative: mutex-based model

```
#include <mutex>

class SafeConfig
{
public:
    void set(ConfigValue v)
    {
        std::lock_guard<std::mutex> lock(m);
        root = std::move(v);
    }

    ConfigValue get_copy() const
    {
        std::lock_guard<std::mutex> lock(m);
        return root;
    }

private:
    mutable std::mutex m;
    ConfigValue root;
};
```

20.3.6 Modern recommendation

Prefer immutable snapshots with atomic replacement for high-read scenarios. Use mutexes only when mutation is frequent or unavoidable.

20.4 Exporting the library as modules

C++20 modules provide a modern alternative to traditional header/source separation. They reduce compile times and eliminate macro-related issues.

20.4.1 Module interface unit

```
// config.ixx
export module config;

import <string>;
import <variant>;
import <vector>;
import <map>;

export struct ConfigValue
{
    using storage_type = std::variant<
        std::nullptr_t,
        bool,
        std::int64_t,
        double,
        std::string,
        std::vector<ConfigValue>,
```

```
        std::map<std::string, ConfigValue>
    >;

    storage_type value;
};

export ConfigValue parse_json(const std::string& text);
```

20.4.2 Module implementation unit

```
// config.cpp
module config;

ConfigValue parse_json(const std::string& text)
{
    // simplified stub
    return ConfigValue{};
}
```

20.4.3 Using the module

```
import config;
#include <iostream>

int main()
{
    ConfigValue v = parse_json("{\"a\":1}");
    std::cout << "Parsed\n";
}
```

20.4.4 MSVC build instructions

```
cl /std:c++20 /experimental:module /c config.ixx
cl /std:c++20 /experimental:module config.cpp main.cpp
```

20.4.5 CMake with modules (basic)

```
cmake_minimum_required(VERSION 3.28)
project(ConfigModule LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)

add_library(config)
target_sources(config
    PUBLIC
        FILE_SET cxx_modules TYPE CXX_MODULES FILES config.ixx
    PRIVATE
        config.cpp
)

add_executable(app main.cpp)
target_link_libraries(app PRIVATE config)
```

20.4.6 Advantages of modules

1. Faster compilation times
2. Stronger encapsulation
3. Elimination of header inclusion order issues

4. Clear interface boundaries

20.5 Conclusion

Testing, error handling, thread safety, and modularization are essential for transforming a configuration system into a production-grade library.

1. Unit tests ensure correctness and regression safety.
2. Structured error handling improves diagnostics and reliability.
3. Immutable snapshot-based design enables safe concurrency.
4. C++20 modules provide modern packaging and compilation benefits.

A well-designed system combines these elements into a cohesive architecture that is robust, maintainable, and scalable across real-world applications.

Part VIII

Ending of the Volume

Conclusion The Future of Object-Oriented Design in Modern C++

Lessons learned from modern C++ object design

Modern C++ object design has evolved far beyond the older style in which classes were treated primarily as containers for methods and raw-resource ownership was managed manually. The strongest lesson of Modern C++ is that object-oriented design in this language is no longer centered on inheritance alone. Instead, it is centered on explicit ownership, deterministic lifetime, value semantics where appropriate, narrow interfaces, generic composition, and careful selection between runtime and compile-time abstraction.
The modern object model of C++ remains powerful precisely because it is not trapped inside one paradigm. A class in Modern C++ may represent:

1. A value type with clear invariants
2. A RAII guard that manages a resource
3. A polymorphic interface boundary
4. A policy carrier for compile-time behavior
5. A lightweight aggregate used in data-oriented systems

The deeper lesson is that modern object design is not about making everything a class hierarchy. It is about making types express ownership, lifetime, semantics, and correctness as clearly as possible.

Ownership over raw pointers

One of the most important shifts in Modern C++ is the move away from raw owning pointers. Raw pointers still have an essential place in the language, but their meaning is now much narrower and more disciplined than in older styles of code.
A raw pointer should usually express observation or non-owning access, not ownership.

The old problem

Historically, many C++ designs placed heap allocation and deletion directly inside business logic:

```
class Logger
{
public:
    void write(const std::string& msg)
    {
        std::cout << msg << '\n';
    }
};

class Service
{
public:
    Service()
        : logger(new Logger)
```

```
    {
    }

    ~Service()
    {
        delete logger;
    }

    void run()
    {
        logger->write("running");
    }

private:
    Logger* logger;
};
```

Although this code may appear simple, it creates several structural problems:

1. Ownership is expressed only informally.
2. Copy and move behavior become dangerous unless carefully defined.
3. Exception safety becomes fragile.
4. Manual `delete` invites leaks and double-deletion bugs.

The modern ownership model

Modern C++ expresses ownership explicitly with standard library tools.
If an object owns exactly one heap allocation, `std::unique_ptr` is usually the correct representation:

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

class Logger
{
public:
    void write(const std::string& msg)
    {
        std::cout << msg << '\n';
    }
};

class Service
{
public:
    Service()
        : logger(std::make_unique<Logger>())
    {
    }

    void run()
    {
        logger->write("running");
    }

private:
    std::unique_ptr<Logger> logger;
};
```

This version is stronger because:

1. Ownership is explicit in the type.
2. Destruction is automatic and exception-safe.
3. Move semantics work naturally.
4. Copying is disallowed by default unless intentionally designed.

If ownership is genuinely shared, `std::shared_ptr` may be used, but modern design treats it as a deliberate tool rather than a default convenience. Shared ownership should communicate real shared lifetime, not avoidance of design decisions.

Raw pointers still matter

Modern C++ does not eliminate raw pointers. It clarifies their role.
A raw pointer is often appropriate for:

1. Optional non-owning access
2. Interfacing with low-level APIs
3. Traversal without lifetime transfer
4. Expressing that null is a meaningful state

For example:

```
class Renderer
{
public:
    void attach_logger(Logger* logger_) noexcept
```

```
    {
        logger = logger_;
    }

    void draw()
    {
        if (logger)
            logger->write("draw call");
    }

private:
    Logger* logger{};
};
```

Here, the raw pointer does not own the `Logger`. That is an appropriate and modern use.

The architectural lesson

Ownership must be visible in the type system. Once ownership is explicit, many other design decisions become easier:

1. APIs become clearer
2. Testing becomes easier
3. Resource leaks become rarer
4. Move operations become natural
5. Lifetime bugs become easier to reason about

This is one of the most important lessons of modern object design.

RAII as the backbone of reliability

RAII remains the deepest and most characteristically C++ design principle. It is not merely a memory-management trick. It is the foundation of deterministic cleanup, exception safety, invariant preservation, and reliable systems programming.
The central idea is simple: a resource is acquired during object initialization and released in the destructor.

The power of deterministic lifetime

Because local objects are destroyed automatically at the end of scope, RAII ties correctness to lexical structure.

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

std::mutex g_mutex;

void write_log(const std::string& path, const std::string& message)
{
    std::lock_guard<std::mutex> lock(g_mutex);
    std::ofstream out(path);

    if (!out)
        throw std::runtime_error("failed to open file");

    out << message << '\n';
}
```

This function demonstrates why RAII is such a strong architectural principle:

1. The mutex is unlocked automatically.
2. The file is closed automatically.
3. Exceptions do not bypass cleanup.
4. The code remains compact and readable.

RAII is broader than memory

A modern C++ programmer should think of RAII as the general model for any scarce or correctness-critical resource:

1. Files
2. Mutexes
3. Threads and join behavior
4. Sockets
5. Handles
6. Database connections
7. Temporary state transitions

A custom RAII wrapper for a Windows handle illustrates the same pattern:

```
#include <windows.h>

class UniqueHandle
{
public:
```

```
    UniqueHandle() noexcept = default;

    explicit UniqueHandle(HANDLE h) noexcept
        : handle(h)
    {
    }

    ~UniqueHandle()
    {
        reset();
    }

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;

    UniqueHandle(UniqueHandle&& other) noexcept
        : handle(other.handle)
    {
        other.handle = INVALID_HANDLE_VALUE;
    }

    UniqueHandle& operator=(UniqueHandle&& other) noexcept
    {
        if (this != &other)
        {
            reset();
            handle = other.handle;
            other.handle = INVALID_HANDLE_VALUE;
        }
```

```
        return *this;
    }

    HANDLE get() const noexcept
    {
        return handle;
    }

    void reset(HANDLE h = INVALID_HANDLE_VALUE) noexcept
    {
        if (handle != nullptr && handle != INVALID_HANDLE_VALUE)
            CloseHandle(handle);

        handle = h;
    }

private:
    HANDLE handle{INVALID_HANDLE_VALUE};
};
```

This is object-oriented design at its strongest: the type itself enforces resource correctness.

RAII and invariants

RAII is also about preserving class invariants. A well-designed RAII type should never exist in a partially valid state after successful construction.
For example, a scoped transaction guard or a lock wrapper can ensure that a system remains in a valid protocol state throughout object lifetime.

```
#include <iostream>
```

```
class Transaction
{
public:
    Transaction()
    {
        std::cout << "begin transaction\n";
    }

    ~Transaction()
    {
        if (!committed)
            std::cout << "rollback transaction\n";
    }

    void commit()
    {
        committed = true;
        std::cout << "commit transaction\n";
    }

private:
    bool committed{false};
};
```

The destructor acts as a correctness boundary. This is a profound architectural idea that continues to distinguish C++ from many other languages.

The architectural lesson

RAII is not optional decoration. In serious C++ systems, it is the backbone of reliability. It turns lifetime into a correctness mechanism and allows programmers to write code that remains safe in the presence of early returns, failures, and exceptions.

Prefer composition and static polymorphism

Another central lesson from modern object design is that inheritance is no longer the default mechanism for reuse. Composition and static polymorphism often provide clearer, safer, and more optimizable structures.

Why heavy inheritance became less attractive

Large inheritance hierarchies tend to create:

1. Tight coupling between base and derived classes
2. Fragile base-class effects
3. Difficult-to-track implicit behavior
4. Heap-heavy pointer-based designs
5. Indirection where a simpler structure would suffice

This does not make inheritance obsolete, but it means modern C++ uses it more selectively.

Composition as the primary assembly model

Composition builds a type from smaller parts instead of forcing semantic relationships through inheritance.

```
#include <iostream>
#include <string>

class Engine
{
public:
    void start() const
    {
        std::cout << "engine started\n";
    }
};

class Radio
{
public:
    void play() const
    {
        std::cout << "radio playing\n";
    }
};

class Car
{
public:
    void start_trip()
    {
        engine.start();
        radio.play();
    }
```

```
private:
    Engine engine;
    Radio radio;
};
```

This design is explicit and easy to extend. `Car` is not pretending to be an `Engine` or a `Radio`; it is built from them.

Static polymorphism and zero-overhead abstraction

When behavior variation is known at compile time, templates, concepts, and CRTP often replace runtime inheritance.

```
#include <concepts>
#include <iostream>
#include <string>

template<typename Sink>
concept MessageSink =
    requires(Sink s, const std::string& msg)
    {
        { s.write(msg) } -> std::same_as<void>;
    };

class ConsoleSink
{
public:
    void write(const std::string& msg)
    {
```

```
        std::cout << msg << '\n';
    }
};

template<MessageSink Sink>
class Reporter
{
public:
    explicit Reporter(Sink sink_)
        : sink(std::move(sink_))
    {
    }

    void report(const std::string& msg)
    {
        sink.write(msg);
    }

private:
    Sink sink;
};

int main()
{
    Reporter reporter(ConsoleSink{});
    reporter.report("static polymorphism");
}
```

This style provides several advantages:

1. No virtual dispatch

2. Strong compile-time checking
3. Inlining opportunities
4. Clearer ownership and layout

The architectural lesson

Modern C++ prefers composition for structural assembly and static polymorphism for compile-time behavior variation when runtime substitutability is not required. This is one of the clearest departures from older inheritance-heavy object-oriented design.

