Modern C++ Refresher Course

Comprehensive Reference - C++11 through C++23

Aung K. Soe Claude Sonnet 4.5 (AI Assistant)

January 3, 2026

Contents

1 Modern C++ Examples - Comprehensive Reference **7**
1.1 **Document Information** . . . 7
1.2 **About This Project** . . . 7
1.2.1 Key Features: . . . 7
1.3 **Quick Start** . . . 8
1.3.1 Prerequisites . . . 8
1.3.2 Build Instructions . . . 8
1.4 **Navigation & Documentation** . . . 8
1.4.1 Detailed Index . . . 8
1.4.2 Supporting Documentation . . . 8
1.4.3 Automatic Updates . . . 9
1.5 **Complete Examples - Alphabetical Reference** . . . 9
1.5.1 A . . . 9
1.5.2 B . . . 9
1.5.3 C . . . 9
1.5.4 D . . . 11
1.5.5 E . . . 11
1.5.6 F . . . 12
1.5.7 G . . . 13
1.5.8 I . . . 13
1.5.9 L . . . 13
1.5.10 M . . . 13
1.5.11 N . . . 14
1.5.12 O . . . 14
1.5.13 P . . . 14
1.5.14 R . . . 15
1.5.15 S . . . 16
1.5.16 T . . . 16
1.5.17 V . . . 17
1.5.18 **Interview Preparation** . . . 17
1.6 **Project Structure** . . . 18
1.7 **Finding Examples by Topic** . . . 18
1.7.1 By C++ Standard . . . 18

1.7.2 By Application Domain . . . 19
1.7.3 By Design Pattern . . . 19
1.8 **Testing & Running Examples** . . . 19
1.9 **Dependencies** . . . 19
1.10 **Contributing** . . . 20
1.11 **License** . . . 20
1.12 **Contact & Support** . . . 20
1.13 **Acknowledgments** . . . 20
1.14 **Additional Resources** . . . 20
1.14.1 Official Documentation . . . 20
1.14.2 Safety Standards . . . 20
1.14.3 Books . . . 21

2 Source Code: ARMInstructionSets.cpp 21

3 Source Code: AdvancedExceptionHandling.cpp 33

4 Source Code: AsioAndModernCppConcurrency.cpp 48

5 Source Code: AsioMultipleContexts.cpp 61

6 Source Code: BinarySearch.cpp 75

7 Source Code: CRTPvsVirtualFunctions.cpp 77

8 Source Code: CameraModule.cppm 92

9 Source Code: ConceptsExamples.cpp 100

10 Source Code: ConfigLoaderAndChecker.cpp 108

11 Source Code: Cpp11Examples.cpp 121

12 Source Code: Cpp14Examples.cpp 131

13 Source Code: Cpp17Concurrency.cpp 137

14 Source Code: Cpp17Examples.cpp 154

15 Source Code: Cpp20Examples.cpp 163

16 Source Code: Cpp23Examples.cpp 173

17 Source Code: CppWrappingCLibrary.cpp 188

18 Source Code: CreatingCApiFromCpp.cpp 202

19 Source Code: DependencyInjection.cpp 214

20 Source Code: DiamondProblem.cpp 225

21 Source Code: EigenSensorFusion.cpp 237

22 Source Code: EmbeddedSystemsAvoid.cpp 256

23 Source Code: EmbeddedSystemsProgramming.cpp 269

24 Source Code: ErrorHandling.cpp 284

25 Source Code: ErrorHandlingStroustrup.cpp 297

26 Source Code: EventDrivenProgramming_Inheritance.cpp 312

27 Source Code: EventDrivenProgramming_Lambdas.cpp 323

28 Source Code: ExceptionWithSourceLocation.cpp 334

29 Source Code: FindCountOfCommonNodes.cpp 349

30 Source Code: FindFirstCommonNode.cpp 352

31 Source Code: FindMToLastElement.cpp 355

32 Source Code: FindMaxNoOfConsecutiveOnesFromIntArray.cpp 358

33 Source Code: FragileBaseClass.cpp 360

34 Source Code: FunctionalSafetyISO26262.cpp 374

35 Source Code: FuturePromiseAsync.cpp 395

36 Source Code: GenericLambdas.cpp 408

37 Source Code: InheritanceTypes.cpp 414

38 Source Code: InsertAndDeleteNodes.cpp 425

39 Source Code: LambdaCaptures.cpp 433

40 Source Code: MISRACppDemo.cpp 443

41 Source Code: MockInterview.cpp 457

42 Source Code: MoveSemantics.cpp 474

43 Source Code: MoveSematicsPerfectForwarding.cpp 486

44 Source Code: MultiThreadedMicroservices.cpp 499

45 Source Code: NVIIdiomTemplateMethod.cpp 517

46 Source Code: NlohmannJsonExample.cpp 532

47 Source Code: NoexceptBestPractices.cpp 548

48 Source Code: ObjectSlicingCpp20.cpp **562**

49 Source Code: ObjectSlicingSmartPtr.cpp **571**

50 Source Code: OptionalExamples.cpp **579**

51 Source Code: PerfectForwardingAndRequires.cpp **587**

52 Source Code: PimplIdiom.cpp **599**

53 Source Code: ProtobufExample.cpp **619**

54 Source Code: Pybind11Example.cpp **635**

55 Source Code: ROMability.cpp **648**

56 Source Code: RangesExamples.cpp **661**

57 Source Code: RealTimeProgramming.cpp **669**

58 Source Code: ResourceLeaks.cpp **698**

59 Source Code: RestApiExample.cpp **719**

60 Source Code: RuleOf3_5_0.cpp **731**

61 Source Code: RuntimePolymorphism.cpp **749**

62 Source Code: SOLIDPrinciples.cpp **763**

63 Source Code: STLContainersNoHeap.cpp **777**

64 Source Code: SafetyCriticalSTLContainers.cpp **790**

65 Source Code: SearchAnagramsDictionary.cpp **806**

66 Source Code: SinglyLinkedList.cpp **808**

67 Source Code: StopTokenExample.cpp **811**

68 Source Code: StructuredBindings.cpp **826**

69 Source Code: SystemInteractionAndParsing.cpp **833**

70 Source Code: TemplatedCameraInterface.cpp **879**

71 Source Code: TemplatedCameraModules.cpp **891**

72 Source Code: ThreadPoolExamples.cpp **898**

73 Source Code: TuplesAndStructuredBindings.cpp **916**

74 Source Code: VariadicTemplateRecursion.cpp **927**

75 Source Code: VirtualFunctionsInTemplates.cpp 947

76 Appendix: Comprehensive Index 960

77 Modern C++ Examples - Comprehensive Index 960
77.1 Table of Contents . . . 960
77.2 Interview Preparation . . . 960
77.2.1 Complete C++ Interview Guide . . . 960
77.3 C++ Standards Features . . . 961
77.3.1 C++11 . . . 961
77.3.2 C++14 . . . 961
77.3.3 C++17 . . . 961
77.3.4 C++20 . . . 961
77.3.5 C++23 . . . 962
77.4 Design Patterns & Idioms . . . 962
77.4.1 CRTP (Curiously Recurring Template Pattern) . . . 962
77.4.2 Pimpl (Pointer to Implementation) . . . 962
77.4.3 NVI (Non-Virtual Interface) . . . 962
77.4.4 RAII (Resource Acquisition Is Initialization) . . . 962
77.4.5 Dependency Injection . . . 962
77.4.6 SOLID Principles . . . 963
77.4.7 Factory Pattern . . . 963
77.5 Object-Oriented Programming . . . 963
77.5.1 Inheritance . . . 963
77.5.2 Polymorphism . . . 963
77.5.3 Object Slicing . . . 963
77.6 Memory Management . . . 963
77.6.1 Smart Pointers . . . 963
77.6.2 Move Semantics . . . 964
77.6.3 Memory Pools & Allocators . . . 964
77.6.4 Resource Management . . . 964
77.7 Concurrency & Parallelism . . . 964
77.7.1 Threading . . . 964
77.7.2 Parallel Algorithms . . . 964
77.7.3 Future/Promise/Async . . . 964
77.7.4 Stop Tokens . . . 965
77.7.5 Microservices . . . 965
77.8 Real-Time & Embedded Systems . . . 965
77.8.1 Real-Time Programming . . . 965
77.8.2 Embedded Systems . . . 965
77.8.3 ARM & Architecture . . . 965
77.8.4 ROM Placement . . . 965
77.9 Safety-Critical & Standards . . . 965
77.9.1 MISRA C++ . . . 965
77.9.2 ISO 26262 (Automotive) . . . 965
77.9.3 STL for Safety-Critical . . . 966
77.9.4 AUTOSAR . . . 966
77.10 STL Containers & Algorithms . . . 966

77.10.1 Container Usage . . . 966
77.10.2 std::bitset . . . 966
77.10.3 std::list::splice() . . . 966
77.10.4 Algorithms . . . 966
77.11 Template Metaprogramming . . . 966
77.11.1 Templates & Concepts . . . 966
77.11.2 CRTP (Static Polymorphism) . . . 967
77.11.3 Variadic Templates . . . 967
77.11.4 Template Interface . . . 967
77.12 Error Handling . . . 967
77.12.1 Exceptions . . . 967
77.12.2 noexcept . . . 967
77.12.3 Error Codes & Optional . . . 967
77.13 Interoperability . . . 967
77.13.1 C Interop . . . 967
77.13.2 Python Binding . . . 968
77.13.3 Protobuf . . . 968
77.13.4 JSON . . . 968
77.13.5 REST API . . . 968
77.14 Performance & Optimization . . . 968
77.14.1 Performance Comparison . . . 968
77.14.2 Cache Locality . . . 968
77.14.3 Compile-Time Computation . . . 968
77.15 Common Problems & Solutions . . . 969
77.15.1 Fragile Base Class . . . 969
77.15.2 Diamond Problem . . . 969
77.15.3 Object Slicing . . . 969
77.15.4 Resource Leaks . . . 969
77.15.5 Virtual Functions in Templates . . . 969
77.16 Keywords & Language Features . . . 969
77.16.1 Keywords . . . 969
77.17 Event-Driven Programming . . . 970
77.17.1 Event Systems . . . 970
77.18 Lambda Expressions . . . 971
77.18.1 Lambda Basics . . . 971
77.19 Data Structures & Algorithms . . . 971
77.19.1 Linked Lists . . . 971
77.19.2 Search & Sort . . . 971
77.19.3 Array Algorithms . . . 971
77.20 Tuples & Structured Bindings . . . 971
77.20.1 Tuples . . . 971
77.20.2 Structured Bindings . . . 972
77.21 Configuration & Parsing . . . 972
77.21.1 Configuration . . . 972
77.21.2 System Interaction . . . 972
77.22 Sensor Fusion & Scientific Computing . . . 972
77.22.1 Eigen Library . . . 972
77.23 Quick Reference Tables . . . 972

77.23.1 By Language Version . . . 972
77.23.2 By Use Case . . . 972
77.23.3 By Problem You're Trying to Solve . . . 973
77.24 Additional Resources . . . 973
77.24.1 Documentation Files . . . 973
77.24.2 Build Scripts . . . 974
77.25 How to Use This Index . . . 974
77.26 Contributing . . . 974

1 Modern C++ Examples - Comprehensive Reference

A comprehensive collection of Modern C++ examples covering C++11 through C++23, design patterns, embedded systems, real-time programming, and safety-critical development.

1.1 Document Information

Version: 1.0.0
Last Updated: January 3, 2026
Authours: - **Aung K. Soe** - Primary Developer - **Claude Sonnet 4.5** - AI Assistant & Documentation Contributor

Repository: https://github.com/AungKyawSoe-Tech/ModernCppExamples

---

1.2 About This Project

This repository contains over 70 comprehensive C++ examples demonstrating modern C++ features, design patterns, best practices, and real-world applications. Each example is self-contained, well-documented, and includes practical use cases.

1.2.1 Key Features:

- **Complete C++ Standards Coverage**: Examples from C++11 through C++23
- **Safety-Critical Systems**: MISRA C++, AUTOSAR C++14, ISO 26262, DO-178C
- **Real-Time Programming**: Deterministic timing, WCET analysis, bounded memory
- **Design Patterns**: Gang of Four, CRTP, Pimpl, NVI, SOLID principles
- **Embedded Systems**: ARM instruction sets, ROMability, resource constraints
- **Modern Concurrency**: C++20 coroutines, thread pools, async/await
- **Interoperability**: Python bindings (pybind11), C API wrapping, Protocol Buffers
- **Performance**: Move semantics, perfect forwarding, template metaprogramming

---

1.3 Quick Start

1.3.1 Prerequisites

- C++20 or later compiler (GCC 10+, Clang 10+, MSVC 2019+)
- CMake 3.20 or later
- Optional: Python 3.8+ (for pybind11 examples)
- Optional: Protocol Buffers compiler (for protobuf examples)

1.3.2 Build Instructions

1.3.2.1 Linux/Mac:

```
# Clone the repository
git clone https://github.com/AungKyawSoe-Tech/ModernCppExamples.git
cd ModernCppExamples

# Build all examples
mkdir build && cd build
cmake ..
make -j$(nproc)

# Run a specific example
./bin/Cpp20Examples
```

1.3.2.2 Windows (PowerShell):

```
# Clone the repository
git clone https://github.com/AungKyawSoe-Tech/ModernCppExamples.git
cd ModernCppExamples

# Build all examples
mkdir build
cd build
cmake ..
cmake --build . --config Release

# Run a specific example
.\bin\Release\Cpp20Examples.exe
```

1.4 Navigation & Documentation

1.4.1 Detailed Index

For comprehensive topic mapping, concept lookup, and cross-references, see: - **INDEX.md** - Complete index mapping concepts, keywords, features, and problems to specific examples

1.4.2 Supporting Documentation

Additional detailed guides are available in the `MarkDownDocuments/` folder: - CPP11.md - C++11 features guide - CPP14.md - C++14 enhancements - CPP17.md - C++17 major features - CPP20.md - C++20 revolutionary features - CPP23.md - C++23 latest additions - EventDrivenProgramming.md - Event-driven architecture - MultiThreadedMicroservices.md - Microservices patterns -

Protobuf.md - Protocol Buffers integration - Pybind11.md - Python bindings - SECURITY.md - Security best practices

1.4.3 Automatic Updates

When you add new source files, the documentation automatically updates: - **README.md** - Alphabetical listing auto-updates (see AUTO_UPDATE_GUIDE.md) - **INDEX.md** - Template generated for manual topic categorization - **PDF** - Automatically includes all new files

See: AUTO_UPDATE_GUIDE.md for complete details on adding new examples.

1.5 Complete Examples - Alphabetical Reference

Below is a complete alphabetical listing of all C++ example files in this repository. Each entry links to the source file with a brief description of its contents.

1.5.1 A

1. **AdvancedExceptionHandling.cpp**
   - AdvancedExceptionHandling.cpp
   - Namespaces: std
   - Classes: DatabaseException, TracedException, StackTrace
2. **ARMInstructionSets.cpp**
   - compiled as Thumb
   - Functions: explain_alignment_issues, explain_interworking, explain_instruction_sets
3. **AsioAndModernCppConcurrency.cpp**
   - Note: This example demonstrates ASIO concepts using standard C++ features.
   - Namespaces: std
   - Classes: SimulatedAsioTimer, HybridServer, EventLoop
4. **AsioMultipleContexts.cpp**
   - …
   - Namespaces: SimulatedAsio, std
   - Classes: io_context, executor_work_guard, steady_timer

1.5.2 B

5. **BinarySearch.cpp**
   - Modern C++ example demonstrating various features
   - Classes: Solution
   - Functions: main

1.5.3 C

6. **CameraModule.cppm**
   - ==========
   - Classes: ImageProcessor, Image, Camera
7. **ConceptsExamples.cpp**
   - ==========

- Classes: Rectangle, Point2D, Circle
- Functions: print_sum, example_basic_concept, example_standard_concepts

8. **ConfigLoaderAndChecker.cpp**
   - ConfigLoaderAndChecker.cpp
   - Namespaces: std, fs
   - Classes: ConfigManager, ConfigMonitor
9. **Cpp11Examples.cpp**
   - ==========
   - Classes: Color, Traffic, MoveableResource
   - Functions: example_auto, foo
10. **Cpp14Examples.cpp**

- ==========
- Namespaces: std
- Functions: example_generic_lambdas, example_lambda_capture_initializers, example_binary_literals

11. **Cpp17Concurrency.cpp**

- ==========
- Namespaces: std
- Classes: AsyncFileProcessor, CancellableFuture, ThreadPool

12. **Cpp17Examples.cpp**

- ==========
- Namespaces: fs, A
- Functions: example_template_argument_deduction, example_structured_bindings, sum_fold

13. **Cpp20Examples.cpp**

- ==========
- Classes: Color
- Functions: example_concepts, example_designated_initializers, example_three_way_comparison

14. **Cpp23Examples.cpp**

- ==========
- Classes: GPIOPin, MemoryRegister, SensorError
- Functions: demonstrate_expected, demonstrate_byteswap

15. **CppWrappingCLibrary.cpp**

- ==========
- Classes: Socket, TcpServer, SocketError
- Functions: demonstrate_raii, demonstrate_nodiscard, demonstrate_noexcept

16. **CreatingCApiFromCpp.cpp**

- Create a new image
- Namespaces: image_processing
- Classes: Image

17. **CRTPvsVirtualFunctions.cpp**

- CRTP vs Virtual Functions: Static vs Dynamic Polymorphism
- Namespaces: static_functions, virtual_functions
- Classes: Dog, AnimalBase, Base

1.5.4 D

18. **DependencyInjection.cpp**

- ==========
- Classes: ConsoleLogger, ILogger, FileLogger
- Functions: example_traditional_di, example_constructor_injection, example_interface_injection

19. **DiamondProblem.cpp**

- DiamondProblem.cpp
- Namespaces: virtual_inheritance_solution, diamond_problem
- Classes: InputDevice, OutputDevice, Device

1.5.5 E

20. **EigenSensorFusion.cpp**

- EXPECTED OUTPUT:
- Namespaces: Eigen
- Classes: ComplementaryFilter, ParticleFilter, KalmanFilter

21. **EmbeddedSystemsAvoid.cpp**

- ==========
- Classes: BaseSensor, TempSensor, SensorBase
- Functions: demonstrate_vector_problems, demonstrate_string_problems, demonstrate_heap_fragmentation

22. **EmbeddedSystemsProgramming.cpp**

- ==========
- Namespaces: BitOps
- Classes: MemoryPool, PinMode, CircularBuffer

23. **ErrorHandling.cpp**

- ==========
- Classes: runtime_error, CameraException, CameraNotConnectedException
- Functions: demonstrate_custom_exceptions, demonstrate_basic_exception_handling, demonstrate_exception_hierarchy

24. **ErrorHandlingStroustrup.cpp**

- ==========
- Classes: FileHandler_GOOD, FileHandler_BAD, ConfigParser
- Functions: demonstrate_return_values, demonstrate_exceptions, demonstrate_termination

25. **EventDrivenProgramming_Inheritance.cpp**

- ==========

- Classes: IObserver, ConsoleObserver, CountingObserver
- Functions: example_inheritance_observer, example_listener_pattern, example_event_hierarchy

26. **EventDrivenProgramming_Lambdas.cpp**

- ==========
- Classes: Subject, Signal, Button
- Functions: example_variant_dispatcher, example_signal_slot, example_lambda_observer

27. **ExceptionWithSourceLocation.cpp**

- ==========
- Classes: DetailedException, SourceLocationException, BaseLocationException
- Functions: demonstrate_detailed_location, demonstrate_basic_location, some_function_that_fails

1.5.6 F

28. **FindCountOfCommonNodes.cpp**

- Modern C++ example demonstrating various features
- Namespaces: std
- Classes: Solution

29. **FindFirstCommonNode.cpp**

- Modern C++ example demonstrating various features
- Namespaces: std
- Classes: Solution

30. **FindMaxNoOfConsecutiveOnesFromIntArray.cpp**

- Instructions:
- Functions: main

31. **FindMToLastElement.cpp**

- Modern C++ example demonstrating various features
- Namespaces: std
- Classes: Solution

32. **FragileBaseClass.cpp**

- FragileBaseClass.cpp
- Namespaces: composition_solution, fragile_example
- Classes: Counter_V1, LoggingCounter_V1, Counter_V2

33. **FunctionalSafetyISO26262.cpp**

- FunctionalSafetyISO26262.cpp
- Classes: SafetyResult, SafetyState, ASILLevel
- Functions: demonstrate_safe_data_types, demonstrate_redundancy_voting, demonstrate_asil_levels

34. **FuturePromiseAsync.cpp**

- FuturePromiseAsync.cpp

- Namespaces: std, async_examples
- Classes: RequestProcessor, SimpleThreadPool

1.5.7 G

35. **GenericLambdas.cpp**

- ==========
- Functions: example_basic_generic_lambda, example_generic_lambda_operations, example_generic_lambda_containers

1.5.8 I

36. **InheritanceTypes.cpp**

- ==========
- Classes: Animal, Dog, Timer
- Functions: example_public_inheritance, example_private_inheritance, example_protected_inheritance

37. **InsertAndDeleteNodes.cpp**

- PrintListNode(pInHead);
- Namespaces: std
- Classes: Solution

1.5.9 L

38. **LambdaCaptures.cpp**

- ==========
- Classes: ExampleClass, ExampleClass2, Button
- Functions: example_capture_all_by_reference, example_no_capture, example_capture_all_by_value

1.5.10 M

39. **MISRACppDemo.cpp**

- ==========
- Namespaces: expressions_and_operators, types_and_declarations
- Classes: Resource, GoodBase, BadBase

40. **MoveSemantics.cpp**

- ==========
- Classes: MoveOnlyType, LargeObject, Resource
- Functions: demonstrate_copy_vs_move, demonstrate_std_move, demonstrate_lvalue_rvalue

41. **MoveSematicsPerfectForwarding.cpp**

- ==========
- Namespaces: lvalues_rvalues, move_semantics
- Classes: Resource, Widget

42. **MultiThreadedMicroservices.cpp**

- MultiThreadedMicroservices.cpp
- Namespaces: std
- Classes: ThreadType, ThreadContext, Logger

1.5.11 N

42. **NlohmannJsonExample.cpp**

- ==========
- Functions: example_basic_json, example_arrays, example_custom_types

42. **NoexceptBestPractices.cpp**

- ==========
- Namespaces: always_use_noexcept, what_is_noexcept
- Classes: Resource, Swappable, MoveableResource

42. **NVIIdiomTemplateMethod.cpp**

- NVIIdiomTemplateMethod.cpp
- Namespaces: std, basic_nvi
- Classes: CSVProcessor, JSONProcessor, DataProcessor

1.5.12 O

42. **ObjectSlicingCpp20.cpp**

- ==========
- Classes: Shape, Rectangle, Circle
- Functions: draw_shape, process_shape, example_concepts_prevent_slicing

42. **ObjectSlicingSmartPtr.cpp**

- ==========
- Classes: Shape, Rectangle, Circle
- Functions: example_shared_ptr_slicing_wrong, example_container_slicing_wrong, example_classic_slicing

42. **OptionalExamples.cpp**

- ==========
- Functions: example_basic_optional, example_optional_return, example_value_or

1.5.13 P

42. **PerfectForwardingAndRequires.cpp**

- tests
- Classes: ThreadSafeQueue, Serializable, Register
- Functions: demonstrate_perfect_forwarding, call_with_forward, call_by_value

42. **PimplIdiom.cpp**

- ==========
- Namespaces: what_is_pimpl, modern_cpp_pimpl

- Classes: WidgetCpp98, WidgetNoPimpl, WidgetWithPimpl

42. **ProtobufExample.cpp**

- ==========
- Functions: example_binary_deserialization, example_binary_serialization, example_create_sensor_reading

42. **Pybind11Example.cpp**

- PYTHON USAGE EXAMPLES:
- Namespaces: py
- Classes: Vector2D, Shape, Circle

1.5.14 R

42. **RangesExamples.cpp**

- ==========
- Namespaces: views, ranges
- Functions: example_filter_view, example_basic_ranges, example_transform_view

42. **RealTimeProgramming.cpp**

- RealTimeProgramming.cpp
- Namespaces: big_o_notation, thread_architecture
- Classes: RealTimeFlagManager, SensorSystem, SensorStatus

42. **ResourceLeaks.cpp**

- EXPECTED OUTPUT HIGHLIGHTS:
- Namespaces: ModernCpp, OldCpp
- Classes: Node, ExpensiveResource, ResourceIntensive

42. **RestApiExample.cpp**

- ==========
- Classes: CurlHandle, CurlHeaders, RestClient
- Functions: example_create_post, example_update_post, example_json_placeholder_api

42. **ROMability.cpp**

- ROMability.cpp
- Namespaces: const_vs_constexpr, rom_arrays
- Functions: demonstrate, get_runtime_value

42. **RuleOf3_5_0.cpp**

- no-op
- Classes: RuleOfFiveExample, RuleOfThreeExample, RuleOfZeroExample
- Functions: example_rule_of_zero, example_rule_of_five, example_rule_of_three

42. **RuntimePolymorphism.cpp**

- ==========
- Classes: IShape, Rectangle, Circle

- Functions: example_virtual_destructor, example_virtual_with_default, example_pure_virtual

1.5.15 S

42. **SafetyCriticalSTLContainers.cpp**

- ==========
- Namespaces: container_classification, containers_to_avoid
- Classes: SafetyPoolAllocator

42. **SearchAnagramsDictionary.cpp**

- Example program
- Functions: main

42. **SinglyLinkedList.cpp**

- pHead
- Functions: print, main

42. **SOLIDPrinciples.cpp**

- ==========
- Namespaces: open_closed, single_responsibility
- Classes: Employee, BadEmployee, TaxCalculator

42. **STLContainersNoHeap.cpp**

- ==========
- Classes: PoolAllocator, FixedVector, StackAllocator
- Functions: example_std_array, example_std_span, process_data

42. **StopTokenExample.cpp**

- StopTokenExample.cpp
- Namespaces: std, old_vs_new
- Classes: RESTService, OldStyleThread, NewStyleThread

42. **StructuredBindings.cpp**

- ==========
- Functions: example_array_bindings, example_tuple_bindings, example_basic_structured_bindings

42. **SystemInteractionAndParsing.cpp**

- ==========
- Namespaces: fs
- Classes: BankAccount, ThreadSafeLogger, SharedCounter

1.5.16 T

42. **TemplatedCameraInterface.cpp**

- ==========
- Classes: Camera8bit, Image, Camera
- Functions: demonstrate_16bit_camera, demonstrate_float_camera, demonstrate_8bit_camera

42. **TemplatedCameraModules.cpp**

- ==========
- Functions: demonstrate_camera_capture, demonstrate_concepts, display_camera_info

42. **ThreadPoolExamples.cpp**

- ThreadPoolExamples.cpp
- Namespaces: std
- Classes: BasicThreadPool, TaskPriority, ThreadPoolWithFutures

42. **TuplesAndStructuredBindings.cpp**

- ==========
- Functions: demonstrate_structured_bindings, demonstrate_basic_tuples, demonstrate_tie

1.5.17 V

42. **VariadicTemplateRecursion.cpp**

- ==========
- Classes: GPIO_Pin
- Functions: demonstrate_basic_recursion, print_recursive

42. **VirtualFunctionsInTemplates.cpp**

- VIRTUAL FUNCTIONS IN TEMPLATES: THE CODE BLOAT PROBLEM
- Namespaces: the_problem, bad_example_t80
- Classes: template, Vector, DemoInstantiation

1.5.18 Interview Preparation

75. **MockInterview.cpp**

- **Complete C++ Interview Questions & Answers** - Comprehensive mock interview preparation guide
- **Topics Covered:**
  - Section 1: C++ Fundamentals (pointers vs references, const correctness, RAII)
  - Section 2: Memory Management (smart pointers, memory alignment, custom allocators)
  - Section 3: OOP & Design Patterns (virtual dispatch, CRTP, PIMPL)
  - Section 4: Templates & Metaprogramming (SFINAE, variadic templates, compile-time computation)
  - Section 5: Concurrency & Multithreading (singleton, producer-consumer, atomic operations)
  - Section 6: Performance & Optimization (move semantics, RVO/NRVO, copy elision)
- **Key Demonstrations:** 25+ interview questions with detailed explanations and runnable code
- **Namespaces:** std
- **Classes:** FileHandler, Data, Widget, ThreadSafeSingleton, ProducerConsumer, LargeObject, Expensive
- **Functions:** pointers_vs_references, const_correctness, raii_demo, smart_pointers_advanced, memory_alignment, virtual_dispatch_details, crtp_mixins_demo, sfinae_demo, vari-

adic_demo, compile_time_demo, singleton_demo, producer_consumer_demo, atomic_operations_demo, move_semantics_performance, copy_elision_demo

---

1.6 Project Structure

```
ModernCppExamples/
 src/                        # All C++ source files (.cpp, .cppm)
    Cpp11Examples.cpp       # C++11 features
    Cpp20Examples.cpp       # C++20 features
    RealTimeProgramming.cpp
    SafetyCriticalSTLContainers.cpp
    PimplIdiom.cpp
    ... (70+ examples)

 MarkDownDocuments/          # Detailed documentation
    CPP20.md
    EventDrivenProgramming.md
    MultiThreadedMicroservices.md
    Protobuf.md
    Pybind11.md
    SECURITY.md
    ...

 scripts/                    # Build and utility scripts
    build.sh                # Main build script
    build_modules.sh        # C++20 modules build
    build_protobuf.sh       # Protocol Buffers build
    build_pybind.sh         # Pybind11 build
    run_all.sh              # Run all examples

 proto/                      # Protocol Buffer definitions
    sensor_data.proto

 build/                      # Build output directory
    bin/                    # Compiled executables

 CMakeLists.txt              # CMake configuration
 INDEX.md                    # Comprehensive topic index
 README.md                   # This file
 config.json                 # Configuration file
```

1.7 Finding Examples by Topic

For detailed topic mapping, see INDEX.md. Quick topic guide:

1.7.1 By C++ Standard

- **C++11**: Cpp11Examples.cpp, LambdaCaptures.cpp, MoveSemantics.cpp
- **C++14**: Cpp14Examples.cpp, GenericLambdas.cpp

- **C++17**: Cpp17Examples.cpp, StructuredBindings.cpp, OptionalExamples.cpp
- **C++20**: Cpp20Examples.cpp, ConceptsExamples.cpp, RangesExamples.cpp, CameraModule.cppm
- **C++23**: Cpp23Examples.cpp

1.7.2 By Application Domain

- **Embedded Systems**: EmbeddedSystemsProgramming.cpp, ROMability.cpp, ARMInstructionSets.cpp
- **Real-Time Systems**: RealTimeProgramming.cpp, STLContainersNoHeap.cpp
- **Safety-Critical**: SafetyCriticalSTLContainers.cpp, FunctionalSafetyISO26262.cpp, MISRACppDemo.cpp
- **Concurrency**: Cpp17Concurrency.cpp, FuturePromiseAsync.cpp, ThreadPoolExamples.cpp

1.7.3 By Design Pattern

- **CRTP**: CRTPvsVirtualFunctions.cpp
- **Pimpl**: PimplIdiom.cpp
- **NVI**: NVIIdiomTemplateMethod.cpp
- **Dependency Injection**: DependencyInjection.cpp
- **SOLID**: SOLIDPrinciples.cpp

1.8 Testing & Running Examples

Each example is a standalone executable. After building:

```
cd build/bin

# Run a specific example
./Cpp20Examples

# Run all examples (Linux/Mac)
cd ../..
./scripts/run_all.sh

# Run all examples (Windows PowerShell)
Get-ChildItem .\bin\Release\*.exe | ForEach-Object { & $_.FullName }
```

1.9 Dependencies

Most examples have no external dependencies. Optional dependencies for specific examples:

- **Boost.Asio**: AsioAndModernCppConcurrency.cpp, AsioMultipleContexts.cpp
- **Eigen**: EigenSensorFusion.cpp
- **nlohmann/json**: NlohmannJsonExample.cpp
- **Protocol Buffers**: ProtobufExample.cpp
- **pybind11**: Pybind11Example.cpp

Build scripts automatically check for these dependencies.

1.10 Contributing

Contributions are welcome! Please follow these guidelines:

1. **Code Style**: Follow the existing code style (Google C++ Style Guide)
2. **Documentation**: Each example should have clear comments explaining the concepts
3. **Compilation**: Ensure code compiles with C++20 standard
4. **Testing**: Test on multiple compilers if possible
5. **Commit Messages**: Use clear, descriptive commit messages

---

1.11 License

This project is licensed under the MIT License. See LICENSE file for details.

1.12 Contact & Support

For questions, suggestions, or issues: - **GitHub Issues**: https://github.com/AungKyawSoe-Tech/ModernCppExamples/issues - **Email**: aungksoe.mobile@gmail.com

---

1.13 Acknowledgments

Special thanks to: - The C++ Standards Committee for continuously evolving the language - Open source contributors: Boost, Eigen, nlohmann/json, pybind11 - The C++ community for valuable feedback and suggestions

---

1.14 Additional Resources

1.14.1 Official Documentation

- C++ Reference
- ISO C++ Standard
- C++ Core Guidelines

1.14.2 Safety Standards

- MISRA C++:2008
- AUTOSAR C++14
- ISO 26262 - Automotive functional safety
- DO-178C - Airborne software safety

1.14.3 Books

- "Effective Modern C++" by Scott Meyers
- "C++ Concurrency in Action" by Anthony Williams
- "Real-Time C++" by Christopher Kormanyos
- "C++ Move Semantics" by Nicolai Josuttis

Last Updated: January 3, 2026
Version: 1.0.0
Repository: https://github.com/AungKyawSoe-Tech/ModernCppExamples

ewpage

2 Source Code: ARMInstructionSets.cpp

File: `src/ARMInstructionSets.cpp`
Repository: View on GitHub

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

// ==================================================================
// ARM INSTRUCTION SETS: ARM, THUMB, THUMB-2
// ==================================================================
// This example discusses instruction set modes, alignment issues,
// and compiler switches for ARM Cortex-M processors.
//
// IMPORTANT: This is a DEMONSTRATION for understanding concepts.
// The actual crashes occur on ARM hardware, not x86/x64.
// Compiler flags discussed apply to ARM cross-compilers (arm-none-eabi-gcc).
// ==================================================================

// ==================================================================
// 1. ARM INSTRUCTION SET OVERVIEW
// ==================================================================

void explain_instruction_sets() {
    std::cout << "\n=== 1. ARM INSTRUCTION SETS OVERVIEW ===" << std::endl;

    std::cout << "\n THREE INSTRUCTION SET MODES:" << std::endl;

    std::cout << "\n1. ARM (A32) - 32-bit instructions:" << std::endl;
    std::cout << " •   Original ARM instruction set" << std::endl;
    std::cout << " •   Fixed 32-bit instruction size" << std::endl;
    std::cout << " •   Most powerful, but code size larger" << std::endl;
    std::cout << " •   Used in: ARM7, ARM9, ARM11, Cortex-A" << std::endl;
    std::cout << " •   Example instruction: ADD r0, r1, r2  (32 bits)" << std
        ::endl;

    std::cout << "\n2. THUMB (T16) - 16-bit instructions:" << std::endl;
```

```
    std::cout << " •   Compact 16-bit instruction subset" << std::endl;
    std::cout << " •   ~30% smaller code size vs ARM" << std::endl;
    std::cout << " •   Limited register access (r0-r7)" << std::endl;
    std::cout << " •   Used in: ARM7TDMI, ARM9, early Cortex-M0" << std::endl;
    std::cout << " •   Example: ADD r0, r1  (16 bits, 2-operand only)" << std
        ::endl;

    std::cout << "\n3. THUMB-2 (T32) - Mixed 16/32-bit:" << std::endl;
    std::cout << " •   Mix of 16-bit and 32-bit instructions" << std::endl;
    std::cout << " •   Best of both worlds: compact + powerful" << std::endl;
    std::cout << " •   Full register access, conditional execution" << std::
        endl;
    std::cout << " •   Used in: Cortex-M3, M4, M7, M33 (modern MCUs)" << std::
        endl;
    std::cout << " •   Example: ADD r0, r1, r2 (16 or 32 bits as needed)" <<
        std::endl;

    std::cout << "\n MODERN CORTEX-M PROCESSORS:" << std::endl;
    std::cout << " •   Cortex-M0/M0+: Thumb + subset of Thumb-2" << std::endl;
    std::cout << " •   Cortex-M3/M4/M7: Thumb-2 only (no ARM mode)" << std::
        endl;
    std::cout << " •   Cortex-M33/M55: Thumb-2 + DSP + optional FPU" << std::
        endl;
}

// ==================================================================
// 2. ALIGNMENT ISSUES - THE MAIN CAUSE OF CRASHES
// ==================================================================

// Simulated aligned data structure
struct __attribute__((aligned(4))) AlignedData {
    uint32_t value1;
    uint32_t value2;
    uint32_t value3;
};

// Potentially misaligned data
struct UnalignedData {
    uint8_t byte;      // 1 byte
    uint32_t value;    // Should be 4-byte aligned, but might not be!
    uint16_t halfword; // 2 bytes
};

void explain_alignment_issues() {
    std::cout << "\n=== 2. ALIGNMENT ISSUES (MAIN CRASH CAUSE) ===" << std::
        endl;

    std::cout << "\n ARM ALIGNMENT REQUIREMENTS:" << std::endl;
    std::cout << " •   uint8_t  (byte):      1-byte aligned (any address)" <<
        std::endl;
    std::cout << " •   uint16_t (halfword):  2-byte aligned (address % 2 == 0)
        " << std::endl;
    std::cout << " •   uint32_t (word):      4-byte aligned (address % 4 == 0)
        " << std::endl;
```

```
    std::cout << " •   uint64_t (doubleword): 8-byte aligned (address % 8 == 0)" << std::endl;

    std::cout << "\n WHAT HAPPENS ON MISALIGNED ACCESS:" << std::endl;
    std::cout << " •   Cortex-M0/M0+: HARDFAULT (crash)" << std::endl;
    std::cout << " •   Cortex-M3/M4/M7: May work BUT slower (extra cycles)" << std::endl;
    std::cout << " •   Depends on UNALIGN_TRP bit in CCR register" << std::endl;

    // Demonstrate alignment
    std::cout << "\n ALIGNMENT EXAMPLES:" << std::endl;

    AlignedData aligned;
    std::cout << "  AlignedData size:      " << sizeof(AlignedData) << " bytes" << std::endl;
    std::cout << "  Address of value1:     " << reinterpret_cast<uintptr_t>(&aligned.value1)
              << " (mod 4 = " << (reinterpret_cast<uintptr_t>(&aligned.value1) % 4) << ")" << std::endl;

    UnalignedData unaligned;
    std::cout << "\n  UnalignedData size:    " << sizeof(UnalignedData) << " bytes" << std::endl;
    std::cout << "  Address of byte:       " << reinterpret_cast<uintptr_t>(&unaligned.byte) << std::endl;
    std::cout << "  Address of value:      " << reinterpret_cast<uintptr_t>(&unaligned.value)
              << " (mod 4 = " << (reinterpret_cast<uintptr_t>(&unaligned.value) % 4) << ")" << std::endl;

    std::cout << "\n PROBLEM: If unaligned.value is at address 0x20000001:" << std::endl;
    std::cout << " •   ARM instruction: LDR r0, [r1]  (load word from unaligned address)" << std::endl;
    std::cout << " •   Result: HARDFAULT on Cortex-M0, silent corruption on some others" << std::endl;
}

// ================================================================
// 3. INTERWORKING ISSUES - STATE SWITCHING
// ================================================================

void explain_interworking() {
    std::cout << "\n=== 3. INTERWORKING ISSUES (STATE SWITCHING) ===" << std::endl;

    std::cout << "\n HOW PROCESSOR KNOWS WHICH MODE:" << std::endl;
    std::cout << " •   Function address LSB (Least Significant Bit):" << std::endl;
    std::cout << "     - LSB = 0: ARM mode (A32 instructions)" << std::endl;
    std::cout << "     - LSB = 1: Thumb/Thumb-2 mode (T16/T32 instructions)" << std::endl;
    std::cout << " •   BX/BLX instructions switch modes automatically" << std
```

```
        ::endl;

    std::cout << "\n CRASH SCENARIOS:" << std::endl;
    std::cout << "\n  1. WRONG FUNCTION ADDRESS:" << std::endl;
    std::cout << "     // C code" << std::endl;
    std::cout << "     void my_function() { /* compiled as Thumb */ }" << std::endl;
    std::cout << "     " << std::endl;
    std::cout << "     // Assembly calling without setting LSB" << std::endl;
    std::cout << "     LDR r0, =my_function  ; Address without LSB=1" << std::endl;
    std::cout << "     BLX r0                ;  Tries to execute as ARM!" << std::endl;
    std::cout << "     " << std::endl;
    std::cout << "       CORRECT:" << std::endl;
    std::cout << "     LDR r0, =my_function+1  ; Set LSB to indicate Thumb" << std::endl;
    std::cout << "     BLX r0                  ;  Correctly switches to Thumb" << std::endl;

    std::cout << "\n  2. INTERRUPT SERVICE ROUTINES (ISR):" << std::endl;
    std::cout << " •     Some ARM7/ARM9 enter exceptions in ARM mode" << std::endl;
    std::cout << " •     If ISR compiled as Thumb: CRASH!" << std::endl;
    std::cout << " •     Solution: Force ISR to ARM mode or ensure processor supports Thumb ISRs" << std::endl;

    std::cout << "\n  3. ASSEMBLY CODE MISMATCH:" << std::endl;
    std::cout << "     // File: startup.s (ARM mode)" << std::endl;
    std::cout << "     .arm" << std::endl;
    std::cout << "     Reset_Handler:" << std::endl;
    std::cout << "         LDR sp, =_estack" << std::endl;
    std::cout << "         BL main           ;  Calls main as ARM!" << std::endl;
    std::cout << "     " << std::endl;
    std::cout << "       CORRECT:" << std::endl;
    std::cout << "     .thumb              ; Switch to Thumb mode" << std::endl;
    std::cout << "     Reset_Handler:" << std::endl;
    std::cout << "         LDR sp, =_estack" << std::endl;
    std::cout << "         BLX main          ;  Properly switches to main's mode" << std::endl;
}

// ==================================================================
// 4. COMPILER SWITCHES AND FLAGS
// ==================================================================

void explain_compiler_switches() {
    std::cout << "\n=== 4. COMPILER SWITCHES FOR ARM ===" << std::endl;

    std::cout << "\n ARM-NONE-EABI-GCC FLAGS:" << std::endl;

    std::cout << "\n1. INSTRUCTION SET MODE:" << std::endl;
```

```
    std::cout << "   -marm" << std::endl;
    std::cout << " •     Generate ARM (A32) instructions" << std::endl;
    std::cout << " •     Larger code size, faster execution" << std::endl;
    std::cout << " •     Use for: ARM7, ARM9, Cortex-A" << std::endl;
    std::cout << " •     Example: arm-none-eabi-gcc -marm -mcpu=arm7tdmi main.c" << std::endl;

    std::cout << "\n   -mthumb" << std::endl;
    std::cout << " •     Generate Thumb/Thumb-2 instructions" << std::endl;
    std::cout << " •     Smaller code size, good performance" << std::endl;
    std::cout << " •     DEFAULT for Cortex-M (they only support Thumb-2)" << std::endl;
    std::cout << " •     Example: arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 main.c" << std::endl;

    std::cout << "\n2. CPU/ARCHITECTURE:" << std::endl;
    std::cout << "   -mcpu=cortex-m0      # Cortex-M0 (Thumb subset)" << std::endl;
    std::cout << "   -mcpu=cortex-m3      # Cortex-M3 (Thumb-2)" << std::endl;
    std::cout << "   -mcpu=cortex-m4      # Cortex-M4 (Thumb-2 + DSP)" << std::endl;
    std::cout << "   -mcpu=cortex-m7      # Cortex-M7 (Thumb-2 + DSP + FPU)" << std::endl;

    std::cout << "\n3. ALIGNMENT OPTIONS:" << std::endl;
    std::cout << "   -mno-unaligned-access" << std::endl;
    std::cout << " •     Disable unaligned memory access" << std::endl;
    std::cout << " •     Compiler generates multi-instruction sequences" << std::endl;
    std::cout << " •     Slower but safer on Cortex-M0" << std::endl;
    std::cout << " •     Example: arm-none-eabi-gcc -mthumb -mno-unaligned-access" << std::endl;

    std::cout << "\n   -munaligned-access (default on M3/M4/M7)" << std::endl;
    std::cout << " •     Allow unaligned memory access" << std::endl;
    std::cout << " •     Hardware handles it (with performance penalty)" << std::endl;
    std::cout << " •     Use for: Cortex-M3 and above" << std::endl;

    std::cout << "\n4. INTERWORKING:" << std::endl;
    std::cout << "   -mthumb-interwork" << std::endl;
    std::cout << " •     Generate code that can call between ARM and Thumb" << std::endl;
    std::cout << " •     Required for mixed ARM/Thumb projects" << std::endl;
    std::cout << " •     Not needed for Cortex-M (Thumb-2 only)" << std::endl;
}

// ================================================================
// 5. PRACTICAL EXAMPLE: ALIGNMENT CRASH SIMULATION
// ================================================================

// This simulates what happens on ARM hardware
void demonstrate_alignment_crash() {
    std::cout << "\n=== 5. ALIGNMENT CRASH SIMULATION ===" << std::endl;
```

```

    std::cout << "\n NOTE: On x86/x64, unaligned access works (slower)" << std::endl;
    std::cout << "   On ARM Cortex-M0: Would cause HARDFAULT" << std::endl;

    // Create a buffer with known alignment
    alignas(4) uint8_t buffer[16];

    // Write data at aligned address (safe)
    std::cout << "\n ALIGNED ACCESS (address % 4 == 0):" << std::endl;
    uint32_t* aligned_ptr = reinterpret_cast<uint32_t*>(&buffer[0]);
    *aligned_ptr = 0x12345678;
    std::cout << "   Address: " << reinterpret_cast<uintptr_t>(aligned_ptr)
              << " (aligned)" << std::endl;
    std::cout << "   Value written: 0x" << std::hex << *aligned_ptr << std::dec << std::endl;
    std::cout << "   Result:  SUCCESS (no crash on ARM)" << std::endl;

    // Try unaligned access (dangerous on ARM!)
    std::cout << "\n UNALIGNED ACCESS (address % 4 != 0):" << std::endl;
    uint32_t* unaligned_ptr = reinterpret_cast<uint32_t*>(&buffer[1]);
    std::cout << "   Address: " << reinterpret_cast<uintptr_t>(unaligned_ptr)
              << " (unaligned, offset by 1)" << std::endl;

    // On x86/x64 this works, on ARM Cortex-M0 it would HARDFAULT
    std::cout << "   Attempting to write uint32_t at unaligned address..." << std::endl;
    *unaligned_ptr = 0xAABBCCDD;  // Works on x86, HARDFAULT on Cortex-M0!
    std::cout << "   Result on x86:  Works (slow)" << std::endl;
    std::cout << "   Result on ARM Cortex-M0:  HARDFAULT! (crash)" << std::endl;
    std::cout << "   Result on ARM Cortex-M4:  Works but slow (multiple bus cycles)" << std::endl;
}

// ==================================================================
// 6. SOLUTION: PORTABLE UNALIGNED ACCESS
// ==================================================================

//  SAFE: Portable unaligned read (works on all platforms)
uint32_t read_uint32_unaligned(const uint8_t* ptr) {
    uint32_t value;
    std::memcpy(&value, ptr, sizeof(value));  // Compiler optimizes this
    return value;
}

//  SAFE: Portable unaligned write
void write_uint32_unaligned(uint8_t* ptr, uint32_t value) {
    std::memcpy(ptr, &value, sizeof(value));
}

//  UNSAFE: Direct cast (crashes on unaligned ARM)
uint32_t read_uint32_unsafe(const uint8_t* ptr) {
    return *reinterpret_cast<const uint32_t*>(ptr);  // HARDFAULT if unaligned
```

```
        !
}

void demonstrate_portable_unaligned() {
    std::cout << "\n=== 6. PORTABLE UNALIGNED ACCESS ===" << std::endl;

    alignas(4) uint8_t buffer[16] = {};

    std::cout << "\n SAFE METHOD (using memcpy):" << std::endl;
    std::cout << "   uint32_t read_uint32_unaligned(const uint8_t* ptr) {" << std::endl;
    std::cout << "       uint32_t value;" << std::endl;
    std::cout << "       std::memcpy(&value, ptr, sizeof(value));" << std::endl;
    std::cout << "       return value;" << std::endl;
    std::cout << "   }" << std::endl;

    write_uint32_unaligned(&buffer[1], 0x12345678);
    uint32_t safe_value = read_uint32_unaligned(&buffer[1]);
    std::cout << "\n   Written to unaligned address using memcpy" << std::endl;
    std::cout << "   Value: 0x" << std::hex << safe_value << std::dec << std::endl;
    std::cout << "    Works on ALL platforms (x86, ARM, RISC-V)" << std::endl;
    std::cout << "    Compiler optimizes memcpy to efficient code" << std::endl;

    std::cout << "\n UNSAFE METHOD (direct cast):" << std::endl;
    std::cout << "   uint32_t value = *reinterpret_cast<uint32_t*>(ptr);" << std::endl;
    std::cout << "    HARDFAULT on ARM Cortex-M0 if unaligned!" << std::endl;
}

// ==================================================================
// 7. ASSEMBLY DIRECTIVES FOR MIXED CODE
// ==================================================================

void explain_assembly_directives() {
    std::cout << "\n=== 7. ASSEMBLY DIRECTIVES ===" << std::endl;

    std::cout << "\n ARM ASSEMBLY DIRECTIVES:" << std::endl;

    std::cout << "\n1. SET INSTRUCTION MODE:" << std::endl;
    std::cout << "   .arm              ; Switch to ARM (A32) mode" << std::endl;
    std::cout << "   .thumb            ; Switch to Thumb/Thumb-2 (T16/T32) mode" << std::endl;

    std::cout << "\n2. MARK THUMB FUNCTIONS:" << std::endl;
    std::cout << "   .thumb_func       ; Next symbol is a Thumb function" << std::endl;
    std::cout << "   " << std::endl;
    std::cout << "   Example:" << std::endl;
```

```
    std::cout << "    .thumb" << std::endl;
    std::cout << "    .thumb_func" << std::endl;
    std::cout << "    my_function:" << std::endl;
    std::cout << "        PUSH {lr}" << std::endl;
    std::cout << "        ; ... function code ..." << std::endl;
    std::cout << "        POP {pc}" << std::endl;

    std::cout << "\n3. ALIGNMENT DIRECTIVES:" << std::endl;
    std::cout << "    .align 2          ; Align to 2^2 = 4 bytes (word)" << std::endl;
    std::cout << "    .align 3          ; Align to 2^3 = 8 bytes (doubleword)" << std::endl;
    std::cout << "    " << std::endl;
    std::cout << "    Example:" << std::endl;
    std::cout << "    .align 2          ; Force 4-byte alignment" << std::endl;
    std::cout << "    my_data:" << std::endl;
    std::cout << "        .word 0x12345678" << std::endl;
}

// ================================================================
// 8. REAL-WORLD EXAMPLE: PROTOCOL PARSING
// ================================================================

// Common scenario: Parsing network packets with misaligned fields
struct __attribute__((packed)) NetworkPacket {
    uint8_t  header;      // 1 byte
    uint32_t timestamp;   // 4 bytes (might be misaligned!)
    uint16_t length;      // 2 bytes
    uint32_t crc;         // 4 bytes (might be misaligned!)
};

void demonstrate_real_world_scenario() {
    std::cout << "\n=== 8. REAL-WORLD: PROTOCOL PARSING ===" << std::endl;

    std::cout << "\n SCENARIO: Parsing network packet on Cortex-M4" << std::endl;

    // Simulate receiving a packet
    alignas(4) uint8_t rx_buffer[16] = {
        0xAA,                          // header (1 byte)
        0x78, 0x56, 0x34, 0x12,        // timestamp (4 bytes, at offset 1 = UNALIGNED!)
        0x10, 0x00,                    // length (2 bytes)
        0xDD, 0xCC, 0xBB, 0xAA         // crc (4 bytes)
    };

    std::cout << "\n WRONG WAY (direct struct cast):" << std::endl;
    std::cout << "    NetworkPacket* pkt = (NetworkPacket*)rx_buffer;" << std::endl;
    std::cout << "    uint32_t ts = pkt->timestamp;  //  Unaligned access!" << std::endl;
    std::cout << "    " << std::endl;
    std::cout << "    Result: HARDFAULT on Cortex-M0, slow on Cortex-M4" << std::endl;
```

```

    std::cout << "\n CORRECT WAY (memcpy for unaligned fields):" << std::endl;
    uint8_t header = rx_buffer[0];
    uint32_t timestamp;
    uint16_t length;
    uint32_t crc;

    std::memcpy(&timestamp, &rx_buffer[1], sizeof(timestamp));
    std::memcpy(&length, &rx_buffer[5], sizeof(length));
    std::memcpy(&crc, &rx_buffer[7], sizeof(crc));

    std::cout << "   Header: 0x" << std::hex << static_cast<int>(header) << std::endl;
    std::cout << "   Timestamp: 0x" << timestamp << std::endl;
    std::cout << "   Length: 0x" << length << std::endl;
    std::cout << "   CRC: 0x" << crc << std::dec << std::endl;
    std::cout << "    Works on ALL ARM processors!" << std::endl;

    std::cout << "\n COMPILER OPTIMIZATIONS:" << std::endl;
    std::cout << " •   With -O2: memcpy() compiles to efficient LDR/STR" << std::endl;
    std::cout << " •   No function call overhead" << std::endl;
    std::cout << " •   Compiler knows about alignment and handles it" << std::endl;
}

// ==================================================================
// 9. COMPILER FLAG EXAMPLES
// ==================================================================

void show_compiler_examples() {
    std::cout << "\n=== 9. PRACTICAL COMPILER COMMAND EXAMPLES ===" << std::endl;

    std::cout << "\n TYPICAL MAKEFILE FLAGS:" << std::endl;

    std::cout << "\n1. CORTEX-M0 PROJECT (strict alignment):" << std::endl;
    std::cout << "   CFLAGS = -mcpu=cortex-m0 \\" << std::endl;
    std::cout << "            -mthumb \\" << std::endl;
    std::cout << "            -mno-unaligned-access \\" << std::endl;
    std::cout << "            -O2 -Wall" << std::endl;

    std::cout << "\n2. CORTEX-M4 PROJECT (with FPU):" << std::endl;
    std::cout << "   CFLAGS = -mcpu=cortex-m4 \\" << std::endl;
    std::cout << "            -mthumb \\" << std::endl;
    std::cout << "            -mfpu=fpv4-sp-d16 \\" << std::endl;
    std::cout << "            -mfloat-abi=hard \\" << std::endl;
    std::cout << "            -munaligned-access \\" << std::endl;
    std::cout << "            -O2 -Wall" << std::endl;

    std::cout << "\n3. MIXED ARM/THUMB PROJECT (ARM7):" << std::endl;
    std::cout << "   CFLAGS = -mcpu=arm7tdmi \\" << std::endl;
    std::cout << "            -mthumb \\" << std::endl;
```

```
    std::cout << "            -mthumb-interwork \\" << std::endl;
    std::cout << "            -O2 -Wall" << std::endl;

    std::cout << "\n4. ASSEMBLY FILE COMPILATION:" << std::endl;
    std::cout << "   # For Thumb-2 code" << std::endl;
    std::cout << "   arm-none-eabi-as -mcpu=cortex-m4 \\" << std::endl;
    std::cout << "                    -mthumb \\" << std::endl;
    std::cout << "                    startup.s -o startup.o" << std::endl;
}

// ================================================================
// 10. DEBUGGING HARDFAULTS
// ================================================================

void explain_debugging_hardfaults() {
    std::cout << "\n=== 10. DEBUGGING HARDFAULTS ON ARM ===" << std::endl;

    std::cout << "\ n WHEN YOU GET A HARDFAULT:" << std::endl;

    std::cout << "\n1. CHECK FAULT STATUS REGISTERS:" << std::endl;
    std::cout << " •   HFSR (HardFault Status Register) at 0xE000ED2C" << std::endl;
    std::cout << " •   CFSR (Configurable Fault Status) at 0xE000ED28" << std::endl;
    std::cout << " •   MMFAR (MemManage Fault Address) at 0xE000ED34" << std::endl;
    std::cout << " •   BFAR (BusFault Address) at 0xE000ED38" << std::endl;

    std::cout << "\n2. COMMON FAULT CAUSES:" << std::endl;
    std::cout << "   IBUSERR (bit 0 of CFSR):  Instruction bus error" << std::endl;
    std::cout << " →     Called function at wrong address or wrong mode" << std::endl;
    std::cout << "   " << std::endl;
    std::cout << "   PRECISERR (bit 1 of CFSR): Data bus error" << std::endl;
    std::cout << " →     Unaligned access on Cortex-M0" << std::endl;
    std::cout << "   " << std::endl;
    std::cout << "   IACCVIOL (bit 0 of CFSR): MPU violation" << std::endl;
    std::cout << " →     Tried to execute from non-executable memory" << std::endl;

    std::cout << "\n3. DEBUGGER INSPECTION:" << std::endl;
    std::cout << "   (gdb) info registers  # Check PC, LR, SP" << std::endl;
    std::cout << "   (gdb) x/4x $sp        # Stack contents" << std::endl;
    std::cout << "   (gdb) bt              # Backtrace" << std::endl;
    std::cout << "   " << std::endl;
    std::cout << "   Look for:" << std::endl;
    std::cout << " •   PC (Program Counter) - where crash occurred" << std::endl;
    std::cout << " •   LR (Link Register) - return address (LSB = mode)" << std::endl;
    std::cout << " •   Unaligned addresses (check if address % 4 != 0)" << std::endl;
}
```

```

// ==================================================================
// MAIN
// ==================================================================

int main() {
    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  ARM INSTRUCTION SETS: ARM, THUMB, THUMB-2" << std::endl;
    std::cout << "  Alignment Issues and Compiler Switches" << std::endl;
    std::cout << " ==================================================================" << std::endl;

    explain_instruction_sets();
    explain_alignment_issues();
    explain_interworking();
    explain_compiler_switches();
    demonstrate_alignment_crash();
    demonstrate_portable_unaligned();
    explain_assembly_directives();
    demonstrate_real_world_scenario();
    show_compiler_examples();
    explain_debugging_hardfaults();

    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  SUMMARY: AVOIDING ARM THUMB-2 CRASHES" << std::endl;
    std::cout << " ==================================================================" << std::endl;

    std::cout << "\ n KEY TAKEAWAYS:" << std::endl;

    std::cout << "\n1. INSTRUCTION SET MODES:" << std::endl;
    std::cout << " •   ARM (A32): 32-bit, powerful, larger code" << std::endl;
    std::cout << " •   Thumb (T16): 16-bit, compact, limited" << std::endl;
    std::cout << " •   Thumb-2 (T32): 16/32-bit mix, best of both" << std::endl;
    std::cout << " •   Cortex-M: Thumb-2 only (no ARM mode)" << std::endl;

    std::cout << "\n2. ALIGNMENT REQUIREMENTS:" << std::endl;
    std::cout << " •   uint32_t MUST be 4-byte aligned on Cortex-M0" << std::endl;
    std::cout << " •   Cortex-M3/M4/M7 can handle unaligned (but slower)" << std::endl;
    std::cout << " •   Use std::memcpy() for portable unaligned access" << std::endl;

    std::cout << "\n3. INTERWORKING:" << std::endl;
    std::cout << " •   Function address LSB indicates mode (0=ARM, 1=Thumb)" << std::endl;
```

```
    std::cout << " •   Use BLX instruction for mode switching" << std::endl;
    std::cout << " •   Mark assembly functions with .thumb_func" << std::endl;

    std::cout << "\n4. COMPILER FLAGS:" << std::endl;
    std::cout << "   -mthumb                    # Use Thumb-2 instructions" << std::endl;
    std::cout << "   -mcpu=cortex-m4            # Specify processor" << std::endl;
    std::cout << "   -mno-unaligned-access      # Disable unaligned (M0)" << std::endl;
    std::cout << "   -munaligned-access         # Enable unaligned (M3+)" << std::endl;

    std::cout << "\n5. BEST PRACTICES:" << std::endl;
    std::cout << "     Use std::memcpy() for unaligned access" << std::endl;
    std::cout << "     Align structs with __attribute__((aligned(4)))" << std::endl;
    std::cout << "     Use __attribute__((packed)) carefully" << std::endl;
    std::cout << "     Compile with correct -mcpu flag" << std::endl;
    std::cout << "     Test on actual hardware (not just simulator)" << std::endl;
    std::cout << "     Never cast unaligned pointers directly" << std::endl;
    std::cout << "     Don't mix ARM/Thumb without proper interworking" << std::endl;

    std::cout << "\ n DEBUGGING TIPS:" << std::endl;
    std::cout << "   1. Enable HardFault handler to print fault registers" << std::endl;
    std::cout << "   2. Check CFSR register to identify fault type" << std::endl;
    std::cout << "   3. Look for unaligned addresses in fault address registers" << std::endl;
    std::cout << "   4. Verify -mcpu matches your actual hardware" << std::endl;
    std::cout << "   5. Check linker script for correct alignment" << std::endl;

    std::cout << "\n ==============================================================\n" << std::endl;

    return 0;
}
```

3 Source Code: AdvancedExceptionHandling.cpp

File: `src/AdvancedExceptionHandling.cpp`
Repository: View on GitHub

```cpp
// AdvancedExceptionHandling.cpp
// Comprehensive exception handling with real stack traces, nested exceptions,
// and advanced error reporting patterns

#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
#include <memory>
#include <sstream>
#include <iomanip>
#include <chrono>
#include <functional>
#include <exception>
#include <typeinfo>
#include <cstdlib>
#include <cstring>

// Platform-specific includes for stack traces
#ifdef __linux__
    #include <execinfo.h>
    #include <cxxabi.h>
    #include <unistd.h>
#elif _WIN32
    #include <windows.h>
    #include <dbghelp.h>
    #pragma comment(lib, "dbghelp.lib")
#endif

using namespace std::chrono;

// ==========================================================================
// SECTION 1: Stack Trace Capture (Platform-Specific)
// ==========================================================================

class StackTrace {
private:
    static constexpr size_t MAX_FRAMES = 64;
    std::vector<std::string> frames_;

public:
    StackTrace() {
        capture();
    }

```

```
    void capture() {
        frames_.clear();

#ifdef __linux__
        // Linux: Use backtrace() and backtrace_symbols()
        void* buffer[MAX_FRAMES];
        int frame_count = backtrace(buffer, MAX_FRAMES);

        char** symbols = backtrace_symbols(buffer, frame_count);
        if (symbols) {
            for (int i = 0; i < frame_count; ++i) {
                frames_.push_back(demangle_symbol(symbols[i]));
            }
            free(symbols);
        }

#elif _WIN32
        // Windows: Use CaptureStackBackTrace() and SymFromAddr()
        void* buffer[MAX_FRAMES];
        HANDLE process = GetCurrentProcess();
        SymInitialize(process, NULL, TRUE);

        WORD frame_count = CaptureStackBackTrace(0, MAX_FRAMES, buffer, NULL);

        SYMBOL_INFO* symbol = (SYMBOL_INFO*)calloc(sizeof(SYMBOL_INFO) + 256 *
            sizeof(char), 1);
        if (symbol) {
            symbol->MaxNameLen = 255;
            symbol->SizeOfStruct = sizeof(SYMBOL_INFO);

            for (WORD i = 0; i < frame_count; ++i) {
                if (SymFromAddr(process, (DWORD64)(buffer[i]), 0, symbol)) {
                    frames_.push_back(std::string(symbol->Name));
                } else {
                    std::ostringstream oss;
                    oss << "0x" << std::hex << (uint64_t)buffer[i];
                    frames_.push_back(oss.str());
                }
            }
            free(symbol);
        }

        SymCleanup(process);
#else
        frames_.push_back("[Stack trace not available on this platform]");
#endif
    }

    const std::vector<std::string>& get_frames() const {
        return frames_;
    }

    std::string to_string() const {
        std::ostringstream oss;
```

```
        oss << "Stack Trace (" << frames_.size() << " frames):\n";
        for (size_t i = 0; i < frames_.size(); ++i) {
            oss << "  #" << std::setw(2) << i << ": " << frames_[i] << "\n";
        }
        return oss.str();
    }

private:
#ifdef __linux__
    std::string demangle_symbol(const char* mangled) {
        std::string result = mangled;

        // Extract the mangled name between '(' and '+'
        const char* begin = strchr(mangled, '(');
        const char* end = strchr(mangled, '+');

        if (begin && end && begin < end) {
            begin++;  // Skip '('
            std::string mangled_name(begin, end - begin);

            int status;
            char* demangled = abi::__cxa_demangle(mangled_name.c_str(),
                nullptr, nullptr, &status);

            if (status == 0 && demangled) {
                result = demangled;
                free(demangled);
            }
        }

        return result;
    }
#endif
};

void demonstrate_stack_trace_capture() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 1. Real Stack Trace Capture ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Capturing stack trace from current location...\n\n";

    StackTrace trace;
    std::cout << trace.to_string();

    std::cout << "\n Stack trace captured using platform-specific APIs\n";
#ifdef __linux__
    std::cout << "  Linux: backtrace() + abi::__cxa_demangle()\n";
#elif _WIN32
    std::cout << "  Windows: CaptureStackBackTrace() + SymFromAddr()\n";
#else
    std::cout << "   Generic fallback (no native stack trace support)\n";
#endif
}
```

```

//
    ==========================================================================
// SECTION 2: Exception with Stack Trace
//
    ==========================================================================

class TracedException : public std::runtime_error {
private:
    std::string file_;
    int line_;
    std::string function_;
    StackTrace stack_trace_;
    std::string formatted_message_;

    std::string format_message() const {
        std::ostringstream oss;

        // Extract filename only
        size_t last_slash = file_.find_last_of("/\\");
        std::string filename = (last_slash != std::string::npos)
            ? file_.substr(last_slash + 1) : file_;

        oss << "\                              n\n";
        oss << "  EXCEPTION THROWN WITH STACK TRACE\n";
        oss << "                              \n";
        oss << "  Location: " << filename << ":" << line_ << "\n";
        oss << "  Function: " << function_ << "()\n";
        oss << "  Message:  " << what() << "\n";
        oss << "                              \n";
        oss << "  " << stack_trace_.to_string();
        oss << "                              \n";

        return oss.str();
    }

public:
    TracedException(const std::string& file, int line, const std::string& func
        ,
                    const std::string& message)
        : std::runtime_error(message),
          file_(file),
          line_(line),
          function_(func),
          stack_trace_(),
          formatted_message_(format_message()) {}

    const std::string& get_formatted_message() const {
        return formatted_message_;
    }

    const StackTrace& get_stack_trace() const {
```

```
        return stack_trace_;
    }

    const std::string& get_file() const { return file_; }
    int get_line() const { return line_; }
    const std::string& get_function() const { return function_; }
};

#define THROW_TRACED(message) \
    throw TracedException(__FILE__, __LINE__, __FUNCTION__, message)

void nested_function_3() {
    std::cout << "  [3] About to throw exception...\n";
    THROW_TRACED("Critical error in nested function");
}

void nested_function_2() {
    std::cout << "  [2] Calling nested_function_3()...\n";
    nested_function_3();
}

void nested_function_1() {
    std::cout << "  [1] Calling nested_function_2()...\n";
    nested_function_2();
}

void demonstrate_exception_with_stack_trace() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 2. Exception with Stack Trace ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Calling nested functions to build call stack...\n\n";

    try {
        nested_function_1();
    }
    catch (const TracedException& e) {
        std::cout << "\n Exception caught!\n";
        std::cout << e.get_formatted_message();
    }
}

// ==========================================================================
// SECTION 3: Nested Exceptions (C++11)
// ==========================================================================

class DatabaseException : public std::runtime_error {
public:
    explicit DatabaseException(const std::string& msg)
        : std::runtime_error("Database Error: " + msg) {}
```

```
};

class ConnectionException : public std::runtime_error {
public:
    explicit ConnectionException(const std::string& msg)
        : std::runtime_error("Connection Error: " + msg) {}
};

void database_operation() {
    std::cout << "    Attempting database query...\n";
    throw DatabaseException("Query timeout after 30 seconds");
}

void connect_to_database() {
    std::cout << "  Attempting to connect to database...\n";

    try {
        database_operation();
    }
    catch (const DatabaseException& e) {
        std::cout << "      Database operation failed!\n";
        // Wrap the DatabaseException in a ConnectionException
        std::throw_with_nested(ConnectionException("Failed to establish
            connection"));
    }
}

void print_nested_exception(const std::exception& e, int level = 0) {
    std::string indent(level * 2, ' ');
    std::cout << indent << "  " << e.what() << "\n";

    try {
        std::rethrow_if_nested(e);
    }
    catch (const std::exception& nested) {
        print_nested_exception(nested, level + 1);
    }
}

void demonstrate_nested_exceptions() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 3. Nested Exceptions (C++11) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Concept: Wrap exceptions to preserve context\n\n";

    try {
        connect_to_database();
    }
    catch (const std::exception& e) {
        std::cout << "\n Exception caught with nested context:\n\n";
        print_nested_exception(e);
    }

```

```
    std::cout << "\n Original exception preserved inside wrapper\n";
    std::cout << "  Use std::throw_with_nested() and std::rethrow_if_nested()\n";
}

// ==========================================================================
// SECTION 4: Exception Guarantee Levels
// ==========================================================================

template<typename T>
class Container {
private:
    std::unique_ptr<T[]> data_;
    size_t size_;
    size_t capacity_;

public:
    Container() : data_(nullptr), size_(0), capacity_(0) {}

    // Basic guarantee: valid state but data may be lost
    void push_back_basic(const T& value) {
        if (size_ == capacity_) {
            size_t new_capacity = (capacity_ == 0) ? 1 : capacity_ * 2;
            auto new_data = std::make_unique<T[]>(new_capacity);

            // If copy throws here, we're in an inconsistent state
            for (size_t i = 0; i < size_; ++i) {
                new_data[i] = data_[i];  // May throw!
            }

            data_ = std::move(new_data);
            capacity_ = new_capacity;
        }

        data_[size_++] = value;
    }

    // Strong guarantee: operation succeeds completely or has no effect
    void push_back_strong(const T& value) {
        if (size_ == capacity_) {
            size_t new_capacity = (capacity_ == 0) ? 1 : capacity_ * 2;
            auto new_data = std::make_unique<T[]>(new_capacity);

            // Copy all elements (may throw)
            for (size_t i = 0; i < size_; ++i) {
                new_data[i] = data_[i];
            }

            // Only if successful, commit changes (no-throw from here)
            data_ = std::move(new_data);
```

```
            capacity_ = new_capacity;
        }

        data_[size_++] = value;
    }

    // No-throw guarantee: never throws
    size_t size() const noexcept {
        return size_;
    }

    // No-throw guarantee
    void clear() noexcept {
        size_ = 0;
    }
};

void demonstrate_exception_guarantees() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 4. Exception Safety Guarantees ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. No-throw guarantee (noexcept):\n";
    std::cout << " •   Function never throws\n";
    std::cout << " •   Example: size(), clear(), swap()\n";
    std::cout << " •   Mark with 'noexcept' keyword\n\n";

    std::cout << "2. Strong guarantee:\n";
    std::cout << " •   Operation succeeds completely OR has no effect\n";
    std::cout << " •   Example: push_back() using copy-and-swap\n";
    std::cout << " •   All or nothing - no partial state\n\n";

    std::cout << "3. Basic guarantee:\n";
    std::cout << " •   Object left in valid state if exception thrown\n";
    std::cout << " •   Example: push_back() that may leave capacity changed\n";
    std::cout << " •   No resource leaks, but data may be lost\n\n";

    std::cout << "4. No guarantee:\n";
    std::cout << " •   May leave object in invalid state\n";
    std::cout << " •     AVOID - leads to crashes and undefined behavior\n\n";

    Container<int> cont;
    cont.push_back_strong(42);
    std::cout << "  Container with strong guarantee used successfully\n";
}

// ==========================================================================
// SECTION 5: RAII and Exception Safety
// ==========================================================================
```

```

class FileRAII {
private:
    FILE* file_;
    std::string filename_;

public:
    explicit FileRAII(const std::string& filename)
        : file_(nullptr), filename_(filename) {

        file_ = fopen(filename.c_str(), "w");
        if (!file_) {
            throw std::runtime_error("Failed to open file: " + filename);
        }
        std::cout << "  [RAII] File opened: " << filename << "\n";
    }

    ~FileRAII() {
        if (file_) {
            fclose(file_);
            std::cout << "  [RAII] File closed: " << filename_ << "\n";
        }
    }

    void write(const std::string& data) {
        if (!file_) {
            throw std::runtime_error("File not open");
        }

        if (data == "ERROR") {
            throw std::runtime_error("Simulated write error");
        }

        fprintf(file_, "%s\n", data.c_str());
    }

    // Delete copy/move to enforce RAII
    FileRAII(const FileRAII&) = delete;
    FileRAII& operator=(const FileRAII&) = delete;
};

void demonstrate_raii_exception_safety() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 5. RAII and Exception Safety ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Without RAII (manual cleanup - leak on exception):\n\n";
    try {
        FILE* f = fopen("/tmp/test.txt", "w");
        std::cout << "  File opened\n";
        // If exception thrown here, file never closed!
        throw std::runtime_error("Oops!");
        fclose(f);  // Never executed
    }
```

```
    catch (...) {
        std::cout << "    File leaked - never closed!\n\n";
    }

    std::cout << "With RAII (automatic cleanup - safe):\n\n";
    try {
        FileRAII file("/tmp/test_raii.txt");
        file.write("Hello");
        file.write("ERROR");  // Throws exception
        file.write("World");  // Never executed
    }
    catch (const std::exception& e) {
        std::cout << "     Exception: " << e.what() << "\n";
        std::cout << "    RAII destructor automatically closed file!\n";
    }

    std::cout << "\n RAII guarantees resource cleanup even with exceptions\n";
}

// ==========================================================================
// SECTION 6: std::exception_ptr and Thread Exception Propagation
// ==========================================================================

#include <thread>

std::exception_ptr worker_exception = nullptr;

void worker_thread_function() {
    try {
        std::cout << "  [Worker] Doing some work...\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        // Simulate error
        throw std::runtime_error("Worker thread encountered an error!");
    }
    catch (...) {
        std::cout << "  [Worker] Exception caught, storing for main thread\n";
        worker_exception = std::current_exception();
    }
}

void demonstrate_exception_ptr() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 6. std::exception_ptr (Thread Exception Propagation) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Problem: Exceptions can't cross thread boundaries\n";
    std::cout << "Solution: std::exception_ptr + std::current_exception()\n\n"
```

```
        ;

    std::cout << "Starting worker thread...\n";

    worker_exception = nullptr;
    std::thread worker(worker_thread_function);
    worker.join();

    std::cout << "\nMain thread checking for worker exception...\n";

    if (worker_exception) {
        try {
            std::rethrow_exception(worker_exception);
        }
        catch (const std::exception& e) {
            std::cout << "  Worker exception received: " << e.what() << "\n";
        }
    }

    std::cout << "\ n Exception successfully propagated from worker to main thread\n";
}

// ==========================================================================
// SECTION 7: Custom Exception Hierarchy
// ==========================================================================

class ApplicationException : public std::runtime_error {
protected:
    int error_code_;
    system_clock::time_point timestamp_;

public:
    ApplicationException(int code, const std::string& msg)
        : std::runtime_error(msg),
          error_code_(code),
          timestamp_(system_clock::now()) {}

    int get_error_code() const { return error_code_; }

    std::string get_timestamp() const {
        auto time = system_clock::to_time_t(timestamp_);
        std::ostringstream oss;
        oss << std::put_time(std::localtime(&time), "%Y-%m-%d %H:%M:%S");
        return oss.str();
    }
};

class NetworkError : public ApplicationException {
public:
```

```
    NetworkError(int code, const std::string& msg)
        : ApplicationException(code, msg) {}
};

class FileSystemError : public ApplicationException {
public:
    FileSystemError(int code, const std::string& msg)
        : ApplicationException(code, msg) {}
};

class BusinessLogicError : public ApplicationException {
public:
    BusinessLogicError(int code, const std::string& msg)
        : ApplicationException(code, msg) {}
};

void demonstrate_exception_hierarchy() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 7. Custom Exception Hierarchy ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Exception Hierarchy:\n";
    std::cout << "  ApplicationException (base)\n";
    std::cout << "    NetworkError\n";
    std::cout << "    FileSystemError\n";
    std::cout << "    BusinessLogicError\n\n";

    try {
        throw NetworkError(503, "Service unavailable");
    }
    catch (const NetworkError& e) {
        std::cout << "Caught NetworkError:\n";
        std::cout << "  Code:      " << e.get_error_code() << "\n";
        std::cout << "  Message:   " << e.what() << "\n";
        std::cout << "  Timestamp: " << e.get_timestamp() << "\n";
    }
    catch (const ApplicationException& e) {
        std::cout << "Caught generic ApplicationException\n";
    }

    std::cout << "\n Specific catch before generic catch\n";
    std::cout << "  Polymorphic exception handling\n";
}

// ==========================================================================
// SECTION 8: Function Try Blocks (Constructor Exception Handling)
// ==========================================================================

class Resource {
public:
```

```
    Resource(int id) {
        std::cout << "    [Resource " << id << "] Constructed\n";
        if (id == 2) {
            throw std::runtime_error("Resource 2 construction failed!");
        }
    }

    ~Resource() {
        std::cout << "    [Resource] Destroyed\n";
    }
};

class ComponentWithResources {
private:
    Resource res1_;
    Resource res2_;
    Resource res3_;

public:
    // Function try block for constructor
    ComponentWithResources()
    try : res1_(1), res2_(2), res3_(3)  // res2 will throw!
    {
        std::cout << "  Constructor body executed\n";
    }
    catch (const std::exception& e) {
        std::cout << "    Constructor caught: " << e.what() << "\n";
        std::cout << "    res1_ will be automatically destroyed\n";
        // Note: Exception is automatically rethrown after this catch
    }
};

void demonstrate_function_try_blocks() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 8. Function Try Blocks (Constructor Exception Handling) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Attempting to construct ComponentWithResources...\n\n";

    try {
        ComponentWithResources component;
    }
    catch (const std::exception& e) {
        std::cout << "\n Exception propagated to caller\n";
        std::cout << "  Partially constructed members automatically cleaned up\n";
    }
}

// ==========================================================================
// SECTION 9: Best Practices Summary
```

```cpp
// ==========================================================================

void demonstrate_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 9. Exception Handling Best Practices ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DO:\n";
    std::cout << "   \n";
    std::cout << "   1. Use RAII for resource management\n";
    std::cout << "   2. Catch by const reference: catch (const Ex& e)\n";
    std::cout << "   3. Throw by value: throw MyException(...)\n";
    std::cout << "   4. Inherit from std::exception hierarchy\n";
    std::cout << "   5. Mark functions noexcept when appropriate\n";
    std::cout << "   6. Use specific exceptions before generic ones\n";
    std::cout << "   7. Provide meaningful error messages\n";
    std::cout << "   8. Document what exceptions functions can throw\n";
    std::cout << "   9. Use stack traces for debugging\n";
    std::cout << "   10. Test exception paths thoroughly\n\n";

    std::cout << "  DON'T:\n";
    std::cout << "     \n";
    std::cout << "   1. Throw from destructors (noexcept by default in C++11)\n";
    std::cout << "   2. Catch (...) without rethrowing or logging\n";
    std::cout << "   3. Use exceptions for normal control flow\n";
    std::cout << "   4. Return error codes for exceptional situations\n";
    std::cout << "   5. Ignore exceptions silently\n";
    std::cout << "   6. Throw pointers: throw new Ex()  \n";
    std::cout << "   7. Catch by value (causes slicing)\n";
    std::cout << "   8. Mix exceptions and error codes inconsistently\n";
    std::cout << "   9. Throw from noexcept functions\n";
    std::cout << "   10. Forget to clean up resources (use RAII!)\n\n";

    std::cout << "EXCEPTION vs ERROR CODES:\n";
    std::cout << "            \n";
    std::cout << "Use Exceptions when:\n";
    std::cout << " •  Error is truly exceptional\n";
    std::cout << " •  Can't continue execution\n";
    std::cout << " •  Need to propagate through many layers\n";
    std::cout << " •  RAII cleanup is needed\n\n";

    std::cout << "Use Error Codes when:\n";
    std::cout << " •  Expected/recoverable errors\n";
    std::cout << " •  Performance-critical hot paths\n";
    std::cout << " •  C API compatibility\n";
    std::cout << " •  Safety-critical systems (ISO 26262)\n";
}

// ==========================================================================
```

```
// MAIN FUNCTION
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "         Advanced Exception Handling with Stack Traces          \n";
    std::cout << "                                                                  \n";
    std::cout << "   Comprehensive guide to modern C++ exception handling patterns    \n";
    std::cout << "                                    \n";

    demonstrate_stack_trace_capture();
    demonstrate_exception_with_stack_trace();
    demonstrate_nested_exceptions();
    demonstrate_exception_guarantees();
    demonstrate_raii_exception_safety();
    demonstrate_exception_ptr();
    demonstrate_exception_hierarchy();
    demonstrate_function_try_blocks();
    demonstrate_best_practices();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All exception handling demonstrations completed!\n";
    std::cout << "\nKEY TAKEAWAYS:\n";
    std::cout << "  1. Real stack traces available on Linux/Windows\n";
    std::cout << "  2. RAII guarantees cleanup even with exceptions\n";
    std::cout << "  3. Nested exceptions preserve error context\n";
    std::cout << "  4. std::exception_ptr for thread exception propagation\n";
    std::cout << "  5. Always catch by const reference\n";
    std::cout << "  6. Use noexcept for no-throw guarantees\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

4 Source Code: AsioAndModernCppConcurrency.cpp

File: `src/AsioAndModernCppConcurrency.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <thread>
#include <future>
#include <chrono>
#include <vector>
#include <string>
#include <mutex>
#include <memory>
#include <functional>

// Note: This example demonstrates ASIO concepts using standard C++ features.
// For actual ASIO usage, install: https://think-async.com/Asio/
// Standalone ASIO: #include <asio.hpp>
// Boost.ASIO: #include <boost/asio.hpp>

// ==================================================================
// ASIO AND MODERN C++ CONCURRENCY COMPARISON
// ==================================================================

// This example shows the concepts and patterns without requiring ASIO installation
// It demonstrates when to use ASIO vs C++ standard library features

using namespace std::chrono_literals;

// ==================================================================
// 1. THE FUNDAMENTAL DIFFERENCE: I/O vs CPU
// ==================================================================

void example_io_vs_cpu_bound() {
    std::cout << "\n=== 1. I/O-BOUND vs CPU-BOUND OPERATIONS ===" << std::endl;

    std::cout << "\n--- I/O-BOUND OPERATIONS (ASIO's domain) ---" << std::endl;
    std::cout << "• Waiting for network data (recv/send)" << std::endl;
    std::cout << "• Reading/writing files" << std::endl;
    std::cout << "• Waiting for timers to expire" << std::endl;
    std::cout << "• Database queries" << std::endl;
    std::cout << "• HTTP requests/responses" << std::endl;
    std::cout << "\nCharacteristics:" << std::endl;
    std::cout << " → Thread spends most time WAITING" << std::endl;
    std::cout << " → CPU is mostly idle" << std::endl;
    std::cout << " → One thread can handle thousands of operations" << std::endl;
    std::cout << " → Need event multiplexing (epoll/IOCP)" << std::endl;

    std::cout << "\n--- CPU-BOUND OPERATIONS (std::async/thread domain) ---" << std::endl;
```

```
    std::cout << "• Image processing" << std::endl;
    std::cout << "• Video encoding" << std::endl;
    std::cout << "• Mathematical computations" << std::endl;
    std::cout << "• Data compression" << std::endl;
    std::cout << "• Machine learning inference" << std::endl;
    std::cout << "\nCharacteristics:" << std::endl;
    std::cout << " → Thread spends most time COMPUTING" << std::endl;
    std::cout << " → CPU is fully utilized" << std::endl;
    std::cout << " → Need multiple threads/cores for parallelism" << std::endl;
    std::cout << " → Thread count  CPU core count for best performance" << std::endl;
}

// ==================================================================
// 2. SIMULATED ASIO-STYLE TIMER vs std::this_thread::sleep_for
// ==================================================================

// Simulate ASIO timer callback pattern
class SimulatedAsioTimer {
private:
    std::chrono::milliseconds duration;
    std::function<void()> callback;
    std::jthread timer_thread;

public:
    SimulatedAsioTimer(std::chrono::milliseconds ms) : duration(ms) {}

    // ASIO-style async_wait (non-blocking)
    void async_wait(std::function<void()> handler) {
        callback = std::move(handler);

        // Start timer in background (simulates event loop)
        timer_thread = std::jthread([this]() {
            std::this_thread::sleep_for(duration);
            if (callback) {
                callback();
            }
        });
    }

    void cancel() {
        // In real ASIO, this would cancel the pending operation
        timer_thread.request_stop();
    }
};

void example_async_timer_patterns() {
    std::cout << "\n=== 2. ASYNC TIMER PATTERNS ===" << std::endl;

    std::cout << "\n--- Pattern 1: Blocking (std::this_thread::sleep_for) ---" << std::endl;
    auto start = std::chrono::steady_clock::now();
    std::cout << "Starting blocking sleep..." << std::endl;
```

```
    std::this_thread::sleep_for(100ms);
    std::cout << "Sleep completed (thread was blocked)" << std::endl;
    auto end = std::chrono::steady_clock::now();
    std::cout << "Time: " << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "ms" << std::endl;

    std::cout << "\n--- Pattern 2: Async Timer (ASIO-style) ---" << std::endl;
    start = std::chrono::steady_clock::now();
    std::cout << "Starting async timer..." << std::endl;

    SimulatedAsioTimer timer(100ms);
    timer.async_wait([start]() {
        auto end = std::chrono::steady_clock::now();
        std::cout << "Timer callback executed (thread was NOT blocked)" << std::endl;
        std::cout << "Time: " << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "ms" << std::endl;
    });

    std::cout << "async_wait returned immediately (non-blocking)" << std::endl;
    std::cout << "Main thread can do other work..." << std::endl;

    // Wait for timer to complete
    std::this_thread::sleep_for(150ms);

    std::cout << "\n KEY DIFFERENCE:" << std::endl;
    std::cout << " • Blocking: Thread frozen, can't do other work" << std::endl;
    std::cout << " • ASIO async: Thread free, callback executes when ready" << std::endl;
    std::cout << " • ASIO: One thread can manage thousands of timers" << std::endl;
}

// ==================================================================
// 3. CALLBACK-BASED vs FUTURE-BASED ASYNC
// ==================================================================

// Simulate async network read with callback (ASIO pattern)
void async_read_callback(const std::string& data, std::function<void(const std::string&)> callback) {
    std::jthread([data, callback = std::move(callback)]() {
        std::this_thread::sleep_for(50ms);  // Simulate I/O delay
        callback(data);
    }).detach();
}

// Future-based async read (std::async pattern)
std::future<std::string> async_read_future(const std::string& data) {
    return std::async(std::launch::async, [data]() {
        std::this_thread::sleep_for(50ms);  // Simulate I/O delay
        return data;
    });
```

```
}

void example_callback_vs_future() {
    std::cout << "\n=== 3. CALLBACK-BASED vs FUTURE-BASED ASYNC ===" << std::endl;

    std::cout << "\n--- Callback-Based (ASIO pattern) ---" << std::endl;
    std::cout << "Initiating async read with callback..." << std::endl;

    async_read_callback("Data from network", [](const std::string& result) {
        std::cout << "Callback received: " << result << std::endl;
    });

    std::cout << "async_read_callback returned immediately" << std::endl;
    std::cout << "Main thread continues..." << std::endl;
    std::this_thread::sleep_for(100ms);  // Wait for callback

    std::cout << "\n--- Future-Based (std::async pattern) ---" << std::endl;
    std::cout << "Initiating async read with future..." << std::endl;

    auto future = async_read_future("Data from computation");
    std::cout << "async_read_future returned immediately" << std::endl;
    std::cout << "Main thread can do work before getting result..." << std::endl;

    // Get result (blocks until ready)
    std::string result = future.get();
    std::cout << "Future result: " << result << std::endl;

    std::cout << "\ n TRADE-OFFS:" << std::endl;
    std::cout << "  Callbacks (ASIO):" << std::endl;
    std::cout << "    + Very efficient for I/O (no thread per operation)" << std::endl;
    std::cout << "    + Composable (chain callbacks)" << std::endl;
    std::cout << "    - Callback hell (deep nesting)" << std::endl;
    std::cout << "    - Error handling more complex" << std::endl;
    std::cout << "\n  Futures (std::async):" << std::endl;
    std::cout << "    + Simpler synchronous-style code" << std::endl;
    std::cout << "    + Easier error handling (exceptions)" << std::endl;
    std::cout << "    - Creates threads (expensive for many I/O ops)" << std::endl;
    std::cout << "    - Doesn't scale for thousands of concurrent operations" << std::endl;
}

// ==================================================================
// 4. EVENT LOOP CONCEPT (CORE OF ASIO)
// ==================================================================

// Simplified event loop to demonstrate the concept
class EventLoop {
private:
    std::vector<std::function<void()>> callbacks;
    std::mutex mutex;
```

```
    bool running = false;

public:
    void post(std::function<void()> callback) {
        std::lock_guard<std::mutex> lock(mutex);
        callbacks.push_back(std::move(callback));
    }

    void run() {
        running = true;
        std::cout << "Event loop started" << std::endl;

        while (running) {
            std::vector<std::function<void()>> work;

            {
                std::lock_guard<std::mutex> lock(mutex);
                work.swap(callbacks);
            }

            for (auto& callback : work) {
                callback();
            }

            if (work.empty() && !running) {
                break;
            }

            std::this_thread::sleep_for(10ms);  // Poll interval
        }

        std::cout << "Event loop stopped" << std::endl;
    }

    void stop() {
        running = false;
    }
};

void example_event_loop() {
    std::cout << "\n=== 4. EVENT LOOP CONCEPT (ASIO's HEART) ===" << std::endl
        ;

    EventLoop loop;

    // Run event loop in background thread
    std::jthread loop_thread([&loop]() {
        loop.run();
    });

    std::cout << "\nPosting work to event loop..." << std::endl;

    // Post work items (like ASIO handlers)
    loop.post([]() {
```

```
        std::cout << " → Handler 1 executed" << std::endl;
    });

    loop.post([]() {
        std::cout << " → Handler 2 executed" << std::endl;
    });

    loop.post([]() {
        std::cout << " → Handler 3 executed" << std::endl;
    });

    std::this_thread::sleep_for(50ms);

    loop.post([]() {
        std::cout << " → Handler 4 executed (posted later)" << std::endl;
    });

    std::this_thread::sleep_for(50ms);
    loop.stop();

    std::cout << "\n EVENT LOOP BENEFITS:" << std::endl;
    std::cout << " • Single thread processes all callbacks sequentially" << std::endl;
    std::cout << " • No race conditions within event loop" << std::endl;
    std::cout << " • Can handle thousands of I/O operations efficiently" << std::endl;
    std::cout << " • Real ASIO: Uses OS-level primitives (epoll/IOCP)" << std::endl;
}

// ================================================================
// 5. WHEN TO USE WHAT: DECISION MATRIX
// ================================================================

void example_decision_matrix() {
    std::cout << "\n=== 5. DECISION MATRIX: ASIO vs C++ STD ===" << std::endl;

    std::cout << "\                              n" << std::endl;
    std::cout << " Use Case                          ASIO        C++ Std      " << std::endl;
    std::cout << "                              " << std::endl;
    std::cout << " TCP/UDP networking                                      " << std::endl;
    std::cout << " HTTP server/client                                      " << std::endl;
    std::cout << " WebSocket                                               " << std::endl;
    std::cout << " Async timers                                            " << std::endl;
    std::cout << " Serial port I/O                                         " << std::endl;
    std::cout << " Many concurrent connections                             " << std::endl;
    std::cout << " Image processing                                        " <<
```

```
        std::endl;
    std::cout << "  Parallel computation                                    " << std::endl;
    std::cout << "  Thread synchronization                                  " << std::endl;
    std::cout << "  Protect shared data                          (mutex)  " << std::endl;
    std::cout << "  Wait for task completion                    (future)  " << std::endl;
    std::cout << "                                " << std::endl;

    std::cout << "\n COMBINED USAGE (BEST PRACTICE):" << std::endl;
    std::cout << "                                " << std::endl;
    std::cout << "  Component                             Solution                " << std::endl;
    std::cout << "                                " << std::endl;
    std::cout << "  Network I/O                   ASIO (io_context)               " << std::endl;
    std::cout << "  CPU-heavy processing          std::async / thread pool        " << std::endl;
    std::cout << "  Shared cache                  std::mutex                      " << std::endl;
    std::cout << "  Thread coordination           std::semaphore / std::latch     " << std::endl;
    std::cout << "  Rate limiting                 ASIO timers + std::semaphore    " << std::endl;
    std::cout << "                                " << std::endl;
}

// ==================================================================
// 6. SCALABILITY COMPARISON
// ==================================================================

void example_scalability() {
    std::cout << "\n=== 6. SCALABILITY: ASIO vs std::thread ===" << std::endl;

    std::cout << "\n--- Scenario: Handle 10,000 concurrent connections ---" << std::endl;

    std::cout << "\n APPROACH 1: One thread per connection (std::thread)" << std::endl;
    std::cout << " •  Need 10,000 threads" << std::endl;
    std::cout << " •  Each thread: ~1MB stack = 10GB memory" << std::endl;
    std::cout << " •  Context switching overhead" << std::endl;
    std::cout << " •  OS thread limit (~32k on Linux)" << std::endl;
    std::cout << " •  Result: System collapse  " << std::endl;

    std::cout << "\n APPROACH 2: Event-driven with ASIO" << std::endl;
    std::cout << " •  1-4 threads (typically)" << std::endl;
    std::cout << " •  Event multiplexing (epoll/IOCP)" << std::endl;
    std::cout << " •  Each connection: ~few KB state" << std::endl;
    std::cout << " •  Total memory: <100MB" << std::endl;
    std::cout << " •  Result: Handles load easily  " << std::endl;

```

```
    std::cout << "\n PERFORMANCE COMPARISON:" << std::endl;
    std::cout << "  Connections | std::thread | ASIO      " << std::endl;
    std::cout << "  ------------|-------------|----------" << std::endl;
    std::cout << "  10          | OK          | OK        " << std::endl;
    std::cout << "  100         | Struggling  | OK        " << std::endl;
    std::cout << "  1,000       | Failing     | OK        " << std::endl;
    std::cout << "  10,000      | Impossible  | Good      " << std::endl;
    std::cout << "  100,000     | N/A         | Possible  " << std::endl;
}

// ================================================================
// 7. INTEGRATION PATTERNS
// ================================================================

// Pattern: ASIO for I/O + std::async for CPU work
class HybridServer {
private:
    std::mutex shared_cache_mutex;
    std::vector<std::string> shared_cache;

public:
    // Simulated: ASIO receives network request
    void on_request_received(const std::string& data) {
        std::cout << "  [ASIO] Received request: " << data << std::endl;

        // CPU-intensive work: offload to std::async
        auto future = std::async(std::launch::async, [this, data]() {
            return process_data(data);
        });

        std::cout << "  [ASIO] Offloaded to worker thread, can handle more
           requests" << std::endl;

        // Get result (in real app, would be callback)
        std::string result = future.get();

        // ASIO would send response here
        std::cout << "  [ASIO] Sending response: " << result << std::endl;
    }

private:
    std::string process_data(const std::string& data) {
        // Simulate CPU-intensive work
        std::this_thread::sleep_for(50ms);
        std::string result = "Processed: " + data;

        // Update shared cache (use mutex for thread safety)
        {
            std::lock_guard<std::mutex> lock(shared_cache_mutex);
            shared_cache.push_back(result);
        }

        return result;
    }
```

```
};

void example_integration_patterns() {
    std::cout << "\n=== 7. INTEGRATION PATTERNS: ASIO + C++ STD ===" << std::endl;

    std::cout << "\nPattern: I/O with ASIO + CPU work with std::async" << std::endl;

    HybridServer server;

    // Simulate multiple requests
    server.on_request_received("Request-1");
    server.on_request_received("Request-2");
    server.on_request_received("Request-3");

    std::cout << "\n HYBRID APPROACH:" << std::endl;
    std::cout << "  1. ASIO handles network I/O (event loop)" << std::endl;
    std::cout << "  2. Offload CPU work to std::async" << std::endl;
    std::cout << "  3. Use std::mutex for shared data" << std::endl;
    std::cout << "  4. ASIO remains responsive during CPU work" << std::endl;
}

// ==================================================================
// 8. C++20 COROUTINES + ASIO (THE FUTURE)
// ==================================================================

void example_coroutines_concept() {
    std::cout << "\n=== 8. C++20 COROUTINES + ASIO (MODERN PATTERN) ===" << std::endl;

    std::cout << "\n--- Without Coroutines (Callback Hell) ---" << std::endl;
    std::cout << "async_read(socket, buffer, [](error, bytes) {" << std::endl;
    std::cout << "    async_write(socket, data, [](error, bytes) {" << std::endl;
    std::cout << "        async_read(socket, response, [](error, bytes) {" << std::endl;
    std::cout << "            // 3 levels deep already!" << std::endl;
    std::cout << "        });" << std::endl;
    std::cout << "    });" << std::endl;
    std::cout << "});" << std::endl;

    std::cout << "\n--- With C++20 Coroutines (Sequential Code) ---" << std::endl;
    std::cout << "asio::awaitable<void> handle_connection() {" << std::endl;
    std::cout << "    auto bytes = co_await async_read(socket, buffer);  // Looks sync!" << std::endl;
    std::cout << "    co_await async_write(socket, data);                // But it's async" << std::endl;
    std::cout << "    co_await async_read(socket, response);             // No nesting!" << std::endl;
    std::cout << "}" << std::endl;

    std::cout << "\n COROUTINES BENEFITS:" << std::endl;
```

```
    std::cout << " •  Write async code that looks synchronous" << std::endl;
    std::cout << " •  No callback nesting (no hell)" << std::endl;
    std::cout << " •  Exception handling works naturally (try/catch)" << std::endl;
    std::cout << " •  Still non-blocking (efficient as callbacks)" << std::endl;
    std::cout << " •  ASIO has full coroutine support (co_await)" << std::endl;
}

// ================================================================
// 9. REAL-WORLD ARCHITECTURE EXAMPLE
// ================================================================

void example_architecture() {
    std::cout << "\n=== 9. REAL-WORLD ARCHITECTURE: WEB SERVER ===" << std::endl;

    std::cout << "\                        n" << std::endl;
    std::cout << "            High-Performance Web Server          " << std::endl;
    std::cout << "                       " << std::endl;
    std::cout << "                                                    " << std::endl;
    std::cout << "                                 " << std::endl;
    std::cout << "                                                    " << std::endl;
    std::cout << "                                                    " << std::endl;
    std::cout << "                                 " << std::endl;
    std::cout << "   ASIO          C++ Std         C++ Std         " << std::endl;
    std::cout << "                                                   " << std::endl;
    std::cout << "  Layer           Layer           Layer          " << std::endl;
    std::cout << "                                 " << std::endl;

    std::cout << "\n ASIO LAYER:" << std::endl;
    std::cout << " •  io_context (1-4 threads)" << std::endl;
    std::cout << " •  Accept connections (10k+ concurrent)" << std::endl;
    std::cout << " •  Parse HTTP requests" << std::endl;
    std::cout << " •  Send HTTP responses" << std::endl;
    std::cout << " •  Timers for keepalive/timeout" << std::endl;

    std::cout << "\n CPU PROCESSING LAYER:" << std::endl;
    std::cout << " •  std::jthread pool (N = CPU cores)" << std::endl;
    std::cout << " •  Process business logic" << std::endl;
    std::cout << " •  Image/video processing" << std::endl;
    std::cout << " •  Compute-intensive tasks" << std::endl;
    std::cout << " •  Post results back to ASIO" << std::endl;

    std::cout << "\n SYNCHRONIZATION LAYER:" << std::endl;
    std::cout << " •  std::mutex: Protect shared cache" << std::endl;
```

```
    std::cout << " •  std::shared_mutex: Read-write locks" << std::endl;
    std::cout << " •  std::atomic: Lock-free counters" << std::endl;
    std::cout << " •  std::semaphore: Rate limiting" << std::endl;
    std::cout << " •  std::latch: Wait for initialization" << std::endl;

    std::cout << "\n BENEFITS OF HYBRID APPROACH:" << std::endl;
    std::cout << "    Scales to 100k+ connections (ASIO)" << std::endl;
    std::cout << "    Utilizes all CPU cores (std::jthread)" << std::endl;
    std::cout << "    Thread-safe shared state (std::mutex)" << std::endl;
    std::cout << "    Best of both worlds!" << std::endl;
}

// ==================================================================
// 10. SUMMARY AND RECOMMENDATIONS
// ==================================================================

void example_summary() {
    std::cout << "\n=== 10. SUMMARY: ASIO vs C++ STANDARD LIBRARY ===" << std
        ::endl;

    std::cout << "\n KEY TAKEAWAYS:" << std::endl;
    std::cout << "\n1. DIFFERENT PROBLEMS, DIFFERENT TOOLS:" << std::endl;
    std::cout << " •   ASIO: Asynchronous I/O (network, timers, file I/O)" <<
        std::endl;
    std::cout << " •   C++ Std: Threading, CPU parallelism, synchronization"
        << std::endl;
    std::cout << " •   They COMPLEMENT each other, not compete!" << std::endl;

    std::cout << "\n2. WHEN TO USE ASIO:" << std::endl;
    std::cout << "     Building network servers/clients" << std::endl;
    std::cout << "     Need to handle thousands of concurrent connections" <<
        std::endl;
    std::cout << "     I/O-bound operations (waiting > computing)" << std::
        endl;
    std::cout << "     Event-driven architecture" << std::endl;
    std::cout << "     Cross-platform async I/O" << std::endl;

    std::cout << "\n3. WHEN TO USE C++ STANDARD LIBRARY:" << std::endl;
    std::cout << "     CPU-bound parallel computations" << std::endl;
    std::cout << "     Thread-safe data structures" << std::endl;
    std::cout << "     Simple threading needs" << std::endl;
    std::cout << "     Waiting for computation results (futures)" << std::endl
        ;
    std::cout << "     Thread coordination (semaphores, latches)" << std::endl
        ;

    std::cout << "\n4. BEST PRACTICES (2026):" << std::endl;
    std::cout << "     Use ASIO for I/O multiplexing" << std::endl;
    std::cout << "     Use std::async/thread_pool for CPU work" << std::endl;
    std::cout << "     Use std::mutex for thread-safe shared data" << std::
        endl;
    std::cout << "     Use C++20 coroutines with ASIO (co_await)" << std::endl
        ;
    std::cout << "     Combine: ASIO event loop + std threading primitives" <<
```

```
        std::endl;

    std::cout << "\n5. GETTING STARTED:" << std::endl;
    std::cout << " •   Install: Standalone ASIO (https://think-async.com/Asio/)" << std::endl;
    std::cout << " •   Or: Boost.ASIO (apt install libboost-all-dev)" << std::endl;
    std::cout << " •   Learn: Start with timers, then sockets" << std::endl;
    std::cout << " •   Pattern: Event loop in 1-4 threads" << std::endl;
    std::cout << " •   Modern: Use ASIO with C++20 coroutines" << std::endl;

    std::cout << "\n6. COMMON MISTAKES:" << std::endl;
    std::cout << "     Using std::thread for thousands of connections" << std::endl;
    std::cout << "     Using ASIO for CPU-intensive work" << std::endl;
    std::cout << "     Blocking ASIO event loop with long operations" << std::endl;
    std::cout << "     Not protecting shared data between ASIO and worker threads" << std::endl;

    std::cout << "\ n GOLDEN RULE:" << std::endl;
    std::cout << "   \"ASIO for waiting, std::thread for computing\"" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  ASIO AND MODERN C++ CONCURRENCY: COMPREHENSIVE GUIDE" << std::endl;
    std::cout << " ==================================================================" << std::endl;
    std::cout << "\nNote: This demonstrates concepts using standard C++ features." << std::endl;
    std::cout << "For actual ASIO usage, install from: https://think-async.com/Asio/" << std::endl;

    example_io_vs_cpu_bound();
    example_async_timer_patterns();
    example_callback_vs_future();
    example_event_loop();
    example_decision_matrix();
    example_scalability();
    example_integration_patterns();
    example_coroutines_concept();
    example_architecture();
    example_summary();

```

```
    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  FINAL VERDICT: ASIO vs C++ STANDARD LIBRARY" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\n WINNER: BOTH (They're Partners, Not Competitors!)" << std::endl;

    std::cout << "\n COMPARISON TABLE:" << std::endl;
    std::cout << "\                          n" << std::endl;
    std::cout << "  Feature                ASIO           C++ Std       " << std::endl;
    std::cout << "                          " << std::endl;
    std::cout << "  Network I/O                                  " << std::endl;
    std::cout << "  Async Timers                                 " << std::endl;
    std::cout << "  Scalability                                 " << std::endl;
    std::cout << "  CPU Parallelism                             " << std::endl;
    std::cout << "  Thread Safety                                " << std::endl;
    std::cout << "  Ease of Use                                 " << std::endl;
    std::cout << "  Learning Curve       Steep         Moderate     " << std::endl;
    std::cout << "  Standardization      Not yet         Standard      " << std::endl;
    std::cout << "  Maturity             20+ years       Modern C++     " << std::endl;
    std::cout << "                          " << std::endl;

    std::cout << "\n LEARNING PATH:" << std::endl;
    std::cout << "  1. Master C++ std threading first (mutex, thread, async)" << std::endl;
    std::cout << "  2. Understand I/O vs CPU bound problems" << std::endl;
    std::cout << "  3. Learn ASIO basics (timers, then sockets)" << std::endl;
    std::cout << "  4. Study ASIO examples and patterns" << std::endl;
    std::cout << "  5. Explore ASIO + C++20 coroutines" << std::endl;
    std::cout << "  6. Build hybrid systems (ASIO + std)" << std::endl;

    std::cout << "\n NEXT STEPS:" << std::endl;
    std::cout << " •  Install Standalone ASIO or Boost.ASIO" << std::endl;
    std::cout << " •  Try ASIO timer examples" << std::endl;
    std::cout << " •  Build a simple TCP echo server" << std::endl;
    std::cout << " •  Experiment with coroutines (co_await)" << std::endl;
    std::cout << " •  Study beast (HTTP on top of ASIO)" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