Balancing dynamic and static polymorphism

A mature Modern C++ library or application almost never chooses one form of polymorphism exclusively. Instead, it balances dynamic and static techniques according to architectural boundaries.

Dynamic polymorphism still matters

Dynamic polymorphism remains essential when:

1. Runtime substitution is required
2. An interface must remain open to independently developed implementations
3. Plugins or loadable modules are involved
4. Binary boundaries and stable abstract contracts matter

A classical example remains valid:

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

class ILogger
{
public:
    virtual ~ILogger() = default;
    virtual void write(const std::string& msg) = 0;
};

class ConsoleLogger : public ILogger
{
public:
    void write(const std::string& msg) override
    {
        std::cout << "[console] " << msg << '\n';
    }
};

class Application
{
public:
    explicit Application(std::unique_ptr<ILogger> logger_)
        : logger(std::move(logger_))
    {
    }

    void run()
```

```
    {
        logger->write("application started");
    }

private:
    std::unique_ptr<ILogger> logger;
};
```

This remains the right model when the implementation must be selected at runtime.

Static polymorphism remains stronger internally

Inside performance-sensitive subsystems, internal algorithms, views, small adapters, and policy-driven types, static polymorphism is frequently better.

```
#include <iostream>

struct FastPolicy
{
    static int compute(int x)
    {
        return x * 2 + 1;
    }
};

template<typename Policy>
class Engine
{
public:
    int run(int x) const
```

```
    {
        return Policy::compute(x);
    }
};

int main()
{
    Engine<FastPolicy> e;
    std::cout << e.run(10) << '\n';
}
```

A practical balance

A powerful architectural rule is:

1. Use dynamic polymorphism at runtime boundaries.
2. Use static polymorphism inside the implementation core.

For example, a library might expose a runtime plugin interface but internally use templates, policies, and variant-based dispatch to keep the core efficient and maintainable.

Variant-based middle ground

Between classical virtual dispatch and pure templates, `std::variant` provides an important middle ground for closed sets of alternatives.

```
#include <iostream>
#include <variant>

struct JsonParser
```

```
{
    void parse() const
    {
        std::cout << "parse JSON\n";
    }
};

struct IniParser
{
    void parse() const
    {
        std::cout << "parse INI\n";
    }
};

using Parser = std::variant<JsonParser, IniParser>;

int main()
{
    Parser parser = JsonParser{};

    std::visit([](const auto& p)
    {
        p.parse();
    }, parser);
}
```

This style is excellent when the set of possibilities is intentionally closed and operations are easier to add than new types.

The architectural lesson

Polymorphism in Modern C++ is no longer a single mechanism. It is a toolbox:

1. Virtual dispatch for open runtime interfaces
2. Templates and concepts for compile-time abstraction
3. `std::variant` for closed alternative sets
4. Type erasure for value-like runtime abstraction without intrusive inheritance

The future of object-oriented design in C++ depends on choosing among these precisely rather than treating inheritance as the automatic answer.

Designing future-proof C++ libraries

Future-proof library design does not mean predicting every future language feature. It means building APIs and internal structures that remain maintainable, extensible, and performant as the language and ecosystem evolve.

Make ownership explicit

A future-proof library should make it obvious:

1. Who owns a resource
2. Whether sharing is intended
3. Whether null is meaningful
4. Whether lifetime transfer occurs

This means using references, values, `std::unique_ptr`, and `std::shared_ptr` intentionally rather than interchangeably.

Keep interfaces narrow

Narrow interfaces are easier to preserve over time. Large interfaces become brittle and discourage internal evolution.

```
class ISerializer
{
public:
    virtual ~ISerializer() = default;
    virtual std::string serialize() const = 0;
};
```

A focused interface like this is easier to keep stable than a large kitchen-sink abstraction.

Prefer values when possible

Many libraries become easier to use and easier to evolve when they are value-oriented rather than heap-heavy and pointer-centric.

```
#include <string>

struct Endpoint
{
    std::string host;
    int port{};
};
```

A library built from strong value types often achieves:

1. Better locality
2. Clearer semantics

3. Simpler testing
4. Fewer lifetime bugs

Separate interface from implementation

For larger libraries, especially binary-delivered ones, internal representation should remain separate from the public contract. Pimpl remains relevant for this reason.

```
#include <memory>

class Widget
{
public:
    Widget();
    ~Widget();
    Widget(Widget&&) noexcept;
    Widget& operator=(Widget&&) noexcept;

    void draw();

private:
    struct Impl;
    std::unique_ptr<Impl> impl;
};
```

This allows internal change without exposing all details in public headers.

Design with modules in mind

C++20 modules encourage cleaner separation of public and private surfaces. A future-proof library should already think in terms of exported interfaces and hidden implementation units.

A simple module interface illustrates this direction:

```
export module mylib;

import <string>;

export class Greeter
{
public:
    explicit Greeter(std::string name_);
    std::string greet() const;

private:
    std::string name;
};
```

This style improves boundary clarity and reduces accidental exposure.

Avoid accidental ABI traps

Libraries meant for long-term distribution should be careful with:

1. Exposed object layouts
2. Inlined implementation-heavy interfaces
3. Exception type boundaries across binaries
4. Compiler- and standard-library-specific assumptions

A modern library should distinguish clearly between source-level design quality and binary-compatibility strategy.

Prefer extensibility through layers

A good library often separates:

1. Stable public API
2. Customization points
3. Internal optimized machinery

For example, a public runtime interface may coexist with internal concept-constrained algorithms and variant-based representations. This layered architecture is far more future-proof than one monolithic design.

The architectural lesson

Future-proof libraries are not rigid. They are explicit about semantics, careful about boundaries, narrow in their contracts, and flexible in their implementation strategy.

Where modern C++ architecture is heading

Modern C++ architecture is moving toward a model that combines safety, zero-overhead abstraction, modular boundaries, and more deliberate use of polymorphism.

Value-oriented design is growing

Even in strongly object-oriented systems, value types are becoming more central. Instead of treating every entity as a heap-allocated polymorphic object, modern systems increasingly use:

1. Value aggregates
2. Variant-based state

3. Immutable snapshots
4. Data-oriented component sets

This trend is especially visible in configuration systems, parsers, protocol layers, compilers, ECS-based engines, and high-performance backend services.

Static abstraction is expanding

Concepts, improved constexpr support, policy-based design, and template-driven APIs continue to push more correctness to compile time.
This does not mean that everything should become template-heavy. It means that more architectural choices can be encoded precisely without runtime cost.

Runtime abstraction is becoming more selective

Runtime polymorphism is not disappearing, but it is becoming more deliberate. Modern code increasingly reserves virtual dispatch and type erasure for places where runtime openness is genuinely needed:

1. Plugin systems
2. Stable service interfaces
3. Callback boundaries
4. User-extensible frameworks

Modules are reshaping library structure

Modules push C++ toward cleaner physical architecture:

1. More explicit public surfaces
2. Reduced preprocessor coupling
3. Better compilation scaling
4. More intentional dependency structure

As module adoption grows, library architecture is likely to become more disciplined and easier to scale across large code bases.

Concurrency-aware design is becoming standard

Modern libraries increasingly assume multi-threaded use. This pushes architecture toward:

1. Immutable shared data
2. Snapshot replacement
3. RAII locking discipline
4. Clear ownership transfer
5. Reduced shared mutable state

This trend reinforces the broader lesson that lifetime and ownership are architectural concerns, not merely local implementation details.

Hybrid architectures are the future

The strongest future C++ systems are likely to be hybrid rather than doctrinaire. They will combine:

1. Value-oriented internal data

2. RAII-based resource safety
3. Static polymorphism for internal flexibility
4. Dynamic polymorphism for extension boundaries
5. Modules for physical organization
6. Selective use of type erasure and variant-based design

This hybrid model is the real destination of modern C++ architecture.

Final reflection

Object-oriented design in Modern C++ has not ended. It has matured.
The language has moved beyond the era in which object orientation was treated mainly as inheritance plus virtual functions. Today, a stronger and more technically grounded understanding has emerged:

1. Ownership must be explicit.
2. Lifetime must be a correctness tool.
3. RAII must govern scarce resources.
4. Composition should be preferred over hierarchy when semantic substitution is not required.
5. Static and dynamic polymorphism should be balanced rather than opposed.
6. Libraries should be designed as layered systems with clear boundaries.

The future of object-oriented design in Modern C++ is therefore not a retreat from objects. It is a refinement of what objects should mean in a systems language.

A modern C++ object is not merely a bag of methods. It is a semantic unit that can express ownership, invariants, lifetime, policy, value, and abstraction boundary with precision.

That is why Modern C++ remains uniquely powerful. It does not force programmers into one model of software design. Instead, it allows them to combine object-oriented design, generic programming, RAII, value semantics, and systems-level control into one coherent engineering discipline.

This volume has shown that modern C++ object design is strongest when it is guided by clarity rather than ceremony, by semantics rather than habit, and by architectural intent rather than by inherited tradition.

That is also where the future lies.

Appendices

Appendix A The C++ Object Model and ABI Details

Introduction

The C++ object model is one of the most important foundations behind class design, virtual dispatch, inheritance, ABI compatibility, and binary interoperability. At the source-code level, the language intentionally abstracts away many low-level details. However, at the implementation level, every compiler must choose concrete rules for object layout, virtual table organization, pointer adjustment, calling conventions, RTTI attachment, exception metadata, and construction behavior.

For advanced library authors, framework engineers, compiler users, reverse engineers, and developers building binary-distributed C++ components, these details matter. They influence:

1. Binary compatibility across compilers and toolchains
2. DLL and shared-library boundaries
3. Performance of polymorphic dispatch
4. Debugging and memory inspection
5. Multiple inheritance and virtual inheritance behavior

6. Plugin and interface architecture choices

This appendix focuses on the practical object-model view relevant to Modern C++ programmers, especially when comparing the GCC-family ABI model commonly associated with the Itanium C++ ABI and the Microsoft Visual C++ ABI used by MSVC on Windows.

What an ABI means in C++

An Application Binary Interface defines how separately compiled machine-code fragments interoperate. In C++, the ABI governs far more than function calling alone. In object-oriented code, it also determines:

1. Name mangling
2. Class layout
3. Placement of virtual function pointers
4. Virtual table structure
5. Base-class offsets
6. RTTI organization
7. Exception-handling metadata
8. Construction and destruction rules for polymorphic objects

Two compilers may both accept the same conforming C++ source code and still produce incompatible binary layouts. This is why binary-level interoperability in C++ is toolchain-sensitive.

GCC ABI vs MSVC ABI

High-level difference

On most Unix-like systems, GCC and Clang commonly follow the Itanium C++ ABI for class layout, vtables, RTTI structure, and related object-model rules. On Windows, MSVC uses a different ABI model with its own conventions for object layout, virtual base handling, construction displacement support, and symbol representation.
This means that even when two compilers both implement the same language semantics, they may not agree on:

1. Exact vtable contents and ordering
2. Location of hidden pointers in objects
3. Representation of RTTI and related metadata
4. Base adjustment mechanisms for multiple inheritance
5. Virtual-base support fields such as MSVC `vtordisp` in relevant cases

As a consequence, C++ classes with virtual functions, inheritance, or nontrivial layout should not be assumed binary-compatible across GCC-family and MSVC toolchains.

The Itanium-style model in practice

The Itanium C++ ABI is not limited to the historical Itanium processor. It defines a portable C++ ABI model that has been widely used by GCC and Clang on many platforms. It gives explicit rules for:

1. Virtual table layout
2. RTTI association

3. Primary-base selection
4. Base-subobject placement
5. Adjustment of `this` during virtual calls
6. Pointer-to-member representation

This model is highly documented and especially valuable for studying class layout formally.

The MSVC model in practice

MSVC uses a distinct ABI on Windows. The compiler and debugger ecosystem expose concepts such as:

1. `vftable`
2. `vbtable`
3. `vtordisp`
4. construction and destruction displacement behavior

These are not just naming differences. They reflect real implementation choices in how MSVC handles virtual inheritance, base adjustment, and virtual calls during object construction and destruction.

One especially important practical detail is that MSVC object layout can be affected by `/vd` settings or `#pragma vtordisp`, and those settings can change the binary shape of relevant classes. Therefore, mixing object files built with inconsistent settings can create binary compatibility problems.

Practical consequence for library design

The most important engineering lesson is simple:

1. Do not expose compiler-dependent C++ object layouts across mixed-toolchain binary boundaries.
2. Avoid exporting complex polymorphic class hierarchies unless the producer and consumer use the same toolchain family and compatible settings.
3. Prefer C-style APIs, opaque handles, or narrow abstract interfaces at ABI-sensitive boundaries.

Fundamental object layout concepts

Before comparing ABI families in more detail, it is useful to review the common structural ideas implemented by all mainstream C++ compilers.

Data members

At the lowest level, an object stores its non-static data members. Subject to alignment and padding, these appear within the object storage itself.