5 Source Code: AsioMultipleContexts.cpp

File: `src/AsioMultipleContexts.cpp`
Repository: View on GitHub

```cpp
// AsioMultipleContexts.cpp
// Comprehensive educational example of using multiple io_context objects in standalone ASIO
// Demonstrates LAN/WAN separation, thread pooling, and priority-based I/O handling

#include <iostream>
#include <thread>
#include <vector>
#include <memory>
#include <chrono>
#include <string>
#include <atomic>
#include <iomanip>
#include <sstream>
#include <mutex>
#include <functional>
#include <condition_variable>

// Note: This example uses simulated ASIO patterns for educational purposes
// For actual ASIO usage, install standalone ASIO: https://think-async.com/Asio/
// Then: #include <asio.hpp>
// And replace SimulatedAsio classes with real asio::io_context, asio::steady_timer, etc.

using namespace std::chrono_literals;

// ==============================================================================
// SIMULATED ASIO CLASSES (for demonstration without requiring ASIO installation)
// ==============================================================================
// In real code, replace these with actual ASIO classes

namespace SimulatedAsio {

// Simulate asio::io_context
class io_context {
private:
    std::atomic<bool> stopped_{false};
    std::atomic<int> work_count_{0};
    std::vector<std::function<void()>> pending_work_;
    std::mutex work_mutex_;
    std::condition_variable work_cv_;
    std::string name_;  // For debugging
```

```

public:
    explicit io_context(const std::string& name = "io_context") : name_(name)
        {}

    // Main event loop - blocks until no more work
    void run() {
        std::cout << "[" << name_ << "] Thread " << std::this_thread::get_id()
                  << " calling run()\n";

        while (!stopped_) {
            std::function<void()> work;

            {
                std::unique_lock<std::mutex> lock(work_mutex_);
                work_cv_.wait_for(lock, 50ms, [this] {
                    return !pending_work_.empty() || stopped_;
                });

                if (stopped_ && pending_work_.empty()) {
                    break;
                }

                if (!pending_work_.empty()) {
                    work = std::move(pending_work_.back());
                    pending_work_.pop_back();
                }
            }

            if (work) {
                work();
                --work_count_;
            }

            // If no more work, exit
            if (work_count_ == 0 && pending_work_.empty()) {
                break;
            }
        }

        std::cout << "[" << name_ << "] Thread " << std::this_thread::get_id()
                  << " exiting run()\n";
    }

    // Post work to the io_context
    void post(std::function<void()> handler) {
        {
            std::lock_guard<std::mutex> lock(work_mutex_);
            pending_work_.push_back(std::move(handler));
            ++work_count_;
        }
        work_cv_.notify_one();
    }

```

```
    // Stop the io_context
    void stop() {
        stopped_ = true;
        work_cv_.notify_all();
    }

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

// Simulate asio::steady_timer
class steady_timer {
private:
    io_context& io_;
    std::chrono::milliseconds duration_;
    std::jthread timer_thread_;

public:
    steady_timer(io_context& io, std::chrono::milliseconds ms)
        : io_(io), duration_(ms) {}

    void async_wait(std::function<void()> handler) {
        // Start timer in background, then post handler to io_context
        timer_thread_ = std::jthread([this, h = std::move(handler)]() {
            std::this_thread::sleep_for(duration_);
            io_.post(h);  // Post to io_context when timer expires
        });
    }
};

// Work guard to keep io_context alive
template<typename Executor>
class executor_work_guard {
private:
    io_context& io_;

public:
    explicit executor_work_guard(io_context& io) : io_(io) {}
    ~executor_work_guard() = default;
};

} // namespace SimulatedAsio

// ==========================================================================
// SECTION 1: Basic Multiple io_context Pattern
// ==========================================================================

void demonstrate_basic_multiple_contexts() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 1. Basic Multiple io_context Pattern ===\n";
    std::cout << std::string(70, '=') << "\n\n";
```

```

    std::cout << "Concept: Two separate io_context instances, each with its own thread\n";
    std::cout << "Use case: Isolate different types of I/O operations\n\n";

    SimulatedAsio::io_context io1("io_context_1");
    SimulatedAsio::io_context io2("io_context_2");

    // Post work to both contexts
    io1.post([]() {
        std::cout << "  Work executing on io_context_1\n";
    });

    io2.post([]() {
        std::cout << "  Work executing on io_context_2\n";
    });

    // Run each io_context in a separate thread
    std::thread t1([&io1]() { io1.run(); });
    std::thread t2([&io2]() { io2.run(); });

    t1.join();
    t2.join();

    std::cout << "\n Both contexts completed independently\n";
    std::cout << "  Each thread called run() on its assigned io_context\n";
}

// ============================================================================
// SECTION 2: Thread Pool Pattern (Multiple threads, one io_context)
// ============================================================================

void demonstrate_thread_pool_pattern() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 2. Thread Pool Pattern (Multiple threads share one io_context) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Concept: Multiple threads call run() on the SAME io_context\n";
    std::cout << "Benefit: Work is automatically distributed across threads\n\n";

    SimulatedAsio::io_context io("thread_pool");

    // Post 6 tasks
    for (int i = 1; i <= 6; ++i) {
        io.post([i]() {
            std::cout << "  Task " << i << " executing on thread "
                      << std::this_thread::get_id() << "\n";
```

```
            std::this_thread::sleep_for(100ms);  // Simulate work
        });
    }

    // Create thread pool: 3 threads all calling run() on the same io_context
    std::vector<std::thread> thread_pool;
    for (int i = 0; i < 3; ++i) {
        thread_pool.emplace_back([&io]() {
            io.run();
        });
    }

    // Wait for all threads
    for (auto& t : thread_pool) {
        t.join();
    }

    std::cout << "\n 6 tasks distributed across 3 threads automatically\n";
    std::cout << "  This is the standard ASIO thread pool pattern\n";
}

// ==========================================================================
// SECTION 3: LAN vs WAN Separation (The Main Example!)
// ==========================================================================

// Simulate a network connection
class Connection {
private:
    std::string name_;
    std::string type_;  // "LAN" or "WAN"
    int request_count_;

public:
    Connection(const std::string& name, const std::string& type)
        : name_(name), type_(type), request_count_(0) {}

    void handle_request(int request_id) {
        ++request_count_;
        std::cout << "  [" << type_ << " - " << name_ << "] Processing request #"
                  << request_id << " (total: " << request_count_ << ")\n";

        // Simulate processing time
        if (type_ == "LAN") {
            std::this_thread::sleep_for(50ms);   // Fast LAN processing
        } else {
            std::this_thread::sleep_for(150ms);  // Slower WAN processing
        }
    }

```

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

void demonstrate_lan_wan_separation() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 3. LAN vs WAN Separation (Dual io_context) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Scenario: Web server handling both LAN and WAN traffic\n";
    std::cout << "Architecture:\n";
    std::cout << " • io_lan → Fast local network clients (prioritized)\n";
    std::cout << " • io_wan → Slower internet clients (lower priority)\n";
    std::cout << "Benefits:\n";
    std::cout << " • LAN traffic never blocked by slow WAN connections\n";
    std::cout << " • Can apply different rate limits per context\n";
    std::cout << " • Independent thread pools for each network\n\n";

    // Create two separate io_context instances
    SimulatedAsio::io_context io_lan("io_LAN");
    SimulatedAsio::io_context io_wan("io_WAN");

    // Create connections
    auto lan_client1 = std::make_shared<Connection>("InternalAPI", "LAN");
    auto lan_client2 = std::make_shared<Connection>("Dashboard", "LAN");
    auto wan_client1 = std::make_shared<Connection>("PublicAPI", "WAN");
    auto wan_client2 = std::make_shared<Connection>("MobileApp", "WAN");

    std::cout << "Posting work to both contexts...\n\n";

    // Post LAN work (fast, high priority)
    for (int i = 1; i <= 3; ++i) {
        io_lan.post([lan_client1, i]() {
            lan_client1->handle_request(i);
        });

        io_lan.post([lan_client2, i]() {
            lan_client2->handle_request(i);
        });
    }

    // Post WAN work (slower, lower priority)
    for (int i = 1; i <= 3; ++i) {
        io_wan.post([wan_client1, i]() {
            wan_client1->handle_request(i);
        });

        io_wan.post([wan_client2, i]() {
            wan_client2->handle_request(i);
        });
    }

    // Run LAN with 2 threads (higher capacity for fast traffic)
    std::vector<std::thread> lan_threads;
    lan_threads.emplace_back([&io_lan]() { io_lan.run(); });
```

```
    lan_threads.emplace_back([&io_lan]() { io_lan.run(); });

    // Run WAN with 1 thread (lower capacity, prevent resource exhaustion)
    std::thread wan_thread([&io_wan]() { io_wan.run(); });

    // Wait for completion
    for (auto& t : lan_threads) {
        t.join();
    }
    wan_thread.join();

    std::cout << "\n LAN traffic processed with 2 threads (fast completion)\n";
    std::cout << "  WAN traffic processed with 1 thread (controlled rate)\n";
    std::cout << "  Neither context blocked the other\n";
}

// ==========================================================================
// SECTION 4: Priority-Based io_context (Critical vs Normal)
// ==========================================================================

class PriorityServer {
private:
    SimulatedAsio::io_context io_critical_;
    SimulatedAsio::io_context io_normal_;
    std::atomic<int> critical_count_{0};
    std::atomic<int> normal_count_{0};

public:
    PriorityServer() : io_critical_("io_CRITICAL"), io_normal_("io_NORMAL") {}

    void handle_critical_request(const std::string& request) {
        io_critical_.post([this, request]() {
            int id = ++critical_count_;
            std::cout << "  [CRITICAL] Request " << id << ": " << request
                      << " (thread " << std::this_thread::get_id() << ")\n";
            std::this_thread::sleep_for(30ms);  // Fast processing
        });
    }

    void handle_normal_request(const std::string& request) {
        io_normal_.post([this, request]() {
            int id = ++normal_count_;
            std::cout << "  [NORMAL]   Request " << id << ": " << request
                      << " (thread " << std::this_thread::get_id() << ")\n";
            std::this_thread::sleep_for(80ms);  // Slower processing
        });
    }

    void run() {
```

```
        // Critical requests get dedicated thread
        std::thread critical_thread([this]() {
            io_critical_.run();
        });

        // Normal requests share a thread pool
        std::thread normal_thread([this]() {
            io_normal_.run();
        });

        critical_thread.join();
        normal_thread.join();
    }
};

void demonstrate_priority_separation() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 4. Priority-Based Separation (Critical vs Normal) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Scenario: System with time-critical and regular operations\n";
    std::cout << "Examples:\n";
    std::cout << " •  Critical: Heartbeats, alarms, safety-critical commands\n";
    std::cout << " •  Normal:   Data logging, statistics, background tasks\n\n";

    PriorityServer server;

    // Mix critical and normal requests
    server.handle_critical_request("Emergency Stop");
    server.handle_normal_request("Log statistics");
    server.handle_critical_request("Heartbeat");
    server.handle_normal_request("Update dashboard");
    server.handle_critical_request("Safety check");
    server.handle_normal_request("Backup data");

    std::cout << "\nProcessing requests...\n\n";
    server.run();

    std::cout << "\ n Critical requests never delayed by normal traffic\n";
    std::cout << "  Each context can have different threading strategies\n";
}

// ==========================================================================
// SECTION 5: Service-Type Separation (Clients, Database, Logging)
// ==========================================================================

```

```cpp
void demonstrate_service_type_separation() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 5. Service-Type Separation ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Architecture: One io_context per service type\n\n";

    SimulatedAsio::io_context io_clients("io_CLIENTS");
    SimulatedAsio::io_context io_database("io_DATABASE");
    SimulatedAsio::io_context io_logging("io_LOGGING");

    // Client connections (high concurrency)
    for (int i = 1; i <= 4; ++i) {
        io_clients.post([i]() {
            std::cout << "  [CLIENTS]  Handling client " << i << " connection\n";
            std::this_thread::sleep_for(60ms);
        });
    }

    // Database operations (controlled concurrency)
    for (int i = 1; i <= 2; ++i) {
        io_database.post([i]() {
            std::cout << "  [DATABASE] Executing query " << i << "\n";
            std::this_thread::sleep_for(100ms);
        });
    }

    // Logging operations (background, low priority)
    for (int i = 1; i <= 3; ++i) {
        io_logging.post([i]() {
            std::cout << "  [LOGGING]  Writing log entry " << i << "\n";
            std::this_thread::sleep_for(40ms);
        });
    }

    std::cout << "Starting service threads...\n\n";

    // Each service gets its own thread configuration
    std::thread client_thread([&]() { io_clients.run(); });
    std::thread db_thread([&]() { io_database.run(); });
    std::thread log_thread([&]() { io_logging.run(); });

    client_thread.join();
    db_thread.join();
    log_thread.join();

    std::cout << "\n Services isolated from each other\n";
    std::cout << "  Database load doesn't impact client connections\n";
    std::cout << "  Logging doesn't block critical services\n";
}

// ==========================================================================
```

```cpp
// SECTION 6: When to Call run() - Summary
// ==========================================================================

void demonstrate_run_call_patterns() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 6. When to Call io_context.run() - Complete Guide ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "PATTERN 1: Single Thread per io_context\n";
    std::cout << "                    \n";
    std::cout << "  asio::io_context io1, io2;\n";
    std::cout << "  std::thread t1([&]() { io1.run(); });  // Thread 1 → io1\n";
    std::cout << "  std::thread t2([&]() { io2.run(); });  // Thread 2 → io2\n";
    std::cout << "  Use: Separate concerns (LAN/WAN, Client/DB)\n\n";

    std::cout << "PATTERN 2: Thread Pool (Multiple threads, one io_context)\n";
    std::cout << "                            \n";
    std::cout << "  asio::io_context io;\n";
    std::cout << "  std::thread t1([&]() { io.run(); });  // Thread 1 → io\n";
    std::cout << "  std::thread t2([&]() { io.run(); });  // Thread 2 → io\n";
    std::cout << "  std::thread t3([&]() { io.run(); });  // Thread 3 → io\n";
    std::cout << "  Use: Distribute work across threads automatically\n\n";

    std::cout << "PATTERN 3: Main Thread runs io_context\n";
    std::cout << "                   \n";
    std::cout << "  asio::io_context io;\n";
    std::cout << "  // Post all work...\n";
    std::cout << "  io.run();  // Main thread blocks here\n";
    std::cout << "  Use: Simple single-threaded servers\n\n";

    std::cout << "PATTERN 4: Keep io_context Alive\n";
    std::cout << "                \n";
    std::cout << "  asio::io_context io;\n";
    std::cout << "  auto guard = asio::make_work_guard(io);\n";
    std::cout << "  std::thread t([&]() { io.run(); });  // Won't exit immediately\n";
    std::cout << "  // ... post work later ...\n";
    std::cout << "  guard.reset();  // Allow run() to exit when work done\n";
    std::cout << "  Use: Long-running services with dynamic work\n\n";

    std::cout << "KEY RULES:\n";
    std::cout << "     \n";
    std::cout << "  1. run() BLOCKS until no more work (or stopped)\n";
    std::cout << "  2. Multiple threads CAN call run() on SAME io_context (thread pool)\n";
    std::cout << "  3. One thread should NOT call run() on MULTIPLE io_contexts\n";
```

```
    std::cout << "  4. run() returns when work queue is empty (use work_guard to prevent)\n";
    std::cout << "  5. Call stop() to force run() to exit early\n\n";
}

// ============================================================================
// SECTION 7: Real-World Example - Microservice Gateway
// ============================================================================

void demonstrate_microservice_gateway() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 7. Real-World Example: Microservice API Gateway ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Architecture:\n";
    std::cout << "           \n";
    std::cout << "      io_frontend ←    Public internet clients (rate limited)\n";
    std::cout << "      (1 thread)    \n";
    std::cout << "           \n";
    std::cout << " ↓         \n";
    std::cout << "           \n";
    std::cout << "      io_internal ←    Internal microservices (fast LAN)\n";
    std::cout << "      (4 threads)   \n";
    std::cout << "           \n";
    std::cout << " ↓         \n";
    std::cout << "           \n";
    std::cout << "      io_database ←    Database connections (controlled)\n";
    std::cout << "      (2 threads)   \n";
    std::cout << "           \n\n";

    SimulatedAsio::io_context io_frontend("FRONTEND");
    SimulatedAsio::io_context io_internal("INTERNAL");
    SimulatedAsio::io_context io_database("DATABASE");

    // Simulate incoming requests
    std::cout << "Processing requests...\n\n";

    io_frontend.post([]() {
        std::cout << "  [FRONTEND] Public client → Authentication\n";
        std::this_thread::sleep_for(50ms);
    });

    io_internal.post([]() {
        std::cout << "  [INTERNAL] Calling user-service microservice\n";
        std::this_thread::sleep_for(30ms);
    });

    io_internal.post([]() {
        std::cout << "  [INTERNAL] Calling order-service microservice\n";
```

```
        std::this_thread::sleep_for(30ms);
    });

    io_database.post([]() {
        std::cout << "  [DATABASE] Query user preferences\n";
        std::this_thread::sleep_for(80ms);
    });

    // Thread allocation reflects importance and load
    std::thread frontend_thread([&]() { io_frontend.run(); });

    std::vector<std::thread> internal_threads;
    for (int i = 0; i < 2; ++i) {  // 2 threads for internal services
        internal_threads.emplace_back([&]() { io_internal.run(); });
    }

    std::thread db_thread([&]() { io_database.run(); });

    frontend_thread.join();
    for (auto& t : internal_threads) { t.join(); }
    db_thread.join();

    std::cout << "\n Frontend protected from internal service overload\n";
    std::cout << "  Internal services have high concurrency (4 threads)\n";
    std::cout << "  Database access controlled (2 threads max)\n";
    std::cout << "  Each layer can be scaled independently\n";
}

// ==========================================================================
// SECTION 8: Common Pitfalls and Best Practices
// ==========================================================================

void demonstrate_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 8. Common Pitfalls and Best Practices ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  PITFALL 1: run() exits immediately (no work posted)\n";
    std::cout << "    asio::io_context io;\n";
    std::cout << "    std::thread t([&]() { io.run(); });  // ← Exits immediately!\n";
    std::cout << "    io.post([]() { /* ... */ });  // ← Too late, thread already exited\n\n";
    std::cout << "  FIX: Post work BEFORE starting thread, or use work_guard\n\n";

    std::cout << "  PITFALL 2: One thread calling run() on multiple contexts\n";
    std::cout << "    io1.run();  // Blocks until io1 is done\n";
    std::cout << "    io2.run();  // ← io2 never runs until io1 finishes!\n\n";
```

```
    std::cout << "  FIX: Use separate threads for each io_context\n\n";

    std::cout << "  PITFALL 3: Forgetting to join threads\n";
    std::cout << "    std::thread t([&]() { io.run(); });\n";
    std::cout << "    // ← main() exits, std::terminate() called!\n\n";
    std::cout << "  FIX: Always join or detach threads\n\n";

    std::cout << "  PITFALL 4: Sharing mutable state without synchronization\n";
    std::cout << "    int counter = 0;  // ← Multiple threads accessing!\n";
    std::cout << "    io.post([&]() { ++counter; });  // ← RACE CONDITION\n\n";
    std::cout << "  FIX: Use std::atomic or std::mutex for shared state\n\n";

    std::cout << "BEST PRACTICES:\n";
    std::cout << "        \n";
    std::cout << "  1. Use multiple io_contexts to SEPARATE concerns (not for parallelism)\n";
    std::cout << "  2. For parallelism, use THREAD POOL (multiple threads, one io_context)\n";
    std::cout << "  3. Number of threads  number of CPU cores (for CPU-bound work)\n";
    std::cout << "  4. For I/O-bound work, can have many more threads than cores\n";
    std::cout << "  5. Always use strand for sequential execution across threads\n";
    std::cout << "  6. Profile before optimizing - one io_context is often sufficient\n\n";
}

// ============================================================================
// MAIN FUNCTION
// ============================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "          ASIO Multiple io_context - Educational Examples                    \n";
    std::cout << "                                                                                \n";
    std::cout << "  Learn when and how to use multiple io_context objects in ASIO    \n";
    std::cout << "                                    \n";

    demonstrate_basic_multiple_contexts();
    demonstrate_thread_pool_pattern();
    demonstrate_lan_wan_separation();         // ← Your requested example!
    demonstrate_priority_separation();
    demonstrate_service_type_separation();
```

```
    demonstrate_run_call_patterns();
    demonstrate_microservice_gateway();
    demonstrate_best_practices();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << "\nKEY TAKEAWAY:\n";
    std::cout << "  Multiple io_context = SEPARATION (isolate different
        concerns)\n";
    std::cout << "  Multiple threads on one io_context = PARALLELISM (
        distribute work)\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

6 Source Code: BinarySearch.cpp

File: `src/BinarySearch.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <vector>
#include <iterator>

class Solution {
public:
    int binarySearch(std::vector<int> array, int target)   {
        std::vector<int>::iterator     vecBIT = array.begin();
        std::vector<int>::iterator     vecEIT = array.end();
        std::vector<int>::iterator     vecIT = vecBIT + (vecEIT - vecBIT) / 2;

        //-------/\----target---------/\-------target -------/\-----------------------
        //      vecBIT                vecIT                 vecEIT

        while (vecBIT <= vecEIT) {
              if (*vecIT == target) {
                  return std::distance(array.begin(), vecIT); // Return distance between iterators
              } else if (*vecIT < target) {
                  vecBIT = vecIT + 1;
                  vecIT = vecBIT + (vecEIT - vecBIT) / 2;
              } else {
                  vecEIT = vecIT - 1;
                  vecIT = vecBIT + (vecEIT - vecBIT) / 2;
              }
        }

        if (vecBIT > vecEIT) return -1;
    }

    int binarySearchRecursion(std::vector<int> array, int vecBIT, int vecEIT, int target) {

        if (vecBIT <= vecEIT) {
            int    vecIT = vecBIT + (vecEIT - vecBIT) / 2;

            if (array[vecIT] == target)
              return vecIT;
            else if (array[vecIT] < target)
              return binarySearchRecursion(array, vecIT + 1, vecEIT, target);
            else
              return binarySearchRecursion(array, vecBIT, vecIT - 1, target);
        }
        else
            return -1;
    }
};

```

```
int main () {
    Solution testSolution;
    int n = 3;
    std::vector<int> searchArray(1); // initialize vector of n elements with 0

    // push_back another n - 1 elements
    for (int i = 1; i < n; i ++)
        searchArray.push_back(i);

    // n element in vector now
    for (std::vector<int>::const_iterator vecIT = searchArray.begin(); vecIT <
        searchArray.end(); vecIT ++) {
        std::cout << *vecIT << std::endl;
    }
    std::cout << std::endl;

    std::cout << std::distance(searchArray.begin(), searchArray.end()) << "\n"
        << std::endl;

    for (int i = 0; i <= n; i ++) {
        std::cout << testSolution.binarySearch(searchArray, i) << std::endl;
        std::cout << testSolution.binarySearchRecursion(searchArray, 0, n, i)
            << std::endl;
    }

    return 0;
}
```

7 Source Code: CRTPvsVirtualFunctions.cpp

File: `src/CRTPvsVirtualFunctions.cpp`
Repository: View on GitHub

```cpp
/*
 * CRTP vs Virtual Functions: Static vs Dynamic Polymorphism
 *
 * This example demonstrates:
 * 1. How virtual functions work (vtable mechanism)
 * 2. Performance overhead of virtual functions
 * 3. CRTP (Curiously Recurring Template Pattern) as zero-overhead alternative
 * 4. When to use each approach
 * 5. Benchmarking and trade-offs
 */

#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <chrono>
#include <iomanip>

// ==========================================================================
// SECTION 1: VIRTUAL FUNCTIONS AND VTABLE MECHANISM
// ==========================================================================

namespace virtual_functions {

/*
 * VTABLE (Virtual Function Table) EXPLANATION:
 *
 * When you use virtual functions, the compiler creates:
 *
 * 1. VTABLE (per class):
 *    - Static array of function pointers
 *    - One vtable per polymorphic class
 *    - Stored in read-only memory
 *
 * 2. VPTR (per object):
 *    - Hidden pointer member in each object
 *    - Points to the class's vtable
 *    - Added automatically by compiler
 *    - Size overhead: 8 bytes (64-bit system)
 *
 * MEMORY LAYOUT EXAMPLE:
 *
 * class Base {
 *     virtual void foo();
 *     int data;
```

```
 * };
 *
 * Object memory:  [vptr][data]
 *                   |
 *                   v
 * Vtable:        [&Base::foo][type_info*]
 *
 * CALL MECHANISM:
 * obj.foo() becomes:
 *   1. Load vptr from object
 *   2. Index into vtable
 *   3. Load function pointer
 *   4. Call through pointer
 *
 * PERFORMANCE COSTS:
 * • 2-3 extra memory loads (vptr + function pointer)
 * • Cannot inline (compiler doesn't know target at compile-time)
 * • Prevents devirtualization optimizations
 * • Cache misses (vtable might not be in cache)
 * • Branch prediction harder (indirect call)
 */

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

    // Virtual function - uses vtable dispatch
    virtual void speak() const = 0;
    virtual std::string getName() const = 0;

    // Virtual function with implementation
    virtual void describe() const {
        std::cout << "   I am a " << getName() << "\n";
    }
};

class Dog : public AnimalBase {
public:
    void speak() const override {
        std::cout << "    Woof! Woof!\n";
    }

    std::string getName() const override {
        return "Dog";
    }
};

class Cat : public AnimalBase {
public:
    void speak() const override {
        std::cout << "    Meow! Meow!\n";
    }

    std::string getName() const override {
```

```
        return "Cat";
    }
};

class Bird : public AnimalBase {
public:
    void speak() const override {
        std::cout << "     Tweet! Tweet!\n";
    }

    std::string getName() const override {
        return "Bird";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 1: VIRTUAL FUNCTIONS (DYNAMIC POLYMORPHISM)\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  VTABLE MECHANISM:\n";
    std::cout << " •   Each object has hidden vptr (8 bytes overhead)\n";
    std::cout << " •   vptr points to class vtable\n";
    std::cout << " •   Function calls: load vptr -> index vtable -> call\n";
    std::cout << " •   Runtime overhead: 2-3 memory loads per virtual call\n\n";

    // Demonstrate dynamic polymorphism
    std::vector<std::unique_ptr<AnimalBase>> animals;
    animals.push_back(std::make_unique<Dog>());
    animals.push_back(std::make_unique<Cat>());
    animals.push_back(std::make_unique<Bird>());

    std::cout << "  ADVANTAGE: Runtime polymorphism (heterogeneous containers)\n";
    for (const auto& animal : animals) {
        animal->describe();
        animal->speak();
    }

    std::cout << "\ n  SIZE OVERHEAD:\n";
    std::cout << "   sizeof(AnimalBase*) = " << sizeof(AnimalBase*) << " bytes\n";
    std::cout << "   sizeof(Dog) = " << sizeof(Dog) << " bytes (includes vptr)\n";
    std::cout << "   sizeof(Cat) = " << sizeof(Cat) << " bytes (includes vptr)\n";
}

} // namespace virtual_functions

// ==========================================================================
```

```cpp
// SECTION 2: STATIC MEMBER FUNCTIONS AND VTABLES
// ==============================================================================

namespace static_functions {

/*
 * STATIC MEMBER FUNCTIONS AND VTABLES:
 *
 * KEY FACT: Static member functions CANNOT be virtual!
 *
 * WHY?
 * • Static functions don't have 'this' pointer
 * • No object instance to determine which vtable to use
 * • Called via class name, not object
 * • Resolved at compile-time, not runtime
 *
 * IMPLICATION:
 * • Static functions don't appear in vtable
 * • Only ONE implementation exists (class-level, not object-level)
 * • Cannot be overridden in derived classes
 * • Zero runtime overhead (direct call)
 *
 * WHEN TO USE:
 * • Factory functions
 * • Utility functions that don't need object state
 * • Performance-critical code that doesn't need polymorphism
 */

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

    // Virtual function - in vtable
    virtual void instanceMethod() const {
        std::cout << "   Base::instanceMethod() - uses vtable\n";
    }

    // Static function - NOT in vtable, cannot be virtual
    static void staticMethod() {
        std::cout << "   Base::staticMethod() - no vtable, direct call\n";
    }

    // Non-virtual function - NOT in vtable
    void nonVirtualMethod() const {
        std::cout << "   Base::nonVirtualMethod() - no vtable, direct call\n";
    }
};

class Derived : public Base {
public:
    // Overrides virtual function - has entry in Derived's vtable
    void instanceMethod() const override {
```

```
        std::cout << "   Derived::instanceMethod() - uses vtable\n";
    }

    // CANNOT override static function (compile error if we tried)
    // This is a NEW function, not an override
    static void staticMethod() {
        std::cout << "   Derived::staticMethod() - different function!\n";
    }

    // Hides base class function (NOT override, no virtual)
    void nonVirtualMethod() const {
        std::cout << "   Derived::nonVirtualMethod() - hides base, no vtable\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 2: STATIC FUNCTIONS AND VTABLES\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  STATIC FUNCTIONS:\n";
    std::cout << " •   Cannot be virtual (no 'this' pointer)\n";
    std::cout << " •   NOT in vtable (only ONE implementation)\n";
    std::cout << " •   Called via class name, resolved at compile-time\n";
    std::cout << " •   Zero runtime overhead\n\n";

    Base* ptr = new Derived();

    std::cout << "  VIRTUAL FUNCTION (uses vtable):\n";
    ptr->instanceMethod(); // Calls Derived::instanceMethod via vtable

    std::cout << "\n STATIC FUNCTION (no vtable, direct call):\n";
    Base::staticMethod();    // Calls Base::staticMethod
    Derived::staticMethod(); // Calls Derived::staticMethod (different function!)

    std::cout << "\n  NON-VIRTUAL FUNCTION (no override, no vtable):\n";
    ptr->nonVirtualMethod(); // Calls Base::nonVirtualMethod (no polymorphism!)

    std::cout << "\n KEY INSIGHT:\n";
    std::cout << " •   Virtual functions: ONE entry in vtable per class\n";
    std::cout << " •   Static functions: ONE function for entire class hierarchy\n";
    std::cout << " •   Static means 'belongs to class, not object'\n";

    delete ptr;
}

} // namespace static_functions

// ==========================================================================
```

```cpp
// SECTION 3: CRTP - STATIC POLYMORPHISM (ZERO OVERHEAD)
// ==========================================================================

namespace crtp_pattern {

/*
 * CRTP: Curiously Recurring Template Pattern
 *
 * HOW IT WORKS:
 * • Base class is template parameterized by derived class
 * • Base uses static_cast to call derived methods
 * • Compiler resolves calls at compile-time
 * • NO vtable, NO vptr, NO runtime overhead
 *
 * CALL MECHANISM:
 * obj.speak() becomes:
 *   1. Compiler knows exact type at compile-time
 *   2. Direct function call (no indirection)
 *   3. Can inline the function
 *   4. Zero runtime overhead
 *
 * BENEFITS:
 * • Zero runtime overhead (no vtable lookup)
 * • Functions can be inlined
 * • No memory overhead (no vptr)
 * • Compiler can optimize aggressively
 *
 * DRAWBACKS:
 * • Cannot use heterogeneous containers (no common base type)
 * • Type known at compile-time (no runtime polymorphism)
 * • More complex code (template metaprogramming)
 * • Longer compile times
 */

// CRTP Base class
template <typename Derived>
class Animal {
public:
    void speak() const {
        // Static cast to derived type - resolved at compile-time
        static_cast<const Derived*>(this)->speak_impl();
    }

    std::string getName() const {
        return static_cast<const Derived*>(this)->getName_impl();
    }

    void describe() const {
        std::cout << "   I am a " << getName() << "\n";
    }

```

```
    // Optional: Provide default implementation
    void defaultBehavior() const {
        std::cout << "   All animals need food and water\n";
    }
};

class Dog : public Animal<Dog> {
public:
    void speak_impl() const {
        std::cout << "    Woof! Woof!\n";
    }

    std::string getName_impl() const {
        return "Dog";
    }
};

class Cat : public Animal<Cat> {
public:
    void speak_impl() const {
        std::cout << "    Meow! Meow!\n";
    }

    std::string getName_impl() const {
        return "Cat";
    }
};

class Bird : public Animal<Bird> {
public:
    void speak_impl() const {
        std::cout << "    Tweet! Tweet!\n";
    }

    std::string getName_impl() const {
        return "Bird";
    }
};

// Template function that works with any Animal<T>
template <typename T>
void makeAnimalSpeak(const Animal<T>& animal) {
    animal.describe();
    animal.speak();
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 3: CRTP - STATIC POLYMORPHISM (ZERO OVERHEAD)\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  CRTP MECHANISM:\n";
    std::cout << " •   Compile-time polymorphism (no runtime overhead)\n";
    std::cout << " •   No vtable, no vptr (zero memory overhead)\n";
```

```
    std::cout << " •   Direct function calls (can be inlined)\n";
    std::cout << " •   Compiler knows exact type at compile-time\n\n";

    Dog d;
    Cat c;
    Bird b;

    std::cout << "  DIRECT CALLS (statically resolved):\n";
    d.speak(); // Compiler knows this is Dog::speak_impl at compile-time
    c.speak(); // Compiler knows this is Cat::speak_impl at compile-time
    b.speak(); // Compiler knows this is Bird::speak_impl at compile-time

    std::cout << "\n TEMPLATE FUNCTION (works with any Animal<T>):\n";
    makeAnimalSpeak(d);
    makeAnimalSpeak(c);
    makeAnimalSpeak(b);

    std::cout << "\n  SIZE COMPARISON:\n";
    std::cout << "   sizeof(Dog) = " << sizeof(Dog) << " bytes (NO vptr!)\n";
    std::cout << "   sizeof(Cat) = " << sizeof(Cat) << " bytes (NO vptr!)\n";
    std::cout << "   Compare to virtual_functions::Dog = "
              << sizeof(virtual_functions::Dog) << " bytes (with vptr)\n";

    std::cout << "\n LIMITATION: Cannot create heterogeneous container:\n";
    std::cout << "   // std::vector<Animal<??>> animals; // Won't work!\n";
    std::cout << "   // Dog and Cat are DIFFERENT types (Animal<Dog> vs Animal
        <Cat>)\n";
}

} // namespace crtp_pattern

//
    ==========================================================================
// SECTION 4: PERFORMANCE COMPARISON
//
    ==========================================================================

namespace performance_comparison {

// Virtual function version
class VirtualAnimal {
public:
    virtual ~VirtualAnimal() = default;
    virtual int compute(int x) const = 0;
};

class VirtualDog : public VirtualAnimal {
public:
    int compute(int x) const override {
        return x * x + x; // Simple computation
    }
};
```

```

// CRTP version
template <typename Derived>
class CRTPAnimal {
public:
    int compute(int x) const {
        return static_cast<const Derived*>(this)->compute_impl(x);
    }
};

class CRTPDog : public CRTPAnimal<CRTPDog> {
public:
    int compute_impl(int x) const {
        return x * x + x; // Same computation
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 4: PERFORMANCE BENCHMARKING\n";
    std::cout << std::string(70, '=') << "\n\n";

    constexpr int ITERATIONS = 10'000'000;

    // Benchmark virtual functions
    auto start = std::chrono::high_resolution_clock::now();
    {
        VirtualDog dog;
        VirtualAnimal* ptr = &dog;
        long long sum = 0;
        for (int i = 0; i < ITERATIONS; ++i) {
            sum += ptr->compute(i % 100); // Virtual call
        }
        // Use sum to prevent optimization
        if (sum > 0) {}
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto virtual_time = std::chrono::duration_cast<std::chrono::milliseconds>(
        end - start).count();

    // Benchmark CRTP
    start = std::chrono::high_resolution_clock::now();
    {
        CRTPDog dog;
        long long sum = 0;
        for (int i = 0; i < ITERATIONS; ++i) {
            sum += dog.compute(i % 100); // Static call
        }
        // Use sum to prevent optimization
        if (sum > 0) {}
    }
    end = std::chrono::high_resolution_clock::now();
    auto crtp_time = std::chrono::duration_cast<std::chrono::milliseconds>(end
        - start).count();
```

```

    std::cout << "  BENCHMARK: " << ITERATIONS << " function calls\n\n";
    std::cout << "   Virtual functions:  " << std::setw(5) << virtual_time << " ms\n";
    std::cout << "   CRTP (static):      " << std::setw(5) << crtp_time << " ms\n";
    std::cout << "   Speedup:            " << std::setw(5) << std::fixed << std::setprecision(2)
              << (double)virtual_time / crtp_time << "x\n\n";

    std::cout << "  WHY CRTP IS FASTER:\n";
    std::cout << " •   No vtable lookup (2-3 memory loads eliminated)\n";
    std::cout << " •   Function can be inlined (compiler sees implementation)\n";
    std::cout << " •   Better cache locality (no vtable indirection)\n";
    std::cout << " •   Better branch prediction (direct call)\n";
    std::cout << " •   Enables other compiler optimizations\n";
}

} // namespace performance_comparison

// ============================================================================
// SECTION 5: WHEN TO USE EACH APPROACH
// ============================================================================

namespace when_to_use {

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 5: WHEN TO USE VIRTUAL FUNCTIONS VS CRTP\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  USE VIRTUAL FUNCTIONS WHEN:\n\n";
    std::cout << "    1 RUNTIME POLYMORPHISM NEEDED:\n";
    std::cout << " •      Heterogeneous containers (vector<Base*>)\n";
    std::cout << " •      Plugin systems (load types at runtime)\n";
    std::cout << " •      Factory patterns\n";
    std::cout << " •      Type not known until runtime\n\n";

    std::cout << "    2 INTERFACE-BASED DESIGN:\n";
    std::cout << " •      Defining abstract interfaces\n";
    std::cout << " •      Separating interface from implementation\n";
    std::cout << " •      Dependency injection\n\n";

    std::cout << "    3 BINARY COMPATIBILITY:\n";
    std::cout << " •      DLLs/shared libraries\n";
    std::cout << " •      ABI stability requirements\n";
    std::cout << " •      Plugin systems across compilation units\n\n";

    std::cout << "    4 CODE SIMPLICITY:\n";
```

```
    std::cout << " •      Simpler code (no template metaprogramming)\n";
    std::cout << " •      Faster compile times\n";
    std::cout << " •      More familiar to developers\n\n";

    std::cout << "    5 PERFORMANCE NOT CRITICAL:\n";
    std::cout << " •      UI code, configuration, initialization\n";
    std::cout << " •      Virtual call overhead is negligible\n";
    std::cout << " •      Clarity more important than speed\n\n";

    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  USE CRTP (STATIC POLYMORPHISM) WHEN:\n\n";
    std::cout << "    1 PERFORMANCE CRITICAL:\n";
    std::cout << " •      Hot loops (inner loops, tight iterations)\n";
    std::cout << " •      Real-time systems (low latency required)\n";
    std::cout << " •      Game engines (per-frame calculations)\n";
    std::cout << " •      High-frequency trading systems\n\n";

    std::cout << "    2 MEMORY CONSTRAINED:\n";
    std::cout << " •      Embedded systems (every byte counts)\n";
    std::cout << " •      Large arrays of objects (vptr overhead * N)\n";
    std::cout << " •      Cache-sensitive code\n\n";

    std::cout << "    3 COMPILE-TIME POLYMORPHISM SUFFICIENT:\n";
    std::cout << " •      Type known at compile-time\n";
    std::cout << " •      Template containers (vector<Dog>, vector<Cat>)\n";
    std::cout << " •      Generic algorithms\n\n";

    std::cout << "    4 INLINING REQUIRED:\n";
    std::cout << " •      Small functions that must be inlined\n";
    std::cout << " •      Zero-overhead abstractions needed\n";
    std::cout << " •      Maximum compiler optimization wanted\n\n";

    std::cout << "    5 TEMPLATE-BASED LIBRARIES:\n";
    std::cout << " •      STL-style libraries (iterators, algorithms)\n";
    std::cout << " •      Expression templates\n";
    std::cout << " •      Policy-based design\n\n";

    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "   TRADE-OFFS SUMMARY:\n\n";
    std::cout << "   VIRTUAL FUNCTIONS:\n";
    std::cout << "     Runtime polymorphism\n";
    std::cout << "     Heterogeneous containers\n";
    std::cout << "     Simpler code\n";
    std::cout << "     Faster compile times\n";
    std::cout << "     Runtime overhead (vtable lookup)\n";
    std::cout << "     Memory overhead (vptr per object)\n";
    std::cout << "     Cannot inline\n\n";

    std::cout << "   CRTP (STATIC POLYMORPHISM):\n";
    std::cout << "     Zero runtime overhead\n";
    std::cout << "     Zero memory overhead\n";
    std::cout << "     Can inline functions\n";
```

```
    std::cout << "     Maximum performance\n";
    std::cout << "     No runtime polymorphism\n";
    std::cout << "     No heterogeneous containers\n";
    std::cout << "     More complex code\n";
    std::cout << "     Longer compile times\n\n";

    std::cout << "  GOLDEN RULE:\n";
    std::cout << "    'Use virtual functions by default for flexibility.\n";
    std::cout << "     Use CRTP only when performance profiling shows\n";
    std::cout << "     virtual function overhead is a bottleneck.'\n\n";

    std::cout << "   PREMATURE OPTIMIZATION WARNING:\n";
    std::cout << " •   Don't use CRTP everywhere \"just in case\"\n";
    std::cout << " •   Profile first, optimize later\n";
    std::cout << " •   Virtual function overhead is often negligible\n";
    std::cout << " •   Code clarity often more valuable than tiny speedup\n";
}

} // namespace when_to_use

// ==========================================================================
// SECTION 6: HYBRID APPROACH - BEST OF BOTH WORLDS
// ==========================================================================

namespace hybrid_approach {

/*
 * HYBRID APPROACH: Combine virtual functions with CRTP
 *
 * STRATEGY:
 * • Use virtual functions for high-level interfaces
 * • Use CRTP for performance-critical inner operations
 * • Get both flexibility and performance
 */

// Virtual base for runtime polymorphism
class RenderableBase {
public:
    virtual ~RenderableBase() = default;
    virtual void render() const = 0;
};

// CRTP for performance-critical operations
template <typename Derived>
class FastOperations {
public:
    void processPixel(int x, int y) const {
        static_cast<const Derived*>(this)->processPixel_impl(x, y);
    }
};
```

```

// Concrete class uses both
class Sprite : public RenderableBase, public FastOperations<Sprite> {
private:
    mutable int pixel_count = 0;

public:
    // Virtual function for high-level interface
    void render() const override {
        std::cout << "   [Sprite] Rendering...\n";
        // Performance-critical inner loop uses CRTP
        for (int y = 0; y < 100; ++y) {
            for (int x = 0; x < 100; ++x) {
                processPixel(x, y); // CRTP call (zero overhead)
            }
        }
        std::cout << "   [Sprite] Processed " << pixel_count << " pixels\n";
    }

    // CRTP implementation (inlined, zero overhead)
    void processPixel_impl(int x, int y) const {
        // Performance-critical pixel processing
        ++pixel_count;
        // Actual pixel operations would go here
        (void)x; (void)y; // Suppress unused warnings
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 6: HYBRID APPROACH - BEST OF BOTH WORLDS\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  STRATEGY:\n";
    std::cout << " •   Virtual functions for high-level interface\n";
    std::cout << " •   CRTP for performance-critical inner loops\n";
    std::cout << " •   Get both flexibility AND performance\n\n";

    std::cout << "  DEMONSTRATION:\n";
    Sprite sprite;
    RenderableBase* ptr = &sprite;

    // Virtual call for high-level operation
    ptr->render(); // Virtual call (small overhead, called once)
                   // Inner loop uses CRTP (zero overhead, called 10000x)

    std::cout << "\ n BENEFIT:\n";
    std::cout << " •   Can store in vector<RenderableBase*> (runtime
        polymorphism)\n";
    std::cout << " •   Inner loop has zero overhead (CRTP inlining)\n";
    std::cout << " •   Best of both worlds!\n";
}

} // namespace hybrid_approach
```

```cpp

// ==============================================================================
// MAIN
// ==============================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "           CRTP vs VIRTUAL FUNCTIONS IN MODERN C++          \n";
    std::cout << "          Static vs Dynamic Polymorphism Trade-offs         \n";
    std::cout << "                                    \n";

    virtual_functions::demonstrate();
    static_functions::demonstrate();
    crtp_pattern::demonstrate();
    performance_comparison::demonstrate();
    when_to_use::demonstrate();
    hybrid_approach::demonstrate();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "KEY TAKEAWAYS:\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << " 1 VIRTUAL FUNCTIONS:\n";
    std::cout << " •   Use vtable (per class) and vptr (per object)\n";
    std::cout << " •   Runtime overhead: 2-3 memory loads per call\n";
    std::cout << " •   Memory overhead: 8 bytes per object (vptr)\n";
    std::cout << " •   Cannot inline virtual calls\n";
    std::cout << " •   Enable runtime polymorphism\n\n";

    std::cout << " 2 STATIC MEMBER FUNCTIONS:\n";
    std::cout << " •   Cannot be virtual (no 'this' pointer)\n";
    std::cout << " •   Only ONE function per class hierarchy\n";
    std::cout << " •   Zero runtime overhead (direct call)\n";
    std::cout << " •   Not in vtable\n\n";

    std::cout << " 3 CRTP (CURIOUSLY RECURRING TEMPLATE PATTERN):\n";
    std::cout << " •   Zero runtime overhead (compile-time resolution)\n";
    std::cout << " •   Zero memory overhead (no vptr)\n";
    std::cout << " •   Functions can be inlined\n";
    std::cout << " •   No heterogeneous containers\n";
    std::cout << " •   More complex code\n\n";

    std::cout << " 4 WHEN TO USE WHAT:\n";
    std::cout << " •   Virtual: Flexibility, runtime polymorphism, simplicity\n";
    std::cout << " •   CRTP: Performance-critical code, compile-time known types\n";
```

```
    std::cout << " •   Hybrid: High-level virtual + low-level CRTP\n\n";

    std::cout << " 5 PERFORMANCE:\n";
    std::cout << " •   Virtual overhead often negligible in real applications\n";
    std::cout << " •   CRTP can be 2-10x faster in tight loops\n";
    std::cout << " •   Profile before optimizing!\n\n";

    std::cout << "                                    \n";
    std::cout << "            ALL CONCEPTS DEMONSTRATED SUCCESSFULLY!\n";
    std::cout << "                                    \n\n";

    return 0;
}
```

8 Source Code: CameraModule.cppm

File: `src/CameraModule.cppm`
Repository: View on GitHub

```
// ==================================================================
// C++20 MODULE: CAMERA INTERFACE WITH TEMPLATES
// ==================================================================
// Demonstrates C++20 modules with export/import
// Also includes concepts for type constraints
// ==================================================================

module;

// Global module fragment - for #include directives
#include <vector>
#include <memory>
#include <string>
#include <cstdint>
#include <type_traits>
#include <algorithm>
#include <cmath>
#include <utility>
#include <concepts>

export module camera;

// ==================================================================
// C++20 CONCEPTS FOR PIXEL TYPES
// ==================================================================

export template<typename T>
concept PixelType = std::is_arithmetic_v<T> && (
    std::is_same_v<T, uint8_t> ||
    std::is_same_v<T, uint16_t> ||
    std::is_same_v<T, uint32_t> ||
    std::is_same_v<T, float> ||
    std::is_same_v<T, double>
);

export template<typename T>
concept IntegerPixel = PixelType<T> && std::is_integral_v<T>;

export template<typename T>
concept FloatingPixel = PixelType<T> && std::is_floating_point_v<T>;

// ==================================================================
// IMAGE CLASS (EXPORTED)
// ==================================================================

export template<PixelType T>
class Image {
private:
    size_t width;
```

```
    size_t height;
    std::vector<T> pixels;

public:
    Image(size_t w, size_t h)
        : width(w), height(h), pixels(w * h) {}

    Image(size_t w, size_t h, T initial_value)
        : width(w), height(h), pixels(w * h, initial_value) {}

    // Accessors
    [[nodiscard]] size_t get_width() const noexcept { return width; }
    [[nodiscard]] size_t get_height() const noexcept { return height; }
    [[nodiscard]] size_t get_size() const noexcept { return pixels.size(); }

    // Pixel access
    T& at(size_t x, size_t y) {
        return pixels[y * width + x];
    }

    const T& at(size_t x, size_t y) const {
        return pixels[y * width + x];
    }

    // Raw data access
    [[nodiscard]] T* data() noexcept { return pixels.data(); }
    [[nodiscard]] const T* data() const noexcept { return pixels.data(); }

    // Memory size
    [[nodiscard]] size_t memory_bytes() const noexcept {
        return pixels.size() * sizeof(T);
    }

    // Fill with value
    void fill(T value) {
        std::fill(pixels.begin(), pixels.end(), value);
    }

    // C++20: Three-way comparison
    auto operator<=>(const Image&) const = default;
};

// ==================================================================
// CAMERA INTERFACE (EXPORTED)
// ==================================================================

export template<PixelType T>
class Camera {
private:
    size_t width;
    size_t height;
    std::string camera_name;

public:
```

```
    Camera(const std::string& name, size_t w, size_t h)
        : camera_name(name), width(w), height(h) {}

    virtual ~Camera() = default;

    // Pure virtual: capture image
    virtual Image<T> capture() = 0;

    // Configuration
    [[nodiscard]] size_t get_width() const noexcept { return width; }
    [[nodiscard]] size_t get_height() const noexcept { return height; }
    [[nodiscard]] const std::string& get_name() const noexcept { return
        camera_name; }

    // Get pixel type information
    [[nodiscard]] static constexpr size_t bits_per_pixel() noexcept {
        return sizeof(T) * 8;
    }

    [[nodiscard]] static constexpr bool is_floating_point() noexcept {
        return std::is_floating_point_v<T>;
    }

    [[nodiscard]] static constexpr bool is_integer() noexcept {
        return std::is_integral_v<T>;
    }
};

// ==================================================================
// IMAGE PROCESSOR (EXPORTED)
// ==================================================================

export template<PixelType T>
class ImageProcessor {
public:
    // Calculate average pixel value
    [[nodiscard]] static double calculate_mean(const Image<T>& img) {
        double sum = 0.0;
        const T* data = img.data();
        size_t size = img.get_size();

        for (size_t i = 0; i < size; ++i) {
            sum += static_cast<double>(data[i]);
        }

        return sum / size;
    }

    // Find min and max pixel values
    [[nodiscard]] static std::pair<T, T> find_min_max(const Image<T>& img) {
        const T* data = img.data();
        size_t size = img.get_size();

        T min_val = data[0];
```

```
        T max_val = data[0];

        for (size_t i = 1; i < size; ++i) {
            if (data[i] < min_val) min_val = data[i];
            if (data[i] > max_val) max_val = data[i];
        }

        return {min_val, max_val};
    }

    // Scale pixel values
    [[nodiscard]] static Image<T> scale(const Image<T>& img, double factor) {
        Image<T> result(img.get_width(), img.get_height());

        for (size_t y = 0; y < img.get_height(); ++y) {
            for (size_t x = 0; x < img.get_width(); ++x) {
                double scaled = static_cast<double>(img.at(x, y)) * factor;
                result.at(x, y) = static_cast<T>(scaled);
            }
        }

        return result;
    }

    // Threshold operation - using concept constraint
    [[nodiscard]] static Image<T> threshold(const Image<T>& img, T
        threshold_value)
        requires IntegerPixel<T>
    {
        Image<T> result(img.get_width(), img.get_height());

        for (size_t y = 0; y < img.get_height(); ++y) {
            for (size_t x = 0; x < img.get_width(); ++x) {
                result.at(x, y) = (img.at(x, y) >= threshold_value) ?
                    threshold_value : T(0);
            }
        }

        return result;
    }

    // Normalize operation - only for floating point images
    [[nodiscard]] static Image<T> normalize(const Image<T>& img)
        requires FloatingPixel<T>
    {
        auto [min_val, max_val] = find_min_max(img);
        T range = max_val - min_val;

        if (range == T(0)) return img;

        Image<T> result(img.get_width(), img.get_height());

        for (size_t y = 0; y < img.get_height(); ++y) {
            for (size_t x = 0; x < img.get_width(); ++x) {
```

```
                result.at(x, y) = (img.at(x, y) - min_val) / range;
            }
        }

        return result;
    }
};

// ==================================================================
// TYPE CONVERSION (EXPORTED)
// ==================================================================

export template<PixelType DestType, PixelType SrcType>
[[nodiscard]] Image<DestType> convert_image(const Image<SrcType>& src) {
    Image<DestType> dest(src.get_width(), src.get_height());

    // Find source range
    auto [min_val, max_val] = ImageProcessor<SrcType>::find_min_max(src);
    double src_range = static_cast<double>(max_val) - static_cast<double>(
        min_val);

    // Determine destination range
    double dest_min, dest_max;
    if constexpr (std::is_floating_point_v<DestType>) {
        dest_min = 0.0;
        dest_max = 1.0;
    } else if constexpr (std::is_same_v<DestType, uint8_t>) {
        dest_min = 0.0;
        dest_max = 255.0;
    } else if constexpr (std::is_same_v<DestType, uint16_t>) {
        dest_min = 0.0;
        dest_max = 65535.0;
    } else {
        dest_min = 0.0;
        dest_max = 1.0;
    }

    double dest_range = dest_max - dest_min;