```
struct Plain
{
    int    a;
    double b;
    char   c;
};
```

A conceptual layout might be:

```
+------------------------+
| int a                  |
+------------------------+
| padding                |
+------------------------+
| double b               |
+------------------------+
| char c                 |
+------------------------+
| tail padding           |
+------------------------+
```

The exact offsets depend on alignment rules for the target platform and ABI.

Base subobjects

When inheritance is used, the complete object contains one or more base-class subobjects plus any additional members from the most-derived type.

```
struct Base
{
    int x;
};

struct Derived : Base
{
    int y;
};
```

Conceptually:

```
Derived object
+------------------------+
| Base::x                |
+------------------------+
| Derived::y             |
+------------------------+
```

This is the simplest case of single inheritance.

Virtual function pointer

For polymorphic classes, the implementation typically stores a hidden pointer within the object that identifies the virtual dispatch structure associated with the dynamic type.
In informal terminology, programmers often call this hidden field the `vptr`. It points to some implementation-defined virtual table structure.

```
struct Poly
{
    virtual void f();
    int x;
};
```

A conceptual layout is often visualized as:

```
Poly object
+------------------------+
| hidden vptr            |
+------------------------+
| int x                  |
+------------------------+
| padding                |
+------------------------+
```

The standard does not require this exact layout, but it is the dominant implementation model.

Object layout diagrams

Single inheritance, non-polymorphic

```
struct A
{
    int a;
};

struct B : A
{
    int b;
};
```

Conceptual memory layout of `B`:

```
B complete object
+------------------------+
| A::a                   |
+------------------------+
| B::b                   |
+------------------------+
```

No hidden virtual-dispatch support is required because there are no virtual functions.

Single inheritance, polymorphic base

```
struct A
{
```

```
    virtual void f();
    int a;
};

struct B : A
{
    void f() override;
    int b;
};
```

Typical conceptual layout:

```
B complete object
+------------------------+
| hidden vptr            |
+------------------------+
| A::a                   |
+------------------------+
| B::b                   |
+------------------------+
| padding                |
+------------------------+
```

The `vptr` usually appears at the start of the primary polymorphic subobject, although the exact rules belong to the ABI and class-layout algorithm.

Multiple inheritance

```
struct Left
{
    virtual void lf();
```

```
    int l;
};

struct Right
{
    virtual void rf();
    int r;
};

struct Derived : Left, Right
{
    void lf() override;
    void rf() override;
    int d;
};
```

A conceptual layout may resemble:

```
Derived complete object
+------------------------+
| Left subobject         |
|   hidden vptr          |
|   Left::l              |
+------------------------+
| Right subobject        |
|   hidden vptr          |
|   Right::r             |
+------------------------+
| Derived::d             |
+------------------------+
```

```
| padding                |
+------------------------+
```

In multiple inheritance, there may be more than one virtual-dispatch-related pointer-bearing subobject, because each polymorphic base may need its own dispatch context.

Virtual inheritance

```
struct VBase
{
    virtual void vf();
    int v;
};

struct Left : virtual VBase
{
    int l;
};

struct Right : virtual VBase
{
    int r;
};

struct Most : Left, Right
{
    int m;
};
```

Conceptually, the complete object contains one shared `VBase` subobject:

```
Most complete object
+------------------------+
| Left-specific part     |
+------------------------+
| Right-specific part    |
+------------------------+
| Most::m                |
+------------------------+
| shared VBase subobject |
|   hidden vptr          |
|   VBase::v             |
+------------------------+
```

The real implementation is more complex because the compiler must support locating the shared virtual base correctly from different subobject viewpoints.

GCC ABI vs MSVC ABI in class layout terms

Primary-base oriented Itanium-style layout

The Itanium ABI defines formal rules for choosing a primary base and organizing the virtual table structure relative to that choice. In practice, this leads to a documented and systematic model for placing polymorphic base subobjects and arranging associated virtual-table entries. A major advantage of the Itanium ABI for study is that it describes the layout process explicitly, including concepts such as:

1. primary base
2. offset-to-top
3. RTTI pointer association

4. virtual call and virtual base offsets

When reading GCC or Clang-generated layouts on non-Windows platforms, this model is often a good mental framework.

MSVC and Windows-specific object-model features

MSVC uses different internal conventions. In practice, developers on Windows will encounter terminology such as:

1. `vftable` for a virtual-function table
2. `vbtable` for virtual-base navigation support
3. `vtordisp` for constructor/destructor-related displacement support in relevant virtual-inheritance cases

These mechanisms are particularly important in classes that involve virtual bases and overridden virtual functions called during construction or destruction.

Binary-compatibility caution

Even when source inheritance graphs look identical, the low-level representation may differ substantially between the Itanium-style and MSVC worlds. Therefore:

1. A class layout report from GCC or Clang should not be assumed to match MSVC.
2. Pointer adjustment code generated for one ABI should not be assumed to match another.
3. Exported polymorphic interfaces must be treated as ABI-sensitive design decisions.

Virtual dispatch internals

What virtual dispatch really does

A virtual call uses the dynamic type of the object, not merely the static type visible at the call site. To implement this, the compiler must:

1. Find the correct virtual-dispatch structure associated with the object or subobject
2. Select the correct slot for the virtual function
3. Possibly adjust `this` before entering the final function body

At a high level, a call such as:

```
Base* p = new Derived;
p->f();
```

conceptually behaves like:

```
1. read hidden dispatch pointer from object
2. locate slot corresponding to Base::f
3. fetch final function target for dynamic type
4. call that function, adjusting this if required
```

The exact machine sequence is ABI- and optimizer-dependent, but this is the core model.

Single-inheritance dispatch

In simple single inheritance, dispatch is conceptually straightforward because the base-subobject pointer often already points to the start of the relevant polymorphic object representation.
Conceptual sketch:

```
p --> [vptr][data...]
        |
        +--> vtable
             + slot 0 -> Derived::f
```

If `f()` does not require any special base adjustment, the call is direct through the table entry.

Multiple-inheritance dispatch

In multiple inheritance, a pointer to one base may not equal the start address of the most-derived object. That means a virtual call may need pointer adjustment.
Example:

```
struct Left
{
    virtual void f();
};

struct Right
{
    virtual void g();
};

struct Derived : Left, Right
{
    void f() override;
    void g() override;
};
```

A `Right*` pointing into a `Derived` object typically points at the `Right` subobject, not necessarily at byte offset zero of the complete object. Therefore, calling an override may require:

1. finding the correct table for the `Right` subobject view
2. adjusting `this` so that the called override receives the correct object pointer

Adjustor thunks

Compilers often use small helper entry points commonly called *adjustor thunks*. Their job is usually to adjust `this` and then transfer control to the real overriding function body. Conceptually:

```
Right* ----virtual call----> thunk
thunk:
    adjust this to Derived*
    jump to Derived::g
```

This is one of the central low-level mechanisms behind multiple-inheritance virtual dispatch.

Virtual-base dispatch complexity

Virtual inheritance adds more complexity because the location of the shared virtual base is not fixed as a simple compile-time constant from every subobject viewpoint. The generated code may need to consult ABI-defined offset structures in order to locate the correct shared base. This is one reason why virtual inheritance tends to be more expensive in implementation complexity and often in runtime cost than simple single inheritance.

Inside a virtual table conceptually

Although exact contents differ by ABI, a typical conceptual virtual table picture for a polymorphic class includes:

1. Information that allows navigation relative to the complete object

2. RTTI association
3. Function pointers or function-entry descriptors
4. In more complex hierarchies, auxiliary offsets related to base adjustment

A simplified teaching diagram:

```
virtual table (conceptual)
+----------------------------+
| metadata / offsets         |
+----------------------------+
| RTTI association           |
+----------------------------+
| slot for virtual f()       |
+----------------------------+
| slot for virtual g()       |
+----------------------------+
| slot for virtual h()       |
+----------------------------+
```

This is only a pedagogical model. Real layouts differ across ABI families.

Construction and destruction behavior

Why construction complicates dispatch

During construction and destruction, the object is not yet, or is no longer, in the fully formed most-derived state. As a result, compilers must carefully manage what virtual dispatch means during these phases.

At the language level, virtual calls in constructors and destructors dispatch according to the currently active subobject construction/destruction stage, not as though the object were already fully most-derived in every sense.

MSVC vtordisp relevance

In MSVC, the compiler may introduce hidden `vtordisp` fields in certain class layouts involving virtual bases and overridden virtual functions. These exist to support correct dispatch and adjustment during construction and destruction scenarios.
This has two practical consequences:

1. Object layout may change depending on compiler settings related to construction displacement support.
2. Mixing modules built with inconsistent settings can create ABI incompatibility.

This is a distinctly important point for Windows binary distribution.

RTTI and polymorphic objects

Runtime type information is tightly connected to polymorphic object models. In practice, RTTI support allows operations such as:

1. `dynamic_cast`
2. `typeid` on polymorphic glvalues

At the ABI level, this requires compiler-generated metadata associated with polymorphic class representations.
For programmers, the practical lesson is that RTTI is not free. It depends on real ABI structures and metadata layouts, and those are compiler-specific.

Object layout and empty base optimization

One important layout optimization is that empty base subobjects may often occupy no additional space beyond what is needed to preserve correct object identity rules.

```
struct Empty
{
};

struct X : Empty
{
    int value;
};
```

Conceptually, `X` may often be laid out approximately like:

```
X complete object
+------------------------+
| int value              |
+------------------------+
```

This is not something the programmer should assume blindly for every scenario, but it is a key optimization idea in modern class-layout implementations.

Why object layout diagrams are only conceptual

A critical professional rule is that hand-drawn object layout diagrams are teaching tools, not ABI contracts, unless they are derived directly from one specific compiler, target, and build configuration.
Actual layout can vary with:

1. target architecture
2. pointer size
3. alignment rules

4. compiler version
5. inheritance graph
6. virtual inheritance
7. compiler switches

Therefore, the diagrams in this appendix should be understood as structural explanations, not absolute byte-for-byte promises.

Practical Windows inspection techniques

On Windows with MSVC, one of the best practical ways to study real object layout is to inspect compiler-generated class layout reports.
A widely used command-line technique is:

```
cl /std:c++20 /EHsc /d1reportSingleClassLayoutMyClass example.cpp
```

or, for broader reporting:

```
cl /std:c++20 /EHsc /d1reportAllClassLayout example.cpp
```

These switches are intended for inspection and diagnostics and are extremely useful when learning MSVC object layout in practice.

Practical GCC and Clang inspection techniques

With GCC or Clang-family toolchains, practical ABI investigation often involves:

1. reading generated assembly
2. using symbol tools

3. checking `sizeof`, `alignof`, and pointer-difference experiments
4. consulting the Itanium ABI rules for interpretation

A simple exploratory program:

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

struct A
{
    virtual void f() {}
    int a{1};
};

struct B
{
    virtual void g() {}
    int b{2};
};

struct C : A, B
{
    int c{3};
};

int main()
{
    C obj;
```

```
    std::cout << "sizeof(A) = " << sizeof(A) << '\n';
    std::cout << "sizeof(B) = " << sizeof(B) << '\n';
    std::cout << "sizeof(C) = " << sizeof(C) << '\n';

    A* pa = &obj;
    B* pb = &obj;

    std::cout << "A* = " << static_cast<void*>(pa) << '\n';
    std::cout << "B* = " << static_cast<void*>(pb) << '\n';
}
```

This kind of experiment makes base-subobject offsets visible.