    // Convert with proper scaling
    if (src_range == 0.0) {
        dest.fill(static_cast<DestType>(dest_min));
        return dest;
    }

    for (size_t y = 0; y < src.get_height(); ++y) {
        for (size_t x = 0; x < src.get_width(); ++x) {
            double normalized = (static_cast<double>(src.at(x, y)) -
                static_cast<double>(min_val)) / src_range;
            double scaled = normalized * dest_range + dest_min;
            dest.at(x, y) = static_cast<DestType>(scaled);
        }
    }

```

```
    return dest;
}

// ==================================================================
// CAMERA IMPLEMENTATIONS (EXPORTED)
// ==================================================================

export class Camera8bit : public Camera<uint8_t> {
public:
    Camera8bit(const std::string& name, size_t w, size_t h)
        : Camera<uint8_t>(name, w, h) {}

    Image<uint8_t> capture() override {
        Image<uint8_t> img(get_width(), get_height());

        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                uint8_t value = static_cast<uint8_t>(
                    (x * 255.0 / get_width()) * 0.5 +
                    (y * 255.0 / get_height()) * 0.5
                );
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

export class Camera16bit : public Camera<uint16_t> {
public:
    Camera16bit(const std::string& name, size_t w, size_t h)
        : Camera<uint16_t>(name, w, h) {}

    Image<uint16_t> capture() override {
        Image<uint16_t> img(get_width(), get_height());

        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                uint16_t value = static_cast<uint16_t>(
                    (x * 65535.0 / get_width()) * 0.3 +
                    (y * 65535.0 / get_height()) * 0.7
                );
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

export class CameraFloat : public Camera<float> {
public:
    CameraFloat(const std::string& name, size_t w, size_t h)
        : Camera<float>(name, w, h) {}

```

```
    Image<float> capture() override {
        Image<float> img(get_width(), get_height());

        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                float value =
                    0.5f + 0.5f * std::sin(x * 0.1f) * std::cos(y * 0.1f);
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

export class CameraDouble : public Camera<double> {
public:
    CameraDouble(const std::string& name, size_t w, size_t h)
        : Camera<double>(name, w, h) {}

    Image<double> capture() override {
        Image<double> img(get_width(), get_height());

        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                double value =
                    std::sin(x * 0.05) * std::cos(y * 0.05) +
                    std::exp(-((x - get_width()/2.0) * (x - get_width()/2.0) +
                               (y - get_height()/2.0) * (y - get_height()/2.0)
                                   ) / 1000.0);
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

// ==================================================================
// CAMERA HANDLER (EXPORTED)
// ==================================================================

export template<PixelType T>
class CameraHandler {
private:
    std::unique_ptr<Camera<T>> camera;

public:
    CameraHandler(std::unique_ptr<Camera<T>> cam)
        : camera(std::move(cam)) {}

    [[nodiscard]] const Camera<T>* get_camera() const noexcept {
        return camera.get();
    }

    [[nodiscard]] Image<T> capture() {
```

```
        return camera->capture();
    }

    [[nodiscard]] static constexpr size_t pixel_size() noexcept {
        return sizeof(T);
    }

    [[nodiscard]] static constexpr size_t bits_per_pixel() noexcept {
        return sizeof(T) * 8;
    }
};
```

9 Source Code: ConceptsExamples.cpp

File: `src/ConceptsExamples.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <concepts>
#include <type_traits>
#include <algorithm>
#include <iterator>

// ==================================================================
// 1. BASIC CONCEPT DEFINITION
// ==================================================================
template<typename T>
concept Numeric = std::is_arithmetic_v<T>;

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

void example_basic_concept() {
    std::cout << "\n=== 1. BASIC CONCEPT DEFINITION ===" << std::endl;

    std::cout << "add(10, 20) = " << add(10, 20) << std::endl;
    std::cout << "add(3.14, 2.86) = " << add(3.14, 2.86) << std::endl;

    // This would fail to compile:
    // add(std::string("hello"), std::string("world"));
}

// ==================================================================
// 2. STANDARD LIBRARY CONCEPTS
// ==================================================================
template<std::integral T>
T multiply(T a, T b) {
    return a * b;
}

template<std::floating_point T>
T divide(T a, T b) {
    return a / b;
}

void example_standard_concepts() {
    std::cout << "\n=== 2. STANDARD LIBRARY CONCEPTS ===" << std::endl;

    std::cout << "multiply(5, 6) = " << multiply(5, 6) << std::endl;
    std::cout << "divide(10.0, 3.0) = " << divide(10.0, 3.0) << std::endl;

    // multiply(3.14, 2.0);  // ERROR: requires integral type
```

```
    // divide(10, 3);        // ERROR: requires floating point type
}

// ====================================================================
// 3. COMPOUND CONCEPTS
// ====================================================================
template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

template<typename T>
concept Printable = requires(T t) {
    { std::cout << t } -> std::convertible_to<std::ostream&>;
};

template<typename T>
concept AddableAndPrintable = Addable<T> && Printable<T>;

template<AddableAndPrintable T>
void print_sum(T a, T b) {
    std::cout << a << " + " << b << " = " << (a + b) << std::endl;
}

void example_compound_concepts() {
    std::cout << "\n=== 3. COMPOUND CONCEPTS ===" << std::endl;

    print_sum(10, 20);
    print_sum(3.14, 2.86);
    print_sum(std::string("Hello "), std::string("World"));
}

// ====================================================================
// 4. REQUIRES CLAUSE
// ====================================================================
template<typename T>
T max_value(T a, T b) requires std::totally_ordered<T> {
    return (a > b) ? a : b;
}

template<typename T>
requires std::integral<T>
T factorial(T n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

void example_requires_clause() {
    std::cout << "\n=== 4. REQUIRES CLAUSE ===" << std::endl;

    std::cout << "max(10, 20) = " << max_value(10, 20) << std::endl;
    std::cout << "max(3.14, 2.71) = " << max_value(3.14, 2.71) << std::endl;

    std::cout << "factorial(5) = " << factorial(5) << std::endl;
```

```
}

// ==================================================================
// 5. CONCEPTS WITH MULTIPLE REQUIREMENTS
// ==================================================================
template<typename T>
concept Container = requires(T t) {
    typename T::value_type;
    typename T::iterator;
    { t.begin() } -> std::same_as<typename T::iterator>;
    { t.end() } -> std::same_as<typename T::iterator>;
    { t.size() } -> std::convertible_to<std::size_t>;
};

template<Container C>
void print_container(const C& container, const std::string& name) {
    std::cout << name << " (size " << container.size() << "): ";
    for (const auto& elem : container) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;
}

void example_container_concept() {
    std::cout << "\n=== 5. CONCEPTS WITH MULTIPLE REQUIREMENTS ===" << std::
        endl;

    std::vector<int> vec = {1, 2, 3, 4, 5};
    print_container(vec, "Vector");

    std::vector<std::string> words = {"Hello", "Modern", "C++"};
    print_container(words, "Strings");
}

// ==================================================================
// 6. CONCEPT SUBSUMPTION
// ==================================================================
template<typename T>
concept Number = std::is_arithmetic_v<T>;

template<typename T>
concept Integer = Number<T> && std::is_integral_v<T>;

template<Number T>
void process(T value) {
    std::cout << "Processing number: " << value << std::endl;
}

template<Integer T>
void process(T value) {
    std::cout << "Processing integer: " << value << " (doubled: " << value * 2
        << ")" << std::endl;
}

```

```
void example_concept_subsumption() {
    std::cout << "\n=== 6. CONCEPT SUBSUMPTION ===" << std::endl;

    process(42);        // Calls Integer version
    process(3.14);      // Calls Number version
}

// ==================================================================
// 7. CUSTOM CONCEPT FOR RANGES
// ==================================================================
template<typename T>
concept Sortable = requires(T container) {
    { container.begin() } -> std::input_or_output_iterator;
    { container.end() } -> std::input_or_output_iterator;
    requires std::sortable<decltype(container.begin())>;
};

template<Sortable C>
void sort_and_print(C& container) {
    std::sort(container.begin(), container.end());
    std::cout << "Sorted: ";
    for (const auto& elem : container) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;
}

void example_sortable_concept() {
    std::cout << "\n=== 7. CUSTOM CONCEPT FOR RANGES ===" << std::endl;

    std::vector<int> numbers = {5, 2, 8, 1, 9};
    sort_and_print(numbers);
}

// ==================================================================
// 8. CONCEPT WITH CLASS TEMPLATE
// ==================================================================
template<typename T>
concept Multipliable = requires(T a, T b) {
    { a * b } -> std::convertible_to<T>;
};

template<Multipliable T>
class Point2D {
private:
    T x, y;

public:
    Point2D(T x_val, T y_val) : x(x_val), y(y_val) {}

    T dot_product(const Point2D& other) const {
        return x * other.x + y * other.y;
    }

```

```
    void print() const {
        std::cout << "Point(" << x << ", " << y << ")" << std::endl;
    }
};

void example_concept_class_template() {
    std::cout << "\n=== 8. CONCEPT WITH CLASS TEMPLATE ===" << std::endl;

    Point2D<int> p1(3, 4);
    Point2D<int> p2(5, 6);

    p1.print();
    p2.print();
    std::cout << "Dot product: " << p1.dot_product(p2) << std::endl;
}

// ==================================================================
// 9. CONCEPTS WITH LOGICAL OPERATORS
// ==================================================================
template<typename T>
concept SignedIntegral = std::integral<T> && std::is_signed_v<T>;

template<typename T>
concept UnsignedIntegral = std::integral<T> && std::is_unsigned_v<T>;

template<SignedIntegral T>
T abs_value(T value) {
    return value < 0 ? -value : value;
}

template<UnsignedIntegral T>
T abs_value(T value) {
    return value;  // Already positive
}

void example_logical_concepts() {
    std::cout << "\n=== 9. CONCEPTS WITH LOGICAL OPERATORS ===" << std::endl;

    std::cout << "abs(-42) = " << abs_value(-42) << std::endl;
    std::cout << "abs(42u) = " << abs_value(42u) << std::endl;
}

// ==================================================================
// 10. CONCEPT FOR CALLABLE TYPES
// ==================================================================
template<typename F, typename T>
concept Predicate = std::predicate<F, T>;

template<typename C, typename Pred>
requires Container<C> && Predicate<Pred, typename C::value_type>
void filter_and_print(const C& container, Pred pred, const std::string& name)
    {
    std::cout << name << ": ";
    for (const auto& elem : container) {
```

```
        if (pred(elem)) {
            std::cout << elem << " ";
        }
    }
    std::cout << std::endl;
}

void example_predicate_concept() {
    std::cout << "\n=== 10. CONCEPT FOR CALLABLE TYPES ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    filter_and_print(numbers, [](int n) { return n % 2 == 0; }, "Even numbers"
        );
    filter_and_print(numbers, [](int n) { return n > 5; }, "Greater than 5");
}

// ==================================================================
// 11. CONCEPT FOR CUSTOM TYPES
// ==================================================================
template<typename T>
concept Drawable = requires(T obj) {
    { obj.draw() } -> std::same_as<void>;
    { obj.get_name() } -> std::convertible_to<std::string>;
};

class Circle {
public:
    void draw() const {
        std::cout << "Drawing a circle" << std::endl;
    }

    std::string get_name() const {
        return "Circle";
    }
};

class Rectangle {
public:
    void draw() const {
        std::cout << "Drawing a rectangle" << std::endl;
    }

    std::string get_name() const {
        return "Rectangle";
    }
};

template<Drawable T>
void render(const T& shape) {
    std::cout << "Rendering: " << shape.get_name() << std::endl;
    shape.draw();
}

```

```
void example_custom_type_concept() {
    std::cout << "\n=== 11. CONCEPT FOR CUSTOM TYPES ===" << std::endl;

    Circle circle;
    Rectangle rect;

    render(circle);
    render(rect);
}

// ==================================================================
// 12. ADVANCED CONCEPT WITH NESTED REQUIREMENTS
// ==================================================================
template<typename T>
concept Arithmetic = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
    { a - b } -> std::convertible_to<T>;
    { a * b } -> std::convertible_to<T>;
    { a / b } -> std::convertible_to<T>;
};

template<Arithmetic T>
class Calculator {
private:
    T value;

public:
    Calculator(T val) : value(val) {}

    T add(T other) { return value + other; }
    T subtract(T other) { return value - other; }
    T multiply(T other) { return value * other; }
    T divide(T other) { return value / other; }

    void print() const {
        std::cout << "Calculator value: " << value << std::endl;
    }
};

void example_arithmetic_concept() {
    std::cout << "\n=== 12. ADVANCED CONCEPT WITH NESTED REQUIREMENTS ===" <<
        std::endl;

    Calculator<int> calc_int(100);
    calc_int.print();
    std::cout << "100 + 50 = " << calc_int.add(50) << std::endl;
    std::cout << "100 * 2 = " << calc_int.multiply(2) << std::endl;

    Calculator<double> calc_double(10.5);
    calc_double.print();
    std::cout << "10.5 / 2.0 = " << calc_double.divide(2.0) << std::endl;
}

// ==================================================================
```

```
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n=================================================" << std::endl;
    std::cout << "    C++20 CONCEPTS EXAMPLES" << std::endl;
    std::cout << "=================================================" << std::endl;

    example_basic_concept();
    example_standard_concepts();
    example_compound_concepts();
    example_requires_clause();
    example_container_concept();
    example_concept_subsumption();
    example_sortable_concept();
    example_concept_class_template();
    example_logical_concepts();
    example_predicate_concept();
    example_custom_type_concept();
    example_arithmetic_concept();

    std::cout << "\n=================================================" << std::endl;
    std::cout << "    ALL EXAMPLES COMPLETED" << std::endl;
    std::cout << "=================================================\n" << std::endl;

    return 0;
}
```

10 Source Code: ConfigLoaderAndChecker.cpp

File: `src/ConfigLoaderAndChecker.cpp`
Repository: View on GitHub

```cpp
// ConfigLoaderAndChecker.cpp
// Demonstrates dynamic configuration loading and monitoring using std::unordered_multimap
//
// FEATURES:
// 1. Load key-value pairs from JSON config file into unordered_multimap
// 2. Monitor config.json for changes (file modification time)
// 3. Automatically reload and update configuration on changes
// 4. Handle new keys, modified values, and deleted keys
// 5. Thread-safe configuration access
// 6. JSON validation and error handling
// 7. Real-time configuration updates without restart

#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>
#include <vector>
#include <set>
#include <chrono>
#include <thread>
#include <mutex>
#include <atomic>
#include <iomanip>
#include <filesystem>
#include <nlohmann/json.hpp>

namespace fs = std::filesystem;
using namespace std::chrono;
using namespace std::chrono_literals;

// ============================================================================
// SECTION 1: Using nlohmann/json library for JSON parsing
// ============================================================================

using json = nlohmann::json;

// ============================================================================
// SECTION 2: Thread-Safe Configuration Manager with unordered_multimap
// ============================================================================

```

```
class ConfigManager {
private:
    std::unordered_multimap<std::string, std::string> config_data_;
    std::unordered_multimap<std::string, std::string> previous_config_data_;
    mutable std::mutex mutex_;
    fs::file_time_type last_write_time_;
    std::string config_file_path_;
    std::atomic<int> version_{0};

public:
    explicit ConfigManager(const std::string& config_file)
        : config_file_path_(config_file) {

        if (fs::exists(config_file_path_)) {
            last_write_time_ = fs::last_write_time(config_file_path_);
        }
    }

    // Load configuration from file
    bool load() {
        try {
            std::ifstream file(config_file_path_);
            if (!file.is_open()) {
                std::cerr << " Cannot open config file: " <<
                    config_file_path_ << "\n";
                return false;
            }

            // Parse JSON using nlohmann/json
            json parsed_json;
            file >> parsed_json;
            file.close();

            // Update configuration
            std::lock_guard<std::mutex> lock(mutex_);

            // Save previous config for change detection
            previous_config_data_ = config_data_;
            config_data_.clear();

            // Support both flat objects and nested structures
            for (auto& [key, value] : parsed_json.items()) {
                // Convert value to string representation
                std::string value_str;
                if (value.is_string()) {
                    value_str = value.get<std::string>();
                } else if (value.is_number()) {
                    value_str = value.dump();
                } else if (value.is_boolean()) {
                    value_str = value.get<bool>() ? "true" : "false";
                } else if (value.is_array()) {
                    // For arrays, insert multiple entries (true multimap
                        usage)
                    for (const auto& item : value) {
```

```
                        if (item.is_string()) {
                            config_data_.insert({key, item.get<std::string>()
                                });
                        } else {
                            config_data_.insert({key, item.dump()});
                        }
                    }
                    continue; // Skip the insert below
                } else {
                    value_str = value.dump();
                }

                config_data_.insert({key, value_str});
            }

            version_++;

            std::cout << " Loaded " << config_data_.size()
                      << " configuration entries (version " << version_ << ")\
                        n";

            return true;

        } catch (const std::exception& e) {
            std::cerr << " Failed to load config: " << e.what() << "\n";
            return false;
        }
    }

    // Reload if file has changed
    bool reload_if_changed() {
        if (!fs::exists(config_file_path_)) {
            std::cerr << "  Config file not found: " << config_file_path_ <<
                "\n";
            return false;
        }

        auto current_write_time = fs::last_write_time(config_file_path_);

        if (current_write_time != last_write_time_) {
            std::cout << "\n Config file modified, reloading...\n";
            last_write_time_ = current_write_time;
            bool loaded = load();
            if (loaded) {
                print_changes();
            }
            return loaded;
        }

        return false;
    }

    // Get value by key (returns first match for multimap)
    std::string get(const std::string& key, const std::string& default_value =
```

```
        "") const {
        std::lock_guard<std::mutex> lock(mutex_);
        auto it = config_data_.find(key);
        if (it != config_data_.end()) {
            return it->second;
        }
        return default_value;
    }

    // Get all values for a key (multimap can have multiple values per key)
    std::vector<std::string> get_all(const std::string& key) const {
        std::lock_guard<std::mutex> lock(mutex_);
        std::vector<std::string> values;

        auto range = config_data_.equal_range(key);
        for (auto it = range.first; it != range.second; ++it) {
            values.push_back(it->second);
        }

        return values;
    }

    // Check if key exists
    bool has_key(const std::string& key) const {
        std::lock_guard<std::mutex> lock(mutex_);
        return config_data_.find(key) != config_data_.end();
    }

    // Get all keys
    std::vector<std::string> get_all_keys() const {
        std::lock_guard<std::mutex> lock(mutex_);
        std::vector<std::string> keys;

        for (const auto& [key, value] : config_data_) {
            keys.push_back(key);
        }

        return keys;
    }

    // Get configuration count
    size_t size() const {
        std::lock_guard<std::mutex> lock(mutex_);
        return config_data_.size();
    }

    // Display all configuration
    void display() const {
        std::lock_guard<std::mutex> lock(mutex_);

        std::cout << "\                            n\n";
        std::cout << "   Current Configuration (Version " << version_ << ")";
        std::cout << std::string(20 - std::to_string(version_.load()).length()
            , ' ') << " \n";
```

```
        std::cout << "                                    \n";

        if (config_data_.empty()) {
            std::cout << "   (empty)                                                    \n";
        } else {
            for (const auto& [key, value] : config_data_) {
                std::string line = "   " + key + ": " + value;
                line += std::string(58 - line.length(), ' ') + " ";
                std::cout << line << "\n";
            }
        }

        std::cout << "                                    \n\n";
    }

    int get_version() const {
        return version_.load();
    }

    std::string get_file_path() const {
        return config_file_path_;
    }

    void print_changes() {
        std::lock_guard<std::mutex> lock(mutex_);

        std::cout << "\n Configuration changes detected:\n";

        // Track unique keys
        std::set<std::string> all_keys;
        for (const auto& [key, _] : previous_config_data_) all_keys.insert(key);
        for (const auto& [key, _] : config_data_) all_keys.insert(key);

        for (const auto& key : all_keys) {
            auto prev_range = previous_config_data_.equal_range(key);
            auto curr_range = config_data_.equal_range(key);

            std::vector<std::string> prev_values;
            for (auto it = prev_range.first; it != prev_range.second; ++it) {
                prev_values.push_back(it->second);
            }

            std::vector<std::string> curr_values;
            for (auto it = curr_range.first; it != curr_range.second; ++it) {
                curr_values.push_back(it->second);
            }

            if (prev_values.empty() && !curr_values.empty()) {
                std::cout << "    ADDED to unordered_multimap: '" << key << "' = ";
                for (size_t i = 0; i < curr_values.size(); ++i) {
                    std::cout << "\"" << curr_values[i] << "\"";
```

```
                    if (i < curr_values.size() - 1) std::cout << ", ";
                }
                if (curr_values.size() > 1) {
                    std::cout << " (" << curr_values.size() << " entries)";
                }
                std::cout << "\n";
            } else if (!prev_values.empty() && curr_values.empty()) {
                std::cout << "    REMOVED from unordered_multimap: '" << key << "' (was: ";
                for (size_t i = 0; i < prev_values.size(); ++i) {
                    std::cout << "\"" << prev_values[i] << "\"";
                    if (i < prev_values.size() - 1) std::cout << ", ";
                }
                std::cout << ")\n";
            } else if (prev_values != curr_values) {
                std::cout << "    MODIFIED in unordered_multimap: '" << key << "'\n";
                std::cout << "      Old: ";
                for (size_t i = 0; i < prev_values.size(); ++i) {
                    std::cout << "\"" << prev_values[i] << "\"";
                    if (i < prev_values.size() - 1) std::cout << ", ";
                }
                std::cout << "\n      New: ";
                for (size_t i = 0; i < curr_values.size(); ++i) {
                    std::cout << "\"" << curr_values[i] << "\"";
                    if (i < curr_values.size() - 1) std::cout << ", ";
                }
                std::cout << "\n";
            }
        }
        std::cout << "\n";
    }
};

// ============================================================================
// SECTION 3: Configuration Monitor (Background Thread)
// ============================================================================

class ConfigMonitor {
private:
    ConfigManager& config_manager_;
    std::atomic<bool> running_{false};
    std::thread monitor_thread_;
    std::chrono::milliseconds check_interval_;

public:
    ConfigMonitor(ConfigManager& manager, std::chrono::milliseconds interval = 600000ms)
        : config_manager_(manager), check_interval_(interval) {}

```

```
    ~ConfigMonitor() {
        stop();
    }

    void start() {
        if (running_) {
            std::cout << "  Monitor already running\n";
            return;
        }

        running_ = true;
        monitor_thread_ = std::thread(&ConfigMonitor::monitor_loop, this);

        std::cout << "  Configuration monitor started (checking every "
                  << check_interval_.count() << "ms)\n";
    }

    void stop() {
        if (!running_) return;

        running_ = false;
        if (monitor_thread_.joinable()) {
            monitor_thread_.join();
        }

        std::cout << "  Configuration monitor stopped\n";
    }

private:
    void monitor_loop() {
        while (running_) {
            config_manager_.reload_if_changed();
            std::this_thread::sleep_for(check_interval_);
        }
    }
};

// ==========================================================================
// SECTION 4: Demonstration Application
// ==========================================================================

void demonstrate_basic_loading() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Demo 1: Basic Configuration Loading ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    ConfigManager config("../config.json");

    std::cout << "Loading initial configuration...\n";
    if (config.load()) {
```

```
        config.display();

        // Access individual values
        std::cout << "Accessing specific values:\n";
        std::cout << "  server_host: " << config.get("server_host", "not found") << "\n";
        std::cout << "  server_port: " << config.get("server_port", "not found") << "\n";
        std::cout << "  debug_mode: " << config.get("debug_mode", "not found") << "\n";
        std::cout << "  max_connections: " << config.get("max_connections", "not found") << "\n";

        // Check if keys exist
        std::cout << "\nKey existence checks:\n";
        std::cout << "  has 'server_host': " << (config.has_key("server_host") ? "Yes" : "No") << "\n";
        std::cout << "  has 'nonexistent': " << (config.has_key("nonexistent") ? "Yes" : "No") << "\n";

        // Get all keys
        std::cout << "\nAll keys: ";
        auto keys = config.get_all_keys();
        for (size_t i = 0; i < keys.size(); ++i) {
            std::cout << keys[i];
            if (i < keys.size() - 1) std::cout << ", ";
        }
        std::cout << "\n";
    }
}

void demonstrate_live_monitoring() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Demo 2: Live Configuration Monitoring ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    ConfigManager config("../config.json");
    std::cout << "  Reading config from: " << fs::absolute("../config.json") << "\n\n";
    config.load();
    config.display();

    // Start monitoring in background - use 10 seconds for demo (production: 10 minutes)
    ConfigMonitor monitor(config, 10000ms);  // Check every 10 seconds for demo
    monitor.start();

    // Read demoTime from config (default to 120 seconds)
    int demo_time = 120;
    std::string demo_time_str = config.get("demoTime");
    if (!demo_time_str.empty()) {
        try {
            demo_time = std::stoi(demo_time_str);
```

```
            std::cout << "  Using demoTime from config: " << demo_time << " seconds\n";
        } catch (...) {
            std::cout << "   Invalid demoTime in config, using default: 120 seconds\n";
        }
    }

    std::cout << "\n Instructions:\n";
    std::cout << "  1. Edit config.json file while this program is running\n";
    std::cout << "  2. Add new key-value pairs: \"new_key\": \"new_value\"\n";
    std::cout << "  3. Modify existing values: \"server_port\": \"9090\"\n";
    std::cout << "  4. Add arrays: \"allowed_ips\": [\"192.168.1.1\", \"10.0.0.1\"]\n";
    std::cout << "  5. Change \"demoTime\": 60 to adjust monitoring duration dynamically\n";
    std::cout << "  6. Save the file and watch for automatic reload!\n";
    std::cout << "\n  App will run for " << demo_time << " seconds (checks config every 10 seconds)...\n";
    std::cout << "  (In production: use 10 minutes / 600000ms)\n\n";

    // Simulate application running and accessing config
    int elapsed = 0;
    while (elapsed < demo_time) {
        std::this_thread::sleep_for(1s);
        elapsed++;

        // Periodically show current config version
        if (elapsed % 10 == 0) {
            std::cout << "  " << elapsed << "s - Config version: " << config.get_version()
                      << " | Entries: " << config.size();

            // Check if demoTime has changed
            std::string current_demo_time_str = config.get("demoTime");
            if (!current_demo_time_str.empty()) {
                try {
                    int new_demo_time = std::stoi(current_demo_time_str);
                    if (new_demo_time != demo_time) {
                        std::cout << " | demoTime changed: " << demo_time << "→" << new_demo_time << "s";
                        demo_time = new_demo_time;
                        std::cout << "\n Runtime adjusted! Will now run until " << demo_time << " seconds\n";
                    } else {
                        std::cout << "\n";
                    }
                } catch (...) {
                    std::cout << "\n";
                }
            } else {
                std::cout << "\n";
            }
        }
```

```
    }

    monitor.stop();

    std::cout << "\nFinal configuration state:\n";
    config.display();
}

void demonstrate_error_handling() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Demo 3: Error Handling ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Test with non-existent file
    std::cout << "1. Loading non-existent file:\n";
    ConfigManager config1("nonexistent.json");
    config1.load();

    // Test with invalid JSON
    std::cout << "\n2. Creating and loading invalid JSON:\n";
    std::ofstream bad_file("bad_config.json");
    bad_file << "{ invalid json without quotes }";
    bad_file.close();

    ConfigManager config2("bad_config.json");
    config2.load();

    // Cleanup
    fs::remove("bad_config.json");

    std::cout << "\n Error handling demonstrated\n";
    std::cout << " • File not found: gracefully handled\n";
    std::cout << " • Invalid JSON: exception caught and reported\n";
}

void demonstrate_multimap_features() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Demo 4: unordered_multimap Features ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Concept: unordered_multimap allows multiple values per key\n";
    std::cout << "Use case: Configuration with arrays or multiple values\n\n";

    std::cout << "Example JSON (if extended):\n";
    std::cout << R"({
  "server": "localhost",
  "allowed_ip": "192.168.1.1",
  "allowed_ip": "192.168.1.2",
  "allowed_ip": "10.0.0.1"
})" << "\n\n";

    std::cout << "With unordered_multimap, you can store multiple 'allowed_ip' values\n";
```

```
    std::cout << "and retrieve them all using get_all('allowed_ip')\n\n";

    std::cout << "Current implementation: Simple key-value pairs\n";
    std::cout << "If duplicate keys in JSON: last one wins (map behavior)\n";
    std::cout << "To enable true multimap: parse JSON arrays into separate entries\n\n";

    ConfigManager config("../config.json");
    config.load();

    std::cout << "Accessing all values for a key:\n";
    auto values = config.get_all("server_host");
    std::cout << "  'server_host' has " << values.size() << " value(s):\n";
    for (const auto& val : values) {
        std::cout << "    - " << val << "\n";
    }
}

void create_sample_config_if_not_exists() {
    if (!fs::exists("config.json")) {
        std::cout << "  Creating sample config.json...\n";
        std::ofstream config_file("config.json");
        config_file << R"({
  "server_host": "localhost",
  "server_port": "8080",
  "debug_mode": "true",
  "max_connections": "100",
  "timeout_seconds": "30",
  "database_url": "mongodb://localhost:27017"
})";
        config_file.close();
        std::cout << "  Sample config.json created\n\n";
    }
}

// ==========================================================================
// SECTION 5: Best Practices Guide
// ==========================================================================

void show_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Configuration Management Best Practices ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DESIGN PATTERNS:\n";
    std::cout << "  1. Singleton pattern for global config access\n";
    std::cout << "  2. Observer pattern for change notifications\n";
    std::cout << "  3. Strategy pattern for different config sources\n";
    std::cout << "  4. Builder pattern for complex configurations\n\n";

```

```
    std::cout << "  THREAD SAFETY:\n";
    std::cout << "   1. Use std::mutex for protecting config access\n";
    std::cout << "   2. std::shared_mutex for read-write locks (C++17)\n";
    std::cout << "   3. std::atomic for version counters\n";
    std::cout << "   4. Lock-free structures for high-performance\n\n";

    std::cout << "  FILE MONITORING:\n";
    std::cout << "   1. Check file modification time (fs::last_write_time)\n";
    std::cout << "   2. Use inotify (Linux) / ReadDirectoryChangesW (Windows)\n";
    std::cout << "   3. Background thread with periodic checks\n";
    std::cout << "   4. Debounce rapid changes (wait for settle)\n\n";

    std::cout << "  ERROR HANDLING:\n";
    std::cout << "   1. Validate JSON syntax before parsing\n";
    std::cout << "   2. Keep previous valid config on parse errors\n";
    std::cout << "   3. Log all config changes and errors\n";
    std::cout << "   4. Provide default values for missing keys\n\n";

    std::cout << "  PERFORMANCE:\n";
    std::cout << "   1. Cache frequently accessed values\n";
    std::cout << "   2. Use unordered_map for O(1) lookups\n";
    std::cout << "   3. Minimize file I/O (check timestamp first)\n";
    std::cout << "   4. Parse JSON incrementally for large files\n\n";

    std::cout << "  PRODUCTION CONSIDERATIONS:\n";
    std::cout << "   1. Support multiple config sources (file, env, CLI)\n";
    std::cout << "   2. Implement config versioning and rollback\n";
    std::cout << "   3. Add config validation (schemas, constraints)\n";
    std::cout << "   4. Enable hot-reload without service restart\n";
    std::cout << "   5.  Using nlohmann/json library (header-only, modern C++)\n";
    std::cout << "   6. Implement config encryption for sensitive data\n";
    std::cout << "   7. Set appropriate monitoring interval (10 min for production)\n\n";
}

// ==========================================================================
// MAIN FUNCTION
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                  \n";
    std::cout << "   Configuration Loader and Live Monitor                  \n";
    std::cout << "   Using std::unordered_multimap + File Watching          \n";
    std::cout << "                                  \n";

```

```
    // Create sample config if it doesn't exist
    create_sample_config_if_not_exists();

    // Run demonstrations
    demonstrate_basic_loading();
    demonstrate_multimap_features();
    demonstrate_error_handling();

    // Interactive monitoring demo
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "Would you like to run the live monitoring demo? (y/n): ";
    char choice;
    std::cin >> choice;

    if (choice == 'y' || choice == 'Y') {
        demonstrate_live_monitoring();
    } else {
        std::cout << "\nSkipping live monitoring demo.\n";
        std::cout << "To test manually:\n";
        std::cout << "  1. Run this program\n";
        std::cout << "  2. In another terminal: echo '{\"test\": \"value\"}' > config.json\n";
        std::cout << "  3. Watch the program detect changes!\n";
    }

    show_best_practices();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << "\nKEY FEATURES DEMONSTRATED:\n";
    std::cout << "    std::unordered_multimap for config storage\n";
    std::cout << "    JSON parsing and validation\n";
    std::cout << "    File modification detection\n";
    std::cout << "    Automatic config reloading\n";
    std::cout << "    Thread-safe access with std::mutex\n";
    std::cout << "    Background monitoring thread\n";
    std::cout << "    Error handling and recovery\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

11 Source Code: Cpp11Examples.cpp

File: `src/Cpp11Examples.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <thread>
#include <chrono>
#include <tuple>
#include <array>
#include <unordered_map>
#include <functional>
#include <algorithm>
#include <numeric>
#include <type_traits>

// ====================================================================
// C++11 COMPREHENSIVE EXAMPLES
// ====================================================================

// ====================================================================
// 1. AUTO TYPE DEDUCTION
// ====================================================================
void example_auto() {
    std::cout << "\n=== 1. AUTO TYPE DEDUCTION ===" << std::endl;

    auto a = 42;              // int
    auto b = 3.14;            // double
    auto c = "hello";         // const char*
    auto d = std::string("world");  // std::string

    std::vector<int> vec = {1, 2, 3};
    auto it = vec.begin();    // std::vector<int>::iterator

    std::cout << "auto deduced types: int, double, const char*, std::string"
        << std::endl;
}

// ====================================================================
// 2. NULLPTR
// ====================================================================
void foo(int) { std::cout << "Called foo(int)" << std::endl; }
void foo(int*) { std::cout << "Called foo(int*)" << std::endl; }

void example_nullptr() {
    std::cout << "\n=== 2. NULLPTR ===" << std::endl;

    int* ptr = nullptr;
    foo(nullptr);  // Calls foo(int*)
    std::cout << "nullptr is " << (ptr == nullptr ? "null" : "not null") <<
        std::endl;
```

```
}

// ==================================================================
// 3. RANGE-BASED FOR LOOPS
// ==================================================================
void example_range_for() {
    std::cout << "\n=== 3. RANGE-BASED FOR LOOPS ===" << std::endl;

    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::cout << "Elements: ";
    for (const auto& elem : vec) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 4. LAMBDA EXPRESSIONS
// ==================================================================
void example_lambdas() {
    std::cout << "\n=== 4. LAMBDA EXPRESSIONS ===" << std::endl;

    int x = 10;
    auto add_x = [x](int y) { return x + y; };
    std::cout << "Lambda add_x(5) = " << add_x(5) << std::endl;

    auto square = [](int n) { return n * n; };
    std::cout << "Lambda square(7) = " << square(7) << std::endl;

    std::vector<int> vec = {1, 2, 3, 4, 5};
    int sum = 0;
    std::for_each(vec.begin(), vec.end(), [&sum](int n) { sum += n; });
    std::cout << "Sum using lambda: " << sum << std::endl;
}

// ==================================================================
// 5. RVALUE REFERENCES AND MOVE SEMANTICS
// ==================================================================
class MoveableResource {
private:
    int* data;
    size_t size;
public:
    MoveableResource(size_t s) : size(s) {
        data = new int[size];
        std::cout << "Allocated resource of size " << size << std::endl;
    }

    ~MoveableResource() {
        delete[] data;
    }

    // Move constructor
    MoveableResource(MoveableResource&& other) noexcept
```

```
        : data(other.data), size(other.size) {
        other.data = nullptr;
        other.size = 0;
        std::cout << "Moved resource" << std::endl;
    }

    // Move assignment
    MoveableResource& operator=(MoveableResource&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            size = other.size;
            other.data = nullptr;
            other.size = 0;
        }
        return *this;
    }
};

void example_move_semantics() {
    std::cout << "\n=== 5. MOVE SEMANTICS ===" << std::endl;

    MoveableResource res1(100);
    MoveableResource res2 = std::move(res1);  // Move constructor
}

// ==================================================================
// 6. VARIADIC TEMPLATES
// ==================================================================
template <typename... Args>
void print_args(Args... args) {
    (void)std::initializer_list<int>{
        (std::cout << args << " ", 0)...
    };
    std::cout << std::endl;
}

template <typename... T>
struct arity {
    static constexpr int value = sizeof...(T);
};

void example_variadic_templates() {
    std::cout << "\n=== 6. VARIADIC TEMPLATES ===" << std::endl;

    print_args(1, "hello", 3.14, "world");
    std::cout << "arity<int, double, char>::value = "
              << arity<int, double, char>::value << std::endl;
}

// ==================================================================
// 7. STRONGLY-TYPED ENUMS
// ==================================================================
enum class Color { Red, Green, Blue };
```

```
enum class Traffic { Red, Yellow, Green };  // No conflict!

void example_strongly_typed_enums() {
    std::cout << "\n=== 7. STRONGLY-TYPED ENUMS ===" << std::endl;

    Color c = Color::Red;
    Traffic t = Traffic::Red;

    // Color and Traffic::Red don't conflict
    std::cout << "Strongly-typed enums prevent naming conflicts" << std::endl;
}

// ==================================================================
// 8. INITIALIZER LISTS
// ==================================================================
int sum_list(const std::initializer_list<int>& list) {
    int total = 0;
    for (auto elem : list) {
        total += elem;
    }
    return total;
}

void example_initializer_lists() {
    std::cout << "\n=== 8. INITIALIZER LISTS ===" << std::endl;

    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::cout << "Vector initialized with {1, 2, 3, 4, 5}" << std::endl;
    std::cout << "sum_list({10, 20, 30}) = " << sum_list({10, 20, 30}) << std
        ::endl;
}

// ==================================================================
// 9. DECLTYPE
// ==================================================================
template <typename X, typename Y>
auto multiply(X x, Y y) -> decltype(x * y) {
    return x * y;
}

void example_decltype() {
    std::cout << "\n=== 9. DECLTYPE ===" << std::endl;

    int a = 5;
    decltype(a) b = 10;  // b is int

    std::cout << "multiply(3, 4) = " << multiply(3, 4) << std::endl;
    std::cout << "multiply(2.5, 4) = " << multiply(2.5, 4) << std::endl;
}

// ==================================================================
// 10. TYPE ALIASES
// ==================================================================
template <typename T>
```

```
using Vec = std::vector<T>;

using String = std::string;

void example_type_aliases() {
    std::cout << "\n=== 10. TYPE ALIASES ===" << std::endl;

    Vec<int> numbers = {1, 2, 3};
    String text = "Hello";

    std::cout << "Vec<int> and String aliases work" << std::endl;
}

// ==================================================================
// 11. CONSTEXPR
// ==================================================================
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

constexpr int square(int n) {
    return n * n;
}

void example_constexpr() {
    std::cout << "\n=== 11. CONSTEXPR ===" << std::endl;

    constexpr int fact5 = factorial(5);
    constexpr int sq7 = square(7);

    std::cout << "constexpr factorial(5) = " << fact5 << std::endl;
    std::cout << "constexpr square(7) = " << sq7 << std::endl;
}

// ==================================================================
// 12. STATIC ASSERTIONS
// ==================================================================
void example_static_assert() {
    std::cout << "\n=== 12. STATIC ASSERTIONS ===" << std::endl;

    static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
    static_assert(sizeof(char) == 1, "char must be 1 byte");

    std::cout << "Static assertions passed" << std::endl;
}

// ==================================================================
// 13. DELEGATING CONSTRUCTORS
// ==================================================================
class Rectangle {
private:
    int width, height;
public:
    Rectangle() : Rectangle(0, 0) {}  // Delegate to another constructor
```

```
    Rectangle(int w) : Rectangle(w, w) {}  // Delegate
    Rectangle(int w, int h) : width(w), height(h) {
        std::cout << "Rectangle(" << width << ", " << height << ")" << std::endl;
    }
};

void example_delegating_constructors() {
    std::cout << "\n=== 13. DELEGATING CONSTRUCTORS ===" << std::endl;

    Rectangle r1;
    Rectangle r2(10);
    Rectangle r3(5, 8);
}

// ==================================================================
// 14. DEFAULT AND DELETE
// ==================================================================
class NonCopyable {
public:
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;
};

void example_default_delete() {
    std::cout << "\n=== 14. DEFAULT AND DELETE ===" << std::endl;

    NonCopyable obj1;
    // NonCopyable obj2 = obj1;  // Error: copy constructor deleted

    std::cout << "Non-copyable class works" << std::endl;
}

// ==================================================================
// 15. OVERRIDE AND FINAL
// ==================================================================
class Base {
public:
    virtual void foo() { std::cout << "Base::foo()" << std::endl; }
    virtual ~Base() = default;
};

class Derived final : public Base {
public:
    void foo() override { std::cout << "Derived::foo()" << std::endl; }
};

void example_override_final() {
    std::cout << "\n=== 15. OVERRIDE AND FINAL ===" << std::endl;

    std::unique_ptr<Base> ptr(new Derived());
    ptr->foo();
}
```

```

// ==================================================================
// 16. SMART POINTERS
// ==================================================================
void example_smart_pointers() {
    std::cout << "\n=== 16. SMART POINTERS ===" << std::endl;

    // unique_ptr
    std::unique_ptr<int> uptr(new int(42));
    std::cout << "*uptr = " << *uptr << std::endl;

    // shared_ptr
    std::shared_ptr<int> sptr1 = std::make_shared<int>(100);
    std::shared_ptr<int> sptr2 = sptr1;
    std::cout << "shared_ptr count: " << sptr1.use_count() << std::endl;

    // weak_ptr
    std::weak_ptr<int> wptr = sptr1;
    std::cout << "weak_ptr valid: " << !wptr.expired() << std::endl;
}

// ==================================================================
// 17. TUPLES
// ==================================================================
void example_tuples() {
    std::cout << "\n=== 17. TUPLES ===" << std::endl;

    std::tuple<int, std::string, double> tup{42, "hello", 3.14};

    std::cout << "Tuple element 0: " << std::get<0>(tup) << std::endl;
    std::cout << "Tuple element 1: " << std::get<1>(tup) << std::endl;
    std::cout << "Tuple element 2: " << std::get<2>(tup) << std::endl;

    int a;
    std::string b;
    double c;
    std::tie(a, b, c) = tup;
    std::cout << "Using tie: " << a << ", " << b << ", " << c << std::endl;
}

// ==================================================================
// 18. STD::ARRAY
// ==================================================================
void example_array() {
    std::cout << "\n=== 18. STD::ARRAY ===" << std::endl;

    std::array<int, 5> arr = {1, 2, 3, 4, 5};

    std::cout << "Array size: " << arr.size() << std::endl;
    std::cout << "Array elements: ";
    for (const auto& elem : arr) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;
```

```
}

// ==================================================================
// 19. UNORDERED CONTAINERS
// ==================================================================
void example_unordered_containers() {
    std::cout << "\n=== 19. UNORDERED CONTAINERS ===" << std::endl;

    std::unordered_map<std::string, int> map;
    map["one"] = 1;
    map["two"] = 2;
    map["three"] = 3;

    std::cout << "Unordered map: ";
    for (const auto& pair : map) {
        std::cout << pair.first << "=" << pair.second << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 20. STD::THREAD
// ==================================================================
void thread_function(int n) {
    std::cout << "Thread " << n << " running" << std::endl;
}

void example_threads() {
    std::cout << "\n=== 20. STD::THREAD ===" << std::endl;

    std::thread t1(thread_function, 1);
    std::thread t2(thread_function, 2);

    t1.join();
    t2.join();

    std::cout << "Threads completed" << std::endl;
}

// ==================================================================
// 21. CHRONO
// ==================================================================
void example_chrono() {
    std::cout << "\n=== 21. STD::CHRONO ===" << std::endl;

    auto start = std::chrono::high_resolution_clock::now();

    // Do some work
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end
        - start);

```

```
    std::cout << "Duration: " << duration.count() << " ms" << std::endl;
}

// ==================================================================
// 22. RAW STRING LITERALS
// ==================================================================
void example_raw_strings() {
    std::cout << "\n=== 22. RAW STRING LITERALS ===" << std::endl;

    std::string normal = "Line 1\\nLine 2\\nLine 3";
    std::string raw = R"(Line 1
Line 2
Line 3)";

    std::cout << "Raw string with newlines preserved" << std::endl;
}

// ==================================================================
// 23. ATTRIBUTES
// ==================================================================
[[noreturn]] void terminate_program() {
    std::exit(0);
}

void example_attributes() {
    std::cout << "\n=== 23. ATTRIBUTES ===" << std::endl;

    std::cout << "[[noreturn]] attribute available" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    C++11 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
    std::cout << "================================================" << std::
        endl;

    example_auto();
    example_nullptr();
    example_range_for();
    example_lambdas();
    example_move_semantics();
    example_variadic_templates();
    example_strongly_typed_enums();
    example_initializer_lists();
    example_decltype();
    example_type_aliases();
    example_constexpr();
    example_static_assert();
    example_delegating_constructors();
    example_default_delete();
```

```
    example_override_final();
    example_smart_pointers();
    example_tuples();
    example_array();
    example_unordered_containers();
    example_threads();
    example_chrono();
    example_raw_strings();
    example_attributes();

    std::cout << "\n==============================================" << std::endl;
    std::cout << "    ALL C++11 EXAMPLES COMPLETED" << std::endl;
    std::cout << "==============================================\n" << std::endl;

    return 0;
}
```

12 Source Code: Cpp14Examples.cpp

File: `src/Cpp14Examples.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <algorithm>
#include <utility>

// ==================================================================
// C++14 LANGUAGE FEATURES
// ==================================================================

// ==================================================================
// 1. BINARY LITERALS
// ==================================================================
void example_binary_literals() {
    std::cout << "\n=== 1. BINARY LITERALS ===" << std::endl;

    int a = 0b110;              // Binary literal
    int b = 0b1111'1111;        // Binary with digit separator
    int c = 0b1010'0101'1100;   // Larger binary number

    std::cout << "0b110 = " << a << std::endl;
    std::cout << "0b1111'1111 = " << b << std::endl;
    std::cout << "0b1010'0101'1100 = " << c << std::endl;
}

// ==================================================================
// 2. GENERIC LAMBDA EXPRESSIONS
// ==================================================================
void example_generic_lambdas() {
    std::cout << "\n=== 2. GENERIC LAMBDA EXPRESSIONS ===" << std::endl;

    auto identity = [](auto x) { return x; };

    int three = identity(3);
    std::string foo = identity(std::string("foo"));
    double pi = identity(3.14);

    std::cout << "Identity(3) = " << three << std::endl;
    std::cout << "Identity(\"foo\") = " << foo << std::endl;
    std::cout << "Identity(3.14) = " << pi << std::endl;

    // Generic lambda with multiple parameters
    auto add = [](auto a, auto b) { return a + b; };
    std::cout << "add(10, 20) = " << add(10, 20) << std::endl;
    std::cout << "add(1.5, 2.5) = " << add(1.5, 2.5) << std::endl;
}

// ==================================================================
```

```
// 3. LAMBDA CAPTURE INITIALIZERS
// ==================================================================
void example_lambda_capture_initializers() {
    std::cout << "\n=== 3. LAMBDA CAPTURE INITIALIZERS ===" << std::endl;

    // Initialize capture with expression
    int factory_value = 2;
    auto f = [x = factory_value * 10] { return x; };
    std::cout << "Lambda with initialized capture: " << f() << std::endl;

    // Generator with mutable capture
    auto generator = [x = 0]() mutable {
        return x++;
    };
    std::cout << "Generator call 1: " << generator() << std::endl;
    std::cout << "Generator call 2: " << generator() << std::endl;
    std::cout << "Generator call 3: " << generator() << std::endl;

    // Move-only types in lambda
    auto ptr = std::make_unique<int>(42);
    auto task = [p = std::move(ptr)] {
        std::cout << "Value in lambda: " << *p << std::endl;
    };
    task();
    std::cout << "Original ptr is " << (ptr ? "valid" : "null") << std::endl;
}

// ==================================================================
// 4. RETURN TYPE DEDUCTION
// ==================================================================
auto multiply(int a, int b) {
    return a * b;  // Return type deduced as int
}

template <typename T>
auto add_values(T a, T b) {
    return a + b;  // Return type deduced from expression
}

void example_return_type_deduction() {
    std::cout << "\n=== 4. RETURN TYPE DEDUCTION ===" << std::endl;

    std::cout << "multiply(5, 6) = " << multiply(5, 6) << std::endl;
    std::cout << "add_values(10, 20) = " << add_values(10, 20) << std::endl;
    std::cout << "add_values(1.5, 2.5) = " << add_values(1.5, 2.5) << std::
        endl;

    // Lambda with auto return type
    auto square = [](auto x) { return x * x; };
    std::cout << "square(7) = " << square(7) << std::endl;
}

// ==================================================================
// 5. DECLTYPE(AUTO)
```

```
// ==================================================================
template <typename Container>
decltype(auto) get_first(Container& c) {
    return c[0];  // Returns reference if c[0] is a reference
}

void example_decltype_auto() {
    std::cout << "\n=== 5. DECLTYPE(AUTO) ===" << std::endl;

    const int x = 0;
    auto x1 = x;              // int
    decltype(auto) x2 = x;    // const int

    std::cout << "auto strips const, decltype(auto) preserves it" << std::endl
        ;

    std::vector<int> vec = {1, 2, 3};
    decltype(auto) first = get_first(vec);
    first = 100;  // Modifies vec[0]
    std::cout << "Modified first element: " << vec[0] << std::endl;
}

// ==================================================================
// 6. RELAXED CONSTEXPR FUNCTIONS
// ==================================================================
constexpr int factorial(int n) {
    // C++14 allows multiple statements in constexpr
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}

constexpr int fibonacci(int n) {
    if (n <= 1) return n;
    int a = 0, b = 1;
    for (int i = 2; i <= n; ++i) {
        int temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}

void example_relaxed_constexpr() {
    std::cout << "\n=== 6. RELAXED CONSTEXPR FUNCTIONS ===" << std::endl;

    constexpr int fact5 = factorial(5);
    constexpr int fib10 = fibonacci(10);

    std::cout << "factorial(5) = " << fact5 << std::endl;
    std::cout << "fibonacci(10) = " << fib10 << std::endl;
}
```

```

// ==================================================================
// 7. VARIABLE TEMPLATES
// ==================================================================
template <typename T>
constexpr T pi = T(3.1415926535897932385);

template <typename T>
constexpr T e = T(2.7182818284590452354);

void example_variable_templates() {
    std::cout << "\n=== 7. VARIABLE TEMPLATES ===" << std::endl;

    std::cout << "pi<float> = " << pi<float> << std::endl;
    std::cout << "pi<double> = " << pi<double> << std::endl;
    std::cout << "e<float> = " << e<float> << std::endl;
    std::cout << "e<double> = " << e<double> << std::endl;
}

// ==================================================================
// 8. [[DEPRECATED]] ATTRIBUTE
// ==================================================================
[[deprecated("Use new_function() instead")]]
void old_function() {
    std::cout << "This is deprecated!" << std::endl;
}

void new_function() {
    std::cout << "This is the new function!" << std::endl;
}

void example_deprecated_attribute() {
    std::cout << "\n=== 8. [[DEPRECATED]] ATTRIBUTE ===" << std::endl;

    // old_function();  // Would generate compiler warning
    new_function();
    std::cout << "Using new function instead of deprecated one" << std::endl;
}

// ==================================================================
// C++14 LIBRARY FEATURES
// ==================================================================

// ==================================================================
// 9. USER-DEFINED LITERALS FOR STANDARD LIBRARY
// ==================================================================
void example_user_defined_literals() {
    std::cout << "\n=== 9. USER-DEFINED LITERALS ===" << std::endl;

    using namespace std::string_literals;

    auto str = "Hello"s;  // std::string

    std::cout << "String literal type: std::string" << std::endl;
```

```
    std::cout << "User-defined string literals work in C++14" << std::endl;
}

// ==================================================================
// 10. STD::MAKE_UNIQUE
// ==================================================================
void example_make_unique() {
    std::cout << "\n=== 10. STD::MAKE_UNIQUE ===" << std::endl;

    auto ptr1 = std::make_unique<int>(42);
    auto ptr2 = std::make_unique<std::string>("Hello");
    auto ptr3 = std::make_unique<std::vector<int>>(5, 10);

    std::cout << "*ptr1 = " << *ptr1 << std::endl;
    std::cout << "*ptr2 = " << *ptr2 << std::endl;
    std::cout << "ptr3 size = " << ptr3->size() << std::endl;

    // Array version
    auto arr = std::make_unique<int[]>(5);
    for (int i = 0; i < 5; ++i) {
        arr[i] = i * 10;
    }
    std::cout << "Array elements: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 11. COMPILE-TIME INTEGER SEQUENCES
// ==================================================================
template <typename T, T... Ints>
void print_sequence(std::integer_sequence<T, Ints...>) {
    std::cout << "Sequence: ";
    int dummy[] = {0, (std::cout << Ints << " ", 0)...};
    (void)dummy;
    std::cout << std::endl;
}

void example_integer_sequences() {
    std::cout << "\n=== 11. COMPILE-TIME INTEGER SEQUENCES ===" << std::endl;

    using Seq = std::integer_sequence<int, 0, 1, 2, 3, 4>;
    using IdxSeq = std::make_index_sequence<5>;

    std::cout << "Integer sequence example (needs C++17 fold for printing)" <<
        std::endl;
    std::cout << "std::make_index_sequence<5> creates indices 0,1,2,3,4" <<
        std::endl;
}

// ==================================================================
// 12. DIGIT SEPARATORS
```

```
// ==================================================================
void example_digit_separators() {
    std::cout << "\n=== 12. DIGIT SEPARATORS ===" << std::endl;

    int decimal = 1'000'000;
    int hex = 0xDEAD'BEEF;
    int binary = 0b1010'1010'1010;
    double floating = 3.141'592'653'589;

    std::cout << "1'000'000 = " << decimal << std::endl;
    std::cout << "0xDEAD'BEEF = " << hex << std::endl;
    std::cout << "0b1010'1010'1010 = " << binary << std::endl;
    std::cout << "3.141'592'653'589 = " << floating << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    C++14 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
    std::cout << "================================================" << std::
        endl;

    // Language Features
    example_binary_literals();
    example_generic_lambdas();
    example_lambda_capture_initializers();
    example_return_type_deduction();
    example_decltype_auto();
    example_relaxed_constexpr();
    example_variable_templates();
    example_deprecated_attribute();

    // Library Features
    example_user_defined_literals();
    example_make_unique();
    example_integer_sequences();
    example_digit_separators();

    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    ALL C++14 EXAMPLES COMPLETED" << std::endl;
    std::cout << "================================================\n" << std::
        endl;

    return 0;
}
```

13 Source Code: Cpp17Concurrency.cpp

File: `src/Cpp17Concurrency.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <future>
#include <thread>
#include <chrono>
#include <vector>
#include <numeric>
#include <algorithm>
#include <string>
#include <random>
#include <mutex>
#include <queue>
#include <condition_variable>
#include <atomic>
#include <type_traits>
#include <variant>
#include <optional>
#include <any>
#include <tuple>
#include <filesystem>
#include <sstream>
#include <fstream>
#include <map>
#include <array>
#include <functional>

using namespace std;
using namespace std::chrono;
using namespace std::chrono_literals;

// ==================================================================
// 1. BASIC FUTURE/PROMISE
// ==================================================================
void example_basic_future_promise() {
    cout << "\n=== 1. BASIC FUTURE/PROMISE ===" << endl;

    promise<int> promise_obj;
    future<int> future_obj = promise_obj.get_future();

    thread t([&promise_obj]() {
        cout << "  Thread: Doing some computation..." << endl;
        this_thread::sleep_for(1s);

        int result = 42;
        cout << "  Thread: Setting promise value: " << result << endl;
        promise_obj.set_value(result);
    });

    cout << "Main: Waiting for result..." << endl;
    int result = future_obj.get();
```

```
    cout << "Main: Got result: " << result << endl;

    t.join();
}

// ==================================================================
// 2. FUTURE WITH EXCEPTION
// ==================================================================
void example_future_exception() {
    cout << "\n=== 2. FUTURE WITH EXCEPTION ===" << endl;

    promise<int> promise_obj;
    future<int> future_obj = promise_obj.get_future();

    thread t([&promise_obj]() {
        try {
            cout << "  Thread: Simulating error..." << endl;
            throw runtime_error("Something went wrong!");
        }
        catch (...) {
            promise_obj.set_exception(current_exception());
        }
    });

    try {
        cout << "Main: Waiting for result..." << endl;
        int result = future_obj.get();
        cout << "Main: Got result: " << result << endl;
    }
    catch (const exception& e) {
        cout << "Main: Caught exception: " << e.what() << endl;
    }

    t.join();
}

// ==================================================================
// 3. ASYNC (HIGH-LEVEL FUTURE)
// ==================================================================
void example_async() {
    cout << "\n=== 3. ASYNC (HIGH-LEVEL FUTURE) ===" << endl;

    auto future1 = async(launch::async, []() {
        cout << "  Async task 1 running..." << endl;
        this_thread::sleep_for(500ms);
        return 42;
    });

    auto future2 = async(launch::async, []() {
        cout << "  Async task 2 running..." << endl;
        this_thread::sleep_for(300ms);
        return 100;
    });

```

```
    auto future3 = async(launch::deferred, []() {
        cout << "  Deferred task running..." << endl;
        return 200;
    });

    cout << "Main: Doing other work..." << endl;
    this_thread::sleep_for(100ms);

    cout << "Result 1: " << future1.get() << endl;
    cout << "Result 2: " << future2.get() << endl;
    cout << "Result 3 (deferred): " << future3.get() << endl;
}

// ==================================================================
// 4. SHARED FUTURE (MULTIPLE WAITERS)
// ==================================================================
void example_shared_future() {
    cout << "\n=== 4. SHARED FUTURE (MULTIPLE WAITERS) ===" << endl;

    promise<int> promise_obj;
    future<int> future_obj = promise_obj.get_future();
    shared_future<int> shared_future_obj = future_obj.share();

    vector<thread> threads;
    for (int i = 0; i < 3; i++) {
        threads.emplace_back([i, shared_future_obj]() {
            cout << "  Thread " << i << ": Waiting for result..." << endl;
            int result = shared_future_obj.get();
            cout << "  Thread " << i << ": Got result: " << result << endl;
        });
    }

    thread producer([&promise_obj]() {
        this_thread::sleep_for(1s);
        cout << "  Producer: Setting value to 999" << endl;
        promise_obj.set_value(999);
    });

    for (auto& t : threads) t.join();
    producer.join();
}

// ==================================================================
// 5. FUTURE WITH TIMEOUT
// ==================================================================
void example_future_timeout() {
    cout << "\n=== 5. FUTURE WITH TIMEOUT ===" << endl;

    promise<int> promise_obj;
    future<int> future_obj = promise_obj.get_future();

    thread slow_worker([&promise_obj]() {
        cout << "  Slow worker: Starting (3 seconds)" << endl;
        this_thread::sleep_for(3s);
```

```
        promise_obj.set_value(42);
        cout << "  Slow worker: Finished" << endl;
    });

    cout << "Main: Waiting with timeout (2 seconds)..." << endl;

    future_status status = future_obj.wait_for(2s);

    if (status == future_status::ready) {
        cout << "Main: Got result: " << future_obj.get() << endl;
    }
    else if (status == future_status::timeout) {
        cout << "Main: Timeout! Task not ready yet" << endl;
    }

    this_thread::sleep_for(2s);
    if (slow_worker.joinable()) {
        if (future_obj.wait_for(0s) == future_status::ready) {
            cout << "Main: Finally got result: " << future_obj.get() << endl;
        }
        slow_worker.join();
    }
}

// ==================================================================
// 6. PACKAGED TASK
// ==================================================================
void example_packaged_task() {
    cout << "\n=== 6. PACKAGED TASK ===" << endl;

    packaged_task<int(int, int)> task([](int a, int b) {
        cout << "  Task executing with " << a << " and " << b << endl;
        this_thread::sleep_for(500ms);
        return a + b;
    });

    future<int> result = task.get_future();

    thread t(move(task), 10, 20);

    cout << "Main: Waiting for packaged task result..." << endl;
    cout << "Result: " << result.get() << endl;

    t.join();
}

// ==================================================================
// 7. ASYNC PARALLEL COMPUTATION
// ==================================================================
void example_parallel_computation() {
    cout << "\n=== 7. ASYNC PARALLEL COMPUTATION ===" << endl;

    auto compute_chunk = [](int start, int end) {
        long long sum = 0;
```

```
        for (int i = start; i <= end; i++) {
            sum += i;
        }
        cout << "  Chunk " << start << "-" << end << ": " << sum << endl;
        return sum;
    };

    const int N = 1000000;
    const int num_threads = 4;
    const int chunk_size = N / num_threads;

    vector<future<long long>> futures;

    auto start_time = high_resolution_clock::now();

    for (int i = 0; i < num_threads; i++) {
        int chunk_start = i * chunk_size + 1;
        int chunk_end = (i == num_threads - 1) ? N : (i + 1) * chunk_size;

        futures.push_back(async(launch::async, compute_chunk,
                                chunk_start, chunk_end));
    }

    long long total = 0;
    for (auto& f : futures) {
        total += f.get();
    }

    auto end_time = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end_time - start_time);

    cout << "Total sum 1.." << N << ": " << total << endl;
    cout << "Parallel time: " << duration.count() << "ms" << endl;

    // Sequential comparison
    start_time = high_resolution_clock::now();
    long long seq_total = 0;
    for (int i = 1; i <= N; i++) {
        seq_total += i;
    }
    end_time = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(end_time - start_time);

    cout << "Sequential sum: " << seq_total << endl;
    cout << "Sequential time: " << duration.count() << "ms" << endl;
}

// ==================================================================
// 8. FUTURE CONTINUATIONS (MANUAL IMPLEMENTATION)
// ==================================================================
void example_future_continuations() {
    cout << "\n=== 8. FUTURE CONTINUATIONS (MANUAL) ===" << endl;

    // Using nested async for continuations
```

```
    auto future1 = async(launch::async, []() {
        cout << "  Step 1: Computing 42..." << endl;
        this_thread::sleep_for(300ms);
        return 42;
    });

    // Continuation 1
    auto future2 = async(launch::async, [future1 = move(future1)]() mutable {
        int x = future1.get();  // Wait for first result
        cout << "  Step 2: Doubling " << x << "..." << endl;
        this_thread::sleep_for(300ms);
        return x * 2;
    });

    // Continuation 2
    auto future3 = async(launch::async, [future2 = move(future2)]() mutable {
        int x = future2.get();  // Wait for second result
        cout << "  Step 3: Adding 100 to " << x << "..." << endl;
        this_thread::sleep_for(300ms);
        return x + 100;
    });

    cout << "Main: Waiting for pipeline..." << endl;
    int result = future3.get();
    cout << "Final result: " << result << endl;
}

// ==================================================================
// 9. WHEN_ALL IMPLEMENTATION (SIMPLIFIED)
// ==================================================================
void example_when_all() {
    cout << "\n=== 9. WHEN_ALL (WAIT FOR ALL FUTURES) ===" << endl;

    auto task1 = async(launch::async, []() {
        this_thread::sleep_for(800ms);
        cout << "  Task 1 complete" << endl;
        return string("Result 1");
    });

    auto task2 = async(launch::async, []() {
        this_thread::sleep_for(500ms);
        cout << "  Task 2 complete" << endl;
        return 42;
    });

    auto task3 = async(launch::async, []() {
        this_thread::sleep_for(300ms);
        cout << "  Task 3 complete" << endl;
        return 3.14;
    });

    cout << "Main: Waiting for all tasks..." << endl;

    // Manual when_all - just get all results
```

```
    string result1 = task1.get();
    int result2 = task2.get();
    double result3 = task3.get();

    cout << "All tasks complete!" << endl;
    cout << "Results: " << result1 << ", " << result2 << ", " << result3 << endl;
}

// ==================================================================
// 10. WHEN_ANY IMPLEMENTATION (SIMPLIFIED)
// ==================================================================
void example_when_any() {
    cout << "\n=== 10. WHEN_ANY (FIRST COMPLETED) ===" << endl;

    vector<future<string>> tasks;

    tasks.push_back(async(launch::async, []() {
        this_thread::sleep_for(1200ms);
        return string("Task 1 (slow)");
    }));

    tasks.push_back(async(launch::async, []() {
        this_thread::sleep_for(700ms);
        return string("Task 2 (medium)");
    }));

    tasks.push_back(async(launch::async, []() {
        this_thread::sleep_for(300ms);
        return string("Task 3 (fast)");
    }));

    cout << "Main: Waiting for first task to complete..." << endl;

    // Simple when_any using wait_for
    while (true) {
        for (size_t i = 0; i < tasks.size(); i++) {
            if (tasks[i].wait_for(chrono::milliseconds(10)) == future_status::ready) {
                cout << "Task " << i << " completed first!" << endl;
                cout << "Result: " << tasks[i].get() << endl;

                // Get remaining results
                for (size_t j = 0; j < tasks.size(); j++) {
                    if (j != i) {
                        cout << "Task " << j << ": " << tasks[j].get() << endl;
                    }
                }
                return;
            }
        }
        this_thread::sleep_for(100ms);
    }
```

```
}

// ==================================================================
// 11. THREAD POOL WITH FUTURES
// ==================================================================
class ThreadPool {
private:
    vector<thread> workers;
    queue<function<void()>> tasks;
    mutex queue_mutex;
    condition_variable condition;
    atomic<bool> stop;

public:
    ThreadPool(size_t num_threads) : stop(false) {
        for (size_t i = 0; i < num_threads; ++i) {
            workers.emplace_back([this] {
                while (true) {
                    function<void()> task;
                    {
                        unique_lock<mutex> lock(queue_mutex);
                        condition.wait(lock, [this] {
                            return stop || !tasks.empty();
                        });

                        if (stop && tasks.empty()) return;

                        task = move(tasks.front());
                        tasks.pop();
                    }
                    task();
                }
            });
        }
    }

    template<class F, class... Args>
    auto enqueue(F&& f, Args&&... args) -> future<invoke_result_t<F, Args...>>
        {
        using return_type = invoke_result_t<F, Args...>;

        auto task = make_shared<packaged_task<return_type()>>(
            bind(forward<F>(f), forward<Args>(args)...)
        );

        future<return_type> result = task->get_future();
        {
            lock_guard<mutex> lock(queue_mutex);
            if (stop) {
                throw runtime_error("enqueue on stopped ThreadPool");
            }
            tasks.emplace([task]() { (*task)(); });
        }
        condition.notify_one();
```

```
        return result;
    }

    ~ThreadPool() {
        {
            lock_guard<mutex> lock(queue_mutex);
            stop = true;
        }
        condition.notify_all();
        for (thread& worker : workers) {
            if (worker.joinable()) worker.join();
        }
    }
};

void example_thread_pool() {
    cout << "\n=== 11. THREAD POOL WITH FUTURES ===" << endl;

    ThreadPool pool(4);
    vector<future<int>> results;

    cout << "Submitting 8 tasks to thread pool..." << endl;

    for (int i = 0; i < 8; i++) {
        results.emplace_back(pool.enqueue([i]() -> int {
            cout << "  Task " << i << " executing" << endl;
            this_thread::sleep_for(chrono::milliseconds(100 * (i % 3 + 1)));
            return i * i;
        }));
    }

    cout << "Collecting results..." << endl;
    for (size_t i = 0; i < results.size(); i++) {
        cout << "Result " << i << ": " << results[i].get() << endl;
    }

    cout << "All tasks completed!" << endl;
}

// ==================================================================
// 12. FUTURE WITH CANCELLATION
// ==================================================================
class CancellableFuture {
private:
    promise<int> promise_obj;
    future<int> future_obj;
    atomic<bool> cancelled;

public:
    CancellableFuture() : cancelled(false) {
        future_obj = promise_obj.get_future();
    }

    void cancel() {
```

```
        cancelled = true;
        promise_obj.set_exception(make_exception_ptr(runtime_error("Cancelled")));
    }

    bool is_cancelled() const { return cancelled; }

    future<int>& get_future() { return future_obj; }

    void execute_async(function<int()> task) {
        thread([this, task]() {
            if (cancelled) return;

            try {
                int result = task();
                if (!cancelled) {
                    promise_obj.set_value(result);
                }
            }
            catch (...) {
                if (!cancelled) {
                    promise_obj.set_exception(current_exception());
                }
            }
        }).detach();
    }
};

void example_cancellable_future() {
    cout << "\n=== 12. CANCELLABLE FUTURE ===" << endl;

    CancellableFuture cf;

    cf.execute_async([]() {
        cout << "  Long task starting (5 seconds)..." << endl;
        for (int i = 0; i < 5; i++) {
            this_thread::sleep_for(1s);
            cout << "  Task progress: " << (i + 1) << "/5" << endl;
        }
        return 42;
    });

    this_thread::sleep_for(2500ms);
    cout << "Main: Cancelling task after 2.5 seconds..." << endl;
    cf.cancel();

    try {
        cout << "Main: Waiting for result..." << endl;
        int result = cf.get_future().get();
        cout << "Main: Got result: " << result << endl;
    }
    catch (const exception& e) {
        cout << "Main: Caught: " << e.what() << endl;
    }
```

```

    this_thread::sleep_for(1s);
}

// ==================================================================
// 13. STD::VARIANT (C++17)
// ==================================================================
void example_variant() {
    cout << "\n=== 13. STD::VARIANT (C++17) ===" << endl;

    variant<int, string, double> value;

    value = 42;
    cout << "Holds int: " << get<int>(value) << endl;

    value = "Hello World";
    cout << "Holds string: " << get<string>(value) << endl;

    value = 3.14159;
    cout << "Holds double: " << get<double>(value) << endl;

    // Visit pattern
    auto visitor = [](auto&& arg) {
        using T = decay_t<decltype(arg)>;
        if constexpr (is_same_v<T, int>) {
            cout << "Visited int: " << arg << endl;
        }
        else if constexpr (is_same_v<T, string>) {
            cout << "Visited string: " << arg << endl;
        }
        else if constexpr (is_same_v<T, double>) {
            cout << "Visited double: " << arg << endl;
        }
    };

    visit(visitor, value);

    value = "Test";
    if (holds_alternative<string>(value)) {
        cout << "Value holds a string" << endl;
    }
}

// ==================================================================
// 14. STD::OPTIONAL (C++17)
// ==================================================================
optional<int> divide(int a, int b) {
    if (b == 0) {
        return nullopt;
    }
    return a / b;
}

void example_optional() {
```

```
    cout << "\n=== 14. STD::OPTIONAL (C++17) ===" << endl;

    auto result1 = divide(10, 2);
    if (result1) {
        cout << "10 / 2 = " << *result1 << endl;
    }

    auto result2 = divide(10, 0);
    if (!result2) {
        cout << "10 / 0 = No result (division by zero)" << endl;
    }

    cout << "Result or default: " << result2.value_or(999) << endl;

    optional<string> name = "Alice";
    cout << "Name: " << name.value() << endl;

    optional<int> empty_opt;
    cout << "Empty optional has value: " << boolalpha << empty_opt.has_value()
        << endl;
}

// ====================================================================
// 15. STD::ANY (C++17)
// ====================================================================
void example_any() {
    cout << "\n=== 15. STD::ANY (C++17) ===" << endl;

    any value;

    value = 42;
    cout << "Any holds int: " << any_cast<int>(value) << endl;

    value = string("Hello");
    cout << "Any holds string: " << any_cast<string>(value) << endl;

    value = 3.14;
    cout << "Any holds double: " << any_cast<double>(value) << endl;

    if (value.type() == typeid(double)) {
        cout << "Value is a double" << endl;
    }

    try {
        cout << any_cast<int>(value) << endl;
    }
    catch (const bad_any_cast& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
}

// ====================================================================
// 16. FILESYSTEM (C++17)
// ====================================================================
```

```
void example_filesystem() {
    cout << "\n=== 16. FILESYSTEM (C++17) ===" << endl;

    namespace fs = std::filesystem;

    cout << "Current path: " << fs::current_path() << endl;

    fs::create_directory("test_dir");
    cout << "Created directory: test_dir" << endl;

    {
        ofstream file("test_dir/test_file.txt");
        file << "Hello Filesystem!" << endl;
    }

    if (fs::exists("test_dir/test_file.txt")) {
        cout << "File exists" << endl;
        cout << "File size: " << fs::file_size("test_dir/test_file.txt") << "
            bytes" << endl;
    }

    cout << "\nDirectory contents:" << endl;
    for (const auto& entry : fs::directory_iterator("test_dir")) {
        cout << "  " << entry.path().filename()
             << " (" << fs::file_size(entry) << " bytes)" << endl;
    }

    fs::remove_all("test_dir");
    cout << "Cleaned up test directory" << endl;
}

// ==================================================================
// 17. STRUCTURED BINDINGS (C++17)
// ==================================================================
void example_structured_bindings() {
    cout << "\n=== 17. STRUCTURED BINDINGS (C++17) ===" << endl;

    // With tuple
    tuple<int, string, double> tup = {42, "Alice", 3.14};
    auto [id, name, score] = tup;
    cout << "Tuple unpacked: " << id << ", " << name << ", " << score << endl;

    // With array
    array<int, 3> arr = {1, 2, 3};
    auto [a, b, c] = arr;
    cout << "Array unpacked: " << a << ", " << b << ", " << c << endl;

    // With struct
    struct Person {
        string name;
        int age;
        string city;
    };

```

```
    Person p = {"Bob", 30, "New York"};
    auto [person_name, age, city] = p;
    cout << "Person: " << person_name << ", " << age << ", " << city << endl;

    // With map
    map<string, int> scores = {{"Alice", 95}, {"Bob", 87}, {"Charlie", 92}};
    for (const auto& [student, score] : scores) {
        cout << student << ": " << score << endl;
    }
}

// ==================================================================
// 18. CONSTEXPR IF (C++17)
// ==================================================================
template<typename T>
auto process_value(T value) {
    if constexpr (is_integral_v<T>) {
        cout << "Processing integer: " << value * 2 << endl;
        return value * 2;
    }
    else if constexpr (is_floating_point_v<T>) {
        cout << "Processing float: " << value * 3.14 << endl;
        return value * 3.14;
    }
    else if constexpr (is_same_v<T, string>) {
        cout << "Processing string: " << value + " processed" << endl;
        return value + " processed";
    }
    else {
        cout << "Unknown type" << endl;
        return value;
    }
}

void example_constexpr_if() {
    cout << "\n=== 18. CONSTEXPR IF (C++17) ===" << endl;

    process_value(42);
    process_value(3.14);
    process_value(string("Hello"));
}

// ==================================================================
// 19. COMPREHENSIVE EXAMPLE: ASYNC FILE PROCESSOR
// ==================================================================
void example_comprehensive_async_processor() {
    cout << "\n=== 19. COMPREHENSIVE EXAMPLE: ASYNC FILE PROCESSOR ===" <<
        endl;

    class AsyncFileProcessor {
    private:
        ThreadPool pool{4};

    public:
```

```
        future<vector<string>> process_files(const vector<string>& filenames) {
            vector<future<string>> futures;

            for (const auto& filename : filenames) {
                futures.push_back(pool.enqueue([filename]() -> string {
                    this_thread::sleep_for(chrono::milliseconds(100 + rand() % 400));

                    if (filename.find("error") != string::npos) {
                        throw runtime_error("Error processing " + filename);
                    }

                    return "Processed: " + filename + " (size: "
                           + to_string(rand() % 1000) + " bytes)";
                }));
            }

            return async(launch::async, [futures = move(futures)]() mutable {
                vector<string> results;
                for (auto& f : futures) {
                    try {
                        results.push_back(f.get());
                    }
                    catch (const exception& e) {
                        results.push_back("ERROR: " + string(e.what()));
                    }
                }
                return results;
            });
        }
    };

    AsyncFileProcessor processor;

    vector<string> files = {
        "document1.txt",
        "image1.png",
        "error_file.txt",
        "data.csv",
        "config.json"
    };

    cout << "Processing " << files.size() << " files asynchronously..." << endl;

    auto start = high_resolution_clock::now();
    auto result_future = processor.process_files(files);

    cout << "Main thread doing other work..." << endl;
    this_thread::sleep_for(200ms);

    auto results = result_future.get();
    auto end = high_resolution_clock::now();
```

```

    cout << "\nProcessing complete in "
         << duration_cast<milliseconds>(end - start).count() << "ms" << endl;
    cout << "\nResults:" << endl;
    for (const auto& result : results) {
        cout << "  " << result << endl;
    }
}

// ==================================================================
// 20. ADDITIONAL EXAMPLE: FUTURE WITH MULTIPLE CONTINUATIONS
// ==================================================================
void example_future_chaining() {
    cout << "\n=== 20. FUTURE CHAINING ===" << endl;

    auto future1 = async(launch::async, []() {
        cout << "  Step 1: Starting with 5" << endl;
        this_thread::sleep_for(200ms);
        return 5 + 10;  // 15
    });

    auto future2 = async(launch::async, [f1 = move(future1)]() mutable {
        int val = f1.get();
        cout << "  Step 2: Processing " << val << endl;
        this_thread::sleep_for(200ms);
        return val * 2;  // 30
    });

    auto future3 = async(launch::async, [f2 = move(future2)]() mutable {
        int val = f2.get();
        cout << "  Step 3: Finalizing " << val << endl;
        this_thread::sleep_for(200ms);
        return val - 5;  // 25
    });

    int result = future3.get();
    cout << "Final result: " << result << endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    cout << "
        ==================================================================="
        << endl;
    cout << "COMPLETE C++17 FEATURES EXAMPLES" << endl;
    cout << "
        ==================================================================="
        << endl;

    srand(time(nullptr));

    try {
```

```
        example_basic_future_promise();
        example_future_exception();
        example_async();
        example_shared_future();
        example_future_timeout();
        example_packaged_task();
        example_parallel_computation();
        example_future_continuations();
        example_when_all();
        example_when_any();
        example_thread_pool();
        example_cancellable_future();
        example_variant();
        example_optional();
        example_any();
        example_filesystem();
        example_structured_bindings();
        example_constexpr_if();
        example_comprehensive_async_processor();
        example_future_chaining();

        cout << "\n
            ==================================================================
            " << endl;
        cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!" << endl;
        cout << "
            ==================================================================
            " << endl;
    }
    catch (const exception& e) {
        cerr << "Error: " << e.what() << endl;
        return 1;
    }

    return 0;
}
```

14 Source Code: Cpp17Examples.cpp

File: `src/Cpp17Examples.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <tuple>
#include <optional>
#include <variant>
#include <any>
#include <string_view>
#include <filesystem>
#include <algorithm>
#include <numeric>
#include <memory>
#include <functional>

// ==================================================================
// C++17 COMPREHENSIVE EXAMPLES
// ==================================================================

// ==================================================================
// 1. TEMPLATE ARGUMENT DEDUCTION FOR CLASS TEMPLATES
// ==================================================================
void example_template_argument_deduction() {
    std::cout << "\n=== 1. TEMPLATE ARGUMENT DEDUCTION ===" << std::endl;

    std::pair p1{1, 2.0};           // std::pair<int, double>
    std::tuple t1{1, "hello", 3.14}; // std::tuple<int, const char*, double>
    std::vector v1{1, 2, 3, 4};     // std::vector<int>

    std::cout << "std::pair deduced: (" << p1.first << ", " << p1.second << ")
        " << std::endl;
    std::cout << "std::vector deduced with " << v1.size() << " elements" <<
        std::endl;
}

// ==================================================================
// 2. STRUCTURED BINDINGS
// ==================================================================
void example_structured_bindings() {
    std::cout << "\n=== 2. STRUCTURED BINDINGS ===" << std::endl;

    std::pair<int, std::string> pair{1, "hello"};
    auto [num, str] = pair;
    std::cout << "Pair: " << num << ", " << str << std::endl;

    std::tuple<int, double, std::string> tup{42, 3.14, "world"};
    auto [i, d, s] = tup;
    std::cout << "Tuple: " << i << ", " << d << ", " << s << std::endl;

```

```
    std::map<int, std::string> m{{1, "one"}, {2, "two"}};
    for (const auto& [key, value] : m) {
        std::cout << "Map: " << key << " -> " << value << std::endl;
    }
}

// ==================================================================
// 3. FOLDING EXPRESSIONS
// ==================================================================
template <typename... Args>
auto sum_fold(Args... args) {
    return (... + args);  // Unary left fold
}

template <typename... Args>
bool all_true(Args... args) {
    return (... && args);  // Unary left fold
}

template <typename... Args>
void print_all(Args... args) {
    ((std::cout << args << " "), ...);  // Binary left fold
    std::cout << std::endl;
}

void example_folding_expressions() {
    std::cout << "\n=== 3. FOLDING EXPRESSIONS ===" << std::endl;

    std::cout << "sum_fold(1, 2, 3, 4, 5) = " << sum_fold(1, 2, 3, 4, 5) << std::endl;
    std::cout << "all_true(true, true, true) = " << std::boolalpha << all_true(true, true, true) << std::endl;
    std::cout << "all_true(true, false, true) = " << all_true(true, false, true) << std::endl;
    std::cout << "print_all: ";
    print_all(1, "hello", 3.14, "world");
}

// ==================================================================
// 4. CONSTEXPR IF
// ==================================================================
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>) {
        return *t;  // Dereference pointer
    } else {
        return t;   // Return value directly
    }
}

void example_constexpr_if() {
    std::cout << "\n=== 4. CONSTEXPR IF ===" << std::endl;

    int x = 42;
```

```
    int* ptr = &x;

    std::cout << "get_value(10) = " << get_value(10) << std::endl;
    std::cout << "get_value(ptr) = " << get_value(ptr) << std::endl;
}

// ==================================================================
// 5. CONSTEXPR LAMBDA
// ==================================================================
void example_constexpr_lambda() {
    std::cout << "\n=== 5. CONSTEXPR LAMBDA ===" << std::endl;

    constexpr auto square = [](int n) { return n * n; };
    constexpr int result = square(5);

    std::cout << "Compile-time square(5) = " << result << std::endl;
    static_assert(square(4) == 16, "Square function test");
}

// ==================================================================
// 6. INLINE VARIABLES
// ==================================================================
inline int global_counter = 0;

struct Config {
    inline static const std::string app_name = "MyApp";
    inline static const int version = 1;
};

void example_inline_variables() {
    std::cout << "\n=== 6. INLINE VARIABLES ===" << std::endl;

    global_counter++;
    std::cout << "Global counter: " << global_counter << std::endl;
    std::cout << "App name: " << Config::app_name << std::endl;
    std::cout << "Version: " << Config::version << std::endl;
}

// ==================================================================
// 7. NESTED NAMESPACES
// ==================================================================
namespace A::B::C {
    void nested_function() {
        std::cout << "Inside A::B::C namespace" << std::endl;
    }
}

void example_nested_namespaces() {
    std::cout << "\n=== 7. NESTED NAMESPACES ===" << std::endl;
    A::B::C::nested_function();
}

// ==================================================================
// 8. SELECTION STATEMENTS WITH INITIALIZER
```

```cpp
// ==================================================================
void example_selection_with_initializer() {
    std::cout << "\n=== 8. SELECTION WITH INITIALIZER ===" << std::endl;

    std::map<int, std::string> m{{1, "one"}, {2, "two"}};

    if (auto it = m.find(1); it != m.end()) {
        std::cout << "Found: " << it->second << std::endl;
    }

    switch (auto value = 42; value) {
        case 42:
            std::cout << "Value is 42" << std::endl;
            break;
        default:
            std::cout << "Value is not 42" << std::endl;
    }
}

// ==================================================================
// 9. [[FALLTHROUGH]], [[NODISCARD]], [[MAYBE_UNUSED]] ATTRIBUTES
// ==================================================================
[[nodiscard]] int important_function() {
    return 42;
}

void example_attributes([[maybe_unused]] int unused_param) {
    std::cout << "\n=== 9. ATTRIBUTES ===" << std::endl;

    int result = important_function();  // Must use result or compiler warns
    std::cout << "Result: " << result << std::endl;

    int value = 2;
    switch (value) {
        case 1:
            std::cout << "Case 1" << std::endl;
            [[fallthrough]];
        case 2:
            std::cout << "Case 2 (maybe from fallthrough)" << std::endl;
            break;
    }
}

// ==================================================================
// 10. STD::OPTIONAL
// ==================================================================
std::optional<int> try_parse_int(const std::string& str) {
    try {
        return std::stoi(str);
    } catch (...) {
        return std::nullopt;
    }
}

```

```
void example_optional() {
    std::cout << "\n=== 10. STD::OPTIONAL ===" << std::endl;

    std::optional<int> opt1 = 42;
    std::optional<int> opt2 = std::nullopt;

    if (opt1) {
        std::cout << "opt1 has value: " << *opt1 << std::endl;
    }

    std::cout << "opt2 value or default: " << opt2.value_or(-1) << std::endl;

    auto result1 = try_parse_int("123");
    auto result2 = try_parse_int("abc");

    std::cout << "Parse '123': " << (result1 ? std::to_string(*result1) : "
        failed") << std::endl;
    std::cout << "Parse 'abc': " << (result2 ? std::to_string(*result2) : "
        failed") << std::endl;
}

// ==================================================================
// 11. STD::VARIANT
// ==================================================================
void example_variant() {
    std::cout << "\n=== 11. STD::VARIANT ===" << std::endl;

    std::variant<int, double, std::string> var;

    var = 42;
    std::cout << "Variant holds int: " << std::get<int>(var) << std::endl;

    var = 3.14;
    std::cout << "Variant holds double: " << std::get<double>(var) << std::
        endl;

    var = "hello";
    std::cout << "Variant holds string: " << std::get<std::string>(var) << std
        ::endl;

    std::visit([](auto&& arg) {
        std::cout << "Visiting variant: " << arg << std::endl;
    }, var);
}

// ==================================================================
// 12. STD::ANY
// ==================================================================
void example_any() {
    std::cout << "\n=== 12. STD::ANY ===" << std::endl;

    std::any a = 42;
    std::cout << "Any holds int: " << std::any_cast<int>(a) << std::endl;

```

```
    a = std::string("hello");
    std::cout << "Any holds string: " << std::any_cast<std::string>(a) << std
        ::endl;

    if (a.type() == typeid(std::string)) {
        std::cout << "Confirmed: any contains string" << std::endl;
    }
}

// ==================================================================
// 13. STD::STRING_VIEW
// ==================================================================
void print_string_view(std::string_view sv) {
    std::cout << "String view: " << sv << " (length: " << sv.length() << ")"
        << std::endl;
}

void example_string_view() {
    std::cout << "\n=== 13. STD::STRING_VIEW ===" << std::endl;

    std::string str = "Hello, World!";
    std::string_view sv = str;

    print_string_view(sv);
    print_string_view("Literal string");
    print_string_view(sv.substr(0, 5));
}

// ==================================================================
// 14. STD::FILESYSTEM
// ==================================================================
void example_filesystem() {
    std::cout << "\n=== 14. STD::FILESYSTEM ===" << std::endl;

    namespace fs = std::filesystem;

    fs::path p = fs::current_path();
    std::cout << "Current path: " << p << std::endl;

    fs::path example_path = "example.txt";
    std::cout << "Filename: " << example_path.filename() << std::endl;
    std::cout << "Extension: " << example_path.extension() << std::endl;
}

// ==================================================================
// 15. STD::CLAMP
// ==================================================================
void example_clamp() {
    std::cout << "\n=== 15. STD::CLAMP ===" << std::endl;

    int value = 50;
    int clamped = std::clamp(value, 10, 40);
    std::cout << "clamp(50, 10, 40) = " << clamped << std::endl;

```

```
    int value2 = 5;
    int clamped2 = std::clamp(value2, 10, 40);
    std::cout << "clamp(5, 10, 40) = " << clamped2 << std::endl;
}

// ==================================================================
// 16. GCD AND LCM
// ==================================================================
void example_gcd_lcm() {
    std::cout << "\n=== 16. GCD AND LCM ===" << std::endl;

    int a = 12, b = 18;
    std::cout << "gcd(" << a << ", " << b << ") = " << std::gcd(a, b) << std::
        endl;
    std::cout << "lcm(" << a << ", " << b << ") = " << std::lcm(a, b) << std::
        endl;
}

// ==================================================================
// 17. LAMBDA CAPTURE THIS BY VALUE
// ==================================================================
struct MyObject {
    int value = 123;

    auto get_value_copy() {
        return [*this] { return value; };
    }

    auto get_value_ref() {
        return [this] { return value; };
    }
};

void example_lambda_capture_this() {
    std::cout << "\n=== 17. LAMBDA CAPTURE THIS BY VALUE ===" << std::endl;

    MyObject obj;
    auto copy_lambda = obj.get_value_copy();
    auto ref_lambda = obj.get_value_ref();

    obj.value = 456;

    std::cout << "Copy lambda returns: " << copy_lambda() << " (original value
        )" << std::endl;
    std::cout << "Ref lambda returns: " << ref_lambda() << " (modified value)"
         << std::endl;
}

// ==================================================================
// 18. STD::INVOKE
// ==================================================================
int add_func(int a, int b) {
    return a + b;
}
```

```

struct Adder {
    int operator()(int a, int b) const {
        return a + b;
    }
};

void example_invoke() {
    std::cout << "\n=== 18. STD::INVOKE ===" << std::endl;

    std::cout << "invoke(add_func, 3, 4) = " << std::invoke(add_func, 3, 4) << std::endl;

    Adder adder;
    std::cout << "invoke(Adder, 5, 6) = " << std::invoke(adder, 5, 6) << std::endl;

    auto lambda = [](int a, int b) { return a * b; };
    std::cout << "invoke(lambda, 7, 8) = " << std::invoke(lambda, 7, 8) << std::endl;
}

// ==================================================================
// 19. STD::APPLY
// ==================================================================
int multiply(int a, int b, int c) {
    return a * b * c;
}

void example_apply() {
    std::cout << "\n=== 19. STD::APPLY ===" << std::endl;

    std::tuple<int, int, int> args{2, 3, 4};
    int result = std::apply(multiply, args);

    std::cout << "apply(multiply, {2, 3, 4}) = " << result << std::endl;
}

// ==================================================================
// 20. STD::MAKE_FROM_TUPLE
// ==================================================================
struct Point {
    int x, y, z;
    Point(int x, int y, int z) : x(x), y(y), z(z) {}
};

void example_make_from_tuple() {
    std::cout << "\n=== 20. STD::MAKE_FROM_TUPLE ===" << std::endl;

    std::tuple<int, int, int> coords{10, 20, 30};
    auto point = std::make_from_tuple<Point>(coords);

    std::cout << "Point created from tuple: (" << point.x << ", " << point.y << ", " << point.z << ")" << std::endl;
```

```
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n================================================" << std::endl;
    std::cout << "    C++17 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
    std::cout << "================================================" << std::endl;

    example_template_argument_deduction();
    example_structured_bindings();
    example_folding_expressions();
    example_constexpr_if();
    example_constexpr_lambda();
    example_inline_variables();
    example_nested_namespaces();
    example_selection_with_initializer();
    example_attributes(42);
    example_optional();
    example_variant();
    example_any();
    example_string_view();
    example_filesystem();
    example_clamp();
    example_gcd_lcm();
    example_lambda_capture_this();
    example_invoke();
    example_apply();
    example_make_from_tuple();

    std::cout << "\n================================================" << std::endl;
    std::cout << "    ALL C++17 EXAMPLES COMPLETED" << std::endl;
    std::cout << "================================================\n" << std::endl;

    return 0;
}
```

15 Source Code: Cpp20Examples.cpp

File: `src/Cpp20Examples.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <span>
#include <ranges>
#include <algorithm>
#include <concepts>
#include <compare>
#include <numbers>
#include <bit>
#include <bitset>
#include <array>
#include <set>
#include <numeric>

// ==================================================================
// C++20 COMPREHENSIVE EXAMPLES
// ==================================================================

// ==================================================================
// 1. CONCEPTS
// ==================================================================
template <typename T>
concept Integral = std::is_integral_v<T>;

template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed_v<T>;

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

template <SignedIntegral T>
T negate(T value) {
    return -value;
}

void example_concepts() {
    std::cout << "\n=== 1. CONCEPTS ===" << std::endl;

    std::cout << "add(10, 20) = " << add(10, 20) << std::endl;
    std::cout << "add(5U, 3U) = " << add(5U, 3U) << std::endl;
    std::cout << "negate(42) = " << negate(42) << std::endl;

    // add(3.14, 2.5);  // Error: double doesn't satisfy Integral
}

// ==================================================================
```

```
// 2. THREE-WAY COMPARISON (SPACESHIP OPERATOR)
// ====================================================================
struct Point {
    int x, y;
    auto operator<=>(const Point&) const = default;
};

void example_three_way_comparison() {
    std::cout << "\n=== 2. THREE-WAY COMPARISON ===" << std::endl;

    Point p1{1, 2}, p2{1, 3}, p3{1, 2};

    std::cout << "p1 == p3: " << std::boolalpha << (p1 == p3) << std::endl;
    std::cout << "p1 != p2: " << (p1 != p2) << std::endl;
    std::cout << "p1 < p2: " << (p1 < p2) << std::endl;

    int a = 10, b = 20;
    auto result = a <=> b;
    if (result < 0) {
        std::cout << "a < b" << std::endl;
    }
}

// ====================================================================
// 3. DESIGNATED INITIALIZERS
// ====================================================================
struct Config {
    int width;
    int height;
    std::string title;
};

void example_designated_initializers() {
    std::cout << "\n=== 3. DESIGNATED INITIALIZERS ===" << std::endl;

    Config cfg{
        .width = 1920,
        .height = 1080,
        .title = "My Window"
    };

    std::cout << "Config: " << cfg.width << "x" << cfg.height
              << " - " << cfg.title << std::endl;
}

// ====================================================================
// 4. TEMPLATE SYNTAX FOR LAMBDAS
// ====================================================================
void example_template_lambdas() {
    std::cout << "\n=== 4. TEMPLATE SYNTAX FOR LAMBDAS ===" << std::endl;

    auto generic_add = []<typename T>(T a, T b) {
        return a + b;
    };
```

```

    std::cout << "generic_add(10, 20) = " << generic_add(10, 20) << std::endl;
    std::cout << "generic_add(1.5, 2.5) = " << generic_add(1.5, 2.5) << std::
        endl;

    auto print_type = []<typename T>(T value) {
        if constexpr (std::is_integral_v<T>) {
            std::cout << "Integer: " << value << std::endl;
        } else {
            std::cout << "Non-integer: " << value << std::endl;
        }
    };

    print_type(42);
    print_type(3.14);
}

// ==================================================================
// 5. RANGE-BASED FOR LOOP WITH INITIALIZER
// ==================================================================
void example_range_for_initializer() {
    std::cout << "\n=== 5. RANGE-BASED FOR WITH INITIALIZER ===" << std::endl;

    for (std::vector<int> vec = {1, 2, 3, 4, 5}; auto& elem : vec) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 6. [[LIKELY]] AND [[UNLIKELY]] ATTRIBUTES
// ==================================================================
int predict_branch(int value) {
    if (value > 0) [[likely]] {
        return value * 2;
    } else [[unlikely]] {
        return value * 10;
    }
}

void example_likely_unlikely() {
    std::cout << "\n=== 6. [[LIKELY]] AND [[UNLIKELY]] ===" << std::endl;

    std::cout << "predict_branch(5) = " << predict_branch(5) << std::endl;
    std::cout << "predict_branch(-2) = " << predict_branch(-2) << std::endl;
}

// ==================================================================
// 7. CONSTEXPR VIRTUAL FUNCTIONS
// ==================================================================
struct Base {
    virtual constexpr int get_value() const {
        return 10;
    }
```

```
};

struct Derived : Base {
    constexpr int get_value() const override {
        return 20;
    }
};

void example_constexpr_virtual() {
    std::cout << "\n=== 7. CONSTEXPR VIRTUAL FUNCTIONS ===" << std::endl;

    constexpr Derived d;
    constexpr int value = d.get_value();

    std::cout << "Constexpr virtual function result: " << value << std::endl;
}

// ==================================================================
// 8. EXPLICIT(BOOL)
// ==================================================================
struct MyInt {
    int value;
    explicit(sizeof(int) > 4) MyInt(int v) : value(v) {}
};

void example_explicit_bool() {
    std::cout << "\n=== 8. EXPLICIT(BOOL) ===" << std::endl;

    MyInt mi{42};  // Always requires explicit construction based on condition
    std::cout << "MyInt created with value: " << mi.value << std::endl;
}

// ==================================================================
// 9. IMMEDIATE FUNCTIONS (CONSTEVAL)
// ==================================================================
consteval int square(int n) {
    return n * n;
}

void example_immediate_functions() {
    std::cout << "\n=== 9. IMMEDIATE FUNCTIONS ===" << std::endl;

    constexpr int result = square(5);  // Must be evaluated at compile time
    std::cout << "Consteval square(5) = " << result << std::endl;
}

// ==================================================================
// 10. USING ENUM
// ==================================================================
enum class Color { Red, Green, Blue };

void example_using_enum() {
    std::cout << "\n=== 10. USING ENUM ===" << std::endl;

```

```
    using enum Color;
    Color c = Red;  // No need for Color::Red

    switch (c) {
        using enum Color;  // Can use in switch scope
        case Red:
            std::cout << "Color is Red" << std::endl;
            break;
        case Green:
            std::cout << "Color is Green" << std::endl;
            break;
        case Blue:
            std::cout << "Color is Blue" << std::endl;
            break;
    }
}

// ====================================================================
// 11. CHAR8_T
// ====================================================================
void example_char8_t() {
    std::cout << "\n=== 11. CHAR8_T ===" << std::endl;

    char8_t utf8_char = u8'A';
    const char8_t* utf8_str = u8"Hello UTF-8";

    std::cout << "UTF-8 character type created" << std::endl;
    std::cout << "UTF-8 string type created" << std::endl;
}

// ====================================================================
// 12. CONSTINIT
// ====================================================================
constinit int global_value = 42;

void example_constinit() {
    std::cout << "\n=== 12. CONSTINIT ===" << std::endl;

    std::cout << "Constinit global value: " << global_value << std::endl;
    global_value = 100;  // Can be modified at runtime
    std::cout << "Modified value: " << global_value << std::endl;
}

// ====================================================================
// 13. STD::SPAN
// ====================================================================
void process_span(std::span<int> s) {
    for (auto& elem : s) {
        elem *= 2;
    }
}

void example_span() {
    std::cout << "\n=== 13. STD::SPAN ===" << std::endl;
```

```

    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::span<int> s{vec};

    std::cout << "Before: ";
    for (auto v : vec) std::cout << v << " ";
    std::cout << std::endl;

    process_span(s);

    std::cout << "After: ";
    for (auto v : vec) std::cout << v << " ";
    std::cout << std::endl;
}

// ==================================================================
// 14. RANGES
// ==================================================================
void example_ranges() {
    std::cout << "\n=== 14. RANGES ===" << std::endl;

    std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto even = vec | std::views::filter([](int n) { return n % 2 == 0; });
    auto squared = even | std::views::transform([](int n) { return n * n; });

    std::cout << "Even numbers squared: ";
    for (auto v : squared) {
        std::cout << v << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 15. BIT OPERATIONS
// ==================================================================
void example_bit_operations() {
    std::cout << "\n=== 15. BIT OPERATIONS ===" << std::endl;

    unsigned int value = 0b10110100;

    std::cout << "popcount(0b10110100) = " << std::popcount(value) << std::
        endl;
    std::cout << "has_single_bit(8) = " << std::boolalpha << std::
        has_single_bit(8u) << std::endl;
    std::cout << "bit_width(7) = " << std::bit_width(7u) << std::endl;
    std::cout << "rotl(0b10110100, 2) = " << std::rotl(value, 2) << std::endl;
}

// ==================================================================
// 16. MATH CONSTANTS
// ==================================================================
void example_math_constants() {
    std::cout << "\n=== 16. MATH CONSTANTS ===" << std::endl;
```

```

    std::cout << "pi = " << std::numbers::pi << std::endl;
    std::cout << "e = " << std::numbers::e << std::endl;
    std::cout << "sqrt2 = " << std::numbers::sqrt2 << std::endl;
    std::cout << "ln2 = " << std::numbers::ln2 << std::endl;
}

// ==================================================================
// 17. STD::IS_CONSTANT_EVALUATED
// ==================================================================
constexpr int compute_value() {
    if (std::is_constant_evaluated()) {
        return 42;  // Compile-time
    } else {
        return 100; // Runtime
    }
}

void example_is_constant_evaluated() {
    std::cout << "\n=== 17. STD::IS_CONSTANT_EVALUATED ===" << std::endl;

    constexpr int compile_time = compute_value();
    int runtime = compute_value();

    std::cout << "Compile-time: " << compile_time << std::endl;
    std::cout << "Runtime: " << runtime << std::endl;
}

// ==================================================================
// 18. STARTS_WITH / ENDS_WITH
// ==================================================================
void example_starts_ends_with() {
    std::cout << "\n=== 18. STARTS_WITH / ENDS_WITH ===" << std::endl;

    std::string str = "Hello, World!";

    std::cout << "starts_with('Hello'): " << std::boolalpha
              << str.starts_with("Hello") << std::endl;
    std::cout << "ends_with('World!'): " << str.ends_with("World!") << std::
        endl;
}

// ==================================================================
// 19. ASSOCIATIVE CONTAINER .CONTAINS()
// ==================================================================
void example_contains() {
    std::cout << "\n=== 19. ASSOCIATIVE CONTAINER .CONTAINS() ===" << std::
        endl;

    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::set<int> s{vec.begin(), vec.end()};

    std::cout << "Set contains 3: " << std::boolalpha << s.contains(3) << std
        ::endl;
```

```
    std::cout << "Set contains 10: " << s.contains(10) << std::endl;
}

// ==================================================================
// 20. STD::MIDPOINT
// ==================================================================
void example_midpoint() {
    std::cout << "\n=== 20. STD::MIDPOINT ===" << std::endl;

    std::cout << "midpoint(10, 20) = " << std::midpoint(10, 20) << std::endl;
    std::cout << "midpoint(1.0, 5.0) = " << std::midpoint(1.0, 5.0) << std::
        endl;
}

// ==================================================================
// 21. STD::TO_ARRAY
// ==================================================================
void example_to_array() {
    std::cout << "\n=== 21. STD::TO_ARRAY ===" << std::endl;

    auto arr = std::to_array({1, 2, 3, 4, 5});

    std::cout << "Array from initializer list: ";
    for (auto v : arr) {
        std::cout << v << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 22. LAMBDA CAPTURE OF PARAMETER PACK
// ==================================================================
template <typename... Args>
auto make_lambda(Args... args) {
    return [...args = std::move(args)] {
        ((std::cout << args << " "), ...);
        std::cout << std::endl;
    };
}

void example_lambda_capture_pack() {
    std::cout << "\n=== 22. LAMBDA CAPTURE OF PARAMETER PACK ===" << std::endl
        ;

    auto lambda = make_lambda(1, 2, 3, "hello", 4.5);
    std::cout << "Lambda with captured pack: ";
    lambda();
}

// ==================================================================
// 23. CLASS TYPES IN NON-TYPE TEMPLATE PARAMETERS
// ==================================================================
struct CompileTimeValue {
    int value;
```

```
    constexpr CompileTimeValue(int v) : value(v) {}
};

template <CompileTimeValue val>
void print_compile_time() {
    std::cout << "Compile-time value: " << val.value << std::endl;
}

void example_class_non_type_template() {
    std::cout << "\n=== 23. CLASS TYPES IN NON-TYPE TEMPLATE ===" << std::endl
        ;

    print_compile_time<CompileTimeValue{42}>();
}

// ================================================================
// MAIN FUNCTION
// ================================================================
int main() {
    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    C++20 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
    std::cout << "================================================" << std::
        endl;

    example_concepts();
    example_three_way_comparison();
    example_designated_initializers();
    example_template_lambdas();
    example_range_for_initializer();
    example_likely_unlikely();
    example_constexpr_virtual();
    example_explicit_bool();
    example_immediate_functions();
    example_using_enum();
    example_char8_t();
    example_constinit();
    example_span();
    example_ranges();
    example_bit_operations();
    example_math_constants();
    example_is_constant_evaluated();
    example_starts_ends_with();
    example_contains();
    example_midpoint();
    example_to_array();
    example_lambda_capture_pack();
    example_class_non_type_template();

    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    ALL C++20 EXAMPLES COMPLETED" << std::endl;
    std::cout << "================================================\n" << std::
        endl;
```

```

    return 0;
}
```

16 Source Code: Cpp23Examples.cpp

File: `src/Cpp23Examples.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <optional>
#include <utility>  // for std::to_underlying, std::unreachable
#include <cmath>
#include <cstdint>
#include <array>
#include <bit>

// ====================================================================
// C++23 FEATURES FOR EMBEDDED SYSTEMS, FIRMWARE, AND RTOS
// ====================================================================
// Focused on features useful for:
// - Firmware development
// - Real-Time Operating Systems (RTOS)
// - Embedded Linux platforms
// - Hardware interfacing
// - Resource-constrained environments
// ====================================================================

// ====================================================================
// 1. std::expected<T, E> - ERROR HANDLING WITHOUT EXCEPTIONS
// ====================================================================
// CRITICAL for embedded: No exception overhead!
// Perfect for firmware where exceptions are disabled

#if __cplusplus >= 202302L && __has_include(<expected>)
#include <expected>

enum class SensorError {
    NotConnected,
    ReadTimeout,
    InvalidData,
    CalibrationFailed
};

const char* to_string(SensorError error) {
    switch (error) {
        case SensorError::NotConnected: return "Sensor not connected";
        case SensorError::ReadTimeout: return "Read timeout";
        case SensorError::InvalidData: return "Invalid data";
        case SensorError::CalibrationFailed: return "Calibration failed";
    }
    return "Unknown error";
}

// Return either temperature value or error - no exceptions!
std::expected<float, SensorError> read_temperature_sensor() {
```

```
    // Simulate sensor read
    static int call_count = 0;
    call_count++;

    if (call_count == 1) {
        return std::unexpected(SensorError::NotConnected);
    }
    if (call_count == 2) {
        return std::unexpected(SensorError::ReadTimeout);
    }

    return 23.5f;  // Success
}

// Chaining operations with monadic interface
std::expected<float, SensorError> read_and_convert_to_fahrenheit() {
    return read_temperature_sensor()
        .and_then([](float celsius) -> std::expected<float, SensorError> {
            if (celsius < -273.15f) {
                return std::unexpected(SensorError::InvalidData);
            }
            return celsius * 9.0f / 5.0f + 32.0f;
        });
}

void demonstrate_expected() {
    std::cout << "\n=== 1. std::expected - EXCEPTION-FREE ERROR HANDLING ==="
       << std::endl;
    std::cout << "Perfect for firmware with -fno-exceptions" << std::endl;