ABI-safe design recommendations

Do not expose unstable class layouts unnecessarily

Avoid exporting concrete polymorphic classes from libraries unless all participants are guaranteed to use the same compiler family, compatible settings, and compatible runtime model.

Prefer opaque handles or Pimpl for stable boundaries

If binary stability matters, prefer:

1. opaque pointer handles
2. C APIs
3. Pimpl
4. narrow abstract interfaces

These reduce dependence on compiler-specific layout rules.

Be careful with DLL boundaries on Windows

On Windows, especially with MSVC, configuration mismatches involving runtime library selection, exception model, and layout-affecting options can create subtle bugs even when the source code seems correct.

Use pure abstract interfaces cautiously

Pure abstract interfaces can help isolate implementation details, but they are still ABI-sensitive in practice if producer and consumer toolchains do not agree on the same object-model conventions.

Performance implications of virtual dispatch internals

Indirect call cost

Virtual dispatch introduces at least one layer of indirection. In hot paths, this can affect:

1. branch prediction
2. inlining opportunities
3. instruction-cache behavior

Multiple-inheritance adjustment cost

When `this` adjustment or thunks are required, the dispatch path may become more complex than the simple single-inheritance case.

Virtual inheritance cost

Virtual inheritance often increases both layout complexity and dispatch/navigation overhead because shared virtual-base locations must be found according to ABI-defined support

structures rather than simple fixed offsets.

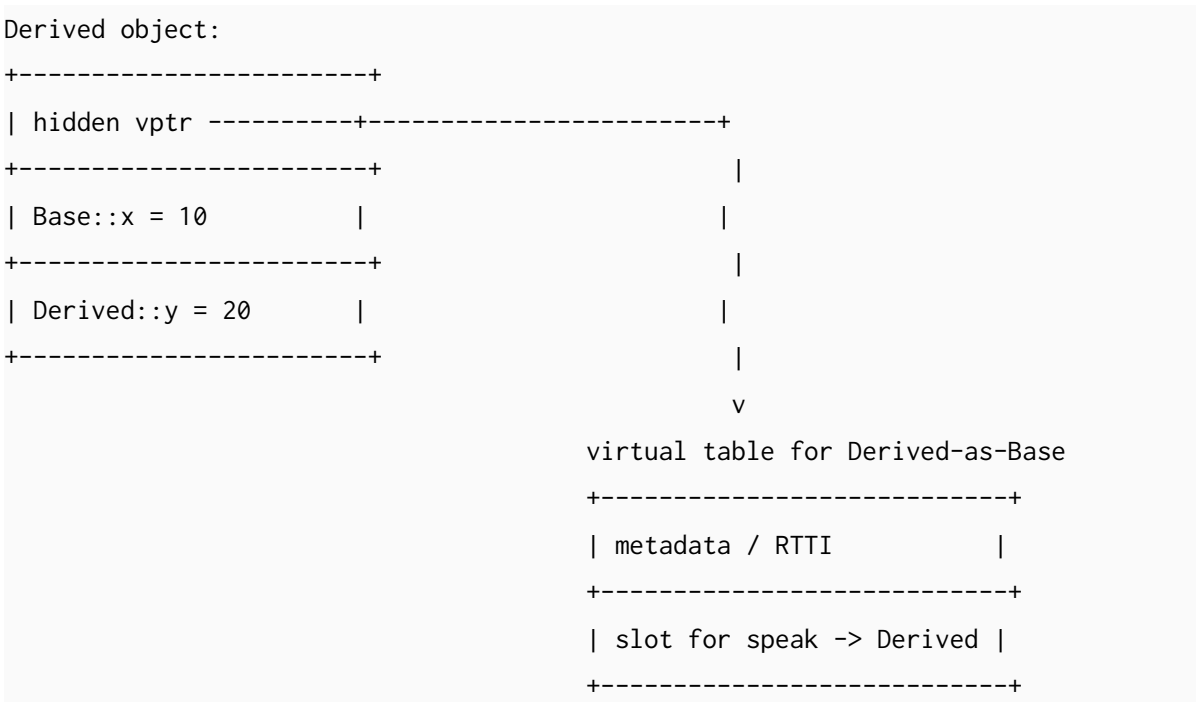
A worked conceptual example

Consider:

```
#include <iostream>

struct Base
{
    virtual void speak()
    {
        std::cout << "Base\n";
    }

    int x{10};
};

struct Derived : Base
{
    void speak() override
    {
        std::cout << "Derived\n";
    }

    int y{20};
};

int main()
{
```

```
    Derived d;
    Base* p = &d;
    p->speak();
}
```

Conceptual runtime picture:



When `p->speak()` executes, the program uses the dispatch pointer stored in the object and resolves the slot corresponding to `speak()`, which leads to the override in `Derived`.

The most important conclusions

The object model and ABI details of C++ lead to several essential conclusions for advanced engineering work:

1. Source-level inheritance does not uniquely determine binary layout; the ABI does.
2. GCC-family and MSVC-family binaries do not share one common C++ object ABI.
3. Virtual dispatch is implemented through concrete hidden structures, pointer adjustments, and sometimes thunks.
4. Multiple inheritance and virtual inheritance are valid language features but materially increase ABI and layout complexity.
5. Construction and destruction are special phases in polymorphic objects and can affect hidden layout support, especially on MSVC.
6. ABI-sensitive library design should minimize dependence on exposed compiler-specific class layouts.

Windows compilation guidance

For Visual Studio 2022, use a modern standard mode and warnings enabled:

```
cl /std:c++20 /EHsc /W4 object_model_demo.cpp
```

To inspect class layout with MSVC:

```
cl /std:c++20 /EHsc /d1reportAllClassLayout object_model_demo.cpp
```

For MinGW-w64 g++ on Windows:

```
g++ -std=c++20 -Wall -Wextra -pedantic object_model_demo.cpp -o
↪  object_model_demo.exe
```

Closing reflection

The C++ object model is one of the reasons C++ remains uniquely powerful and uniquely demanding. It allows high-level abstraction and low-level binary control to coexist, but that coexistence comes with responsibility. A strong Modern C++ programmer must understand not only how inheritance and virtual functions look in source code, but also how they map to object layout, dispatch mechanics, and ABI rules in the real machine-level world.
That understanding is what turns ordinary class design into professional systems-level engineering.

Appendix B Memory Alignment and Cache Behavior

Introduction

Modern C++ performance is deeply tied to how data is laid out in memory and how that memory is accessed by the CPU. While algorithmic complexity remains important, real-world performance is often dominated by cache behavior, memory alignment, and contention between threads.
Modern processors are built around hierarchical memory systems:

1. L1 cache (very fast, very small)
2. L2 cache (fast, larger)
3. L3 cache (shared, slower)
4. Main memory (DRAM, significantly slower)

The difference in latency between L1 cache and main memory can be orders of magnitude. Therefore, struct layout, alignment, and access patterns are critical for high-performance C++ systems.

Memory alignment fundamentals

What alignment means

Memory alignment refers to the requirement that objects are placed at addresses that are multiples of a specific value.

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

struct Example
{
    char  a;
    int   b;
    double c;
};

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

Each type has an alignment requirement:

1. `alignof(T)` gives the alignment requirement
2. The compiler inserts padding to satisfy alignment

Padding and its effect

Consider:

```
struct BadLayout
{
    char   a;
    double b;
    char   c;
};
```

Conceptual layout:

```
+----------------------+
| char a               |
+----------------------+
| padding              |
+----------------------+
| double b             |
+----------------------+
| char c               |
+----------------------+
| tail padding         |
+----------------------+
```

Reordering improves memory usage:

```
struct GoodLayout
{
    double b;
    char   a;
    char   c;
};
```

This reduces padding and improves cache density.

Explicit alignment control

C++ provides alignment control through `alignas`:

```
#include <iostream>
#include <new>

struct alignas(64) CacheAligned
{
    int data;
};

int main()
{
    std::cout << alignof(CacheAligned) << '\n';
}
```

This is especially useful when designing data structures for cache-line isolation.

Cache lines

What is a cache line

A cache line is the smallest unit of memory transferred between main memory and CPU cache.
On most modern systems, this is typically 64 bytes.
When a single byte is accessed, the entire cache line containing that byte is loaded into cache.

Spatial locality

Accessing nearby data is efficient because it resides in the same cache line.

```
#include <vector>

void process(const std::vector<int>& v)
{
    for (size_t i = 0; i < v.size(); ++i)
    {
        // sequential access is cache-friendly
        auto x = v[i];
        (void)x;
    }
}
```

Sequential traversal leverages spatial locality.

Temporal locality

Recently accessed data is likely to be accessed again soon.

```
int sum_twice(const int* data, size_t n)
{
    int result = 0;

    for (size_t i = 0; i < n; ++i)
        result += data[i];

    for (size_t i = 0; i < n; ++i)
        result += data[i];

    return result;
}
```

The second loop benefits from cached data.

False sharing

Definition

False sharing occurs when multiple threads modify different variables that reside on the same cache line. Even though the variables are logically independent, the hardware treats them as shared because they occupy the same cache line.
This causes cache coherence traffic and severe performance degradation.

Example of false sharing

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

struct Counter
{
    int value{0};
};

void increment(Counter& c)
{
    for (int i = 0; i < 1000000; ++i)
        ++c.value;
}

int main()
{
    Counter counters[2];
```

```
    std::thread t1(increment, std::ref(counters[0]));
    std::thread t2(increment, std::ref(counters[1]));

    t1.join();
    t2.join();

    std::cout << counters[0].value << " "
              << counters[1].value << '\n';
}
```

Even though each thread operates on a separate `Counter`, both objects may reside on the same cache line, causing contention.

Fix using cache-line alignment

```
#include <new>

struct alignas(64) PaddedCounter
{
    int value{0};
};
```

This ensures each counter resides on a separate cache line.

Using standard interference sizes

Modern C++ provides:

```
#include <new>

struct alignas(std::hardware_destructive_interference_size)
```

```
SafeCounter
{
    int value{0};
};
```

This expresses intent portably and adapts to platform-specific cache-line sizes.

Cache-friendly data layout

Array of Structures vs Structure of Arrays

Array of Structures (AoS):

```
struct Particle
{
    float x, y, z;
    float vx, vy, vz;
};

std::vector<Particle> particles;
```

Structure of Arrays (SoA):

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

SoA is often more cache-efficient for operations on individual fields:

```
for (size_t i = 0; i < p.x.size(); ++i)
{
    p.x[i] += p.vx[i];
}
```

This avoids loading unused fields.

Hot vs cold data separation

Separate frequently accessed data from rarely used data:

```
struct ColdData
{
    std::string debug_info;
    std::vector<int> logs;
};

struct HotData
{
    int x;
    int y;
    int z;
};

struct Entity
{
    HotData hot;
    ColdData* cold;
};
```

This improves cache utilization for performance-critical paths.

Minimizing pointer chasing

Pointer-heavy structures reduce cache efficiency:

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

Traversal leads to unpredictable memory access.
Flat structures improve locality:

```
#include <vector>

std::vector<int> values;
```

Sequential memory improves cache behavior.

Layout optimization strategies

Member reordering

Always group members by decreasing alignment:

```
struct Optimized
{
    double d;
    int    i;
    char   c;
};
```

This reduces padding and improves packing.

Avoiding excessive padding in arrays

Padding multiplies in arrays:

```
struct Padded
{
    char c;
    double d;
};
```

Each element may contain padding, increasing memory footprint.

Aligning for SIMD and vectorization

Alignment can help vectorized operations:

```
struct alignas(32) VecData
{
    float values[8];
};
```

This allows efficient SIMD loads on supported architectures.

Contiguous storage preference

Prefer containers with contiguous storage:

1. `std::vector`
2. `std::array`
3. `std::span`

Over:

1. linked structures
2. fragmented allocations

Threading and cache coherence

Modern CPUs maintain cache coherence across cores. When one core modifies a cache line, other cores must invalidate or update their copies.
This leads to:

1. coherence traffic
2. pipeline stalls
3. reduced scalability

False sharing amplifies these costs dramatically.

Performance measurement on Windows

Using high-resolution timing:

```
#include <chrono>
#include <iostream>

int main()
{
    auto start = std::chrono::high_resolution_clock::now();

    // workload

    auto end = std::chrono::high_resolution_clock::now();
```

```
    std::cout << std::chrono::duration_cast<std::chrono::microseconds>(end -
    ↪  start).count()
              << " us\n";
}
```

Compile with:

```
cl /std:c++20 /O2 /EHsc cache_test.cpp
```

Key guidelines

1. Prefer contiguous memory layouts.
2. Minimize padding through member ordering.
3. Avoid false sharing using alignment.
4. Separate hot and cold data.
5. Prefer SoA for compute-heavy workloads.
6. Reduce pointer indirection.
7. Use `alignas` and hardware interference constants when needed.
8. Measure performance; do not assume.

Final insight

Memory alignment and cache behavior are not micro-optimizations. They are core architectural concerns in Modern C++.
Understanding how data flows through cache hierarchies allows developers to:

1. design faster systems
2. scale across cores efficiently
3. reduce latency and contention
4. fully utilize modern CPU capabilities

In high-performance C++ systems, data layout is often more important than algorithmic elegance. The most efficient program is the one that respects how hardware actually executes it.

Appendix C Smart Pointer Internals

Introduction

Smart pointers are among the most important building blocks of Modern C++ object design. They encode ownership and lifetime into types, reduce manual memory management, and form the practical basis of RAII-oriented heap usage. Among them, `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr` represent three fundamentally different ownership models:

1. exclusive ownership
2. shared ownership
3. non-owning observation of shared ownership

For this appendix, the focus is on the internal design of shared-ownership smart pointers, especially `std::shared_ptr` and `std::weak_ptr`, because that is where the most interesting implementation machinery exists. In advanced systems work, understanding this machinery is essential for making correct decisions about performance, threading, cycles, and API boundaries.

The ownership model of smart pointers

unique_ptr

`std::unique_ptr` represents exclusive ownership. It usually contains just the managed pointer and, depending on deleter type, possibly deleter state. It does not use reference counting.

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

struct Widget
{
    ~Widget()
    {
        std::cout << "Widget destroyed\n";
    }
};

int main()
{
    std::unique_ptr<Widget> p = std::make_unique<Widget>();
}
```

This model is simple, efficient, and often the preferred default when sharing is unnecessary.

shared_ptr

`std::shared_ptr` represents shared ownership. Multiple `shared_ptr` instances may co-own the same object, and the object remains alive until the last owning instance releases it.

```
#include <iostream>
```

```
#include <memory>

struct Widget
{
    ~Widget()
    {
        std::cout << "Widget destroyed\n";
    }
};

int main()
{
    auto p1 = std::make_shared<Widget>();
    auto p2 = p1;
    auto p3 = p2;

    std::cout << "Three owners exist\n";
}
```

The resource is destroyed exactly once, when the last owner disappears.

weak_ptr

`std::weak_ptr` does not own the object. It observes a shared object without extending its lifetime and can be converted temporarily into a `shared_ptr` via `lock()` if the object still exists.

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

```
int main()
{
    std::weak_ptr<int> w;

    {
        auto p = std::make_shared<int>(42);
        w = p;

        if (auto s = w.lock())
            std::cout << *s << '\n';
    }

    if (w.expired())
        std::cout << "object no longer exists\n";
}
```

This split between owning and non-owning references is central to how Modern C++ expresses lifetime relationships safely.

The internal structure of shared ownership

Two logical parts

A `std::shared_ptr` is best understood as involving two logical components:

1. the stored pointer used for access
2. a shared control structure that tracks ownership and destruction rules

The standard specifies the semantics of shared ownership, but it does not require one exact memory layout for implementations. However, mainstream implementations consistently use a control-block model.

Conceptually:

```
shared_ptr object
+---------------------------+
| stored pointer            |
+---------------------------+
| pointer to control block  |
+---------------------------+

control block
+---------------------------+
| strong reference count    |
+---------------------------+
| weak reference count      |
+---------------------------+
| deleter / allocator info  |
+---------------------------+
| either managed object or  |
| pointer to managed object |
+---------------------------+
```

This model explains almost all important behavior of `shared_ptr` and `weak_ptr`.

Stored pointer versus owned object

A very important subtlety is that the pointer stored inside a `shared_ptr` need not be identical to the address of the object whose lifetime is governed by the control block. Aliasing constructors allow a `shared_ptr` to share ownership with another `shared_ptr` while storing a different pointer value.

```
#include <iostream>
```

```cpp
#include <memory>

struct Pair
{
    int x{10};
    int y{20};
};

int main()
{
    auto whole = std::make_shared<Pair>();

    std::shared_ptr<int> alias(whole, &whole->y);

    std::cout << *alias << '\n';
}
```

Here, `alias` stores a pointer to `whole->y`, but it shares the same ownership control block as `whole`. This is a fundamental reason why the control block and stored pointer must be conceptually distinguished.

Control blocks

Role of the control block

The control block is the heart of `shared_ptr`. Its responsibilities conceptually include:

1. tracking how many owning references exist
2. tracking how many weak observers exist
3. invoking the correct deleter when ownership reaches zero

4. preserving allocator-related state if needed
5. keeping the control structure alive until the last weak observer disappears

In practical terms, the control block outlives the managed object whenever weak observers remain after the last strong owner is gone.

Strong count and weak count

A useful conceptual model is:

1. the strong count counts owning `shared_ptr` instances
2. the weak count counts `weak_ptr` observers, plus usually an internal weak reference while strong ownership exists

The object is destroyed when the strong count reaches zero.
The control block itself is destroyed only when no strong or weak references remain that still require the control structure.

Conceptual control-block lifetime

```
initial state after make_shared:
  strong = 1
  weak   = 1 or implementation-equivalent internal baseline

copy shared_ptr:
  strong++

destroy shared_ptr:
  strong--
```

```
if strong becomes 0:
  destroy managed object
  release owning participation in control structure

destroy weak_ptr:
  weak--

if both ownership tracking conditions are now empty:
  destroy control block
```

Different implementations may organize the bookkeeping slightly differently internally, but this conceptual model is highly useful and matches standard semantics.

Separate allocation versus combined allocation

There are two broad implementation styles for shared ownership creation:

1. separate allocation for object and control block
2. combined allocation where object and control block are created together

Using `shared_ptr<T>(new T(...))` commonly leads to separate allocations.
Using `std::make_shared<T>(...)` typically allows one combined allocation for both control block and object.

```
auto p1 = std::shared_ptr<int>(new int(42));   // often two allocations
auto p2 = std::make_shared<int>(42);           // typically one allocation
```

This is one of the most important performance reasons to prefer `make_shared` when appropriate.

Control block and custom deleters

When a custom deleter is supplied, the control block must retain whatever state is necessary to invoke that deleter correctly.

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

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

int main()
{
    std::shared_ptr<FILE> fp(std::fopen("data.txt", "w"), FileCloser{});
}
```

The deleter is not stored inside the raw file pointer. It is part of the ownership machinery associated with the control block.

Control block and allocators

In advanced construction forms such as `allocate_shared`, allocator state may also become part of the control structure or its construction context.

```
#include <memory>
```

```
int main()
{
    std::allocator<int> alloc;
    auto p = std::allocate_shared<int>(alloc, 99);
}
```

This is another reason the control block is richer than a simple integer counter.

Reference counting algorithms

Basic counting idea

Reference counting is a lifetime-management algorithm based on incrementing a counter when ownership is copied and decrementing it when ownership ends. When the count reaches zero, the resource is destroyed.
A simple conceptual algorithm for strong ownership looks like this:

```
create control block:
    strong = 1

copy shared owner:
    ++strong

destroy shared owner:
    --strong
    if strong == 0:
        destroy managed object
```

However, `shared_ptr` must solve more than this. It also needs:

1. weak observers
2. thread-safe reference-count updates across distinct owners
3. proper behavior for aliasing and custom deleters
4. exception-safe acquisition and release logic

A toy reference-counted smart pointer

A minimal educational example helps illustrate the core idea, even though it omits many real-world requirements.

```
#include <iostream>

template<typename T>
class ToySharedPtr
{
public:
    explicit ToySharedPtr(T* p = nullptr)
        : ptr(p), count(p ? new int(1) : nullptr)
    {
    }

    ToySharedPtr(const ToySharedPtr& other)
        : ptr(other.ptr), count(other.count)
    {
        if (count)
            ++(*count);
    }
```

```
    ToySharedPtr& operator=(const ToySharedPtr& other)
    {
        if (this != &other)
        {
            release();
            ptr = other.ptr;
            count = other.count;
            if (count)
                ++(*count);
        }
        return *this;
    }

    ~ToySharedPtr()
    {
        release();
    }

    T* get() const
    {
        return ptr;
    }

    T& operator*() const
    {
        return *ptr;
    }

    T* operator->() const
```

```
    {
        return ptr;
    }

private:
    void release()
    {
        if (count)
        {
            --(*count);
            if (*count == 0)
            {
                delete ptr;
                delete count;
            }
        }
    }

    T* ptr{};
    int* count{};
};
```

This model teaches the essence of shared ownership, but it is not a real replacement for `std::shared_ptr` because it lacks:

1. weak references
2. thread-safe counting
3. custom deleters

4. allocator support
5. aliasing
6. exception guarantees comparable to the standard library

Why weak counting is needed

Without weak references, a control block could disappear as soon as the last strong owner is destroyed. But `weak_ptr` needs a way to observe that the object used to exist and determine safely whether ownership can still be reacquired.
Therefore the control structure must remain alive even after the object is gone, as long as weak references remain.

lock() and conditional reacquisition

The operation `weak_ptr::lock()` is conceptually a conditional attempt to create a new strong owner from a weak observer. This requires the implementation to test whether strong ownership is still nonzero and, if so, increment it safely.
Conceptually:

```
lock():
    if strong == 0:
        return empty shared_ptr
    else:
        atomically increment strong
        return new shared_ptr sharing control block
```

This is a key reason why the reference-count algorithm for `shared_ptr` is more subtle than plain increment/decrement logic.

Bad reacquisition pattern avoided by weak_ptr

```
#include <memory>

struct Node
{
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;
};
```

Using `weak_ptr` for back-links prevents the cycle from becoming permanently self-owning.

Thread-safety considerations

What is thread-safe and what is not

One of the most important lessons about `shared_ptr` is that its control-block ownership management is thread-safe across distinct smart-pointer objects that share ownership, but this does not mean the managed object itself becomes thread-safe automatically.
For example, copying and destroying separate `shared_ptr` instances that co-own the same object is designed to work safely across threads.
However, concurrent unsynchronized access to the pointee's mutable state is still an ordinary data-race problem.

```
#include <memory>
#include <thread>

struct Data
{
    int value{0};
};
```

```
int main()
{
    auto p = std::make_shared<Data>();
    auto q = p;

    std::thread t1([p]
    {
        // ownership operations are safe
        auto local = p;
        (void)local;
    });

    std::thread t2([q]
    {
        // also safe ownership operations
        auto local = q;
        (void)local;
    });

    t1.join();
    t2.join();
}
```

But this does *not* make the following safe:

```
#include <memory>
#include <thread>

struct Data
```

```
{
    int value{0};
};

int main()
{
    auto p = std::make_shared<Data>();

    std::thread t1([p]
    {
        ++p->value;
    });

    std::thread t2([p]
    {
        ++p->value;
    });

    t1.join();
    t2.join();
}
```

The ownership machinery is safe, but the increments on `value` race unless additional synchronization is used.

Distinct shared_ptr objects versus the same shared_ptr object

It is crucial to distinguish:

1. operations on different `shared_ptr` objects that share ownership

2. concurrent unsynchronized operations on the same `shared_ptr` object instance

Different smart-pointer instances can usually be copied, destroyed, and reset independently in a thread-safe ownership sense.
But one specific `shared_ptr` object, as an object with mutable state of its own, is subject to the ordinary rules for concurrent access unless atomic wrappers or external synchronization are used.

atomic<shared_ptr<T»

Modern C++ provides `std::atomic<shared_ptr<T»` for cases where the smart pointer object itself must be manipulated atomically.

```
#include <atomic>
#include <memory>

struct Config
{
    int version{};
};

int main()
{
    std::atomic<std::shared_ptr<Config>> current;
    current.store(std::make_shared<Config>(Config{1}));

    auto snapshot = current.load();
}
```

This is the correct direction for lock-free-style publication or replacement of a shared configuration snapshot.

Snapshot publication example

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

struct Config
{
    int port{};
};

std::atomic<std::shared_ptr<Config>> g_config;

void writer()
{
    g_config.store(std::make_shared<Config>(Config{8080}));
}

void reader()
{
    auto cfg = g_config.load();
    if (cfg)
        std::cout << cfg->port << '\n';
}

int main()
{
    g_config.store(std::make_shared<Config>(Config{80}));
```

```
    std::thread t1(writer);
    std::thread t2(reader);

    t1.join();
    t2.join();
}
```

This pattern is very useful in read-mostly systems.

atomic<weak_ptr<T»

Modern C++ also provides `std::atomic<weak_ptr<T»`. This is useful in more specialized designs where weak observation state itself must be published or exchanged atomically.

```
#include <atomic>
#include <memory>

struct Object
{
    int id{};
};

int main()
{
    auto p = std::make_shared<Object>(Object{7});

    std::atomic<std::weak_ptr<Object>> w;
    w.store(std::weak_ptr<Object>(p));

    auto observed = w.load();
```

```
    if (auto locked = observed.lock())
    {
        (void)locked;
    }
}
```

Reference-count operations and atomics

In mainstream implementations, reference-count changes for shared ownership are performed with atomic operations or equivalent synchronization-aware primitives so that lifetime management remains correct across threads. This is why `shared_ptr` is heavier than `unique_ptr` and why excessive shared ownership can have measurable performance cost.

Performance cost of thread-safe counting

Every shared-ownership copy may require an atomic increment, and every destruction may require an atomic decrement. This affects performance in hot paths, especially under heavy contention.
Therefore, a strong engineering rule is:

1. prefer `unique_ptr` unless shared ownership is genuinely required
2. use `shared_ptr` deliberately, not habitually

make_shared and allocation behavior

Why make_shared is usually preferred

`make_shared` typically improves performance and exception safety by constructing the object and the control machinery together.

```cpp
#include <memory>

struct Widget
{
    int x;
    explicit Widget(int value) : x(value) {}
};

int main()
{
    auto p = std::make_shared<Widget>(42);
}
```

Advantages commonly include:

1. fewer allocations
2. better locality
3. smaller allocation-management overhead
4. clearer construction style

When make_shared may be undesirable

Despite its benefits, there are cases where direct construction may still be chosen:

1. when a custom deleter is required
2. when object lifetime and control-block placement need different allocation strategies
3. when large objects should not remain co-located with a still-existing control block after object destruction concerns are considered in a design analysis

These cases are rarer but important in advanced systems work.

enable_shared_from_this internals

The problem it solves

Creating a new `shared_ptr` directly from `this` inside an object that is already managed by a different `shared_ptr` is a serious error, because it creates a second, unrelated ownership control block.

```
#include <memory>

struct Bad
{
    std::shared_ptr<Bad> dangerous()
    {
        return std::shared_ptr<Bad>(this); // wrong if already owned elsewhere
    }
};
```

This can lead to double deletion.

The standard solution

`std::enable_shared_from_this<T>` allows an object to safely produce a new `shared_ptr` that shares the existing control block.

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

struct Widget : std::enable_shared_from_this<Widget>
{
    std::shared_ptr<Widget> self()
```

```
    {
        return shared_from_this();
    }
};

int main()
{
    auto p = std::make_shared<Widget>();
    auto q = p->self();

    std::cout << (p == q) << '\n';
}
```

How it works conceptually

The object contains internal state that is linked to the first controlling `shared_ptr`. Later, `shared_from_this()` uses that state to create another owner sharing the same control block rather than inventing a new one.
Conceptually:

```
object derives from enable_shared_from_this
first owning shared_ptr initializes hidden weak association
shared_from_this() locks that association and returns shared owner
```

This makes `enable_shared_from_this` one of the most important smart-pointer support patterns in modern object-oriented C++.

Cycles and ownership graphs

The cycle problem

Reference counting alone cannot reclaim cycles. If two objects own each other through `shared_ptr`, their strong counts never reach zero.

```
#include <memory>

struct B;

struct A
{
    std::shared_ptr<B> b;
};

struct B
{
    std::shared_ptr<A> a;
};

int main()
{
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();

    a->b = b;
    b->a = a;
}
```

This leaks because the cycle is self-sustaining.

Breaking cycles with weak_ptr

```
#include <memory>

struct B;

struct A
{
    std::shared_ptr<B> b;
};

struct B
{
    std::weak_ptr<A> a;
};

int main()
{
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();

    a->b = b;
    b->a = a;
}
```

Now one direction is observational instead of owning, so the cycle can collapse naturally.

Architectural lesson

`shared_ptr` is not a general garbage collector. It is a deterministic shared-ownership mechanism with explicit cycle-breaking responsibility. That design is one of its strengths, but

also one of the most important responsibilities placed on the programmer.

A conceptual miniature control-block implementation

The following toy example is intentionally simplified but illustrates the shape of a real control-block design more closely than the earlier single-counter example.

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

template<typename T>
struct ToyControlBlock
{
    std::atomic<long> strong{1};
    std::atomic<long> weak{1};
    T* ptr{};

    explicit ToyControlBlock(T* p) : ptr(p) {}
};

template<typename T>
class ToySharedPtr;

template<typename T>
class ToyWeakPtr
{
public:
    ToyWeakPtr() = default;

    ToyWeakPtr(const ToySharedPtr<T>& sp)
```

```
        : cb(sp.cb)
    {
        if (cb)
            cb->weak.fetch_add(1, std::memory_order_relaxed);
    }

    ToyWeakPtr(const ToyWeakPtr& other)
        : cb(other.cb)
    {
        if (cb)
            cb->weak.fetch_add(1, std::memory_order_relaxed);
    }

    ~ToyWeakPtr()
    {
        release();
    }

private:
    void release()
    {
        if (!cb)
            return;

        if (cb->weak.fetch_sub(1, std::memory_order_acq_rel) == 1)
            delete cb;
    }

    ToyControlBlock<T>* cb{};
```

```
    friend class ToySharedPtr<T>;
};

template<typename T>
class ToySharedPtr
{
public:
    explicit ToySharedPtr(T* p = nullptr)
        : cb(p ? new ToyControlBlock<T>(p) : nullptr)
    {
    }

    ToySharedPtr(const ToySharedPtr& other)
        : cb(other.cb)
    {
        if (cb)
            cb->strong.fetch_add(1, std::memory_order_relaxed);
    }

    ~ToySharedPtr()
    {
        release();
    }

    T* get() const
    {
        return cb ? cb->ptr : nullptr;
    }
```

```
private:
    void release()
    {
        if (!cb)
            return;

        if (cb->strong.fetch_sub(1, std::memory_order_acq_rel) == 1)
        {
            delete cb->ptr;
            cb->ptr = nullptr;

            if (cb->weak.fetch_sub(1, std::memory_order_acq_rel) == 1)
                delete cb;
        }
    }

    ToyControlBlock<T>* cb{};

    friend class ToyWeakPtr<T>;
};
```

This remains only an educational sketch, but it makes visible the central ideas:

1. two counters
2. object destruction when strong ownership reaches zero
3. control-block destruction only after weak participation also ends

Design guidance for production code

Prefer unique_ptr by default

Use `std::unique_ptr` when exclusive ownership is natural. It is simpler, lighter, and often faster.

Use shared_ptr only for genuine shared lifetime

Do not use `shared_ptr` merely because it seems convenient. Shared ownership should express a real semantic requirement.

Use weak_ptr for back-links and observers

Whenever an ownership graph risks cycles or when observation without lifetime extension is intended, `weak_ptr` is usually the correct tool.

Prefer make_shared when suitable

`make_shared` is usually the best general construction form for ordinary shared ownership because of combined allocation and cleaner construction.

Do not confuse control-block thread safety with object thread safety

A `shared_ptr` can manage lifetime safely across threads, but it does not automatically serialize access to the pointee.

Use atomic<shared_ptr<T» for atomic publication of ownership state

When a shared pointer value itself is published or replaced concurrently, use the atomic smart-pointer facilities or external synchronization.

Windows build guidance

All examples in this appendix can be compiled on Windows with Visual Studio 2022. Recommended settings:

```
Project Properties
  C/C++ -> Language -> C++ Language Standard -> ISO C++20 Standard (/std:c++20)
  C/C++ -> Code Generation -> Enable C++ Exceptions -> Yes (/EHsc)
  C/C++ -> Warning Level -> Level4 (/W4)
  C/C++ -> Optimization -> Maximize Speed (/O2) for Release
```

MSVC command line:

```
cl /std:c++20 /EHsc /W4 smart_ptr_internals.cpp
```

If `atomic<shared_ptr<T>` is used, ensure the installed standard library and compiler mode support the required feature set in your toolchain.

Conclusion

The internals of Modern C++ smart pointers reveal several deep truths about the language and its design philosophy.

1. Smart pointers are not just convenience wrappers; they encode ownership and lifetime into the type system.
2. `shared_ptr` is powered by a control-block model that separates stored pointer access from shared lifetime management.
3. Reference counting is simple in principle but becomes significantly more sophisticated once weak observation, aliasing, custom deleters, and thread-safety are added.

4. Thread-safe ownership management does not imply thread-safe access to the managed object.

5. `weak_ptr` is essential both for safe observation and for breaking ownership cycles.

For advanced Modern C++ design, understanding these internals is more than academic knowledge. It directly shapes how libraries should express ownership, how APIs should cross threads, how object graphs should be modeled, and how reliable systems should manage lifetime without falling back into manual memory control.

Appendix D RAII Patterns Library

Introduction

Resource Acquisition Is Initialization (RAII) is one of the most fundamental and powerful design principles in C++. It binds resource lifetime to object lifetime and guarantees deterministic cleanup at scope exit. Modern C++ extends RAII far beyond memory management into a general-purpose pattern for expressing correctness, rollback semantics, and structured execution flow.
This appendix presents a practical library of RAII patterns widely used in modern systems:

1. Scope guards

2. Deferred execution

3. Transaction wrappers

These patterns form the backbone of robust exception-safe and maintainable systems.