    // First call - error
    auto result1 = read_temperature_sensor();
    if (result1) {
        std::cout << "\n Temperature: " << *result1 << "°C" << std::endl;
    } else {
        std::cout << "\n Error: " << to_string(result1.error()) << std::endl;
    }

    // Second call - error
    auto result2 = read_temperature_sensor();
    if (result2) {
        std::cout << "  Temperature: " << *result2 << "°C" << std::endl;
    } else {
        std::cout << "  Error: " << to_string(result2.error()) << std::endl;
    }

    // Third call - success
    auto result3 = read_temperature_sensor();
    if (result3) {
        std::cout << "  Temperature: " << *result3 << "°C" << std::endl;
    } else {
        std::cout << "  Error: " << to_string(result3.error()) << std::endl;
    }

```

```
    std::cout << "\n Benefits for Embedded:" << std::endl;
    std::cout << " •   Zero exception overhead" << std::endl;
    std::cout << " •   Works with -fno-exceptions" << std::endl;
    std::cout << " •   Explicit error handling" << std::endl;
    std::cout << " •   Deterministic performance" << std::endl;
}

#else
void demonstrate_expected() {
    std::cout << "\n=== 1. std::expected ===" << std::endl;
    std::cout << "   std::expected requires C++23 compiler support" << std::
        endl;
    std::cout << "Alternative: Use std::variant<T, Error> in C++17" << std::
        endl;
}
#endif

// ==================================================================
// 2. std::byteswap - ENDIANNESS CONVERSION FOR HARDWARE
// ==================================================================
// Essential for: Network protocols, file formats, hardware registers

#if __cplusplus >= 202302L && __has_include(<bit>)

void demonstrate_byteswap() {
    std::cout << "\n=== 2. std::byteswap - HARDWARE BYTE ORDER ===" << std::
        endl;
    std::cout << "Critical for network protocols and hardware registers" <<
        std::endl;

    // Reading from network (big-endian) to host (possibly little-endian)
    uint32_t network_value = 0x12345678;
    std::cout << "\nNetwork value (big-endian): 0x" << std::hex <<
        network_value << std::dec << std::endl;

    uint32_t host_value = std::byteswap(network_value);
    std::cout << "After byteswap: 0x" << std::hex << host_value << std::dec <<
         std::endl;

    // 16-bit register value swap
    uint16_t reg16 = 0xABCD;
    std::cout << "\n16-bit register: 0x" << std::hex << reg16 << std::dec <<
        std::endl;
    uint16_t swapped16 = std::byteswap(reg16);
    std::cout << "Swapped: 0x" << std::hex << swapped16 << std::dec << std::
        endl;

    // 64-bit timestamp swap
    uint64_t timestamp = 0x0123456789ABCDEF;
    std::cout << "\n64-bit timestamp: 0x" << std::hex << timestamp << std::dec
         << std::endl;
    uint64_t swapped64 = std::byteswap(timestamp);
    std::cout << "Swapped: 0x" << std::hex << swapped64 << std::dec << std::
        endl;
```

```

    std::cout << "\n Embedded Use Cases:" << std::endl;
    std::cout << " •   Network protocol implementation (TCP/IP)" << std::endl;
    std::cout << " •   File format parsing (headers, metadata)" << std::endl;
    std::cout << " •   Cross-platform binary data exchange" << std::endl;
    std::cout << " •   Hardware register manipulation" << std::endl;
}

#else
void demonstrate_byteswap() {
    std::cout << "\n=== 2. std::byteswap ===" << std::endl;
    std::cout << "   std::byteswap requires C++23" << std::endl;
    std::cout << "Alternative: Use __builtin_bswap32() or manual bit shifts"
        << std::endl;
}
#endif

// ==================================================================
// 3. std::to_underlying - ENUM TO INTEGER (HARDWARE REGISTERS)
// ==================================================================

enum class GPIOPin : uint8_t {
    PIN_0 = 0,
    PIN_1 = 1,
    PIN_2 = 2,
    PIN_3 = 3,
    PIN_LED = 13,
    PIN_BUTTON = 7
};

enum class MemoryRegister : uint32_t {
    STATUS_REG = 0x40000000,
    CONTROL_REG = 0x40000004,
    DATA_REG = 0x40000008,
    CONFIG_REG = 0x4000000C
};

void demonstrate_to_underlying() {
    std::cout << "\n=== 3. std::to_underlying - HARDWARE REGISTER ACCESS ==="
        << std::endl;
    std::cout << "Convert enum class to underlying integer type" << std::endl;

    // GPIO pin access
    GPIOPin led_pin = GPIOPin::PIN_LED;
    auto pin_number = std::to_underlying(led_pin);
    std::cout << "\nLED Pin number: " << static_cast<int>(pin_number) << std::
        endl;

    // Memory-mapped register access
    MemoryRegister status = MemoryRegister::STATUS_REG;
    // Example: volatile uint32_t* status_ptr = reinterpret_cast<volatile
        uint32_t*>(std::to_underlying(status));
    std::cout << "Status register address: 0x" << std::hex << std::
        to_underlying(status) << std::dec << std::endl;
```

```

    // Array indexing with enum
    std::array<const char*, 4> reg_names = {"STATUS", "CONTROL", "DATA", "CONFIG"};
    MemoryRegister reg = MemoryRegister::CONTROL_REG;
    size_t index = (std::to_underlying(reg) - std::to_underlying(MemoryRegister::STATUS_REG)) / 4;
    std::cout << "Register name: " << reg_names[index] << std::endl;

    std::cout << "\n Before C++23:" << std::endl;
    std::cout << "   static_cast<std::underlying_type_t<GPIOPin>>(led_pin)" << std::endl;
    std::cout << "\n C++23:" << std::endl;
    std::cout << "   std::to_underlying(led_pin)  // Much cleaner!" << std::endl;
}

// ==================================================================
// 4. constexpr FOR <cmath> - COMPILE-TIME CALCULATIONS
// ==================================================================

// Lookup table generation at compile time
constexpr std::array<float, 256> generate_sine_lookup_table() {
    std::array<float, 256> table{};
    for (size_t i = 0; i < 256; ++i) {
        // C++23: std::sin is now constexpr!
        table[i] = std::sin(2.0 * 3.14159265358979323846 * i / 256.0);
    }
    return table;
}

// PWM duty cycle calculation at compile time
constexpr uint16_t calculate_pwm_value(float duty_percent) {
    return static_cast<uint16_t>(duty_percent * 65535.0f / 100.0f);
}

void demonstrate_constexpr_cmath() {
    std::cout << "\n=== 4. constexpr <cmath> - COMPILE-TIME CALCULATIONS ===" << std::endl;
    std::cout << "Generate lookup tables at compile time" << std::endl;

    // Sine lookup table generated at compile time
    constexpr auto sine_table = generate_sine_lookup_table();
    std::cout << "\nSine lookup table (256 entries) generated at compile time" << std::endl;
    std::cout << "Sample values:" << std::endl;
    std::cout << "  sin(0°)     " << sine_table[0] << std::endl;
    std::cout << "  sin(90°)    " << sine_table[64] << std::endl;
    std::cout << "  sin(180°)   " << sine_table[128] << std::endl;
    std::cout << "  sin(270°)   " << sine_table[192] << std::endl;

    // PWM values calculated at compile time
    constexpr uint16_t pwm_25 = calculate_pwm_value(25.0f);
    constexpr uint16_t pwm_50 = calculate_pwm_value(50.0f);
```

```
    constexpr uint16_t pwm_75 = calculate_pwm_value(75.0f);

    std::cout << "\nPWM duty cycle values (calculated at compile time):" << std::endl;
    std::cout << "  25%: " << pwm_25 << std::endl;
    std::cout << "  50%: " << pwm_50 << std::endl;
    std::cout << "  75%: " << pwm_75 << std::endl;

    std::cout << "\n Embedded Benefits:" << std::endl;
    std::cout << " •   Zero runtime cost for lookup tables" << std::endl;
    std::cout << " •   Stored in ROM/Flash, not RAM" << std::endl;
    std::cout << " •   Fast table lookups vs real-time calculations" << std::endl;
    std::cout << " •   Deterministic execution time" << std::endl;
}

// ==================================================================
// 5. if consteval - COMPILE-TIME VS RUNTIME PATHS
// ==================================================================

constexpr uint32_t crc32_compute(const char* data, size_t len) {
    if consteval {
        // Compile-time: Simple algorithm
        uint32_t crc = 0xFFFFFFFF;
        for (size_t i = 0; i < len; ++i) {
            crc ^= static_cast<uint32_t>(data[i]);
            for (int j = 0; j < 8; ++j) {
                crc = (crc >> 1) ^ (0xEDB88320 & -(crc & 1));
            }
        }
        return ~crc;
    } else {
        // Runtime: Could use hardware CRC or optimized table lookup
        std::cout << "[Using runtime CRC algorithm]" << std::endl;
        uint32_t crc = 0xFFFFFFFF;
        for (size_t i = 0; i < len; ++i) {
            crc ^= static_cast<uint32_t>(data[i]);
            for (int j = 0; j < 8; ++j) {
                crc = (crc >> 1) ^ (0xEDB88320 & -(crc & 1));
            }
        }
        return ~crc;
    }
}

void demonstrate_if_consteval() {
    std::cout << "\n=== 5. if consteval - COMPILE-TIME VS RUNTIME ===" << std::endl;
    std::cout << "Different code paths for compile-time and runtime" << std::endl;

    // Compile-time CRC calculation
    constexpr const char* firmware_id = "FIRMWARE_V1.2.3";
    constexpr uint32_t compile_time_crc = crc32_compute(firmware_id, 15);
```

```
    std::cout << "\nFirmware ID: " << firmware_id << std::endl;
    std::cout << "CRC32 (compile-time): 0x" << std::hex << compile_time_crc <<
        std::dec << std::endl;

    // Runtime CRC calculation
    const char* runtime_data = "RUNTIME_DATA_PACKET";
    uint32_t runtime_crc = crc32_compute(runtime_data, 19);
    std::cout << "\nRuntime data: " << runtime_data << std::endl;
    std::cout << "CRC32 (runtime): 0x" << std::hex << runtime_crc << std::dec
       << std::endl;

    std::cout << "\n Embedded Applications:" << std::endl;
    std::cout << " •   Compile-time: Simple, portable algorithm" << std::endl;
    std::cout << " •   Runtime: Hardware CRC accelerator" << std::endl;
    std::cout << " •   Firmware verification checksums" << std::endl;
    std::cout << " •   Protocol integrity checks" << std::endl;
}

// ================================================================
// 6. std::unreachable - OPTIMIZATION HINTS FOR COMPILER
// ================================================================

enum class DeviceState : uint8_t {
    IDLE = 0,
    RUNNING = 1,
    STOPPED = 2,
    ERROR = 3
};

[[nodiscard]] constexpr const char* device_state_name(DeviceState state) {
    switch (state) {
        case DeviceState::IDLE:    return "IDLE";
        case DeviceState::RUNNING: return "RUNNING";
        case DeviceState::STOPPED: return "STOPPED";
        case DeviceState::ERROR:   return "ERROR";
    }
    std::unreachable();  // Tell compiler all cases covered
}

uint32_t process_command(uint8_t cmd) {
    // Note: cmd >= 0 always true for unsigned type, kept for documentation
    if (cmd <= 3) {
        return cmd * 100;  // Valid command processing
    }

    // Invalid command - should never happen with validated input
    std::unreachable();  // Optimization hint
}

void demonstrate_unreachable() {
    std::cout << "\n=== 6. std::unreachable - COMPILER OPTIMIZATION ===" <<
       std::endl;
    std::cout << "Tell compiler certain code paths are impossible" << std::
       endl;
```

```

    std::cout << "\nDevice states:" << std::endl;
    std::cout << "  " << device_state_name(DeviceState::IDLE) << std::endl;
    std::cout << "  " << device_state_name(DeviceState::RUNNING) << std::endl;
    std::cout << "  " << device_state_name(DeviceState::STOPPED) << std::endl;
    std::cout << "  " << device_state_name(DeviceState::ERROR) << std::endl;

    std::cout << "\nCommand processing:" << std::endl;
    std::cout << "  Command 0: " << process_command(0) << std::endl;
    std::cout << "  Command 2: " << process_command(2) << std::endl;

    std::cout << "\n Optimization Benefits:" << std::endl;
    std::cout << " •   Eliminates dead code paths" << std::endl;
    std::cout << " •   Better branch prediction" << std::endl;
    std::cout << " •   Smaller code size" << std::endl;
    std::cout << " •   Faster execution" << std::endl;

    std::cout << "\n  WARNING: Reaching std::unreachable() is UB!" << std::
        endl;
}

// ==================================================================
// 7. SIZE_T LITERAL SUFFIX (uz/z) - TYPE-SAFE SIZES
// ==================================================================

void demonstrate_size_literals() {
    std::cout << "\n=== 7. SIZE_T LITERAL SUFFIX - TYPE SAFETY ===" << std::
        endl;
    std::cout << "Avoid unsigned vs signed comparison warnings" << std::endl;

    std::vector<uint32_t> buffer = {1, 2, 3, 4, 5};

    // Old way - warning: comparison between signed and unsigned
    // for (int i = 0; i < buffer.size(); ++i) { }  // Warning!

    // C++23 way - type-safe
    constexpr size_t BUFFER_SIZE = 256uz;  // uz suffix = size_t
    std::array<uint8_t, BUFFER_SIZE> uart_buffer{};

    std::cout << "\nBuffer allocations:" << std::endl;
    std::cout << "  UART buffer: " << BUFFER_SIZE << " bytes" << std::endl;
    std::cout << "  Vector size: " << buffer.size() << " elements" << std::
        endl;
    std::cout << "  Array capacity: " << uart_buffer.size() << " bytes" << std
        ::endl;

    // No warning with uz suffix
    if (buffer.size() > 3uz) {
        std::cout << "  Buffer has more than 3 elements" << std::endl;
    }

    std::cout << "\n Before C++23:" << std::endl;
    std::cout << "   if (buffer.size() > 3)     // May warn" << std::endl;
    std::cout << "   if (buffer.size() > 3u)    // Wrong type (unsigned)" <<
```

```
        std::endl;
    std::cout << "   if (buffer.size() > 3UL)   // Platform dependent" << std::endl;
    std::cout << "\n C++23:" << std::endl;
    std::cout << "   if (buffer.size() > 3uz)   // Correct type!" << std::endl;
}

// ==================================================================
// 8. MULTIDIMENSIONAL SUBSCRIPT OPERATOR
// ==================================================================

template<typename T, size_t Rows, size_t Cols>
class Matrix {
private:
    std::array<T, Rows * Cols> data;

public:
    // C++23: Multi-dimensional subscript operator
    // Note: Requires GCC 14+, Clang 17+, or MSVC 2022 17.8+
    #if defined(__cpp_multidimensional_subscript) && __cpp_multidimensional_subscript >= 202110L
    constexpr T& operator[](size_t row, size_t col) {
        return data[row * Cols + col];
    }

    constexpr const T& operator[](size_t row, size_t col) const {
        return data[row * Cols + col];
    }
    #endif

    // Fallback: Traditional at() function for older compilers
    constexpr T& at(size_t row, size_t col) {
        return data[row * Cols + col];
    }

    constexpr const T& at(size_t row, size_t col) const {
        return data[row * Cols + col];
    }

    constexpr void fill(T value) {
        data.fill(value);
    }
};

void demonstrate_multidim_subscript() {
    std::cout << "\n=== 8. MULTIDIMENSIONAL SUBSCRIPT OPERATOR ===" << std::endl;

    #if defined(__cpp_multidimensional_subscript) && __cpp_multidimensional_subscript >= 202110L
    std::cout << "  Multi-dimensional operator[] supported!" << std::endl;
    std::cout << "Direct matrix[row, col] syntax" << std::endl;

```

```
    Matrix<uint16_t, 3, 3> sensor_data;

    // C++23: Clean syntax for 2D array access
    sensor_data[0, 0] = 100;
    sensor_data[0, 1] = 150;
    sensor_data[0, 2] = 200;
    sensor_data[1, 0] = 250;
    sensor_data[1, 1] = 300;
    sensor_data[1, 2] = 350;
    sensor_data[2, 0] = 400;
    sensor_data[2, 1] = 450;
    sensor_data[2, 2] = 500;

    std::cout << "\n3x3 Sensor data matrix:" << std::endl;
    for (size_t row = 0; row < 3; ++row) {
        std::cout << "  ";
        for (size_t col = 0; col < 3; ++col) {
            std::cout << sensor_data[row, col] << "\t";
        }
        std::cout << std::endl;
    }

    std::cout << "\ n C++23 syntax:" << std::endl;
    std::cout << "   matrix[row, col]      // Direct syntax!" << std::endl;

    #else

    std::cout << "  Multi-dimensional operator[] requires GCC 14+, Clang 17+,
        or MSVC 17.8+" << std::endl;
    std::cout << "Using fallback at(row, col) method for demonstration" << std
        ::endl;

    Matrix<uint16_t, 3, 3> sensor_data;

    // Fallback: Use at() method
    sensor_data.at(0, 0) = 100;
    sensor_data.at(0, 1) = 150;
    sensor_data.at(0, 2) = 200;
    sensor_data.at(1, 0) = 250;
    sensor_data.at(1, 1) = 300;
    sensor_data.at(1, 2) = 350;
    sensor_data.at(2, 0) = 400;
    sensor_data.at(2, 1) = 450;
    sensor_data.at(2, 2) = 500;

    std::cout << "\n3x3 Sensor data matrix (using at() method):" << std::endl;
    for (size_t row = 0; row < 3; ++row) {
        std::cout << "  ";
        for (size_t col = 0; col < 3; ++col) {
            std::cout << sensor_data.at(row, col) << "\t";
        }
        std::cout << std::endl;
    }

```

```
    std::cout << "\n Before C++23:" << std::endl;
    std::cout << "   matrix[row][col]      // Traditional double subscript" << std::endl;
    std::cout << "   matrix.at(row, col)   // Member function" << std::endl;
    std::cout << "\n C++23 (when compiler supports):" << std::endl;
    std::cout << "   matrix[row, col]      // Direct multi-dimensional syntax!" << std::endl;

    #endif
}

// ==================================================================
// 9. PRACTICAL EMBEDDED EXAMPLE: FIRMWARE STATUS SYSTEM
// ==================================================================

#if __cplusplus >= 202302L && __has_include(<expected>)

enum class FirmwareError {
    ConfigInvalid,
    MemoryInsufficient,
    HardwareNotResponding,
    ChecksumMismatch
};

const char* firmware_error_name(FirmwareError err) {
    switch (err) {
        case FirmwareError::ConfigInvalid: return "Configuration invalid";
        case FirmwareError::MemoryInsufficient: return "Insufficient memory";
        case FirmwareError::HardwareNotResponding: return "Hardware not responding";
        case FirmwareError::ChecksumMismatch: return "Checksum mismatch";
    }
    std::unreachable();
}

class FirmwareStatus {
public:
    std::expected<bool, FirmwareError> initialize() {
        std::cout << "  [1] Checking configuration..." << std::endl;
        // Simulate config check

        std::cout << "  [2] Allocating memory..." << std::endl;
        // Simulate memory allocation

        std::cout << "  [3] Initializing hardware..." << std::endl;
        // Simulate hardware init

        std::cout << "  [4] Verifying checksums..." << std::endl;
        constexpr uint32_t expected_crc = 0x12345678;
        constexpr uint32_t actual_crc = 0x12345678;

        if (expected_crc != actual_crc) {
            return std::unexpected(FirmwareError::ChecksumMismatch);
        }
```

```

        return true;
    }

    std::expected<uint32_t, FirmwareError> get_hardware_id() {
        // Simulate hardware ID read
        return 0xABCD1234;
    }
};

void demonstrate_practical_firmware() {
    std::cout << "\n=== 9. PRACTICAL EXAMPLE: FIRMWARE STATUS ===" << std::
        endl;
    std::cout << "C++23 features in embedded firmware" << std::endl;

    FirmwareStatus firmware;

    std::cout << "\nInitializing firmware..." << std::endl;
    auto init_result = firmware.initialize();

    if (init_result) {
        std::cout << "  Firmware initialized successfully" << std::endl;

        auto hw_id = firmware.get_hardware_id();
        if (hw_id) {
            std::cout << "  Hardware ID: 0x" << std::hex << *hw_id << std::dec
                << std::endl;
        }
    } else {
        std::cout << "  Initialization failed: "
                  << firmware_error_name(init_result.error()) << std::endl;
    }

    std::cout << "\n This firmware code:" << std::endl;
    std::cout << " •   Uses std::expected (no exceptions)" << std::endl;
    std::cout << " •   Works with -fno-exceptions" << std::endl;
    std::cout << " •   Deterministic error handling" << std::endl;
    std::cout << " •   Suitable for safety-critical systems" << std::endl;
}

#else
void demonstrate_practical_firmware() {
    std::cout << "\n=== 9. PRACTICAL EXAMPLE: FIRMWARE STATUS ===" << std::
        endl;
    std::cout << "   Requires C++23 std::expected" << std::endl;
}
#endif

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n
```

```
        ================================================================" << std::endl;
    std::cout << "  C++23 FEATURES FOR EMBEDDED SYSTEMS & FIRMWARE" << std::endl;
    std::cout << "
        ================================================================" << std::endl;
    std::cout << "Focused on: Firmware, RTOS, Embedded Linux" << std::endl;

    demonstrate_expected();
    demonstrate_byteswap();
    demonstrate_to_underlying();
    demonstrate_constexpr_cmath();
    demonstrate_if_consteval();
    demonstrate_unreachable();
    demonstrate_size_literals();
    demonstrate_multidim_subscript();
    demonstrate_practical_firmware();

    std::cout << "\n
        ================================================================" << std::endl;
    std::cout << "  C++23 FEATURES SUMMARY FOR EMBEDDED" << std::endl;
    std::cout << "
        ================================================================" << std::endl;

    std::cout << "\n MUST-HAVE FOR FIRMWARE:" << std::endl;
    std::cout << "\n1  std::expected<T, E>" << std::endl;
    std::cout << " •   No exception overhead" << std::endl;
    std::cout << " •   Works with -fno-exceptions" << std::endl;
    std::cout << " •   Explicit error handling" << std::endl;
    std::cout << " •   Perfect for RTOS and firmware" << std::endl;
    std::cout << "   Use case: Sensor reads, hardware init, protocol parsing" << std::endl;

    std::cout << "\n2  std::byteswap" << std::endl;
    std::cout << " •   Endianness conversion" << std::endl;
    std::cout << " •   Network protocol implementation" << std::endl;
    std::cout << " •   Binary file format parsing" << std::endl;
    std::cout << " •   Hardware register access" << std::endl;
    std::cout << "   Use case: TCP/IP stack, file systems, cross-platform data" << std::endl;

    std::cout << "\n3  std::to_underlying" << std::endl;
    std::cout << " •   Cleaner enum to integer conversion" << std::endl;
    std::cout << " •   Memory-mapped register access" << std::endl;
    std::cout << " •   GPIO pin manipulation" << std::endl;
    std::cout << " •   Hardware abstraction layers" << std::endl;
    std::cout << "   Use case: Register maps, pin assignments, state machines" << std::endl;

    std::cout << "\n4  constexpr <cmath>" << std::endl;
    std::cout << " •   Compile-time lookup table generation" << std::endl;
```

```
    std::cout << " •   Stored in ROM, not RAM" << std::endl;
    std::cout << " •   Zero runtime cost" << std::endl;
    std::cout << " •   Fast deterministic execution" << std::endl;
    std::cout << "   Use case: Sine/cosine tables, PWM values, calibration
        data" << std::endl;

    std::cout << "\n5  if consteval" << std::endl;
    std::cout << " •   Different paths for compile-time vs runtime" << std::
        endl;
    std::cout << " •   Compile-time: simple algorithm" << std::endl;
    std::cout << " •   Runtime: hardware accelerator" << std::endl;
    std::cout << "   Use case: CRC calculation, crypto, compression" << std::
        endl;

    std::cout << "\n6  std::unreachable" << std::endl;
    std::cout << " •   Compiler optimization hints" << std::endl;
    std::cout << " •   Smaller code size" << std::endl;
    std::cout << " •   Better branch prediction" << std::endl;
    std::cout << "   Use case: State machines, validated inputs, switch
        statements" << std::endl;

    std::cout << "\n7  Size literal suffix (uz)" << std::endl;
    std::cout << " •   Type-safe size comparisons" << std::endl;
    std::cout << " •   No signed/unsigned warnings" << std::endl;
    std::cout << " •   Cleaner buffer management code" << std::endl;
    std::cout << "   Use case: Buffer sizes, array indexing, memory allocation
        " << std::endl;

    std::cout << "\n8  Multidimensional operator[]" << std::endl;
    std::cout << " •   Direct matrix[row, col] syntax" << std::endl;
    std::cout << " •   Cleaner multi-dimensional arrays" << std::endl;
    std::cout << " •   Natural matrix/tensor access" << std::endl;
    std::cout << "   Use case: Image buffers, sensor arrays, DSP data" << std
        ::endl;

    std::cout << "\n COMPILER SUPPORT (January 2026):" << std::endl;
    std::cout << "\n  GCC 12+:" << std::endl;
    std::cout << " •    std::expected  (use GCC 13+)" << std::endl;
    std::cout << " •    std::byteswap  " << std::endl;
    std::cout << " •    std::to_underlying  " << std::endl;
    std::cout << " •    if consteval  " << std::endl;

    std::cout << "\n  Clang 16+:" << std::endl;
    std::cout << " •    std::expected  " << std::endl;
    std::cout << " •    Most C++23 features  " << std::endl;

    std::cout << "\n  MSVC 2022 (17.6+):" << std::endl;
    std::cout << " •    Excellent C++23 support  " << std::endl;

    std::cout << "\n  ARM Compiler 6:" << std::endl;
    std::cout << " •    Based on Clang, good C++23 support  " << std::endl;

    std::cout << "\n RECOMMENDED EMBEDDED COMPILER FLAGS:" << std::endl;
    std::cout << "  -std=c++23              # Enable C++23" << std::endl;
```

```
    std::cout << "  -fno-exceptions         # Disable exceptions" << std::endl;
    std::cout << "  -fno-rtti               # Disable RTTI" << std::endl;
    std::cout << "  -Os or -O2              # Optimize for size/speed" << std::endl;
    std::cout << "  -flto                   # Link-time optimization" << std::endl;
    std::cout << "  -ffunction-sections     # Dead code elimination" << std::endl;

    std::cout << "\n MIGRATION STRATEGY:" << std::endl;
    std::cout << "  1. Start with std::to_underlying (easy win)" << std::endl;
    std::cout << "  2. Add constexpr lookup tables (ROM savings)" << std::endl;
    std::cout << "  3. Replace error codes with std::expected (safer)" << std::endl;
    std::cout << "  4. Use std::byteswap for protocols (cleaner)" << std::endl;
    std::cout << "  5. Add std::unreachable for optimizations" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

17 Source Code: CppWrappingCLibrary.cpp

File: `src/CppWrappingCLibrary.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// WRAPPING C LIBRARIES IN MODERN C++
// ==================================================================
// This example demonstrates best practices for wrapping C libraries
// (TCP/UDP sockets) in modern C++ with:
// - RAII for automatic resource management
// - noexcept for exception safety
// - [[nodiscard]] for preventing ignored errors
// - extern "C" for C library linkage
// - Strong types and error handling
//
// TOPICS COVERED:
// 1. C library integration (POSIX sockets)
// 2. RAII wrappers for C resources
// 3. noexcept specifications (safe and unsafe uses)
// 4. [[nodiscard]] attribute
// 5. std::optional and std::expected for error handling
// 6. Strong typing over raw C types
// ==================================================================

#include <iostream>
#include <string>
#include <string_view>
#include <optional>
#include <system_error>
#include <cstring>
#include <vector>
#include <memory>
#include <chrono>

// Platform-specific socket headers
#ifdef _WIN32
    #include <winsock2.h>
    #include <ws2tcpip.h>
    #pragma comment(lib, "ws2_32.lib")
    using socket_t = SOCKET;
    constexpr socket_t INVALID_SOCKET_VALUE = INVALID_SOCKET;
    #define CLOSE_SOCKET closesocket
#else
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <unistd.h>
    #include <fcntl.h>
    #include <netdb.h>
    using socket_t = int;
    constexpr socket_t INVALID_SOCKET_VALUE = -1;
    #define CLOSE_SOCKET close
#endif
```

```

// ==================================================================
// SECTION 1: C LIBRARY FUNCTIONS (extern "C" linkage)
// ==================================================================
// These are C functions - they use C calling conventions and linkage
// extern "C" prevents C++ name mangling

extern "C" {
    // Example: Custom C utility function we might have in a C library
    int c_validate_port(int port) {
        return (port > 0 && port <= 65535) ? 1 : 0;
    }

    // Note: socket(), bind(), listen(), etc. are already extern "C"
    // from system headers, so we don't redeclare them
}

// ==================================================================
// SECTION 2: ERROR HANDLING TYPES
// ==================================================================

// Modern C++ error type using std::optional
enum class SocketError {
    Success,
    InvalidSocket,
    BindFailed,
    ListenFailed,
    ConnectFailed,
    SendFailed,
    ReceiveFailed,
    AcceptFailed,
    SocketOptionFailed,
    AddressResolutionFailed,
    TimeoutExpired,
    WouldBlock
};

// Convert error code to string
[[nodiscard]] constexpr std::string_view error_to_string(SocketError error)
    noexcept {
    switch (error) {
        case SocketError::Success: return "Success";
        case SocketError::InvalidSocket: return "Invalid socket";
        case SocketError::BindFailed: return "Bind failed";
        case SocketError::ListenFailed: return "Listen failed";
        case SocketError::ConnectFailed: return "Connect failed";
        case SocketError::SendFailed: return "Send failed";
        case SocketError::ReceiveFailed: return "Receive failed";
        case SocketError::AcceptFailed: return "Accept failed";
        case SocketError::SocketOptionFailed: return "Socket option failed";
        case SocketError::AddressResolutionFailed: return "Address resolution
            failed";
        case SocketError::TimeoutExpired: return "Timeout expired";
        case SocketError::WouldBlock: return "Operation would block";
```

```
        default: return "Unknown error";
    }
}

// Result type for operations that can fail
template<typename T>
using Result = std::optional<T>;

// ==================================================================
// SECTION 3: RAII SOCKET WRAPPER
// ==================================================================
// Modern C++ wrapper that automatically manages socket lifecycle

class Socket {
private:
    socket_t fd_;
    bool is_valid_;

public:
    // Constructor - creates socket
    // noexcept(false) because it may throw
    Socket(int domain = AF_INET, int type = SOCK_STREAM, int protocol = 0)
        : fd_(::socket(domain, type, protocol)),
          is_valid_(fd_ != INVALID_SOCKET_VALUE) {

        if (!is_valid_) {
            std::cerr << "Socket creation failed" << std::endl;
        } else {
            std::cout << "  Socket created (fd=" << fd_ << ")" << std::endl;
        }
    }

    // Tag type for from_fd constructor
    struct from_fd_t {};
    static constexpr from_fd_t from_fd{};

    // Construct from existing socket (for accept())
    Socket(from_fd_t, socket_t fd) noexcept
        : fd_(fd), is_valid_(fd != INVALID_SOCKET_VALUE) {}

    // Destructor - RAII automatically closes socket
    // noexcept because destructors should not throw
    ~Socket() noexcept {
        if (is_valid_ && fd_ != INVALID_SOCKET_VALUE) {
            std::cout << "  Socket closing (fd=" << fd_ << ")" << std::endl;
            CLOSE_SOCKET(fd_);
        }
    }

    // Delete copy operations - sockets are unique resources
    Socket(const Socket&) = delete;
    Socket& operator=(const Socket&) = delete;

    // Move operations - transfer ownership
```

```
    Socket(Socket&& other) noexcept
        : fd_(other.fd_), is_valid_(other.is_valid_) {
        other.fd_ = INVALID_SOCKET_VALUE;
        other.is_valid_ = false;
    }

    Socket& operator=(Socket&& other) noexcept {
        if (this != &other) {
            if (is_valid_) {
                CLOSE_SOCKET(fd_);
            }
            fd_ = other.fd_;
            is_valid_ = other.is_valid_;
            other.fd_ = INVALID_SOCKET_VALUE;
            other.is_valid_ = false;
        }
        return *this;
    }

    // Check if socket is valid
    // [[nodiscard]] - result should not be ignored!
    [[nodiscard]] bool is_valid() const noexcept {
        return is_valid_;
    }

    // Get raw socket descriptor
    // [[nodiscard]] - caller needs this value!
    [[nodiscard]] socket_t get() const noexcept {
        return fd_;
    }

    // Bind to address
    // [[nodiscard]] - error must be checked!
    [[nodiscard]] SocketError bind(const std::string& address, uint16_t port)
        noexcept {
        if (!is_valid_) {
            return SocketError::InvalidSocket;
        }

        sockaddr_in addr{};
        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);

        if (address.empty() || address == "0.0.0.0") {
            addr.sin_addr.s_addr = INADDR_ANY;
        } else {
            addr.sin_addr.s_addr = inet_addr(address.c_str());
        }

        if (::bind(fd_, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) < 0)
            {
            return SocketError::BindFailed;
        }

```

```
        std::cout << "  Socket bound to " << address << ":" << port << std::endl;
        return SocketError::Success;
    }

    // Listen for connections
    // [[nodiscard]] - error must be checked!
    [[nodiscard]] SocketError listen(int backlog = 10) noexcept {
        if (!is_valid_) {
            return SocketError::InvalidSocket;
        }

        if (::listen(fd_, backlog) < 0) {
            return SocketError::ListenFailed;
        }

        std::cout << "  Socket listening (backlog=" << backlog << ")" << std::endl;
        return SocketError::Success;
    }

    // Accept connection - returns new Socket
    // [[nodiscard]] - must handle the result!
    [[nodiscard]] Result<Socket> accept() noexcept {
        if (!is_valid_) {
            return std::nullopt;
        }

        sockaddr_in client_addr{};
        socklen_t addr_len = sizeof(client_addr);

        socket_t client_fd = ::accept(fd_, reinterpret_cast<sockaddr*>(&client_addr), &addr_len);

        if (client_fd == INVALID_SOCKET_VALUE) {
            return std::nullopt;
        }

        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, INET_ADDRSTRLEN);
        std::cout << "  Connection accepted from " << client_ip
                  << ":" << ntohs(client_addr.sin_port) << std::endl;

        return Socket(Socket::from_fd, client_fd);
    }

    // Connect to remote address
    // [[nodiscard]] - error must be checked!
    [[nodiscard]] SocketError connect(const std::string& address, uint16_t port) noexcept {
        if (!is_valid_) {
            return SocketError::InvalidSocket;
        }

```

```
        sockaddr_in addr{};
        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);
        addr.sin_addr.s_addr = inet_addr(address.c_str());

        if (::connect(fd_, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) < 0) {
            return SocketError::ConnectFailed;
        }

        std::cout << "  Connected to " << address << ":" << port << std::endl;
        return SocketError::Success;
    }

    // Send data
    // [[nodiscard]] - must check if send succeeded!
    [[nodiscard]] Result<size_t> send(const std::string& data) noexcept {
        if (!is_valid_) {
            return std::nullopt;
        }

        ssize_t sent = ::send(fd_, data.c_str(), data.length(), 0);

        if (sent < 0) {
            return std::nullopt;
        }

        std::cout << "  Sent " << sent << " bytes" << std::endl;
        return static_cast<size_t>(sent);
    }

    // Receive data
    // [[nodiscard]] - must handle received data!
    [[nodiscard]] Result<std::string> receive(size_t max_length = 4096) noexcept {
        if (!is_valid_) {
            return std::nullopt;
        }

        std::vector<char> buffer(max_length);
        ssize_t received = ::recv(fd_, buffer.data(), buffer.size(), 0);

        if (received <= 0) {
            return std::nullopt;
        }

        std::cout << "  Received " << received << " bytes" << std::endl;
        return std::string(buffer.data(), received);
    }

    // Set socket option
    // [[nodiscard]] - error must be checked!
    [[nodiscard]] SocketError set_reuse_address(bool enable) noexcept {
        if (!is_valid_) {
```

```
            return SocketError::InvalidSocket;
        }

        int opt = enable ? 1 : 0;
        if (setsockopt(fd_, SOL_SOCKET, SO_REUSEADDR,
                       reinterpret_cast<const char*>(&opt), sizeof(opt)) < 0) {
            return SocketError::SocketOptionFailed;
        }

        std::cout << "  SO_REUSEADDR set to " << (enable ? "true" : "false")
            << std::endl;
        return SocketError::Success;
    }
};

// ==================================================================
// SECTION 4: HIGH-LEVEL TCP SERVER CLASS
// ==================================================================

class TcpServer {
private:
    Socket listen_socket_;
    uint16_t port_;
    bool is_running_;

public:
    // Constructor
    // noexcept(false) - may throw if initialization fails critically
    explicit TcpServer(uint16_t port)
        : listen_socket_(AF_INET, SOCK_STREAM, 0),
          port_(port),
          is_running_(false) {

        if (!listen_socket_.is_valid()) {
            throw std::runtime_error("Failed to create listen socket");
        }
    }

    // Start server
    // [[nodiscard]] - must check if server started successfully!
    [[nodiscard]] SocketError start() noexcept {
        // Enable address reuse
        auto err = listen_socket_.set_reuse_address(true);
        if (err != SocketError::Success) {
            return err;
        }

        // Bind to port
        err = listen_socket_.bind("0.0.0.0", port_);
        if (err != SocketError::Success) {
            return err;
        }

        // Start listening
```

```
        err = listen_socket_.listen();
        if (err != SocketError::Success) {
            return err;
        }

        is_running_ = true;
        std::cout << "  TCP Server started on port " << port_ << std::endl;
        return SocketError::Success;
    }

    // Accept one client connection
    // [[nodiscard]] - must handle the client socket!
    [[nodiscard]] Result<Socket> accept_client() noexcept {
        if (!is_running_) {
            return std::nullopt;
        }
        return listen_socket_.accept();
    }

    // Check if server is running
    [[nodiscard]] bool is_running() const noexcept {
        return is_running_;
    }

    // Stop server
    void stop() noexcept {
        is_running_ = false;
        std::cout << "  TCP Server stopped" << std::endl;
    }
};

// ====================================================================
// SECTION 5: HIGH-LEVEL TCP CLIENT CLASS
// ====================================================================

class TcpClient {
private:
    Socket socket_;
    bool is_connected_;

public:
    // Constructor
    TcpClient()
        : socket_(AF_INET, SOCK_STREAM, 0),
          is_connected_(false) {}

    // Connect to server
    // [[nodiscard]] - must check connection status!
    [[nodiscard]] SocketError connect(const std::string& host, uint16_t port)
        noexcept {
        if (!socket_.is_valid()) {
            return SocketError::InvalidSocket;
        }

```

```
        auto err = socket_.connect(host, port);
        if (err == SocketError::Success) {
            is_connected_ = true;
        }
        return err;
    }

    // Send message
    // [[nodiscard]] - must check if send succeeded!
    [[nodiscard]] Result<size_t> send(const std::string& message) noexcept {
        if (!is_connected_) {
            return std::nullopt;
        }
        return socket_.send(message);
    }

    // Receive message
    // [[nodiscard]] - must handle received data!
    [[nodiscard]] Result<std::string> receive() noexcept {
        if (!is_connected_) {
            return std::nullopt;
        }
        return socket_.receive();
    }

    // Check if connected
    [[nodiscard]] bool is_connected() const noexcept {
        return is_connected_;
    }
};

// ==================================================================
// SECTION 6: DEMONSTRATION OF noexcept USAGE
// ==================================================================

void demonstrate_noexcept() {
    std::cout << "\n=== NOEXCEPT USAGE ===" << std::endl;

    std::cout << "\n SAFE noexcept usage:" << std::endl;
    std::cout << " •   Getters that don't throw: is_valid() noexcept" << std::endl;
    std::cout << " •   Destructors: ~Socket() noexcept" << std::endl;
    std::cout << " •   Move operations: Socket(Socket&&) noexcept" << std::endl;
    std::cout << " •   Simple checks: error_to_string() noexcept" << std::endl;

    std::cout << "\n  Operations marked noexcept (but handle errors internally):" << std::endl;
    std::cout << " •   bind() noexcept - returns error code instead of throwing" << std::endl;
    std::cout << " •   send() noexcept - returns std::optional for errors" << std::endl;
    std::cout << " •   receive() noexcept - returns std::optional" << std::
```

```
        endl;

    std::cout << "\n NOT noexcept (may throw):" << std::endl;
    std::cout << " •   Constructor if critical initialization fails" << std::endl;
    std::cout << " •   std::string operations (may throw std::bad_alloc)" << std::endl;

    std::cout << "\n KEY PRINCIPLE:" << std::endl;
    std::cout << "   Mark noexcept when you guarantee no exceptions will escape" << std::endl;
    std::cout << "   Use error codes/std::optional for expected failures" << std::endl;
}

// ==================================================================
// SECTION 7: DEMONSTRATION OF [[nodiscard]]
// ==================================================================

void demonstrate_nodiscard() {
    std::cout << "\n=== [[nodiscard]] ATTRIBUTE ===" << std::endl;

    std::cout << "\n [[nodiscard]] forces checking return values:" << std::endl;

    Socket sock(AF_INET, SOCK_STREAM, 0);

    // GOOD: Checking the return value
    if (auto err = sock.bind("0.0.0.0", 8080); err != SocketError::Success) {
        std::cout << "   Bind error checked: " << error_to_string(err) << std::endl;
    }

    // BAD: Would produce compiler warning (if we uncommented)
    // sock.bind("0.0.0.0", 8080);  // Warning: ignoring return value!

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << "     Prevents forgetting to check errors" << std::endl;
    std::cout << "     Compiler warns about ignored return values" << std::endl;
    std::cout << "     Makes APIs safer to use" << std::endl;
    std::cout << "     Self-documenting - shows value is important" << std::endl;
}

// ==================================================================
// SECTION 8: RAII BENEFITS DEMONSTRATION
// ==================================================================

void demonstrate_raii() {
    std::cout << "\n=== RAII AUTOMATIC RESOURCE MANAGEMENT ===" << std::endl;

    std::cout << "\nCreating socket in scope:" << std::endl;
    {
```

```
        Socket sock(AF_INET, SOCK_STREAM, 0);
        auto err = sock.bind("0.0.0.0", 9999);
        if (err == SocketError::Success) {
            std::cout << "   Socket is active (fd=" << sock.get() << ")" << std::endl;
        }
        // No need to manually close - destructor handles it!
    }
    std::cout << "    Socket automatically closed when out of scope!" << std::endl;

    std::cout << "\n RAII GUARANTEES:" << std::endl;
    std::cout << "    Resource acquired in constructor" << std::endl;
    std::cout << "    Resource released in destructor" << std::endl;
    std::cout << "    Exception-safe - cleanup always happens" << std::endl;
    std::cout << "    No manual cleanup needed" << std::endl;
    std::cout << "    No resource leaks possible" << std::endl;
}

// ================================================================
// SECTION 9: EXTERN "C" USAGE
// ================================================================

void demonstrate_extern_c() {
    std::cout << "\n=== EXTERN \"C\" LINKAGE ===" << std::endl;

    std::cout << "\n Purpose: Interface with C libraries" << std::endl;
    std::cout << " •   Prevents C++ name mangling" << std::endl;
    std::cout << " •   Allows C code to call C++ functions" << std::endl;
    std::cout << " •   Enables linking with C libraries" << std::endl;

    // Call our C function
    int port = 8080;
    int is_valid = c_validate_port(port);
    std::cout << "\n Called C function: c_validate_port(" << port << ") = "
              << (is_valid ? "valid" : "invalid") << std::endl;

    std::cout << "\n USAGE PATTERNS:" << std::endl;
    std::cout << "   extern \"C\" { ... }    - Wrap C declarations" << std::endl;
    std::cout << "   #ifdef __cplusplus     - Conditional compilation" << std::endl;
    std::cout << "   System headers (socket.h) already use extern \"C\"" << std::endl;
}

// ================================================================
// SECTION 10: PRACTICAL EXAMPLES
// ================================================================

void example_echo_server() {
    std::cout << "\n=== EXAMPLE: SIMPLE ECHO SERVER ===" << std::endl;
    std::cout << "Simulating echo server workflow...\n" << std::endl;

```

```
    try {
        TcpServer server(12345);

        // Start server
        if (auto err = server.start(); err != SocketError::Success) {
            std::cout << "Failed to start server: " << error_to_string(err) << std::endl;
            return;
        }

        std::cout << "Echo server would accept connections and echo messages ..." << std::endl;
        std::cout << "(Actual network operations skipped for demo)" << std::endl;

        server.stop();

    } catch (const std::exception& e) {
        std::cout << "Exception: " << e.what() << std::endl;
    }

    std::cout << "\n Server resources automatically cleaned up by RAII!" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "        WRAPPING C LIBRARIES IN MODERN C++                    \n";
    std::cout << "                                                                                    \n";
    std::cout << "   Demonstrates: RAII, noexcept, [[nodiscard]], extern \"C\"                \n";
    std::cout << "   Example: TCP/UDP Socket Wrapper                                  \n";
    std::cout << "                                    \n";

#ifdef _WIN32
    // Initialize Winsock on Windows
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "WSAStartup failed" << std::endl;
        return 1;
    }
    std::cout << "\n Winsock initialized (Windows)" << std::endl;
#else
    std::cout << "\n Using POSIX sockets (Linux/Unix)" << std::endl;
#endif

```

```
    demonstrate_raii();
    demonstrate_noexcept();
    demonstrate_nodiscard();
    demonstrate_extern_c();
    example_echo_server();

    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "BEST PRACTICES SUMMARY:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n1. RAII (Resource Acquisition Is Initialization):" << std
        ::endl;
    std::cout << "     Acquire resources in constructor" << std::endl;
    std::cout << "     Release resources in destructor" << std::endl;
    std::cout << "     Make classes non-copyable for unique resources" << std
        ::endl;
    std::cout << "     Implement move semantics for transfer" << std::endl;

    std::cout << "\n2. noexcept Usage:" << std::endl;
    std::cout << "     Mark destructors noexcept (implicit in C++11+)" << std
        ::endl;
    std::cout << "     Mark move operations noexcept (enables optimizations)"
        << std::endl;
    std::cout << "     Mark operations that handle errors via return codes" <<
         std::endl;
    std::cout << "     DON'T mark operations that may allocate memory" << std
        ::endl;

    std::cout << "\n3. [[nodiscard]] Usage:" << std::endl;
    std::cout << "     Use for error codes that must be checked" << std::endl;
    std::cout << "     Use for expensive operations (no wasted work)" << std::
        endl;
    std::cout << "     Use for functions where ignoring result is likely a bug
        " << std::endl;
    std::cout << "     Makes APIs self-documenting and safer" << std::endl;

    std::cout << "\n4. extern \"C\" Linkage:" << std::endl;
    std::cout << "     Wrap C library includes in extern \"C\" blocks" << std
        ::endl;
    std::cout << "     Use #ifdef __cplusplus for C/C++ compatibility" << std
        ::endl;
    std::cout << "     Only C-compatible functions can be extern \"C\"" << std
        ::endl;
    std::cout << "     No overloading, no classes, no templates" << std::endl;

    std::cout << "\n5. C/C++ Interop Patterns:" << std::endl;
    std::cout << "     Create thin C++ wrapper classes (RAII)" << std::endl;
    std::cout << "     Hide C types behind strong C++ types" << std::endl;
    std::cout << "     Use std::optional for nullable results" << std::endl;
    std::cout << "     Use error codes/exceptions instead of C error globals"
        << std::endl;
    std::cout << "     Provide modern C++ interfaces (string_view, span, etc.)
        " << std::endl;

```

```
    std::cout << "\n6. Modern C++ Features for C Wrappers:" << std::endl;
    std::cout << "     std::optional<T> for operations that may fail" << std::
        endl;
    std::cout << "     std::string_view for non-owning string parameters" <<
        std::endl;
    std::cout << "     std::span<T> for array views (C++20)" << std::endl;
    std::cout << "     std::unique_ptr with custom deleter" << std::endl;
    std::cout << "     enum class for type-safe error codes" << std::endl;

    std::cout << "\n All socket resources properly cleaned up by RAII!\n" <<
        std::endl;

#ifdef _WIN32
    WSACleanup();
    std::cout << "  Winsock cleaned up\n" << std::endl;
#endif

    return 0;
}
```

18 Source Code: CreatingCApiFromCpp.cpp

File: `src/CreatingCApiFromCpp.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// CREATING A C API FROM C++ CODE
// ==================================================================
// This example demonstrates how to expose C++ functionality through
// a C-compatible API. This is the REVERSE direction from the previous
// example - we're wrapping C++ in C, not C in C++.
//
// USE CASES:
// - Creating libraries usable from C code
// - Plugin systems with C interfaces
// - FFI (Foreign Function Interface) for Python, Rust, etc.
// - Legacy code integration
// - Stable ABI across compiler versions
//
// TOPICS COVERED:
// 1. Opaque pointers (pimpl idiom for C API)
// 2. extern "C" for C linkage
// 3. Exception handling across C boundary
// 4. C++ features hidden behind C API
// 5. Memory management strategies
// 6. Header guards and C/C++ compatibility
// ==================================================================

#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <cstring>
#include <stdexcept>
#include <algorithm>
#include <cstdint>

// ==================================================================
// SECTION 1: C++ IMPLEMENTATION (INTERNAL)
// ==================================================================
// This is the actual C++ code with modern features that we'll expose
// through a C API

namespace image_processing {

class Image {
private:
    size_t width_;
    size_t height_;
    std::vector<uint8_t> pixels_;

public:
    Image(size_t width, size_t height)
        : width_(width), height_(height), pixels_(width * height, 0) {
```

```
        std::cout << "  C++ Image created (" << width_ << "x" << height_ << ")
            " << std::endl;
    }

    ~Image() {
        std::cout << "  C++ Image destroyed" << std::endl;
    }

    // C++ features: const correctness, exceptions
    size_t width() const noexcept { return width_; }
    size_t height() const noexcept { return height_; }

    uint8_t& at(size_t x, size_t y) {
        if (x >= width_ || y >= height_) {
            throw std::out_of_range("Pixel coordinates out of range");
        }
        return pixels_[y * width_ + x];
    }

    const uint8_t& at(size_t x, size_t y) const {
        if (x >= width_ || y >= height_) {
            throw std::out_of_range("Pixel coordinates out of range");
        }
        return pixels_[y * width_ + x];
    }

    // Modern C++ algorithm
    void invert() {
        std::transform(pixels_.begin(), pixels_.end(), pixels_.begin(),
                       [](uint8_t pixel) { return 255 - pixel; });
        std::cout << "  Image inverted (C++ algorithm)" << std::endl;
    }

    void fill(uint8_t value) {
        std::fill(pixels_.begin(), pixels_.end(), value);
        std::cout << "  Image filled with value " << static_cast<int>(value)
            << std::endl;
    }

    void apply_threshold(uint8_t threshold) {
        for (auto& pixel : pixels_) {
            pixel = (pixel >= threshold) ? 255 : 0;
        }
        std::cout << "  Threshold applied at " << static_cast<int>(threshold)
            << std::endl;
    }

    // Get raw data
    const uint8_t* data() const noexcept { return pixels_.data(); }
    size_t size() const noexcept { return pixels_.size(); }
};

} // namespace image_processing