Scope guards

Concept

A scope guard is an object that executes a user-provided action when leaving scope. This applies whether the scope exits normally or due to an exception.
This pattern is foundational in modern C++ and has been standardized in C++23 as `std::scope_exit`, `std::scope_fail`, and `std::scope_success`.

Basic custom scope guard

```
#include <utility>

template<typename F>
class ScopeExit
{
public:
    explicit ScopeExit(F&& f)
        : func(std::forward<F>(f)), active(true)
    {
    }

    ~ScopeExit()
    {
        if (active)
            func();
    }

    ScopeExit(const ScopeExit&) = delete;
    ScopeExit& operator=(const ScopeExit&) = delete;
```

```
    ScopeExit(ScopeExit&& other) noexcept
        : func(std::move(other.func)), active(other.active)
    {
        other.active = false;
    }

private:
    F func;
    bool active;
};
```

Usage:

```
#include <iostream>

int main()
{
    ScopeExit guard([]{
        std::cout << "scope exited\n";
    });

    std::cout << "inside scope\n";
}
```

C++23 standard scope guards

Modern C++ provides standardized scope guards:

```
#include <scope>
#include <iostream>
```

```
int main()
{
    auto guard = std::scope_exit([]{
        std::cout << "always executed\n";
    });

    auto on_fail = std::scope_fail([]{
        std::cout << "executed on exception\n";
    });

    auto on_success = std::scope_success([]{
        std::cout << "executed on success\n";
    });
}
```

These are extremely powerful for expressing intent:

1. `scope_exit`: always executes
2. `scope_fail`: executes only if stack unwinding occurs
3. `scope_success`: executes only if no exception occurs

Practical file example

```
#include <cstdio>
#include <scope>

void write_file()
{
```

```
    FILE* f = std::fopen("data.txt", "w");

    if (!f)
        return;

    auto close_file = std::scope_exit([&]{
        std::fclose(f);
    });

    std::fprintf(f, "Hello RAII\n");
}
```

This guarantees the file is closed regardless of control flow.

Deferred execution

Concept

Deferred execution is a generalization of scope guards. It schedules an action to be executed later, typically at scope exit, but with additional control such as cancellation or early execution.

Basic defer utility

```
#include <utility>

template<typename F>
class Defer
{
public:
    explicit Defer(F&& f)
        : func(std::forward<F>(f)), active(true)
```

```
    {
    }

    ~Defer()
    {
        if (active)
            func();
    }

    void cancel() noexcept
    {
        active = false;
    }

    void invoke_now()
    {
        if (active)
        {
            func();
            active = false;
        }
    }

private:
    F func;
    bool active;
};
```

Usage:

```
#include <iostream>
```

```
int main()
{
    Defer d([]{
        std::cout << "cleanup\n";
    });

    std::cout << "working...\n";

    // d.cancel(); // optional
}
```

Multiple deferred actions

```
#include <vector>
#include <functional>

class DeferredStack
{
public:
    void add(std::function<void()> f)
    {
        actions.push_back(std::move(f));
    }

    ~DeferredStack()
    {
        for (auto it = actions.rbegin(); it != actions.rend(); ++it)
            (*it)();
    }
```

```
private:
    std::vector<std::function<void()>> actions;
};
```

Usage:

```
#include <iostream>

int main()
{
    DeferredStack ds;

    ds.add([]{ std::cout << "cleanup 1\n"; });
    ds.add([]{ std::cout << "cleanup 2\n"; });

    std::cout << "main work\n";
}
```

This creates a LIFO cleanup sequence, similar to stack unwinding.

RAII and structured cleanup

Deferred execution is particularly useful for:

1. multi-step resource acquisition
2. rollback logic
3. partial initialization cleanup

```
#include <iostream>

void complex_operation()
{
    DeferredStack cleanup;

    cleanup.add([]{ std::cout << "release step 1\n"; });
    cleanup.add([]{ std::cout << "release step 2\n"; });

    std::cout << "processing...\n";
}
```

Transaction wrappers

Concept

A transaction wrapper encapsulates a sequence of operations that must either:

1. complete successfully and commit
2. fail and rollback automatically

RAII is ideal for expressing this pattern.

Basic transaction guard

```
#include <iostream>

class Transaction
{
public:
```

```
    Transaction()
    {
        std::cout << "begin\n";
    }

    ~Transaction()
    {
        if (!committed)
            rollback();
    }

    void commit()
    {
        committed = true;
        std::cout << "commit\n";
    }

private:
    void rollback()
    {
        std::cout << "rollback\n";
    }

    bool committed{false};
};
```

Usage:

```
void process()
{
```

```
    Transaction tx;

    // perform operations

    tx.commit();
}
```

If an exception occurs or commit is not called, rollback is triggered automatically.

File transaction example

```
#include <cstdio>
#include <string>

class FileTransaction
{
public:
    explicit FileTransaction(const std::string& name)
        : filename(name)
    {
        tempname = filename + ".tmp";
        file = std::fopen(tempname.c_str(), "w");
    }

    ~FileTransaction()
    {
        if (file)
            std::fclose(file);

        if (!committed)
```

```
            std::remove(tempname.c_str());
    }

    FILE* get() const
    {
        return file;
    }

    void commit()
    {
        committed = true;
        std::rename(tempname.c_str(), filename.c_str());
    }

private:
    std::string filename;
    std::string tempname;
    FILE* file{};
    bool committed{false};
};
```

Usage:

```
void write_safe()
{
    FileTransaction tx("output.txt");

    std::fprintf(tx.get(), "safe write\n");

    tx.commit();
```

```
}
```

This ensures atomic file replacement behavior.

Database-style transaction pattern

```
#include <iostream>

class DB
{
public:
    void begin() { std::cout << "db begin\n"; }
    void commit() { std::cout << "db commit\n"; }
    void rollback() { std::cout << "db rollback\n"; }
};

class DBTransaction
{
public:
    explicit DBTransaction(DB& db_) : db(db_)
    {
        db.begin();
    }

    ~DBTransaction()
    {
        if (!done)
            db.rollback();
    }
```

```
    void commit()
    {
        db.commit();
        done = true;
    }

private:
    DB& db;
    bool done{false};
};
```

Usage:

```
void run(DB& db)
{
    DBTransaction tx(db);

    // operations

    tx.commit();
}
```

Exception safety guarantees

Transaction wrappers naturally provide strong exception safety:

1. if all operations succeed commit
2. if any operation fails rollback automatically

This aligns with the strong guarantee:

1. either the operation completes successfully
2. or the system remains unchanged

Combining RAII patterns

These patterns can be composed to build complex systems:

```
#include <iostream>
#include <scope>

void combined()
{
    auto log = std::scope_exit([]{
        std::cout << "logging exit\n";
    });

    Transaction tx;

    // operations

    tx.commit();
}
```

RAII ensures:

1. cleanup happens in reverse order
2. invariants are preserved
3. failure paths are automatically handled

Design guidelines

1. Always bind resource lifetime to object lifetime.
2. Prefer stack-based RAII objects over manual cleanup.
3. Use scope guards for small cleanup actions.
4. Use deferred execution for complex staged cleanup.
5. Use transaction wrappers for multi-step operations.
6. Avoid raw resource management in business logic.
7. Ensure destructors never throw.

Windows build guidance

Compile with Visual Studio 2022:

```
cl /std:c++23 /EHsc /W4 raii_patterns.cpp
```

To use `<scope>`:

1. Enable C++23 standard
2. Ensure latest MSVC toolset is installed

Conclusion

RAII patterns form a reusable design toolkit for Modern C++ systems. They transform complex control flow, resource handling, and failure recovery into structured, deterministic, and maintainable constructs.

1. Scope guards provide localized cleanup.

2. Deferred execution enables structured rollback.

3. Transaction wrappers enforce correctness at system boundaries.

Together, these patterns elevate C++ from a language that merely allows resource management into one that enforces correctness through its type system and object lifetime semantics.

References

Reference Philosophy for This Volume

This volume is grounded in a layered reference strategy designed to balance normative precision, practical engineering judgment, historical context, and library-design depth. The references used throughout the study of Modern C++ object-oriented design, RAII, and polymorphism naturally fall into four major groups:

1. **Normative and near-normative standards material**: the ISO C++ standard texts and public working drafts that define the language and library rules.
2. **Authoritative language-design and expert books**: books written by core language designers and leading experts whose work shaped modern best practices.
3. **Living engineering guidance**: documents such as the C++ Core Guidelines, which continue to evolve with industry practice.
4. **Implementation and ABI documentation**: compiler and ABI references that explain how object layout, virtual dispatch, and binary compatibility work in real systems.

For a serious C++ programmer, no single source is sufficient. The standard defines what the language means. Expert books explain why certain styles work well. Guidelines help organize

engineering discipline. ABI documentation reveals how high-level source constructs are realized in low-level binaries.

ISO C++ Standard (C++20 / C++23 / C++26 drafts)

The ISO C++ standard is the primary normative reference for the language and standard library. It defines the formal semantics of object construction, destruction, inheritance, virtual dispatch, templates, concepts, concurrency primitives, smart pointers, the memory model, modules, and all standard-library facilities that influence object-oriented architecture.
For this volume, the standard should be understood in three layers:

C++20

C++20 represents one of the most important turning points in the history of the language. It introduced major facilities that directly affect architecture and object design:

1. Concepts and constrained templates
2. Modules
3. Coroutines
4. Expanded constexpr capabilities
5. Ranges
6. Calendar and time-zone library additions
7. New synchronization and concurrency support

For object-oriented design, C++20 is especially important because it made compile-time interfaces substantially more expressive and allowed many generic designs to become cleaner, more diagnosable, and more maintainable.

C++23

C++23 continued the refinement of modern design and usability. It strengthened the library ecosystem, improved ergonomics, and added facilities that support safer and clearer code. In practical design work, C++23 matters because it reduces incidental complexity and improves the expressiveness of standard-library-based solutions.
For example, the standardization of RAII-related tools such as scope-guard support reflects the continued movement of C++ toward explicit and structured lifetime-oriented design.

C++26 working-draft direction

The public C++26 drafts are not final standards, but they are critically important for understanding the direction of the language. Working drafts reveal which ideas are stabilizing, which abstractions are becoming central, and how the committee is evolving the language for future systems design.
For a volume focused on object-oriented design in Modern C++, draft materials are especially relevant when examining:

1. the evolution of pattern matching
2. new callable-wrapper families
3. reflection-related discussion directions
4. contracts-related design activity
5. refinement of modules, diagnostics, and library abstractions

How to use the standard effectively

The standard is not a tutorial text. It is a normative technical specification. Therefore, its best use is:

1. to confirm precise language rules
2. to verify subtle semantic points
3. to settle questions about guarantees, preconditions, and behavior
4. to distinguish what is guaranteed by the language from what is merely common implementation practice

A strong Modern C++ programmer should repeatedly consult the standard, but should also pair that consultation with practical engineering texts and compiler documentation.

Bjarne Stroustrup The C++ Programming Language

The C++ Programming Language by Bjarne Stroustrup remains one of the foundational texts in the entire C++ literature. It is uniquely important because it combines three qualities that few books can offer simultaneously:

1. it is written by the designer of the language
2. it explains both the philosophy and mechanics of C++
3. it connects language features to real engineering concerns

This book is especially valuable for understanding the broad intellectual model behind C++:

1. zero-overhead abstraction
2. direct support for user-defined types
3. deterministic lifetime
4. generic programming as a first-class design strategy

5. compatibility with low-level systems concerns

For this volume, the relevance of Stroustrup's work is especially strong in the following areas:

Object-oriented foundations

Stroustrup explains classes, constructors, destructors, inheritance, and virtual functions not as isolated syntax, but as a coherent type-system model for building abstractions. This is vital for understanding why C++ object orientation is different from more runtime-heavy object systems.