```

```
// ==================================================================
// SECTION 2: C API HEADER (WHAT C CODE SEES)
// ==================================================================
// This is what would go in a .h file that C programs include

#ifdef __cplusplus
extern "C" {
#endif

// Opaque handle - C code sees this as incomplete type
// This hides the C++ implementation details
typedef struct ImageHandle* ImageHandle_t;

// Error codes (C-compatible enum)
typedef enum {
    IMAGE_SUCCESS = 0,
    IMAGE_ERROR_INVALID_HANDLE = -1,
    IMAGE_ERROR_INVALID_DIMENSIONS = -2,
    IMAGE_ERROR_OUT_OF_RANGE = -3,
    IMAGE_ERROR_OUT_OF_MEMORY = -4,
    IMAGE_ERROR_UNKNOWN = -99
} ImageError;

// C API functions
// Note: All return error codes, use output parameters for data

/**
 * Create a new image
 * @param width Image width in pixels
 * @param height Image height in pixels
 * @param out_handle Output parameter for image handle
 * @return Error code
 */
ImageError image_create(size_t width, size_t height, ImageHandle_t* out_handle
    );

/**
 * Destroy an image and free resources
 * @param handle Image handle
 * @return Error code
 */
ImageError image_destroy(ImageHandle_t handle);

/**
 * Get image dimensions
 * @param handle Image handle
 * @param out_width Output parameter for width
 * @param out_height Output parameter for height
 * @return Error code
 */
ImageError image_get_dimensions(ImageHandle_t handle, size_t* out_width,
    size_t* out_height);

/**
```

```
 * Set pixel value
 * @param handle Image handle
 * @param x X coordinate
 * @param y Y coordinate
 * @param value Pixel value (0-255)
 * @return Error code
 */
ImageError image_set_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t
    value);

/**
 * Get pixel value
 * @param handle Image handle
 * @param x X coordinate
 * @param y Y coordinate
 * @param out_value Output parameter for pixel value
 * @return Error code
 */
ImageError image_get_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t*
    out_value);

/**
 * Invert all pixel values
 * @param handle Image handle
 * @return Error code
 */
ImageError image_invert(ImageHandle_t handle);

/**
 * Fill image with a value
 * @param handle Image handle
 * @param value Fill value (0-255)
 * @return Error code
 */
ImageError image_fill(ImageHandle_t handle, uint8_t value);

/**
 * Apply threshold to image
 * @param handle Image handle
 * @param threshold Threshold value (0-255)
 * @return Error code
 */
ImageError image_apply_threshold(ImageHandle_t handle, uint8_t threshold);

/**
 * Get error message for error code
 * @param error Error code
 * @return Human-readable error message (static string)
 */
const char* image_error_string(ImageError error);

#ifdef __cplusplus
}
#endif
```

```

// ==================================================================
// SECTION 3: C API IMPLEMENTATION (BRIDGE LAYER)
// ==================================================================
// This bridges between C and C++ code

// Helper function to catch C++ exceptions and convert to error codes
// NOTE: Must be OUTSIDE extern "C" block since it's a template
template<typename Func>
ImageError safe_call(Func&& func) noexcept {
    try {
        func();
        return IMAGE_SUCCESS;
    } catch (const std::out_of_range& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
        return IMAGE_ERROR_OUT_OF_RANGE;
    } catch (const std::bad_alloc& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
        return IMAGE_ERROR_OUT_OF_MEMORY;
    } catch (const std::exception& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
        return IMAGE_ERROR_UNKNOWN;
    } catch (...) {
        std::cerr << "Unknown exception caught" << std::endl;
        return IMAGE_ERROR_UNKNOWN;
    }
}

// The actual implementation uses extern "C"
extern "C" {

ImageError image_create(size_t width, size_t height, ImageHandle_t* out_handle
    ) {
    if (!out_handle) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    if (width == 0 || height == 0) {
        return IMAGE_ERROR_INVALID_DIMENSIONS;
    }

    return safe_call([&]() {
        // Create C++ object and cast to opaque handle
        auto* img = new image_processing::Image(width, height);
        *out_handle = reinterpret_cast<ImageHandle_t>(img);
    });
}

ImageError image_destroy(ImageHandle_t handle) {
    if (!handle) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    return safe_call([&]() {
```

```
        // Cast back to C++ object and delete
        auto* img = reinterpret_cast<image_processing::Image*>(handle);
        delete img;
    });
}

ImageError image_get_dimensions(ImageHandle_t handle, size_t* out_width,
    size_t* out_height) {
    if (!handle || !out_width || !out_height) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    return safe_call([&]() {
        auto* img = reinterpret_cast<image_processing::Image*>(handle);
        *out_width = img->width();
        *out_height = img->height();
    });
}

ImageError image_set_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t
    value) {
    if (!handle) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    return safe_call([&]() {
        auto* img = reinterpret_cast<image_processing::Image*>(handle);
        img->at(x, y) = value;
    });
}

ImageError image_get_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t*
    out_value) {
    if (!handle || !out_value) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    return safe_call([&]() {
        auto* img = reinterpret_cast<image_processing::Image*>(handle);
        *out_value = img->at(x, y);
    });
}

ImageError image_invert(ImageHandle_t handle) {
    if (!handle) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    return safe_call([&]() {
        auto* img = reinterpret_cast<image_processing::Image*>(handle);
        img->invert();
    });
}

```

```
ImageError image_fill(ImageHandle_t handle, uint8_t value) {
    if (!handle) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    return safe_call([&]() {
        auto* img = reinterpret_cast<image_processing::Image*>(handle);
        img->fill(value);
    });
}

ImageError image_apply_threshold(ImageHandle_t handle, uint8_t threshold) {
    if (!handle) {
        return IMAGE_ERROR_INVALID_HANDLE;
    }

    return safe_call([&]() {
        auto* img = reinterpret_cast<image_processing::Image*>(handle);
        img->apply_threshold(threshold);
    });
}

const char* image_error_string(ImageError error) {
    switch (error) {
        case IMAGE_SUCCESS: return "Success";
        case IMAGE_ERROR_INVALID_HANDLE: return "Invalid handle";
        case IMAGE_ERROR_INVALID_DIMENSIONS: return "Invalid dimensions";
        case IMAGE_ERROR_OUT_OF_RANGE: return "Coordinates out of range";
        case IMAGE_ERROR_OUT_OF_MEMORY: return "Out of memory";
        case IMAGE_ERROR_UNKNOWN: return "Unknown error";
        default: return "Invalid error code";
    }
}

} // extern "C"

// ==================================================================
// SECTION 4: DEMONSTRATION - USING THE C API
// ==================================================================

void demonstrate_c_api_usage() {
    std::cout << "\n=== USING C API (SIMULATING C CODE) ===" << std::endl;
    std::cout << "Note: This C++ code simulates how C code would use the API\n
        " << std::endl;

    // In real C code, these would be the only includes needed:
    // #include "image_api.h"

    ImageHandle_t image = nullptr;
    ImageError err;

    // Create image
    std::cout << "1. Creating 10x10 image:" << std::endl;
    err = image_create(10, 10, &image);
```

```
    if (err != IMAGE_SUCCESS) {
        std::cout << "   Error: " << image_error_string(err) << std::endl;
        return;
    }
    std::cout << "    Image created successfully" << std::endl;

    // Get dimensions
    size_t width, height;
    err = image_get_dimensions(image, &width, &height);
    if (err == IMAGE_SUCCESS) {
        std::cout << "\n2. Image dimensions: " << width << "x" << height <<
            std::endl;
    }

    // Fill with value
    std::cout << "\n3. Filling image with value 128:" << std::endl;
    err = image_fill(image, 128);
    if (err != IMAGE_SUCCESS) {
        std::cout << "   Error: " << image_error_string(err) << std::endl;
    }

    // Set some pixels
    std::cout << "\n4. Setting individual pixels:" << std::endl;
    image_set_pixel(image, 0, 0, 255);
    image_set_pixel(image, 1, 1, 200);
    image_set_pixel(image, 2, 2, 150);
    std::cout << "    Pixels set" << std::endl;

    // Get pixel values
    std::cout << "\n5. Reading pixel values:" << std::endl;
    uint8_t value;
    for (int i = 0; i < 3; i++) {
        err = image_get_pixel(image, i, i, &value);
        if (err == IMAGE_SUCCESS) {
            std::cout << "   Pixel(" << i << "," << i << ") = "
                      << static_cast<int>(value) << std::endl;
        }
    }

    // Invert
    std::cout << "\n6. Inverting image:" << std::endl;
    err = image_invert(image);
    if (err != IMAGE_SUCCESS) {
        std::cout << "   Error: " << image_error_string(err) << std::endl;
    }

    // Read inverted values
    std::cout << "\n7. Reading inverted pixel values:" << std::endl;
    for (int i = 0; i < 3; i++) {
        err = image_get_pixel(image, i, i, &value);
        if (err == IMAGE_SUCCESS) {
            std::cout << "   Pixel(" << i << "," << i << ") = "
                      << static_cast<int>(value) << std::endl;
        }
```

```
    }

    // Apply threshold
    std::cout << "\n8. Applying threshold at 100:" << std::endl;
    err = image_apply_threshold(image, 100);

    // Error handling
    std::cout << "\n9. Testing error handling (out of range access):" << std::endl;
    err = image_get_pixel(image, 100, 100, &value);
    if (err != IMAGE_SUCCESS) {
        std::cout << "    Error caught: " << image_error_string(err) << std::endl;
    }

    // Cleanup
    std::cout << "\n10. Destroying image:" << std::endl;
    err = image_destroy(image);
    if (err == IMAGE_SUCCESS) {
        std::cout << "    Image destroyed successfully" << std::endl;
    }
}

// ==================================================================
// SECTION 5: BEST PRACTICES EXPLANATION
// ==================================================================

void explain_best_practices() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "BEST PRACTICES FOR C API FROM C++ CODE:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n1. OPAQUE HANDLES (PIMPL IDIOM FOR C):" << std::endl;
    std::cout << "    Hide C++ implementation details" << std::endl;
    std::cout << "    Use typedef struct Name* Handle_t;" << std::endl;
    std::cout << "    Cast to/from C++ objects internally" << std::endl;
    std::cout << "    Stable ABI - C++ changes don't affect C API" << std::endl;

    std::cout << "\n2. EXCEPTION HANDLING:" << std::endl;
    std::cout << "    NEVER let C++ exceptions cross C boundary!" << std::endl;
    std::cout << "    Catch ALL exceptions in extern \"C\" functions" << std::endl;
    std::cout << "    Convert exceptions to error codes" << std::endl;
    std::cout << "    Use try-catch wrappers consistently" << std::endl;

    std::cout << "\n3. ERROR HANDLING:" << std::endl;
    std::cout << "    Return error codes (int or enum)" << std::endl;
    std::cout << "    Use output parameters for data" << std::endl;
    std::cout << "    Provide error_to_string() function" << std::endl;
    std::cout << "    Check for NULL pointers before dereferencing" << std::endl;

```

```
    std::cout << "\n4. MEMORY MANAGEMENT:" << std::endl;
    std::cout << "     C code creates/destroys via API functions" << std::endl;
    std::cout << "     NEVER expose C++ new/delete directly" << std::endl;
    std::cout << "     Provide create() and destroy() functions" << std::endl;
    std::cout << "     Match create/destroy across DLL boundaries" << std::endl;

    std::cout << "\n5. HEADER GUARDS AND COMPATIBILITY:" << std::endl;
    std::cout << "     Use #ifdef __cplusplus for extern \"C\" blocks" << std::endl;
    std::cout << "     Include guards in all headers" << std::endl;
    std::cout << "     Use C-compatible types (no bool, use int)" << std::endl;
    std::cout << "     No function overloading in C API" << std::endl;

    std::cout << "\n6. C++ FEATURES TO AVOID IN C API:" << std::endl;
    std::cout << "     Classes (use opaque handles instead)" << std::endl;
    std::cout << "     Templates (not C-compatible)" << std::endl;
    std::cout << "     Function overloading (C doesn't support it)" << std::endl;
    std::cout << "     Default arguments (C doesn't support them)" << std::endl;
    std::cout << "     References (use pointers)" << std::endl;
    std::cout << "     bool type (use int, 0 for false, 1 for true)" << std::endl;

    std::cout << "\n7. DOCUMENTATION:" << std::endl;
    std::cout << "     Document ownership semantics clearly" << std::endl;
    std::cout << "     Specify thread-safety guarantees" << std::endl;
    std::cout << "     List all possible error codes" << std::endl;
    std::cout << "     Provide usage examples" << std::endl;

    std::cout << "\n8. TESTING:" << std::endl;
    std::cout << "     Test C API from actual C code" << std::endl;
    std::cout << "     Verify exception safety" << std::endl;
    std::cout << "     Test error conditions" << std::endl;
    std::cout << "     Check for memory leaks" << std::endl;
}

// ==================================================================
// SECTION 6: COMPARISON WITH PREVIOUS EXAMPLE
// ==================================================================

void compare_approaches() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "COMPARISON: C IN C++ vs C++ IN C:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\nPREVIOUS EXAMPLE (Wrapping C in C++):" << std::endl;
    std::cout << "  Purpose: Use C libraries from C++ code" << std::endl;
    std::cout << "  Technique: RAII wrappers, smart pointers" << std::endl;
    std::cout << "  Example: Socket library → C++ Socket class" << std::endl;
    std::cout << "  Benefits: Modern C++ safety, automatic cleanup" << std::
```

```
        endl;

    std::cout << "\nTHIS EXAMPLE (Exposing C++ as C):" << std::endl;
    std::cout << "  Purpose: Make C++ code usable from C" << std::endl;
    std::cout << "  Technique: Opaque handles, extern \"C\"" << std::endl;
    std::cout << "  Example: C++ Image class → C image_* functions" << std::
        endl;
    std::cout << "  Benefits: C compatibility, stable ABI, FFI-ready" << std::
        endl;

    std::cout << "\nWHEN TO USE EACH APPROACH:" << std::endl;
    std::cout << "\nWrapping C in C++ (Previous):" << std::endl;
    std::cout << " •  You have C libraries to use" << std::endl;
    std::cout << " •  You want modern C++ features" << std::endl;
    std::cout << " •  You want automatic resource management" << std::endl;
    std::cout << " •  Internal project use" << std::endl;

    std::cout << "\nExposing C++ as C (This):" << std::endl;
    std::cout << " •  You want to create a C-compatible library" << std::endl;
    std::cout << " •  You need stable ABI across versions" << std::endl;
    std::cout << " •  You want FFI for other languages" << std::endl;
    std::cout << " •  You need to support legacy C code" << std::endl;
    std::cout << " •  Plugin systems with C interfaces" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                  \n";
    std::cout << "         CREATING A C API FROM C++ CODE
                                  \n";
    std::cout << "
                                                                    \n";
    std::cout << "   Demonstrates: Opaque Handles, extern \"C\", Exception
        Safety      \n";
    std::cout << "   Example: Image Processing Library with C Interface
                    \n";
    std::cout << "                                  \n";

    demonstrate_c_api_usage();
    explain_best_practices();
    compare_approaches();

    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "SUMMARY:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\ n KEY TECHNIQUES DEMONSTRATED:" << std::endl;
    std::cout << "   1. Opaque handles for hiding C++ implementation" << std::
        endl;
    std::cout << "   2. extern \"C\" for C-compatible linkage" << std::endl;
```

```
    std::cout << "   3. Exception safety across C/C++ boundary" << std::endl;
    std::cout << "   4. Error code-based error handling" << std::endl;
    std::cout << "   5. Output parameters instead of return values" << std::
        endl;
    std::cout << "   6. Proper memory management (create/destroy)" << std::
        endl;

    std::cout << "\n REAL-WORLD APPLICATIONS:" << std::endl;
    std::cout << " •   Game engine C APIs (Unity, Unreal plugins)" << std::
        endl;
    std::cout << " •   Database drivers (SQLite, PostgreSQL)" << std::endl;
    std::cout << " •   Graphics libraries (Vulkan, OpenGL wrappers)" << std::
        endl;
    std::cout << " •   Compression libraries (zlib, bzip2 style)" << std::endl
        ;
    std::cout << " •   Python/Ruby/Lua bindings via C API" << std::endl;

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << " •   C++ power with C compatibility" << std::endl;
    std::cout << " •   Stable ABI (no name mangling issues)" << std::endl;
    std::cout << " •   Can be used from ANY language with C FFI" << std::endl;
    std::cout << " •   Hide implementation details completely" << std::endl;
    std::cout << " •   Version changes don't break binary compatibility" <<
        std::endl;

    std::cout << "\n All resources properly managed!\n" << std::endl;

    return 0;
}
```

19 Source Code: DependencyInjection.cpp

File: `src/DependencyInjection.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <functional>

// ====================================================================
// DEPENDENCY INJECTION IN C++ - NO PRIVATE INHERITANCE NEEDED
// ====================================================================

// ====================================================================
// 1. TRADITIONAL DI - PUBLIC INHERITANCE FOR INTERFACES
// ====================================================================

// Abstract interface (use PUBLIC inheritance)
class ILogger {
public:
    virtual ~ILogger() = default;
    virtual void log(const std::string& message) = 0;
};

// Concrete implementations
class ConsoleLogger : public ILogger {  //  PUBLIC inheritance
public:
    void log(const std::string& message) override {
        std::cout << "[CONSOLE] " << message << std::endl;
    }
};

class FileLogger : public ILogger {  //  PUBLIC inheritance
private:
    std::string filename;

public:
    FileLogger(const std::string& file) : filename(file) {}

    void log(const std::string& message) override {
        std::cout << "[FILE:" << filename << "] " << message << std::endl;
    }
};

// Service that depends on ILogger - uses COMPOSITION, not inheritance
class UserService {
private:
    std::shared_ptr<ILogger> logger;  //  Has-a relationship via composition

public:
    // Constructor injection
    UserService(std::shared_ptr<ILogger> log) : logger(std::move(log)) {}
```

```

    void createUser(const std::string& username) {
        logger->log("Creating user: " + username);
        // User creation logic...
        logger->log("User created successfully: " + username);
    }

    void deleteUser(const std::string& username) {
        logger->log("Deleting user: " + username);
        // User deletion logic...
        logger->log("User deleted: " + username);
    }
};

void example_traditional_di() {
    std::cout << "\n=== 1. TRADITIONAL DI - PUBLIC INHERITANCE + COMPOSITION ===" << std::endl;
    std::cout << "Pattern: Interface (public) + Composition (has-a)\n" << std::endl;

    // Inject ConsoleLogger
    auto consoleLogger = std::make_shared<ConsoleLogger>();
    UserService service1(consoleLogger);
    service1.createUser("Alice");

    std::cout << std::endl;

    // Inject FileLogger - same service, different dependency
    auto fileLogger = std::make_shared<FileLogger>("users.log");
    UserService service2(fileLogger);
    service2.createUser("Bob");

    std::cout << "\n DEPENDENCY INJECTION USES:" << std::endl;
    std::cout << " •   PUBLIC inheritance for interfaces (ILogger)" << std::endl;
    std::cout << " •   COMPOSITION for dependencies (has-a ILogger)" << std::endl;
    std::cout << " •   NO private inheritance needed!" << std::endl;
}

// ==================================================================
// 2. CONSTRUCTOR INJECTION (MOST COMMON)
// ==================================================================

class IDatabase {
public:
    virtual ~IDatabase() = default;
    virtual void save(const std::string& data) = 0;
    virtual std::string load(const std::string& key) = 0;
};

class MockDatabase : public IDatabase {
public:
    void save(const std::string& data) override {
```

```
        std::cout << "  [MockDB] Saving: " << data << std::endl;
    }

    std::string load(const std::string& key) override {
        return "mock_data_for_" + key;
    }
};

class PostgresDatabase : public IDatabase {
public:
    void save(const std::string& data) override {
        std::cout << "  [PostgreSQL] Saving to database: " << data << std::
            endl;
    }

    std::string load(const std::string& key) override {
        return "postgres_data_for_" + key;
    }
};

// Repository with multiple dependencies injected via constructor
class OrderRepository {
private:
    std::shared_ptr<IDatabase> database;  // Composition
    std::shared_ptr<ILogger> logger;      // Composition

public:
    // Constructor injection of multiple dependencies
    OrderRepository(std::shared_ptr<IDatabase> db, std::shared_ptr<ILogger>
        log)
        : database(std::move(db)), logger(std::move(log)) {}

    void saveOrder(const std::string& orderId) {
        logger->log("Saving order: " + orderId);
        database->save("Order:" + orderId);
        logger->log("Order saved successfully");
    }

    void loadOrder(const std::string& orderId) {
        logger->log("Loading order: " + orderId);
        auto data = database->load(orderId);
        std::cout << "  Loaded: " << data << std::endl;
        logger->log("Order loaded successfully");
    }
};

void example_constructor_injection() {
    std::cout << "\n=== 2. CONSTRUCTOR INJECTION (MULTIPLE DEPENDENCIES) ==="
        << std::endl;
    std::cout << "Pattern: Inject all dependencies via constructor\n" << std::
        endl;

    auto logger = std::make_shared<ConsoleLogger>();
    auto database = std::make_shared<PostgresDatabase>();
```

```

    OrderRepository repo(database, logger);
    repo.saveOrder("ORD-12345");

    std::cout << std::endl;
    repo.loadOrder("ORD-12345");

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << " •   Dependencies are explicit and immutable" << std::endl;
    std::cout << " •   Easy to test (inject mocks)" << std::endl;
    std::cout << " •   No private inheritance - uses composition" << std::endl;
}

// ==================================================================
// 3. INTERFACE INJECTION (STRATEGY PATTERN)
// ==================================================================

class IEmailSender {
public:
    virtual ~IEmailSender() = default;
    virtual void sendEmail(const std::string& to, const std::string& message) = 0;
};

class SmtpEmailSender : public IEmailSender {
public:
    void sendEmail(const std::string& to, const std::string& message) override {
        std::cout << "  [SMTP] Sending email to " << to << ": " << message << std::endl;
    }
};

class MockEmailSender : public IEmailSender {
public:
    void sendEmail(const std::string& to, const std::string& message) override {
        std::cout << "  [MOCK] Would send email to " << to << ": " << message << std::endl;
    }
};

class NotificationService {
private:
    std::shared_ptr<IEmailSender> emailSender;  // Composition

public:
    NotificationService(std::shared_ptr<IEmailSender> sender)
        : emailSender(std::move(sender)) {}

    // Setter injection (optional, less common)
    void setEmailSender(std::shared_ptr<IEmailSender> sender) {
        emailSender = std::move(sender);
```

```
    }

    void notifyUser(const std::string& email, const std::string& notification) {
        std::cout << "Sending notification..." << std::endl;
        emailSender->sendEmail(email, notification);
    }
};

void example_interface_injection() {
    std::cout << "\n=== 3. INTERFACE INJECTION (STRATEGY PATTERN) ===" << std::endl;
    std::cout << "Pattern: Inject behavior through interfaces\n" << std::endl;

    auto smtpSender = std::make_shared<SmtpEmailSender>();
    NotificationService service(smtpSender);

    service.notifyUser("user@example.com", "Your order has shipped!");

    std::cout << "\nSwitching to mock sender (setter injection):" << std::endl;
    auto mockSender = std::make_shared<MockEmailSender>();
    service.setEmailSender(mockSender);
    service.notifyUser("user@example.com", "Your order is delivered!");

    std::cout << "\n KEY POINTS:" << std::endl;
    std::cout << " •   Strategy pattern uses PUBLIC inheritance" << std::endl;
    std::cout << " •   Can switch implementations at runtime" << std::endl;
    std::cout << " •   No private inheritance involved" << std::endl;
}

// ==================================================================
// 4. TEMPLATE-BASED DI (COMPILE-TIME INJECTION)
// ==================================================================

// No interface needed - duck typing via templates
class ConsoleOutput {
public:
    void write(const std::string& msg) {
        std::cout << "[Console] " << msg << std::endl;
    }
};

class FileOutput {
public:
    void write(const std::string& msg) {
        std::cout << "[File] " << msg << std::endl;
    }
};

// Template-based dependency injection (no inheritance at all!)
template<typename TOutput>
class MessageProcessor {
private:
```

```
    TOutput output;  // Composition with template type

public:
    MessageProcessor(TOutput out) : output(std::move(out)) {}

    void process(const std::string& message) {
        std::cout << "Processing message: " << message << std::endl;
        output.write("Processed: " + message);
    }
};

void example_template_di() {
    std::cout << "\n=== 4. TEMPLATE-BASED DI (COMPILE-TIME) ===" << std::endl;
    std::cout << "Pattern: No inheritance, no interfaces - pure templates\n"
        << std::endl;

    ConsoleOutput console;
    MessageProcessor<ConsoleOutput> processor1(console);
    processor1.process("Hello World");

    std::cout << std::endl;

    FileOutput file;
    MessageProcessor<FileOutput> processor2(file);
    processor2.process("Template DI");

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " •   Zero runtime overhead (compile-time binding)" << std::
        endl;
    std::cout << " •   No virtual functions needed" << std::endl;
    std::cout << " •   No inheritance (public OR private) needed!" << std::
        endl;
    std::cout << " •   Duck typing - if it has write(), it works" << std::endl
        ;
}

// ==================================================================
// 5. DI CONTAINER / SERVICE LOCATOR PATTERN
// ==================================================================

class DIContainer {
private:
    std::shared_ptr<ILogger> logger;
    std::shared_ptr<IDatabase> database;
    std::shared_ptr<IEmailSender> emailSender;

public:
    // Register dependencies
    void registerLogger(std::shared_ptr<ILogger> log) {
        logger = std::move(log);
    }

    void registerDatabase(std::shared_ptr<IDatabase> db) {
        database = std::move(db);
```

```
    }

    void registerEmailSender(std::shared_ptr<IEmailSender> sender) {
        emailSender = std::move(sender);
    }

    // Resolve dependencies
    std::shared_ptr<ILogger> getLogger() { return logger; }
    std::shared_ptr<IDatabase> getDatabase() { return database; }
    std::shared_ptr<IEmailSender> getEmailSender() { return emailSender; }

    // Factory method to create services with auto-injected dependencies
    std::shared_ptr<OrderRepository> createOrderRepository() {
        return std::make_shared<OrderRepository>(database, logger);
    }
};

void example_di_container() {
    std::cout << "\n=== 5. DI CONTAINER / SERVICE LOCATOR ===" << std::endl;
    std::cout << "Pattern: Central registry for dependency management\n" <<
        std::endl;

    DIContainer container;

    // Setup (composition root)
    container.registerLogger(std::make_shared<ConsoleLogger>());
    container.registerDatabase(std::make_shared<MockDatabase>());
    container.registerEmailSender(std::make_shared<MockEmailSender>());

    // Resolve dependencies
    auto repo = container.createOrderRepository();
    repo->saveOrder("ORD-99999");

    auto emailSender = container.getEmailSender();
    emailSender->sendEmail("admin@example.com", "System started");

    std::cout << "\n DI CONTAINER USES:" << std::endl;
    std::cout << " •   Stores dependencies as members (composition)" << std::
        endl;
    std::cout << " •   Public inheritance for interfaces only" << std::endl;
    std::cout << " •   No private inheritance needed" << std::endl;
}

// ==================================================================
// 6. FUNCTIONAL DI (C++11/14 STYLE)
// ==================================================================

class PaymentService {
private:
    std::function<void(const std::string&)> logFunc;
    std::function<bool(double)> validateFunc;

public:
    // Inject functions instead of objects
```

```
    PaymentService(
        std::function<void(const std::string&)> logger,
        std::function<bool(double)> validator)
        : logFunc(std::move(logger))
        , validateFunc(std::move(validator)) {}

    void processPayment(double amount) {
        logFunc("Processing payment of $" + std::to_string(amount));

        if (validateFunc(amount)) {
            logFunc("Payment validated and processed");
        } else {
            logFunc("Payment validation failed");
        }
    }
};

void example_functional_di() {
    std::cout << "\n=== 6. FUNCTIONAL DI (INJECT FUNCTIONS) ===" << std::endl;
    std::cout << "Pattern: Inject std::function instead of interfaces\n" <<
        std::endl;

    // Define behaviors as lambdas
    auto logger = [](const std::string& msg) {
        std::cout << "  [Payment Log] " << msg << std::endl;
    };

    auto validator = [](double amount) {
        return amount > 0 && amount < 10000;  // Simple validation
    };

    PaymentService service(logger, validator);
    service.processPayment(150.50);

    std::cout << std::endl;
    service.processPayment(-50.0);  // Invalid

    std::cout << "\n FUNCTIONAL DI:" << std::endl;
    std::cout << " •   No interfaces needed" << std::endl;
    std::cout << " •   No inheritance at all (public or private)" << std::endl
        ;
    std::cout << " •   Lightweight and flexible" << std::endl;
    std::cout << " •   Uses std::function and lambdas" << std::endl;
}

// ==================================================================
// 7. WHY NOT PRIVATE INHERITANCE FOR DI?
// ==================================================================

//   WRONG: Using private inheritance for DI (anti-pattern)
class WrongServiceWithPrivateInheritance : private ILogger {
private:
    // This makes ILogger methods private - defeats the purpose of DI!

```

```
public:
    void doSomething() {
        log("This is confusing!");  // Can only use internally
    }

    //  Cannot inject different logger implementations
    //  Cannot swap loggers at runtime
    //  Tightly coupled to ILogger
};

//  CORRECT: Using composition for DI
class CorrectServiceWithComposition {
private:
    std::shared_ptr<ILogger> logger;  // Injected dependency

public:
    CorrectServiceWithComposition(std::shared_ptr<ILogger> log)
        : logger(std::move(log)) {}

    void doSomething() {
        logger->log("This is the right way!");
    }

    //  Can inject any ILogger implementation
    //  Can swap loggers at runtime
    //  Loosely coupled
};

void example_why_not_private() {
    std::cout << "\n=== 7. WHY NOT PRIVATE INHERITANCE FOR DI? ===" << std::endl;

    std::cout << "\n PRIVATE INHERITANCE FOR DI IS WRONG BECAUSE:" << std::endl;
    std::cout << " •  Cannot inject different implementations" << std::endl;
    std::cout << " •  Tightly coupled to specific base class" << std::endl;
    std::cout << " •  Cannot swap dependencies at runtime" << std::endl;
    std::cout << " •  Violates Dependency Inversion Principle" << std::endl;
    std::cout << " •  Makes testing difficult (can't inject mocks)" << std::endl;

    std::cout << "\n DI REQUIRES LOOSE COUPLING:" << std::endl;
    std::cout << " •  Use interfaces (abstract base classes)" << std::endl;
    std::cout << " •  Use PUBLIC inheritance for interfaces" << std::endl;
    std::cout << " •  Use COMPOSITION to hold dependencies" << std::endl;
    std::cout << " •  Inject dependencies via constructor/setter" << std::endl;
}

// ================================================================
// MAIN FUNCTION
// ================================================================

int main() {
```

```
    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  DEPENDENCY INJECTION IN C++ - NO PRIVATE INHERITANCE!" << std::endl;
    std::cout << " ================================================================" << std::endl;

    example_traditional_di();
    example_constructor_injection();
    example_interface_injection();
    example_template_di();
    example_di_container();
    example_functional_di();
    example_why_not_private();

    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  DEPENDENCY INJECTION SUMMARY" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\ n DI PATTERNS IN C++ (NO PRIVATE INHERITANCE):" << std::endl;
    std::cout << "\n1. INTERFACE-BASED DI (MOST COMMON):" << std::endl;
    std::cout << " •   Define abstract interfaces (ILogger, IDatabase, etc.)" << std::endl;
    std::cout << " •   Implementations use PUBLIC inheritance" << std::endl;
    std::cout << " •   Services use COMPOSITION (has-a relationship)" << std::endl;
    std::cout << " •   Inject via constructor or setter" << std::endl;

    std::cout << "\n2. TEMPLATE-BASED DI:" << std::endl;
    std::cout << " •   No inheritance at all!" << std::endl;
    std::cout << " •   Compile-time polymorphism via templates" << std::endl;
    std::cout << " •   Duck typing - if it quacks like a duck..." << std::endl;
    std::cout << " •   Zero runtime overhead" << std::endl;

    std::cout << "\n3. FUNCTIONAL DI:" << std::endl;
    std::cout << " •   Inject std::function instead of interfaces" << std::endl;
    std::cout << " •   No inheritance needed" << std::endl;
    std::cout << " •   Lightweight and flexible" << std::endl;
    std::cout << " •   Great for simple dependencies" << std::endl;

    std::cout << "\n4. DI CONTAINER:" << std::endl;
    std::cout << " •   Central registry for dependencies" << std::endl;
    std::cout << " •   Auto-wiring of complex object graphs" << std::endl;
    std::cout << " •   Uses composition internally" << std::endl;
    std::cout << " •   Similar to Spring (Java) or Autofac (.NET)" << std::
```

```
        endl;

    std::cout << "\n NEVER USE PRIVATE INHERITANCE FOR DI:" << std::endl;
    std::cout << " •   Private inheritance = tight coupling" << std::endl;
    std::cout << " •   Cannot inject different implementations" << std::endl;
    std::cout << " •   Cannot swap at runtime" << std::endl;
    std::cout << " •   Defeats the entire purpose of DI!" << std::endl;

    std::cout << "\n CORRECT DI APPROACH:" << std::endl;
    std::cout << " •   PUBLIC inheritance for interfaces (polymorphism)" <<
        std::endl;
    std::cout << " •   COMPOSITION for dependencies (has-a)" << std::endl;
    std::cout << " •   Constructor/setter injection" << std::endl;
    std::cout << " •   Inversion of Control (IoC)" << std::endl;

    std::cout << "\n KEY PRINCIPLE:" << std::endl;
    std::cout << "   \"Depend on abstractions, not concretions\"" << std::endl
        ;
    std::cout << "   - Dependency Inversion Principle (SOLID)" << std::endl;

    std::cout << "\n WHEN TO USE EACH:" << std::endl;
    std::cout << " •   Interface-based DI: Runtime polymorphism, complex
        systems" << std::endl;
    std::cout << " •   Template-based DI: Performance-critical code, simple
        deps" << std::endl;
    std::cout << " •   Functional DI: Simple behaviors, lambdas" << std::endl;
    std::cout << " •   DI Container: Large applications with many dependencies
        " << std::endl;
    std::cout << " •   Private inheritance: NEVER for DI!" << std::endl;

    std::cout << "\n
        ================================================================\n" <<
         std::endl;

    return 0;
}
```

20 Source Code: DiamondProblem.cpp

File: `src/DiamondProblem.cpp`
Repository: View on GitHub

```cpp
// DiamondProblem.cpp
// Demonstrates the Diamond Problem (Diamond of Death) in C++ and its solution
//
// The Diamond Problem occurs when a class inherits from two classes that both
// inherit from the same base class, creating a diamond-shaped inheritance hierarchy.
//
// PROBLEM:
// Without virtual inheritance, the derived class contains TWO copies of the
// base class, leading to ambiguity and wasted memory.
//
// SOLUTION:
// Use 'virtual' keyword in inheritance to ensure only ONE copy of the base
// class exists in the final derived class.
//
//     Base
//    /    \
//   A      B
//    \    /
//   Derived
//
// KEY CONCEPTS:
// 1. Multiple Inheritance creates duplicate base class instances
// 2. Virtual Inheritance solves the duplication
// 3. Constructor initialization order with virtual bases
// 4. When to use and when to avoid multiple inheritance

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

// ============================================================================
// SECTION 1: THE PROBLEM - Diamond Without Virtual Inheritance
// ============================================================================

namespace diamond_problem {

class Device {
protected:
    std::string name_;
    int id_;

public:
    Device(const std::string& name, int id)
        : name_(name), id_(id) {
```

```
        std::cout << "    [Device] Constructed: " << name_ << " (ID: " << id_ << ")\n";
    }

    void displayInfo() const {
        std::cout << "    Device: " << name_ << ", ID: " << id_ << "\n";
    }

    int getId() const { return id_; }
};

class InputDevice : public Device {
protected:
    std::string inputType_;

public:
    InputDevice(const std::string& name, int id, const std::string& inputType)
        : Device(name, id), inputType_(inputType) {
        std::cout << "    [InputDevice] Constructed: " << inputType_ << "\n";
    }

    void showInput() const {
        std::cout << "    Input Type: " << inputType_ << "\n";
    }
};

class OutputDevice : public Device {
protected:
    std::string outputType_;

public:
    OutputDevice(const std::string& name, int id, const std::string& outputType)
        : Device(name, id), outputType_(outputType) {
        std::cout << "    [OutputDevice] Constructed: " << outputType_ << "\n";
    }

    void showOutput() const {
        std::cout << "    Output Type: " << outputType_ << "\n";
    }
};

// PROBLEM: IODevice inherits Device TWICE (through InputDevice and OutputDevice)
class IODevice : public InputDevice, public OutputDevice {
public:
    IODevice(const std::string& name, int id)
        : InputDevice(name, id, "Keyboard"),
          OutputDevice(name, id, "Display") {
        std::cout << "    [IODevice] Constructed\n";
    }

    void showInfo() const {
```

```
        // AMBIGUITY: Which Device::displayInfo() to call?
        // AMBIGUITY: Which Device::getId() to call?
        // Compiler error without explicit qualification

        // Must specify which base class path to use:
        std::cout << "  \n  Input Device Info:\n";
        InputDevice::displayInfo();   // Calls Device from InputDevice path
        showInput();

        std::cout << "\n  Output Device Info:\n";
        OutputDevice::displayInfo();  // Calls Device from OutputDevice path
        showOutput();

        std::cout << "\n    Problem: Two copies of Device exist!\n";
        std::cout << "      InputDevice::getId() = " << InputDevice::getId() << "\n";
        std::cout << "      OutputDevice::getId() = " << OutputDevice::getId() << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1: Diamond Problem (Without Virtual Inheritance) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Creating IODevice...\n";
    std::cout << "Notice: Device constructor called TWICE!\n\n";

    IODevice device("MyIODevice", 42);

    std::cout << "\nShowing device information:\n";
    device.showInfo();

    std::cout << "\nMemory Analysis:\n";
    std::cout << "  sizeof(Device) = " << sizeof(Device) << " bytes\n";
    std::cout << "  sizeof(InputDevice) = " << sizeof(InputDevice) << " bytes\n";
    std::cout << "  sizeof(OutputDevice) = " << sizeof(OutputDevice) << " bytes\n";
    std::cout << "  sizeof(IODevice) = " << sizeof(IODevice) << " bytes\n";
    std::cout << "    IODevice contains TWO Device instances (memory waste)\n";
}

} // namespace diamond_problem

// ==========================================================================
// SECTION 2: THE SOLUTION - Virtual Inheritance
// ==========================================================================
```

```

namespace virtual_inheritance_solution {

class Device {
protected:
    std::string name_;
    int id_;

public:
    Device(const std::string& name, int id)
        : name_(name), id_(id) {
        std::cout << "    [Device] Constructed: " << name_ << " (ID: " << id_ << ")\n";
    }

    virtual ~Device() = default;

    void displayInfo() const {
        std::cout << "    Device: " << name_ << ", ID: " << id_ << "\n";
    }

    int getId() const { return id_; }
    std::string getName() const { return name_; }
};

// SOLUTION: Use 'virtual' keyword in inheritance
class InputDevice : virtual public Device {
protected:
    std::string inputType_;

public:
    InputDevice(const std::string& name, int id, const std::string& inputType)
        : Device(name, id), inputType_(inputType) {
        std::cout << "    [InputDevice] Constructed: " << inputType_ << "\n";
    }

    void showInput() const {
        std::cout << "    Input Type: " << inputType_ << "\n";
    }
};

// SOLUTION: Use 'virtual' keyword in inheritance
class OutputDevice : virtual public Device {
protected:
    std::string outputType_;

public:
    OutputDevice(const std::string& name, int id, const std::string& outputType)
        : Device(name, id), outputType_(outputType) {
        std::cout << "    [OutputDevice] Constructed: " << outputType_ << "\n";
    }
```

```

    void showOutput() const {
        std::cout << "    Output Type: " << outputType_ << "\n";
    }
};

// Now IODevice has only ONE copy of Device
class IODevice : public InputDevice, public OutputDevice {
public:
    // IMPORTANT: With virtual inheritance, the most derived class
    // is responsible for initializing the virtual base class
    IODevice(const std::string& name, int id)
        : Device(name, id),  // Must initialize Device directly!
          InputDevice(name, id, "Keyboard"),
          OutputDevice(name, id, "Display") {
        std::cout << "    [IODevice] Constructed\n";
    }

    void showInfo() const {
        // No ambiguity now - only one Device exists
        std::cout << "\n  Device Info:\n";
        displayInfo();  // Unambiguous!

        std::cout << "\n  Input/Output Details:\n";
        showInput();
        showOutput();

        std::cout << "\n   Solution: Only ONE copy of Device exists!\n";
        std::cout << "    getId() = " << getId() << " (no ambiguity)\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 2: Solution - Virtual Inheritance ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Creating IODevice with virtual inheritance...\n";
    std::cout << "Notice: Device constructor called ONCE!\n\n";

    IODevice device("MyIODevice", 42);

    std::cout << "\nShowing device information:\n";
    device.showInfo();

    std::cout << "\nMemory Analysis:\n";
    std::cout << "  sizeof(Device) = " << sizeof(Device) << " bytes\n";
    std::cout << "  sizeof(InputDevice) = " << sizeof(InputDevice) << " bytes\n";
    std::cout << "  sizeof(OutputDevice) = " << sizeof(OutputDevice) << " bytes\n";
    std::cout << "  sizeof(IODevice) = " << sizeof(IODevice) << " bytes\n";
    std::cout << "   IODevice contains only ONE Device instance (no duplication)\n";
```

```
}

} // namespace virtual_inheritance_solution

// ==========================================================================
// SECTION 3: Real-World Example - File System
// ==========================================================================

namespace filesystem_example {

class File {
protected:
    std::string path_;
    size_t size_;

public:
    File(const std::string& path, size_t size)
        : path_(path), size_(size) {
        std::cout << "    [File] Created: " << path_ << " (" << size_ << " bytes)\n";
    }

    virtual ~File() = default;

    std::string getPath() const { return path_; }
    size_t getSize() const { return size_; }

    void displayInfo() const {
        std::cout << "    File: " << path_ << " (" << size_ << " bytes)\n";
    }
};

// Readable file
class ReadableFile : virtual public File {
public:
    ReadableFile(const std::string& path, size_t size)
        : File(path, size) {
        std::cout << "    [ReadableFile] Readable permissions added\n";
    }

    void read() const {
        std::cout << "    Reading from: " << path_ << "\n";
    }
};

// Writable file
class WritableFile : virtual public File {
public:
    WritableFile(const std::string& path, size_t size)
        : File(path, size) {
```

```
        std::cout << "    [WritableFile] Writable permissions added\n";
    }

    void write(const std::string& data) {
        std::cout << "    Writing to: " << path_ << " - Data: \"" << data << "
            \"\n";
        size_ += data.length();
    }
};

// Read-Write file - inherits from both
class ReadWriteFile : public ReadableFile, public WritableFile {
public:
    ReadWriteFile(const std::string& path, size_t size)
        : File(path, size),  // Initialize virtual base
          ReadableFile(path, size),
          WritableFile(path, size) {
        std::cout << "    [ReadWriteFile] Read-Write file created\n";
    }

    void showInfo() const {
        displayInfo();  // No ambiguity
        std::cout << "    Permissions: Read + Write\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 3: Real-World Example - File System ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Creating ReadWriteFile with virtual inheritance:\n\n";
    ReadWriteFile file("document.txt", 1024);

    std::cout << "\nUsing the file:\n";
    file.showInfo();
    file.read();
    file.write("Hello, World!");

    std::cout << "\nFinal state:\n";
    file.displayInfo();
}

} // namespace filesystem_example

//
    ==========================================================================
// SECTION 4: Constructor Initialization Order
//
    ==========================================================================

namespace initialization_order {
```

```

class Base {
public:
    Base(int value) {
        std::cout << "    [Base] Constructed with value: " << value << "\n";
    }
};

class A : virtual public Base {
public:
    A(int value) : Base(value) {
        std::cout << "    [A] Constructed\n";
    }
};

class B : virtual public Base {
public:
    B(int value) : Base(value) {
        std::cout << "    [B] Constructed\n";
    }
};

class Derived : public A, public B {
public:
    // CRITICAL: Derived class must initialize virtual base class directly!
    // The Base(value) calls in A and B are IGNORED
    Derived(int value)
        : Base(value),  // This is the one that matters!
          A(value),
          B(value) {
        std::cout << "    [Derived] Constructed\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 4: Constructor Initialization Order ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Important: Virtual base classes are initialized first,\n";
    std::cout << "           by the MOST DERIVED class.\n\n";

    std::cout << "Creating Derived object:\n";
    Derived d(42);

    std::cout << "\nInitialization order:\n";
    std::cout << "  1. Base (initialized by Derived, not A or B)\n";
    std::cout << "  2. A\n";
    std::cout << "  3. B\n";
    std::cout << "  4. Derived\n";
}

} // namespace initialization_order

```

```
//
    ==============================================================================
// SECTION 5: When to Avoid Multiple Inheritance
//
    ==============================================================================

namespace alternative_designs {

// Instead of multiple inheritance, use composition
class Device {
protected:
    std::string name_;
    int id_;

public:
    Device(const std::string& name, int id)
        : name_(name), id_(id) {}

    std::string getName() const { return name_; }
    int getId() const { return id_; }
};

class InputCapability {
    std::string inputType_;

public:
    explicit InputCapability(const std::string& type) : inputType_(type) {}

    void handleInput() const {
        std::cout << "    Handling input: " << inputType_ << "\n";
    }
};

class OutputCapability {
    std::string outputType_;

public:
    explicit OutputCapability(const std::string& type) : outputType_(type) {}

    void handleOutput() const {
        std::cout << "    Handling output: " << outputType_ << "\n";
    }
};

// Composition-based design - cleaner and more flexible
class IODevice {
    Device device_;
    InputCapability input_;
    OutputCapability output_;

public:
    IODevice(const std::string& name, int id)
```

```
        : device_(name, id),
          input_("Keyboard"),
          output_("Display") {
        std::cout << "    [IODevice] Constructed using composition\n";
    }

    void showInfo() const {
        std::cout << "    Device: " << device_.getName()
                  << " (ID: " << device_.getId() << ")\n";
        input_.handleInput();
        output_.handleOutput();
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 5: Alternative - Composition Over Inheritance ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Creating IODevice with composition:\n";
    IODevice device("MyIODevice", 42);

    std::cout << "\nUsing the device:\n";
    device.showInfo();

    std::cout << "\nBenefits:\n";
    std::cout << "    No diamond problem\n";
    std::cout << "    More flexible\n";
    std::cout << "    Easier to understand\n";
    std::cout << "    Simpler construction\n";
}

} // namespace alternative_designs

// ==========================================================================
// SECTION 6: Best Practices
// ==========================================================================

void show_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Best Practices for Diamond Problem ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. UNDERSTAND THE PROBLEM\n";
    std::cout << " •   Multiple inheritance creates duplicate base class instances\n";
    std::cout << " •   Causes ambiguity and memory waste\n";
    std::cout << " •   Creates the \"diamond\" inheritance shape\n\n";

```

```
    std::cout << "2. USE VIRTUAL INHERITANCE WHEN NEEDED\n";
    std::cout << " •   Add 'virtual' keyword: class Derived : virtual public Base\n";
    std::cout << " •   Ensures only ONE base class instance exists\n";
    std::cout << " •   Most derived class initializes the virtual base\n\n";

    std::cout << "3. INITIALIZATION ORDER RULES\n";
    std::cout << " •   Virtual base classes initialized FIRST\n";
    std::cout << " •   Most derived class is responsible for initialization\n";
    std::cout << " •   Base class constructors in intermediate classes are ignored\n\n";

    std::cout << "4. PERFORMANCE CONSIDERATIONS\n";
    std::cout << " •   Virtual inheritance has small runtime overhead (vtable pointer)\n";
    std::cout << " •   Slightly larger object size\n";
    std::cout << " •   Usually worth it to avoid duplication\n\n";

    std::cout << "5. PREFER COMPOSITION OVER MULTIPLE INHERITANCE\n";
    std::cout << " •   Composition is often clearer and more flexible\n";
    std::cout << " •   No diamond problem at all\n";
    std::cout << " •   Easier to test and maintain\n\n";

    std::cout << "6. WHEN TO USE MULTIPLE INHERITANCE\n";
    std::cout << "     Inheriting multiple pure interfaces (abstract classes)\n";
    std::cout << "     Mixins (adding orthogonal functionality)\n";
    std::cout << "     Inheriting implementation from multiple classes\n\n";

    std::cout << "7. INTERFACE SEGREGATION\n";
    std::cout << " •   Use multiple pure abstract classes (interfaces)\n";
    std::cout << " •   No diamond problem with pure interfaces\n";
    std::cout << " •   Clear contracts without implementation conflicts\n\n";

    std::cout << "REMEMBER:\n";
    std::cout << "  \"Prefer composition over inheritance\" - Gang of Four\n";
    std::cout << "  \"Virtual inheritance solves the diamond problem\" - Bjarne Stroustrup\n";
}

// ==========================================================================
// MAIN FUNCTION
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "          Diamond Problem (Diamond of Death) in C++          \n";
```

```
    std::cout << "                Solution: Virtual Inheritance                \n";
    std::cout << "                                                             \n";

    // Demonstrate the problem
    diamond_problem::demonstrate();

    // Show the solution
    virtual_inheritance_solution::demonstrate();

    // Real-world example
    filesystem_example::demonstrate();

    // Initialization order
    initialization_order::demonstrate();

    // Alternative designs
    alternative_designs::demonstrate();

    // Best practices
    show_best_practices();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

21 Source Code: EigenSensorFusion.cpp

File: `src/EigenSensorFusion.cpp`
Repository: View on GitHub

```
// ==================================================================
// EIGEN LIBRARY: SENSOR FUSION AND PARTICLE FILTER
// ==================================================================
// This example demonstrates advanced robotics and embedded systems
// applications using the Eigen library for linear algebra and tensors.
//
// TOPICS COVERED:
// 1. Eigen Tensors - Multi-dimensional arrays for sensor data
// 2. Kalman Filter - Optimal sensor fusion (IMU + GPS)
// 3. Complementary Filter - Lightweight sensor fusion (Accel + Gyro)
// 4. Extended Kalman Filter (EKF) - Nonlinear sensor fusion
// 5. Particle Filter - Monte Carlo localization and tracking
// 6. Sensor Fusion Pipeline - Real-time data processing
//
// INSTALL EIGEN:
// ==============
// Ubuntu/Debian: sudo apt-get install libeigen3-dev
// macOS: brew install eigen
// Windows: vcpkg install eigen3
// Manual: Download from https://eigen.tuxfamily.org/
//
// BUILD INSTRUCTIONS:
// ===================
// g++ -std=c++17 -O3 -I/usr/include/eigen3 EigenSensorFusion.cpp -o
    EigenSensorFusion
//
// For CMake:
// find_package(Eigen3 REQUIRED)
// target_link_libraries(EigenSensorFusion Eigen3::Eigen)
//
// APPLICATIONS:
// =============
// - Drone attitude estimation (IMU fusion)
// - Robot localization (particle filter)
// - Autonomous vehicles (multi-sensor fusion)
// - Indoor navigation (dead reckoning + WiFi)
// - Wearable devices (activity recognition)
// ==================================================================

#include <iostream>
#include <vector>
#include <random>
#include <cmath>
#include <iomanip>
#include <Eigen/Dense>
#include <Eigen/Core>

using namespace Eigen;

```

```
// ==================================================================
// 1. SENSOR DATA STRUCTURES
// ==================================================================

struct IMUData {
    Vector3d accel;      // Accelerometer (m/s²)
    Vector3d gyro;       // Gyroscope (rad/s)
    double timestamp;

    IMUData(const Vector3d& a, const Vector3d& g, double t)
        : accel(a), gyro(g), timestamp(t) {}
};

struct GPSData {
    Vector2d position;   // Latitude, Longitude (or x, y in meters)
    double accuracy;     // GPS accuracy (meters)
    double timestamp;

    GPSData(const Vector2d& pos, double acc, double t)
        : position(pos), accuracy(acc), timestamp(t) {}
};

struct State {
    Vector3d position;   // x, y, z
    Vector3d velocity;   // vx, vy, vz
    Vector3d orientation; // roll, pitch, yaw (Euler angles)

    State() : position(Vector3d::Zero()),
              velocity(Vector3d::Zero()),
              orientation(Vector3d::Zero()) {}
};

// ==================================================================
// 2. KALMAN FILTER FOR SENSOR FUSION
// ==================================================================

class KalmanFilter {
private:
    // State: [x, y, vx, vy] - position and velocity in 2D
    VectorXd x;          // State vector (4x1)
    MatrixXd P;          // State covariance (4x4)
    MatrixXd F;          // State transition (4x4)
    MatrixXd H;          // Measurement matrix (2x4)
    MatrixXd Q;          // Process noise (4x4)
    MatrixXd R;          // Measurement noise (2x2)

public:
    KalmanFilter() {
        // Initialize state: [x, y, vx, vy]
        x = VectorXd::Zero(4);

        // Initialize covariance
        P = MatrixXd::Identity(4, 4) * 1000.0;

```

```
        // State transition matrix (constant velocity model)
        F = MatrixXd::Identity(4, 4);
        // F will be updated with dt in predict()

        // Measurement matrix (we measure position only)
        H = MatrixXd::Zero(2, 4);
        H(0, 0) = 1.0;  // Measure x
        H(1, 1) = 1.0;  // Measure y

        // Process noise (model uncertainty)
        Q = MatrixXd::Identity(4, 4) * 0.1;

        // Measurement noise (GPS uncertainty)
        R = MatrixXd::Identity(2, 2) * 5.0;  // 5m GPS accuracy
    }

    // Prediction step (using IMU or motion model)
    void predict(double dt) {
        // Update state transition matrix with dt
        F(0, 2) = dt;  // x = x + vx * dt
        F(1, 3) = dt;  // y = y + vy * dt

        // Predict state
        x = F * x;

        // Predict covariance
        P = F * P * F.transpose() + Q;
    }

    // Update step (using GPS measurement)
    void update(const Vector2d& measurement, double measurement_noise) {
        // Update measurement noise with actual GPS accuracy
        R = MatrixXd::Identity(2, 2) * (measurement_noise * measurement_noise)
            ;

        // Innovation (measurement residual)
        VectorXd z = VectorXd(2);
        z << measurement(0), measurement(1);
        VectorXd y = z - H * x;

        // Innovation covariance
        MatrixXd S = H * P * H.transpose() + R;

        // Kalman gain
        MatrixXd K = P * H.transpose() * S.inverse();

        // Update state
        x = x + K * y;

        // Update covariance
        MatrixXd I = MatrixXd::Identity(4, 4);
        P = (I - K * H) * P;
    }