RAII and resource management

Few ideas are more central to Modern C++ than RAII, and Stroustrup's books consistently reinforce that resource management should be embedded into types rather than outsourced to manual discipline.

Balancing paradigms

A major strength of *The C++ Programming Language* is that it does not treat object orientation as the only design mode. It shows how object-oriented programming, generic programming, value-oriented design, and low-level efficiency coexist in one language.

Why this reference remains essential

Even when newer standards add more facilities, Stroustrup's work remains essential because it explains the design philosophy beneath the features. For deep study, the value of the book is not only in what features it lists, but in how it teaches the reader to think about C++ as a language of precise abstractions with explicit trade-offs.

Scott Meyers Effective Modern C++

Scott Meyers' *Effective Modern C++* occupies a distinctive place in the C++ literature. It is not a language specification and not a broad encyclopedia. Instead, it is a concentrated engineering guide that helps programmers avoid subtle mistakes and use Modern C++ features effectively. Its core strength is that it translates new language features into practical advice.

Why it matters for this volume

This volume deals heavily with ownership, move semantics, perfect forwarding, lambda expressions, smart pointers, and the practical consequences of modern language evolution. Meyers' book is one of the most influential references for those topics because it focuses on effective use rather than raw feature description.

Major contributions

The book is especially important in these areas:

1. understanding `auto` and type deduction
2. rvalue references and move semantics
3. universal references and forwarding
4. lambda design and capture strategy
5. `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`
6. the practical meaning of `decltype`
7. performance-sensitive modern idioms

Its enduring value

Although the book is centered on C++11 and C++14, its engineering lessons remain extremely valuable. Many of the best habits required for Modern C++ design were shaped by the discipline Meyers helped popularize:

1. prefer explicit ownership semantics
2. understand deduction precisely
3. distinguish move from copy intentionally
4. treat efficiency and correctness as connected concerns

For readers of this volume, *Effective Modern C++* is best treated as a sharpening text: it does not replace the standard or broader architecture books, but it dramatically improves the precision of day-to-day Modern C++ reasoning.

Herb Sutter Exceptional C++

Herb Sutter's *Exceptional C++* and its related works are landmarks in advanced C++ problem-solving. They are especially important because they approach C++ through carefully chosen puzzles, design scenarios, and engineering refinements rather than only through feature exposition.

Why Herb Sutter matters in Modern C++ design

Sutter has had enormous influence on the evolution of C++ best practices, especially in:

1. exception safety
2. generic design

3. correctness-oriented coding style
4. concurrency and memory-model awareness
5. modern guideline-driven engineering

Relevance to this volume

Exceptional C++ is deeply relevant to a book about OOP, RAII, and polymorphism because it repeatedly reinforces the principles that differentiate strong C++ design from superficial class-oriented programming:

1. resource safety must be structural
2. exception safety must be designed, not improvised
3. abstraction must not hide costs carelessly
4. interfaces must express semantics clearly

Its special strength

Many C++ books describe features. Sutter's work trains judgment. That makes it invaluable for advanced study. It helps the reader learn how to examine a design critically, identify subtle weaknesses, and choose stronger alternatives grounded in language semantics rather than habit.

Why it remains current in spirit

Even when some concrete examples are older, the design reasoning remains highly relevant to Modern C++ because the underlying questions have not disappeared:

1. When is copying safe?

2. When should ownership be explicit?
3. How should interfaces enforce invariants?
4. What guarantees should operations provide under failure?

Those questions remain central to serious library design.

Andrei Alexandrescu Modern C++ Design

Modern C++ Design by Andrei Alexandrescu is one of the most influential advanced C++ books ever written. It had a profound impact on generic library design, policy-based class construction, typelists, compile-time architecture, and the broader idea that sophisticated design patterns can be realized through templates rather than only through runtime inheritance.

Why this book matters

Even though the language has evolved significantly since its publication, the book remains historically and technically essential for understanding the movement from classical design patterns toward template-driven and policy-based architecture.

Major contributions relevant to this volume

The book is particularly important for understanding:

1. policy-based design
2. compile-time composition
3. generic programming as architecture
4. the relationship between design patterns and language power

5. advanced smart-pointer design ideas

Connection to modern object design

This volume repeatedly emphasizes that Modern C++ object-oriented design is no longer synonymous with inheritance-heavy runtime dispatch. Alexandrescu's work is one of the clearest historical demonstrations of that shift. It shows that many patterns once expressed through class hierarchies can instead be expressed through generic composition, policies, and type-level structure.

How to read it today

A modern reader should not treat *Modern C++ Design* as a direct style manual for present-day syntax. Rather, it should be read as a deep architectural text that anticipated many later developments:

1. compile-time polymorphism as a serious design tool
2. generic policy assembly
3. abstraction without mandatory runtime cost

In that sense, it remains one of the most intellectually important references for advanced C++ architecture.

Nicolai Josuttis C++ Move Semantics

Nicolai Josuttis' *C++ Move Semantics* is one of the most authoritative specialized references on one of the most important topics in Modern C++: value transfer, ownership transition, perfect forwarding, move-aware APIs, and the subtle interaction between copy elision, exception guarantees, and library design.

Why move semantics deserves a dedicated reference

Move semantics is not a minor optimization feature. It changed the architecture of Modern C++ libraries and value types. It affects:

1. class design
2. container performance
3. smart-pointer behavior
4. resource-transfer interfaces
5. generic forwarding utilities
6. exception-safe optimization strategies

A superficial treatment is never enough. Josuttis' work is valuable precisely because it goes far beyond superficial treatment.

Why this book matters for this volume

A large part of modern object design depends on understanding when a type should:

1. be movable
2. be copyable
3. suppress copying
4. default operations
5. customize move operations carefully

Josuttis' book is one of the best detailed references for those design questions.

Its special strength

The book is especially useful because it combines:

1. language semantics
2. library consequences
3. corner cases
4. practical examples
5. careful explanation of value categories and forwarding

This makes it particularly relevant for designing high-quality RAII types, containers, wrappers, and modern interfaces.

C++ Core Guidelines

The C++ Core Guidelines are one of the most important living references in the modern ecosystem. Unlike the standard, which defines what the language is, the guidelines focus on how the language should be used to produce code that is safer, more maintainable, and more robust.

Why the guidelines matter

The C++ Core Guidelines are especially important because they organize good practice across a very broad range of concerns:

1. interfaces
2. classes and class hierarchies

3. resource management
4. concurrency
5. generic programming
6. error handling
7. style and maintainability

Relevance to this volume

For a volume on Modern OOP, RAII, and polymorphism, the guidelines are central because they repeatedly reinforce the most important modern lessons:

1. express ownership explicitly
2. prefer RAII
3. avoid naked new and delete in ordinary code
4. keep interfaces narrow and intention-revealing
5. prefer composition where appropriate
6. use dynamic polymorphism only when runtime substitution is truly required
7. use static polymorphism and generic constraints where they improve correctness and efficiency

Why they are different from older style guides

Many older style guides focused on formatting or local coding habits. The C++ Core Guidelines are much broader. They are really an engineering framework. They connect language features, safety, performance, maintainability, and tool-based enforcement.

A practical way to use them

The guidelines are most valuable when used in three modes:

1. as a design checklist during architecture work
2. as a review framework during code inspection
3. as a basis for static-analysis enforcement where tools support them

For serious Modern C++ development, they are among the most useful companion references available.

Compiler ABI Documentation (GCC / Clang / MSVC)

No advanced study of Modern C++ object design is complete without implementation-level ABI documentation. The language standard intentionally does not define exact binary object layout, mangling, vtable layout, RTTI structure, or cross-object-file calling conventions. Those details belong to the ABI layer.

Why ABI references matter

ABI documentation becomes essential when dealing with:

1. object layout
2. virtual dispatch internals
3. multiple inheritance
4. virtual inheritance
5. binary compatibility

6. DLL or shared-library boundaries
7. plugin systems
8. low-level debugging

GCC and the Itanium C++ ABI

For GCC-family implementations, one of the most important references is the Itanium C++ ABI. Despite its name, it is not merely historical processor-specific material. It defines a widely used C++ ABI model and remains the central reference for the object-model behavior used by GCC and many Clang targets.
It is especially important for understanding:

1. vtable organization
2. RTTI placement
3. object layout rules
4. base-subobject adjustment
5. multiple- and virtual-inheritance dispatch mechanisms
6. exception-handling ABI structure

GCC ABI policy notes are also important because they explain how GCC aligns with and manages ABI expectations over time.

Clang and ABI behavior

Clang frequently targets the Itanium-style ABI on many open-source and Unix-like platforms, while also supporting Microsoft-compatible ABI modes on Windows. For that reason, Clang documentation is important not only as a compiler manual but also as a bridge between different ABI worlds.
For advanced developers, Clang is especially useful because it often makes ABI-target distinctions explicit in its documentation and internal references.

MSVC ABI documentation

On Windows, MSVC uses a different ABI model. Official Microsoft documentation is therefore essential for understanding:

1. multiple-base-class layout behavior
2. constructor and destructor displacement handling
3. `vtordisp`
4. the `/vd` compiler option
5. practical object-model consequences for polymorphic classes

This matters enormously for production library design on Windows because the binary shape of classes can be influenced by ABI-related implementation rules and compiler options.

Why ABI references are indispensable in this volume

This volume discusses advanced topics such as:

1. virtual dispatch internals

2. object layout diagrams
3. multiple inheritance
4. Pimpl and ABI stability
5. plugin architectures
6. binary-safe interface design

Those topics cannot be treated seriously from the standard alone. ABI documentation is the necessary bridge between high-level C++ semantics and actual compiled object representation.

How These References Work Together

The references listed in this appendix are not interchangeable. Each one has a different role.

The ISO standard and public working drafts

These define the language and library formally. They answer what the program means.

Stroustrup

This explains the language from the viewpoint of its design, philosophy, and broad engineering use.

Meyers

This sharpens everyday modern usage and highlights subtle correctness and performance issues.

Sutter

This develops advanced engineering judgment, especially in correctness, exception safety, and disciplined design.

Alexandrescu

This expands the reader's understanding of compile-time architecture and template-based design patterns.

Josuttis

This provides deep mastery of move semantics, value transfer, and move-aware interface design.

C++ Core Guidelines

These provide living, practical design discipline for writing robust Modern C++.

ABI documentation

This explains how source-level abstractions become real binary structures in compilers and runtimes.

Recommended Reading Order for Serious Study

For readers who want a structured path through these references, the following order is highly effective:

1. Start with Stroustrup for the broad model of the language.

2. Study Meyers for modern feature discipline.
3. Study the Core Guidelines alongside real code practice.
4. Read Sutter for design judgment and correctness thinking.
5. Read Josuttis for deep understanding of move semantics and ownership transfer.
6. Read Alexandrescu to expand into advanced compile-time architecture.
7. Consult the ISO standard and current working drafts for precise formal confirmation.
8. Consult ABI references whenever class layout, binary compatibility, or virtual dispatch details matter.

Final Reflection on the Reference Base

The reference base of Modern C++ is unusually rich because the language itself spans multiple engineering layers at once. C++ is simultaneously:

1. a language of high-level abstraction
2. a language of precise resource control
3. a language of generic compile-time architecture
4. a language with real binary and ABI consequences

That is why a serious study of Modern C++ cannot rely on one source alone.
The standard gives rigor.
Stroustrup gives philosophy and language intent.
Meyers gives disciplined modern usage.
Sutter gives engineering judgment.

Alexandrescu gives architectural depth in templates and policies.

Josuttis gives mastery of move semantics and value transfer.

The Core Guidelines give living best practice.

ABI documentation gives the binary truth beneath the abstractions.

Together, these references form a complete and trustworthy foundation for the study of Modern C++ object-oriented design, RAII, and polymorphism at a professional and enduring level.