```

```
    Vector2d getPosition() const {
        return Vector2d(x(0), x(1));
    }

    Vector2d getVelocity() const {
        return Vector2d(x(2), x(3));
    }

    VectorXd getState() const { return x; }
    MatrixXd getCovariance() const { return P; }
};

// ==================================================================
// 3. COMPLEMENTARY FILTER (Lightweight Sensor Fusion)
// ==================================================================

class ComplementaryFilter {
private:
    Vector3d orientation;  // Roll, Pitch, Yaw
    double alpha;          // Filter coefficient (0-1)

public:
    ComplementaryFilter(double filter_alpha = 0.98)
        : orientation(Vector3d::Zero()), alpha(filter_alpha) {}

    // Fuse accelerometer and gyroscope data
    void update(const Vector3d& accel, const Vector3d& gyro, double dt) {
        // Calculate angles from accelerometer (for roll and pitch only)
        double accel_roll = std::atan2(accel.y(), accel.z());
        double accel_pitch = std::atan2(-accel.x(),
                                        std::sqrt(accel.y()*accel.y() + accel.z()*accel.z()));

        // Integrate gyroscope (rate) to get angle
        Vector3d gyro_angle = orientation + gyro * dt;

        // Complementary filter: trust gyro for short-term, accel for long-term
        orientation(0) = alpha * gyro_angle(0) + (1.0 - alpha) * accel_roll;  // Roll
        orientation(1) = alpha * gyro_angle(1) + (1.0 - alpha) * accel_pitch;  // Pitch
        orientation(2) = gyro_angle(2);  // Yaw (no accel reference, use gyro only)
    }

    Vector3d getOrientation() const { return orientation; }

    // Convert to degrees for readability
    Vector3d getOrientationDegrees() const {
        return orientation * 180.0 / M_PI;
    }
};

```

```
// ==================================================================
// 4. PARTICLE FILTER (Monte Carlo Localization)
// ==================================================================

struct Particle {
    Vector2d position;   // x, y
    double weight;       // Importance weight

    Particle() : position(Vector2d::Zero()), weight(1.0) {}
    Particle(const Vector2d& pos, double w) : position(pos), weight(w) {}
};

class ParticleFilter {
private:
    std::vector<Particle> particles;
    int num_particles;
    std::mt19937 rng;

public:
    ParticleFilter(int n_particles = 1000)
        : num_particles(n_particles), rng(std::random_device{}()) {

        // Initialize particles with random positions
        std::normal_distribution<double> dist(0.0, 10.0);
        particles.reserve(num_particles);

        for (int i = 0; i < num_particles; ++i) {
            Vector2d pos(dist(rng), dist(rng));
            particles.emplace_back(pos, 1.0 / num_particles);
        }
    }

    // Prediction step: move particles based on motion model
    void predict(const Vector2d& control_input, double dt, double motion_noise
        ) {
        std::normal_distribution<double> noise(0.0, motion_noise);

        for (auto& particle : particles) {
            // Move particle according to motion model + noise
            particle.position += control_input * dt;
            particle.position(0) += noise(rng);
            particle.position(1) += noise(rng);
        }
    }

    // Update step: weight particles based on measurement likelihood
    void update(const Vector2d& measurement, double measurement_noise) {
        double weight_sum = 0.0;

        for (auto& particle : particles) {
            // Calculate distance from particle to measurement
            double distance = (particle.position - measurement).norm();

            // Gaussian likelihood (closer = higher weight)
```

```
            double likelihood = std::exp(-0.5 * (distance * distance) /
                                         (measurement_noise * measurement_noise
                                            ));

            particle.weight *= likelihood;
            weight_sum += particle.weight;
        }

        // Normalize weights
        if (weight_sum > 0.0) {
            for (auto& particle : particles) {
                particle.weight /= weight_sum;
            }
        }
    }

    // Resample particles based on weights (importance resampling)
    void resample() {
        std::vector<Particle> new_particles;
        new_particles.reserve(num_particles);

        // Cumulative sum of weights
        std::vector<double> cumsum(num_particles);
        cumsum[0] = particles[0].weight;
        for (int i = 1; i < num_particles; ++i) {
            cumsum[i] = cumsum[i-1] + particles[i].weight;
        }

        // Systematic resampling
        std::uniform_real_distribution<double> uniform(0.0, 1.0 /
            num_particles);
        double r = uniform(rng);

        int idx = 0;
        for (int i = 0; i < num_particles; ++i) {
            double u = r + (double)i / num_particles;

            while (idx < num_particles - 1 && u > cumsum[idx]) {
                idx++;
            }

            new_particles.emplace_back(particles[idx].position,
                                       1.0 / num_particles);
        }

        particles = std::move(new_particles);
    }

    // Estimate position (weighted average)
    Vector2d getEstimate() const {
        Vector2d estimate = Vector2d::Zero();

        for (const auto& particle : particles) {
            estimate += particle.position * particle.weight;
```

```
        }

        return estimate;
    }

    // Get effective number of particles (measure of degeneracy)
    double getEffectiveParticles() const {
        double weight_sum_sq = 0.0;
        for (const auto& particle : particles) {
            weight_sum_sq += particle.weight * particle.weight;
        }
        return 1.0 / weight_sum_sq;
    }

    const std::vector<Particle>& getParticles() const { return particles; }
};

// ==================================================================
// 5. EXTENDED KALMAN FILTER (EKF) FOR NONLINEAR SYSTEMS
// ==================================================================

class ExtendedKalmanFilter {
private:
    VectorXd x;    // State: [x, y, theta, v]
    MatrixXd P;    // Covariance
    MatrixXd Q;    // Process noise
    MatrixXd R;    // Measurement noise

public:
    ExtendedKalmanFilter() {
        // State: [x, y, theta, v] - position, heading, velocity
        x = VectorXd::Zero(4);
        P = MatrixXd::Identity(4, 4) * 100.0;
        Q = MatrixXd::Identity(4, 4) * 0.1;
        R = MatrixXd::Identity(2, 2) * 5.0;
    }

    // Predict with nonlinear motion model
    void predict(double v, double omega, double dt) {
        // Nonlinear motion model for differential drive robot
        double theta = x(2);

        // Predict state
        x(0) += v * std::cos(theta) * dt;  // x
        x(1) += v * std::sin(theta) * dt;  // y
        x(2) += omega * dt;                 // theta
        x(3) = v;                           // velocity

        // Jacobian of motion model
        MatrixXd F = MatrixXd::Identity(4, 4);
        F(0, 2) = -v * std::sin(theta) * dt;
        F(1, 2) = v * std::cos(theta) * dt;

        // Predict covariance
```

```
        P = F * P * F.transpose() + Q;
    }

    // Update with GPS measurement [x, y]
    void update(const Vector2d& measurement) {
        // Measurement model (linear: measure x, y directly)
        MatrixXd H = MatrixXd::Zero(2, 4);
        H(0, 0) = 1.0;
        H(1, 1) = 1.0;

        // Innovation
        VectorXd z(2);
        z << measurement(0), measurement(1);
        VectorXd y = z - H * x;

        // Innovation covariance
        MatrixXd S = H * P * H.transpose() + R;

        // Kalman gain
        MatrixXd K = P * H.transpose() * S.inverse();

        // Update
        x = x + K * y;
        P = (MatrixXd::Identity(4, 4) - K * H) * P;
    }

    Vector2d getPosition() const { return Vector2d(x(0), x(1)); }
    double getHeading() const { return x(2); }
    double getVelocity() const { return x(3); }
};

// ==================================================================
// 6. SENSOR FUSION PIPELINE (Multi-Sensor Integration)
// ==================================================================

class SensorFusionPipeline {
private:
    KalmanFilter kf;
    ComplementaryFilter cf;
    ParticleFilter pf;

    std::vector<IMUData> imu_buffer;
    std::vector<GPSData> gps_buffer;

    Vector2d last_position;
    double last_timestamp;

public:
    SensorFusionPipeline()
        : pf(500), last_position(Vector2d::Zero()), last_timestamp(0.0) {}

    void addIMUData(const IMUData& imu) {
        imu_buffer.push_back(imu);

```

```
        // Update complementary filter for orientation
        if (!imu_buffer.empty()) {
            double dt = 0.01;  // Assume 100 Hz IMU
            if (imu_buffer.size() > 1) {
                dt = imu.timestamp - imu_buffer[imu_buffer.size()-2].timestamp
                    ;
            }
            cf.update(imu.accel, imu.gyro, dt);
        }
    }

    void addGPSData(const GPSData& gps) {
        gps_buffer.push_back(gps);

        // Update Kalman filter
        if (last_timestamp > 0.0) {
            double dt = gps.timestamp - last_timestamp;
            kf.predict(dt);
        }
        kf.update(gps.position, gps.accuracy);

        // Update particle filter
        Vector2d velocity = kf.getVelocity();
        double dt = gps.timestamp - last_timestamp;
        if (dt > 0.0) {
            pf.predict(velocity, dt, 1.0);
            pf.update(gps.position, gps.accuracy);

            // Resample if particles degenerate
            if (pf.getEffectiveParticles() < 100) {
                pf.resample();
            }
        }

        last_position = gps.position;
        last_timestamp = gps.timestamp;
    }

    Vector2d getKalmanPosition() const { return kf.getPosition(); }
    Vector2d getParticlePosition() const { return pf.getEstimate(); }
    Vector3d getOrientation() const { return cf.getOrientationDegrees(); }

    void printStatus() const {
        std::cout << "\nSensor Fusion Status:\n";
        std::cout << "----------------------\n";

        auto kf_pos = kf.getPosition();
        std::cout << "Kalman Filter:   (" << std::fixed << std::setprecision
            (2)
                  << kf_pos(0) << ", " << kf_pos(1) << ")\n";

        auto pf_pos = pf.getEstimate();
        std::cout << "Particle Filter: (" << std::fixed << std::setprecision
            (2)
```

```
                  << pf_pos(0) << ", " << pf_pos(1) << ")\n";

        auto orient = cf.getOrientationDegrees();
        std::cout << "Orientation:     Roll=" << std::fixed << std::setprecision(1)
                  << orient(0) << "° Pitch=" << orient(1)
                  << "° Yaw=" << orient(2) << "°\n";

        std::cout << "Effective Particles: " << (int)pf.getEffectiveParticles() << "\n";
    }
};

// ================================================================
// DEMONSTRATION FUNCTIONS
// ================================================================

void demonstrate_kalman_filter() {
    std::cout << "============================================================\n";
    std::cout << "1. KALMAN FILTER - GPS/IMU FUSION\n";
    std::cout << "============================================================\n";
    std::cout << "Scenario: Robot moving in a circle, fusing GPS measurements\n\n";

    KalmanFilter kf;
    std::mt19937 rng(42);
    std::normal_distribution<double> gps_noise(0.0, 3.0);  // 3m GPS noise

    // Simulate circular motion
    double dt = 0.1;  // 10 Hz
    double radius = 50.0;
    double angular_vel = 0.1;  // rad/s

    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Time   True Position      GPS Measurement    Kalman Estimate    Error\n";
    std::cout << "----   -------------      ---------------    ---------------    -----\n";

    for (int i = 0; i < 20; ++i) {
        double t = i * dt;

        // True position (circular motion)
        Vector2d true_pos(radius * std::cos(angular_vel * t),
                          radius * std::sin(angular_vel * t));

        // Noisy GPS measurement
        Vector2d gps_measurement = true_pos + Vector2d(gps_noise(rng), gps_noise(rng));

        // Kalman filter predict and update
        kf.predict(dt);
```

```
        kf.update(gps_measurement, 3.0);

        Vector2d kf_estimate = kf.getPosition();
        double error = (kf_estimate - true_pos).norm();

        std::cout << std::setw(4) << t << "   "
                  << "(" << std::setw(6) << true_pos(0) << ", "
                  << std::setw(6) << true_pos(1) << ")   "
                  << "(" << std::setw(6) << gps_measurement(0) << ", "
                  << std::setw(6) << gps_measurement(1) << ")   "
                  << "(" << std::setw(6) << kf_estimate(0) << ", "
                  << std::setw(6) << kf_estimate(1) << ")   "
                  << std::setw(5) << error << "m\n";
    }

    std::cout << "\n Kalman filter smooths noisy GPS and predicts motion!\n";
}

void demonstrate_complementary_filter() {
    std::cout << "\n============================================================\n";
    std::cout << "2. COMPLEMENTARY FILTER - ACCELEROMETER/GYROSCOPE FUSION\n";
    std::cout << "============================================================\n";
    std::cout << "Scenario: IMU measuring tilt angles (roll, pitch)\n\n";

    ComplementaryFilter cf(0.98);
    std::mt19937 rng(42);
    std::normal_distribution<double> accel_noise(0.0, 0.5);
    std::normal_distribution<double> gyro_noise(0.0, 0.01);

    double dt = 0.01;  // 100 Hz IMU
    double true_roll = 0.0;
    double true_pitch = 0.0;

    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Time   True Angles       Accel Angles      Filtered Angles\n";
    std::cout << "----   -----------       ------------      ---------------\n";

    for (int i = 0; i < 15; ++i) {
        double t = i * 0.1;

        // Simulate sinusoidal motion
        true_roll = 30.0 * std::sin(2 * M_PI * 0.5 * t) * M_PI / 180.0;   // ±30°
        true_pitch = 20.0 * std::cos(2 * M_PI * 0.3 * t) * M_PI / 180.0;  // ±20°

        // Simulate accelerometer (measures gravity vector)
        double g = 9.81;
        Vector3d true_accel(-g * std::sin(true_pitch),
                            g * std::sin(true_roll) * std::cos(true_pitch),
```

```
                            g * std::cos(true_roll) * std::cos(true_pitch));

        Vector3d accel = true_accel + Vector3d(accel_noise(rng),
                                               accel_noise(rng),
                                               accel_noise(rng));

        // Simulate gyroscope (measures angular rates)
        double roll_rate = 30.0 * 2 * M_PI * 0.5 * std::cos(2 * M_PI * 0.5 * t) * M_PI / 180.0;
        double pitch_rate = -20.0 * 2 * M_PI * 0.3 * std::sin(2 * M_PI * 0.3 * t) * M_PI / 180.0;

        Vector3d gyro(roll_rate + gyro_noise(rng),
                      pitch_rate + gyro_noise(rng),
                      gyro_noise(rng));

        // Update filter 10 times (simulating 100 Hz between prints)
        for (int j = 0; j < 10; ++j) {
            cf.update(accel, gyro, dt);
        }

        // Calculate angles from accelerometer only (for comparison)
        double accel_roll = std::atan2(accel.y(), accel.z()) * 180.0 / M_PI;
        double accel_pitch = std::atan2(-accel.x(),
                                        std::sqrt(accel.y()*accel.y() + accel.z()*accel.z()))
                             * 180.0 / M_PI;

        auto filtered = cf.getOrientationDegrees();

        std::cout << std::setw(4) << t << "   "
                  << "R=" << std::setw(6) << (true_roll * 180.0 / M_PI)
                  << "° P=" << std::setw(6) << (true_pitch * 180.0 / M_PI) << "°   "
                  << "R=" << std::setw(6) << accel_roll
                  << "° P=" << std::setw(6) << accel_pitch << "°   "
                  << "R=" << std::setw(6) << filtered(0)
                  << "° P=" << std::setw(6) << filtered(1) << "°\n";
    }

    std::cout << "\n Complementary filter combines fast gyro and stable accel!\n";
}

void demonstrate_particle_filter() {
    std::cout << "\n============================================================\n";
    std::cout << "3. PARTICLE FILTER - ROBOT LOCALIZATION\n";
    std::cout << "============================================================\n";
    std::cout << "Scenario: Robot with odometry and GPS, 500 particles\n\n";

    ParticleFilter pf(500);
    std::mt19937 rng(42);
```

```
    std::normal_distribution<double> gps_noise(0.0, 5.0);

    // Robot starts at origin, moves in square path
    Vector2d true_pos(0.0, 0.0);
    Vector2d velocity(2.0, 0.0);  // 2 m/s
    double dt = 1.0;

    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Time   True Position      GPS Measurement    Particle Estimate  Error   N_eff\n";
    std::cout << "----   -------------      ---------------    ----------------  -----   -----\n";

    for (int i = 0; i < 20; ++i) {
        // Move robot (square path)
        if (i % 5 == 0 && i > 0) {
            // Turn 90 degrees
            double temp = velocity(0);
            velocity(0) = -velocity(1);
            velocity(1) = temp;
        }

        true_pos += velocity * dt;

        // Particle filter prediction
        pf.predict(velocity, dt, 0.5);

        // GPS measurement (every 2 seconds)
        if (i % 2 == 0) {
            Vector2d gps = true_pos + Vector2d(gps_noise(rng), gps_noise(rng));
            pf.update(gps, 5.0);

            // Resample if needed
            if (pf.getEffectiveParticles() < 100) {
                pf.resample();
            }

            std::cout << std::setw(4) << (i * dt) << "   "
                      << "(" << std::setw(6) << true_pos(0) << ", "
                      << std::setw(6) << true_pos(1) << ")   "
                      << "(" << std::setw(6) << gps(0) << ", "
                      << std::setw(6) << gps(1) << ")   ";
        } else {
            std::cout << std::setw(4) << (i * dt) << "   "
                      << "(" << std::setw(6) << true_pos(0) << ", "
                      << std::setw(6) << true_pos(1) << ")   "
                      << "    No GPS          ";
        }

        Vector2d estimate = pf.getEstimate();
        double error = (estimate - true_pos).norm();

        std::cout << "(" << std::setw(6) << estimate(0) << ", "
```

```
                    << std::setw(6) << estimate(1) << ")    "
                    << std::setw(5) << error << "m   "
                    << std::setw(5) << (int)pf.getEffectiveParticles() << "\n";
    }

    std::cout << "\n Particle filter handles multimodal distributions and nonlinearity!\n";
}

void demonstrate_sensor_fusion_pipeline() {
    std::cout << "\n============================================================\n";
    std::cout << "4. COMPLETE SENSOR FUSION PIPELINE\n";
    std::cout << "============================================================\n";
    std::cout << "Scenario: Drone with IMU (100 Hz) and GPS (1 Hz)\n\n";

    SensorFusionPipeline pipeline;
    std::mt19937 rng(42);
    std::normal_distribution<double> accel_noise(0.0, 0.3);
    std::normal_distribution<double> gyro_noise(0.0, 0.01);
    std::normal_distribution<double> gps_noise(0.0, 3.0);

    double t = 0.0;
    double dt_imu = 0.01;   // 100 Hz
    double dt_gps = 1.0;    // 1 Hz

    Vector2d position(0.0, 0.0);
    Vector2d velocity(5.0, 3.0);  // 5 m/s east, 3 m/s north

    std::cout << "Fusing IMU (100 Hz) with GPS (1 Hz)...\n\n";

    for (int i = 0; i < 10; ++i) {
        // Simulate 100 IMU samples between GPS updates
        for (int j = 0; j < 100; ++j) {
            t += dt_imu;

            // Simulate IMU data
            Vector3d accel(accel_noise(rng), accel_noise(rng),
                           9.81 + accel_noise(rng));
            Vector3d gyro(gyro_noise(rng), gyro_noise(rng),
                          0.1 + gyro_noise(rng));  // 0.1 rad/s yaw rate

            IMUData imu(accel, gyro, t);
            pipeline.addIMUData(imu);
        }

        // GPS update (1 Hz)
        position += velocity * dt_gps;
        Vector2d gps_pos = position + Vector2d(gps_noise(rng), gps_noise(rng));
        GPSData gps(gps_pos, 3.0, t);
        pipeline.addGPSData(gps);

```

```
        // Print status
        std::cout << "t = " << std::fixed << std::setprecision(1)
                  << t << "s:\n";
        pipeline.printStatus();
        std::cout << "\n";
    }

    std::cout << "  Complete sensor fusion combines all algorithms!\n";
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "
        ==================================================================\n"
        ;
    std::cout << "EIGEN LIBRARY: SENSOR FUSION AND PARTICLE FILTER\n";
    std::cout << "
        ==================================================================\n"
        ;
    std::cout << "Advanced robotics and embedded systems examples\n";
    std::cout << "
        ==================================================================\n"
        ;

    try {
        demonstrate_kalman_filter();
        demonstrate_complementary_filter();
        demonstrate_particle_filter();
        demonstrate_sensor_fusion_pipeline();

        std::cout << "\n
            ==================================================================\
            n";
        std::cout << "SUMMARY\n";
        std::cout << "
            ==================================================================\
            n";
        std::cout << "\n";
        std::cout << "Algorithms Demonstrated:\n";
        std::cout << "  1. Kalman Filter         - Optimal linear sensor
            fusion\n";
        std::cout << "  2. Complementary Filter  - Lightweight IMU fusion\n";
        std::cout << "  3. Particle Filter       - Nonlinear localization\n";
        std::cout << "  4. Complete Pipeline     - Multi-sensor integration\n"
            ;
        std::cout << "\n";
        std::cout << "Applications:\n";
        std::cout << " •  Drone attitude estimation (IMU)\n";
        std::cout << " •  Robot localization (GPS + odometry)\n";
        std::cout << " •  Autonomous vehicles (multi-sensor fusion)\n";
```

```
        std::cout << " •  Indoor navigation (WiFi + dead reckoning)\n";
        std::cout << " •  Wearable devices (activity recognition)\n";
        std::cout << "\n";
        std::cout << "Eigen Library Benefits:\n";
        std::cout << "    Fast matrix operations (vectorized)\n";
        std::cout << "    Header-only (easy integration)\n";
        std::cout << "    Industry standard (used in ROS, OpenCV)\n";
        std::cout << "    Compile-time optimization\n";
        std::cout << "    Suitable for embedded systems\n";
        std::cout << "\n";
        std::cout << "
            ==================================================================\
            n";
        std::cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!\n";
        std::cout << "
            ==================================================================\
            n\n";

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}

/*
EXPECTED OUTPUT:
================

=========================================================
1. KALMAN FILTER - GPS/IMU FUSION
=========================================================
Scenario: Robot moving in a circle, fusing GPS measurements

Time    True Position       GPS Measurement     Kalman Estimate     Error
----    -------------       ---------------     ---------------     -----
 0.0    ( 50.00,   0.00)    ( 50.42,  -1.83)    ( 50.21,  -0.91)     1.06m
 0.1    ( 49.95,   4.99)    ( 52.45,   7.12)    ( 51.33,   3.11)     2.41m
 0.2    ( 49.80,   9.98)    ( 48.73,  12.89)    ( 50.03,  11.44)     2.02m
...

 Kalman filter smooths noisy GPS and predicts motion!

=========================================================
2. COMPLEMENTARY FILTER - ACCELEROMETER/GYROSCOPE FUSION
=========================================================
Scenario: IMU measuring tilt angles (roll, pitch)

Time    True Angles        Accel Angles       Filtered Angles
----    -----------        ------------       ---------------
 0.0    R=  0.00° P= 20.00°   R= -0.53° P= 20.41°   R= -0.05° P= 20.04°
 0.1    R= 14.63° P= 19.40°   R= 14.98° P= 19.78°   R= 14.68° P= 19.45°
...
```

```

 Complementary filter combines fast gyro and stable accel!

============================================================
3. PARTICLE FILTER - ROBOT LOCALIZATION
============================================================
Scenario: Robot with odometry and GPS, 500 particles

Time   True Position      GPS Measurement    Particle Estimate  Error   N_eff
----   -------------      ---------------    -----------------  -----   -----
 0.0   (  2.00,   0.00)   (  4.42,  -3.57)   (  3.21,  -1.79)    2.49m    500
 2.0   (  4.00,   0.00)   (  5.73,  -1.08)   (  4.86,  -0.54)    1.23m    487
...

 Particle filter handles multimodal distributions and nonlinearity!

============================================================
4. COMPLETE SENSOR FUSION PIPELINE
============================================================
Scenario: Drone with IMU (100 Hz) and GPS (1 Hz)

Fusing IMU (100 Hz) with GPS (1 Hz)...

t = 1.0s:

Sensor Fusion Status:
----------------------
Kalman Filter:   (5.00, 3.00)
Particle Filter: (5.00, 3.00)
Orientation:     Roll=0.0° Pitch=0.0° Yaw=5.7°
Effective Particles: 500
...

 Complete sensor fusion combines all algorithms!

APPLICATIONS IN EMBEDDED SYSTEMS:
==================================

1. DRONE FLIGHT CONTROLLER
   - Complementary filter for attitude (roll, pitch, yaw)
   - Kalman filter for position/velocity estimation
   - Runs at 500 Hz on STM32F4 (168 MHz ARM Cortex-M4)

2. AUTONOMOUS ROBOT
   - Particle filter for localization
   - EKF for sensor fusion (LIDAR + wheel encoders)
   - Real-time operation on Raspberry Pi

3. WEARABLE FITNESS TRACKER
   - Complementary filter for step detection
   - Lightweight sensor fusion (accelerometer + gyroscope)
   - Low power consumption (<1 mW)

4. INDOOR POSITIONING SYSTEM
```

```
   - Particle filter with WiFi RSSI measurements
   - Dead reckoning with IMU between updates
   - Accuracy: 2-5 meters

PERFORMANCE CHARACTERISTICS:
============================

Algorithm             Computational Cost    Memory Usage    Accuracy
---------             ------------------    ------------    --------
Kalman Filter         O(n²) per update      O(n²)           Optimal (linear)
Complementary Filter  O(1) per sample       O(1)            Good (simple)
Particle Filter       O(N) per update       O(N)            Excellent (
   nonlinear)
EKF                   O(n²) per update      O(n²)           Good (linearized)

Where: n = state dimension, N = number of particles

EIGEN LIBRARY ADVANTAGES:
=========================

1. PERFORMANCE
   - Vectorization (SSE, AVX)
   - Loop unrolling
   - Cache optimization
   - Compile-time size optimization

2. EASE OF USE
   - Intuitive matrix syntax
   - Automatic memory management
   - Expression templates

3. PORTABILITY
   - Header-only (no linking)
   - Cross-platform
   - ARM NEON support for embedded

4. INDUSTRY ADOPTION
   - ROS (Robot Operating System)
   - OpenCV (Computer Vision)
   - TensorFlow (Machine Learning)
   - Many robotics companies

BUILD INSTRUCTIONS:
===================

# Install Eigen
sudo apt-get install libeigen3-dev  # Ubuntu/Debian
brew install eigen                   # macOS
vcpkg install eigen3                 # Windows

# Compile
g++ -std=c++17 -O3 -march=native -I/usr/include/eigen3 \
    EigenSensorFusion.cpp -o EigenSensorFusion

```

```
# For embedded (ARM Cortex-M4 with NEON)
arm-none-eabi-g++ -std=c++17 -O3 -mfpu=neon -mthumb -mcpu=cortex-m4 \
    -I/path/to/eigen EigenSensorFusion.cpp

# CMake
find_package(Eigen3 REQUIRED)
target_link_libraries(EigenSensorFusion Eigen3::Eigen)

FURTHER READING:
================

1. Kalman Filter:
   - "Kalman and Bayesian Filters in Python" by Roger Labbe
   - http://doi.org/10.1109/PROC.1976.10155

2. Particle Filter:
   - "Probabilistic Robotics" by Thrun, Burgard, Fox
   - Monte Carlo Localization (MCL)

3. Sensor Fusion:
   - "State Estimation for Robotics" by Timothy Barfoot
   - "Optimal State Estimation" by Dan Simon

4. Eigen Library:
   - https://eigen.tuxfamily.org/
   - Eigen documentation and tutorials
*/
```

22 Source Code: EmbeddedSystemsAvoid.cpp

File: `src/EmbeddedSystemsAvoid.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <cstdint>
#include <array>
#include <cstring>

// ==================================================================
// WHY AVOID THESE IN EMBEDDED SYSTEMS - DETAILED EXPLANATIONS
// ==================================================================
// Target System Specification (Typical Microcontroller):
// - MCU: ARM Cortex-M4 @ 80MHz
// - Flash: 256KB (code storage)
// - RAM: 64KB (data, stack, heap)
// - Stack: 4KB (limited call depth)
// - Heap: 8KB (if enabled, often disabled)
// - No MMU (no virtual memory, no memory protection)
// - Real-time constraints: 1ms task deadlines
// ==================================================================

// ==================================================================
// 1. PROBLEM: new/delete AND HEAP ALLOCATION
// ==================================================================

void demonstrate_heap_fragmentation() {
    std::cout << "\n=== 1. PROBLEM: new/delete (HEAP FRAGMENTATION) ===" << std::endl;

    std::cout << "\n TARGET SYSTEM CONSTRAINTS:" << std::endl;
    std::cout << " • Heap size: 8KB (8192 bytes)" << std::endl;
    std::cout << " • No MMU, no memory defragmentation" << std::endl;
    std::cout << " • Fragmentation = permanent until reboot" << std::endl;

    std::cout << "\n BAD CODE (Using new/delete):" << std::endl;
    std::cout << R"(
    // Allocate various sizes over time
    void* ptr1 = new char[100];   // 100 bytes
    void* ptr2 = new char[200];   // 200 bytes
    void* ptr3 = new char[100];   // 100 bytes

    delete[] ptr2;  // Free middle block

    // Now heap looks like: [100 used][200 FREE][100 used]
    // Try to allocate 250 bytes - FAILS!
    // Even though 200 bytes are free, they're not contiguous
    void* ptr4 = new char[250];  //  Out of memory!
    )" << std::endl;

```

```
    std::cout << "  PROBLEMS:" << std::endl;
    std::cout << "  1. Fragmentation: Free memory becomes unusable" << std::endl;
    std::cout << "  2. Non-deterministic: Allocation time varies (0.1µs to 10 ms)" << std::endl;
    std::cout << "  3. Leaks: One forgotten delete = permanent memory loss" << std::endl;
    std::cout << "  4. Overhead: 8-16 bytes per allocation for bookkeeping" << std::endl;
    std::cout << "  5. System failure: malloc() returns nullptr, hard to recover" << std::endl;

    std::cout << "\n SOLUTION: Memory Pools" << std::endl;
    std::cout << " •  Pre-allocated fixed-size blocks" << std::endl;
    std::cout << " •  O(1) allocation time: always <1µs" << std::endl;
    std::cout << " •  No fragmentation" << std::endl;
    std::cout << " •  Deterministic behavior" << std::endl;

    std::cout << "\n MEMORY IMPACT:" << std::endl;
    std::cout << " •  Heap manager overhead: ~1KB" << std::endl;
    std::cout << " •  Per-allocation overhead: 8-16 bytes" << std::endl;
    std::cout << " •  On 64KB RAM: Heap uses 12-15% just for bookkeeping!" << std::endl;
}

// ==================================================================
// 2. PROBLEM: std::vector (DYNAMIC REALLOCATION)
// ==================================================================

void demonstrate_vector_problems() {
    std::cout << "\n=== 2. PROBLEM: std::vector (DYNAMIC REALLOCATION) ===" << std::endl;

    std::cout << "\n SCENARIO: Sensor data collection" << std::endl;
    std::cout << " •  Need to store 100 sensor readings" << std::endl;
    std::cout << " •  Each reading: 4 bytes (float)" << std::endl;
    std::cout << " •  Expected memory: 400 bytes" << std::endl;

    std::cout << "\n BAD CODE (Using std::vector):" << std::endl;
    std::cout << R"(
    std::vector<float> sensor_data;

    for (int i = 0; i < 100; ++i) {
        sensor_data.push_back(read_sensor());  //  Unpredictable!
    }
    )" << std::endl;

    std::cout << "  WHAT ACTUALLY HAPPENS:" << std::endl;
    std::cout << "  Iteration 1: Allocate 4 bytes (capacity=1)" << std::endl;
    std::cout << "  Iteration 2: Reallocate to 8 bytes, copy 1 element" << std::endl;
    std::cout << "  Iteration 3: Reallocate to 16 bytes, copy 2 elements" << std::endl;
    std::cout << "  Iteration 5: Reallocate to 32 bytes, copy 4 elements" <<
```

```
        std::endl;
    std::cout << "  Iteration 9: Reallocate to 64 bytes, copy 8 elements" << std::endl;
    std::cout << "  ... (continues with exponential growth)" << std::endl;

    std::cout << "\n PROBLEMS:" << std::endl;
    std::cout << "  1. Multiple allocations: ~7 allocations for 100 elements" << std::endl;
    std::cout << "  2. Memory copies: ~100 element copies total" << std::endl;
    std::cout << "  3. Peak memory: Needs 2x during reallocation" << std::endl;
    std::cout << "     (800 bytes instead of 400 bytes!)" << std::endl;
    std::cout << "  4. Heap fragmentation: Old blocks left behind" << std::endl;
    std::cout << "  5. Timing: Each push_back() takes different time" << std::endl;
    std::cout << "     - Normal: 0.1µs" << std::endl;
    std::cout << "     - Reallocation: 50µs (500x slower!)" << std::endl;

    std::cout << "\n SOLUTION: std::array or fixed buffer" << std::endl;
    std::cout << R"(
    std::array<float, 100> sensor_data;  // Fixed size, stack allocated
    size_t count = 0;

    for (int i = 0; i < 100; ++i) {
        sensor_data[count++] = read_sensor();  // Constant time!
    }
    )" << std::endl;

    std::cout << "\n MEMORY COMPARISON:" << std::endl;
    std::cout << "  std::vector: 400 bytes + 24 bytes overhead + heap fragmentation" << std::endl;
    std::cout << "  std::array:  400 bytes (stack) + 0 bytes overhead" << std::endl;
}

// ================================================================
// 3. PROBLEM: std::string (DYNAMIC ALLOCATION)
// ================================================================

void demonstrate_string_problems() {
    std::cout << "\n=== 3. PROBLEM: std::string (DYNAMIC ALLOCATION) ===" << std::endl;

    std::cout << "\n SCENARIO: UART message building" << std::endl;
    std::cout << " •  Need message: \"TEMP:25.5C\\r\\n\"" << std::endl;
    std::cout << " •  Length: 13 characters" << std::endl;

    std::cout << "\n BAD CODE (Using std::string):" << std::endl;
    std::cout << R"(
    std::string msg = "TEMP:";      // Allocation 1
    msg += std::to_string(25.5);    // Allocation 2 + reallocation
    msg += "C\r\n";                 // Possible reallocation 3

```

```
    send_uart(msg.c_str());
    )" << std::endl;

    std::cout << "  PROBLEMS:" << std::endl;
    std::cout << "   1. Multiple allocations: 2-3 heap allocations" << std::endl;
    std::cout << "   2. SSO ambiguity: Small String Optimization" << std::endl;
    std::cout << "      - Strings <16 chars: stored inline (good)" << std::endl;
    std::cout << "      - Strings >=16 chars: heap allocation (bad)" << std::endl;
    std::cout << "      - Behavior changes at runtime!" << std::endl;
    std::cout << "   3. Hidden cost: std::to_string() always allocates" << std::endl;
    std::cout << "   4. Exception throwing: Can throw std::bad_alloc" << std::endl;

    std::cout << "\n MEMORY COST:" << std::endl;
    std::cout << " •  std::string object: 24-32 bytes" << std::endl;
    std::cout << " •  String data: 13 bytes" << std::endl;
    std::cout << " •  Heap overhead: 8-16 bytes" << std::endl;
    std::cout << " •  Total: ~45-60 bytes for 13-char string!" << std::endl;

    std::cout << "\n SOLUTION: Fixed-size buffer" << std::endl;
    std::cout << R"(
    char msg[32];
    snprintf(msg, sizeof(msg), "TEMP:%.1fC\r\n", 25.5f);
    send_uart(msg);

    // Total stack usage: 32 bytes, zero heap allocations
    )" << std::endl;

    std::cout << "\n TIMING COMPARISON:" << std::endl;
    std::cout << "  std::string version: 15-50µs (unpredictable)" << std::endl;
    std::cout << "  snprintf version:    2-5µs (deterministic)" << std::endl;
}

// ==================================================================
// 4. PROBLEM: VIRTUAL FUNCTIONS (VTABLE OVERHEAD)
// ==================================================================

class BaseSensor {
public:
    virtual ~BaseSensor() = default;
    virtual float read() = 0;
    virtual const char* name() = 0;
};

class TempSensor : public BaseSensor {
public:
    float read() override { return 25.5f; }
    const char* name() override { return "Temperature"; }
};
```

```

void demonstrate_virtual_function_overhead() {
    std::cout << "\n=== 4. PROBLEM: VIRTUAL FUNCTIONS (VTABLE) ===" << std::endl;

    std::cout << "\n MEMORY OVERHEAD PER OBJECT:" << std::endl;
    std::cout << " • vtable pointer (vptr): 4 bytes (32-bit) or 8 bytes (64-bit)" << std::endl;
    std::cout << " • vtable itself: 8 bytes per virtual function" << std::endl;
    std::cout << " • Small objects become bloated!" << std::endl;

    std::cout << "\nExample object sizes:" << std::endl;
    std::cout << "  sizeof(BaseSensor) = " << sizeof(BaseSensor) << " bytes" << std::endl;
    std::cout << "  sizeof(TempSensor) = " << sizeof(TempSensor) << " bytes" << std::endl;

    std::cout << "\n PROBLEMS:" << std::endl;
    std::cout << "  1. RAM overhead: Every object has vptr (4-8 bytes)" << std::endl;
    std::cout << "     - With 100 sensors: 400-800 bytes wasted!" << std::endl;
    std::cout << "  2. Flash overhead: vtable stored in Flash" << std::endl;
    std::cout << "     - 3 virtual functions = 24 bytes Flash per class" << std::endl;
    std::cout << "  3. Indirection: Function call requires:" << std::endl;
    std::cout << "     a. Load vptr from object (1 memory access)" << std::endl;
    std::cout << "     b. Load function pointer from vtable (2nd memory access)" << std::endl;
    std::cout << "     c. Indirect jump (breaks CPU pipeline)" << std::endl;
    std::cout << "  4. Not inlineable: Compiler can't optimize" << std::endl;
    std::cout << "  5. Cache misses: Vtable and code in different locations" << std::endl;

    std::cout << "\n PERFORMANCE IMPACT:" << std::endl;
    std::cout << "  Direct call:   1-2 CPU cycles" << std::endl;
    std::cout << "  Virtual call:  10-20 CPU cycles (10x slower!)" << std::endl;
    std::cout << " • At 80MHz: Virtual call = 125-250ns overhead" << std::endl;
    std::cout << " • In 1ms task: Can fit 4000 direct calls or 400 virtual calls" << std::endl;

    std::cout << "\n SOLUTION: Static polymorphism (CRTP or templates)" << std::endl;
    std::cout << R"(
    template<typename Derived>
    class SensorBase {
        float read() { return static_cast<Derived*>(this)->read_impl(); }
    };

    // Zero overhead, fully inlineable, no vtable!
```

```
    )" << std::endl;
}

// ==================================================================
// 5. PROBLEM: RTTI (Runtime Type Information)
// ==================================================================

void demonstrate_rtti_overhead() {
    std::cout << "\n=== 5. PROBLEM: RTTI (typeid, dynamic_cast) ===" << std::endl;

    std::cout << "\n OVERHEAD:" << std::endl;
    std::cout << " • Type info in Flash: 20-50 bytes per polymorphic class" << std::endl;
    std::cout << " • Name strings: Variable length (stored in Flash)" << std::endl;
    std::cout << " • Dynamic_cast: String comparison overhead" << std::endl;

    std::cout << "\n BAD CODE:" << std::endl;
    std::cout << R"(
    BaseSensor* sensor = get_sensor();

    // dynamic_cast requires RTTI
    if (TempSensor* temp = dynamic_cast<TempSensor*>(sensor)) {
        // Process temperature sensor
    }
    )" << std::endl;

    std::cout << "\n PROBLEMS:" << std::endl;
    std::cout << "  1. Flash overhead: RTTI adds 5-10KB to binary size" << std::endl;
    std::cout << "     - On 256KB Flash: 2-4% wasted!" << std::endl;
    std::cout << "  2. RAM overhead: Type info structures in RAM" << std::endl;
    std::cout << "  3. CPU overhead: String comparisons for casts" << std::endl;
    std::cout << "  4. Complexity: Name mangling, RTTI tables" << std::endl;

    std::cout << "\n TIMING:" << std::endl;
    std::cout << "  static_cast<>:   0 cycles (compile-time)" << std::endl;
    std::cout << "  dynamic_cast<>:  100-500 cycles (runtime check!)" << std::endl;

    std::cout << "\n SOLUTION: std::variant or manual type tags" << std::endl;
    std::cout << R"(
    enum class SensorType { Temperature, Humidity };

    struct Sensor {
        SensorType type;
        float value;
    };

    // Check type with simple enum comparison (1 cycle)
```

```
    if (sensor.type == SensorType::Temperature) { ... }
    )" << std::endl;

    std::cout << "\n GCC FLAG:" << std::endl;
    std::cout << "  -fno-rtti  (disables RTTI, saves 5-10KB)" << std::endl;
}

// ==================================================================
// 6. PROBLEM: EXCEPTIONS
// ==================================================================

void demonstrate_exception_overhead() {
    std::cout << "\n=== 6. PROBLEM: EXCEPTIONS ===" << std::endl;

    std::cout << "\n OVERHEAD:" << std::endl;
    std::cout << " •  Exception tables in Flash: 10-30KB" << std::endl;
    std::cout << " •  Unwinding code: 5-10KB" << std::endl;
    std::cout << " •  Per-function overhead: Extra instructions for cleanup"
        << std::endl;
    std::cout << " •  Total: 15-40KB on 256KB Flash (6-15%!)" << std::endl;

    std::cout << "\n BAD CODE:" << std::endl;
    std::cout << R"(
    try {
        sensor_data.push_back(value);  // May throw std::bad_alloc
        config.parse(json_string);      // May throw parse_error
        send_network(data);             // May throw network_error
    } catch (const std::exception& e) {
        log_error(e.what());
    }
    )" << std::endl;

    std::cout << "\n PROBLEMS:" << std::endl;
    std::cout << "  1. Code bloat: Exception tables for every function" << std
        ::endl;
    std::cout << "  2. Unpredictable: Stack unwinding takes unknown time" <<
        std::endl;
    std::cout << "     - Normal: 0ns (no exception)" << std::endl;
    std::cout << "     - Exception: 10-1000µs (milliseconds!)" << std::endl;
    std::cout << "  3. RAM for stack: Need extra stack for unwinding" << std::
        endl;
    std::cout << "  4. Real-time unsafe: Can't use in ISRs" << std::endl;
    std::cout << "  5. Hidden control flow: Function can exit anywhere" << std
        ::endl;

    std::cout << "\n REAL-TIME VIOLATION:" << std::endl;
    std::cout << "  Task deadline: 1ms" << std::endl;
    std::cout << "  Exception thrown: 500µs to unwind" << std::endl;
    std::cout << "  Result: DEADLINE MISSED! System fails!" << std::endl;

    std::cout << "\n SOLUTION: Error codes or std::optional" << std::endl;
    std::cout << R"(
    enum class Error { None, OutOfMemory, ParseError, NetworkError };

```

```
    Error result = sensor_data.add(value);
    if (result != Error::None) {
        handle_error(result);  // Explicit, predictable
    }

    // Or use std::optional (C++17)
    std::optional<float> maybe_value = read_sensor();
    if (maybe_value) {
        process(*maybe_value);
    }
    )" << std::endl;

    std::cout << "\n GCC FLAG:" << std::endl;
    std::cout << "  -fno-exceptions  (disables exceptions, saves 15-40KB)" << std::endl;
}

// ==================================================================
// 7. PROBLEM: iostream (CODE BLOAT)
// ==================================================================

void demonstrate_iostream_bloat() {
    std::cout << "\n=== 7. PROBLEM: iostream (CODE BLOAT) ===" << std::endl;

    std::cout << "\n BINARY SIZE IMPACT:" << std::endl;

    std::cout << "\n BAD CODE:" << std::endl;
    std::cout << R"(
    #include <iostream>

    int main() {
        std::cout << "Temperature: " << 25.5 << "C" << std::endl;
        return 0;
    }
    )" << std::endl;

    std::cout << "\n BINARY SIZE:" << std::endl;
    std::cout << "  Without iostream:  2-5KB" << std::endl;
    std::cout << "  With iostream:     60-150KB (!)" << std::endl;
    std::cout << "  Increase: 30-60x larger!" << std::endl;

    std::cout << "\n WHAT'S INCLUDED:" << std::endl;
    std::cout << " •  Locale support: 20-40KB" << std::endl;
    std::cout << " •  Formatting code: 15-30KB" << std::endl;
    std::cout << " •  Stream state management: 10-20KB" << std::endl;
    std::cout << " •  Virtual functions (std::ios_base): vtables" << std::endl;
    std::cout << " •  Static initialization: Global constructors" << std::endl;

    std::cout << "\n SOLUTION: printf/snprintf" << std::endl;
    std::cout << R"(
    #include <stdio.h>

```

```
    int main() {
        printf("Temperature: %.1fC\n", 25.5);
        return 0;
    }

    // Binary size: 5-10KB (10x smaller!)
    )" << std::endl;

    std::cout << "\n COMPARISON (ARM Cortex-M4):" << std::endl;
    std::cout << "  iostream:  80KB Flash + 2KB RAM" << std::endl;
    std::cout << "  printf:    8KB Flash + 512 bytes RAM" << std::endl;
    std::cout << "  snprintf:  6KB Flash + 0 bytes heap" << std::endl;

    std::cout << "\n ON 256KB FLASH:" << std::endl;
    std::cout << " •  iostream uses 31% of Flash!" << std::endl;
    std::cout << " •  printf uses 3% of Flash" << std::endl;
}

// ==================================================================
// 8. PROBLEM: RECURSION (STACK OVERFLOW)
// ==================================================================

uint32_t factorial_recursive(uint32_t n) {
    if (n <= 1) return 1;
    return n * factorial_recursive(n - 1);  //  Stack grows
}

uint32_t factorial_iterative(uint32_t n) {
    uint32_t result = 1;
    for (uint32_t i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;  //  Constant stack
}

void demonstrate_recursion_danger() {
    std::cout << "\n=== 8. PROBLEM: RECURSION (STACK OVERFLOW) ===" << std::
        endl;

    std::cout << "\n STACK CONSTRAINTS:" << std::endl;
    std::cout << " •  Total stack: 4KB (4096 bytes)" << std::endl;
    std::cout << " •  Stack frame: ~32 bytes per function call" << std::endl;
    std::cout << " •  Maximum depth: ~128 calls" << std::endl;
    std::cout << " •  NO PROTECTION: Stack overflow = silent corruption!" <<
        std::endl;

    std::cout << "\n BAD CODE (Recursive factorial):" << std::endl;
    std::cout << R"(
    uint32_t factorial(uint32_t n) {
        if (n <= 1) return 1;
        return n * factorial(n - 1);  // Each call uses stack
    }

    // factorial(10)  = 10 stack frames = 320 bytes   OK
```

```
    // factorial(100) = 100 stack frames = 3200 bytes   Close!
    // factorial(150) = 150 stack frames = 4800 bytes   OVERFLOW!
    )" << std::endl;

    std::cout << "\n WHAT HAPPENS ON OVERFLOW:" << std::endl;
    std::cout << "  1. Stack grows downward into heap" << std::endl;
    std::cout << "  2. Corrupts heap data structures" << std::endl;
    std::cout << "  3. Corrupts global variables" << std::endl;
    std::cout << "  4. System crashes (hard fault)" << std::endl;
    std::cout << "  5. NO WARNING! Silent death!" << std::endl;

    std::cout << "\n REAL EXAMPLE: JSON Parser" << std::endl;
    std::cout << R"(
    void parse_json(JsonNode* node) {
        if (node->type == Object) {
            for (auto& child : node->children) {
                parse_json(child);  //  Recursion!
            }
        }
    }

    // Deeply nested JSON (100 levels) = STACK OVERFLOW!
    )" << std::endl;

    std::cout << "\n SOLUTION: Iterative with explicit stack" << std::endl;
    std::cout << R"(
    uint32_t factorial(uint32_t n) {
        uint32_t result = 1;
        for (uint32_t i = 2; i <= n; ++i) {
            result *= i;
        }
        return result;  // Constant 8 bytes stack
    }
    )" << std::endl;

    std::cout << "\n STACK USAGE COMPARISON:" << std::endl;
    std::cout << "  Recursive:  32 bytes × depth (unbounded)" << std::endl;
    std::cout << "  Iterative:  8-16 bytes (constant)" << std::endl;

    std::cout << "\nTest with n=10:" << std::endl;
    std::cout << "  factorial_recursive(10) = " << factorial_recursive(10) << std::endl;
    std::cout << "  factorial_iterative(10) = " << factorial_iterative(10) << std::endl;
}

// ================================================================
// MAIN FUNCTION
// ================================================================

int main() {
    std::cout << "\n ================================================================" << std::endl;
```

```
    std::cout << "  WHY AVOID THESE IN EMBEDDED SYSTEMS" << std::endl;
    std::cout << "
        ================================================================" <<
        std::endl;

    std::cout << "\n TARGET SYSTEM SPECIFICATION:" << std::endl;
    std::cout << "  MCU: ARM Cortex-M4F @ 80MHz" << std::endl;
    std::cout << "  Flash: 256KB (code + constants)" << std::endl;
    std::cout << "  RAM: 64KB (data + stack + heap)" << std::endl;
    std::cout << "    - Stack: 4KB (main task)" << std::endl;
    std::cout << "    - Heap: 8KB (if enabled)" << std::endl;
    std::cout << "    - .bss/.data: ~52KB available" << std::endl;
    std::cout << "  No MMU (no virtual memory)" << std::endl;
    std::cout << "  No OS (bare metal or lightweight RTOS)" << std::endl;
    std::cout << "  Real-time: 1ms task deadlines" << std::endl;
    std::cout << "  Power: Battery powered (sleep modes important)" << std::
        endl;

    demonstrate_heap_fragmentation();
    demonstrate_vector_problems();
    demonstrate_string_problems();
    demonstrate_virtual_function_overhead();
    demonstrate_rtti_overhead();
    demonstrate_exception_overhead();
    demonstrate_iostream_bloat();
    demonstrate_recursion_danger();

    std::cout << "\n
        ================================================================" <<
        std::endl;
    std::cout << "  SUMMARY: RESOURCE IMPACT" << std::endl;
    std::cout << "
        ================================================================" <<
        std::endl;

    std::cout << "\n FLASH USAGE (256KB total):" << std::endl;
    std::cout << "                      " << std::endl;
    std::cout << "    iostream:       80KB (31%)              " << std::endl;
    std::cout << "    Exceptions:     30KB (12%)              " << std::endl;
    std::cout << "    RTTI:           8KB  (3%)               " << std::endl;
    std::cout << "    Virtual tables: 5KB  (2%)               " << std::endl;
    std::cout << "    TOTAL WASTE:    123KB (48%)            " << std::endl;
    std::cout << "                      " << std::endl;
    std::cout << "    Your code:      133KB (52%)             " << std::endl;
    std::cout << "                      " << std::endl;

    std::cout << "\n RAM USAGE (64KB total):" << std::endl;
    std::cout << "                      " << std::endl;
    std::cout << "    Heap:           8KB  (12.5%)            " << std::endl;
    std::cout << "    Stack:          4KB  (6.25%)           " << std::endl;
    std::cout << "    iostream:       2KB  (3.12%)            " << std::endl;
    std::cout << "    vtable ptrs:    1KB  (1.56%)            " << std::endl;
    std::cout << "                      " << std::endl;
    std::cout << "    Your data:      49KB (76.6%)            " << std::endl;
```

```
    std::cout << "                                " << std::endl;

    std::cout << "\n REAL-TIME IMPACT:" << std::endl;
    std::cout << "  Task deadline: 1ms = 80,000 CPU cycles" << std::endl;
    std::cout << "\n  Direct function call:  2 cycles   (40,000 calls per ms)" << std::endl;
    std::cout << "  Virtual call:          20 cycles  (4,000 calls per ms)" << std::endl;
    std::cout << "  std::vector::push:     50 cycles  (1,600 calls per ms)" << std::endl;
    std::cout << "  std::vector realloc:   4,000 cycles (20 calls per ms)" << std::endl;
    std::cout << "  Exception throw:       40,000 cycles (2 per ms = DEADLINE MISS!)" << std::endl;

    std::cout << "\n BEST PRACTICES FOR EMBEDDED:" << std::endl;
    std::cout << "\n    USE:" << std::endl;
    std::cout << " •    std::array (fixed-size, stack)" << std::endl;
    std::cout << " •    Memory pools (controlled allocation)" << std::endl;
    std::cout << " •    Static polymorphism (CRTP, templates)" << std::endl;
    std::cout << " •    Error codes, std::optional (no exceptions)" << std::endl;
    std::cout << " •    printf/snprintf (smaller than iostream)" << std::endl;
    std::cout << " •    Iterative algorithms (constant stack)" << std::endl;
    std::cout << " •    constexpr (compile-time computation)" << std::endl;
    std::cout << " •    std::variant (type-safe unions)" << std::endl;

    std::cout << "\n    AVOID:" << std::endl;
    std::cout << " •    new/delete, malloc/free" << std::endl;
    std::cout << " •    std::vector, std::string, std::map" << std::endl;
    std::cout << " •    Virtual functions (use sparingly)" << std::endl;
    std::cout << " •    RTTI (typeid, dynamic_cast)" << std::endl;
    std::cout << " •    Exceptions (try/catch/throw)" << std::endl;
    std::cout << " •    iostream (cout, cin, etc.)" << std::endl;
    std::cout << " •    Recursion (unbounded depth)" << std::endl;

    std::cout << "\n COMPILER FLAGS FOR EMBEDDED:" << std::endl;
    std::cout << "  -fno-exceptions      (saves 15-40KB Flash)" << std::endl;
    std::cout << "  -fno-rtti            (saves 5-10KB Flash)" << std::endl;
    std::cout << "  -fno-threadsafe-statics (saves 2-5KB Flash)" << std::endl;
    std::cout << "  -ffunction-sections  (enables dead code removal)" << std::endl;
    std::cout << "  -fdata-sections      (enables dead data removal)" << std::endl;
    std::cout << "  -Wl,--gc-sections    (linker removes unused code)" << std::endl;
    std::cout << "  -Os                  (optimize for size)" << std::endl;

    std::cout << "\n MEMORY SAVINGS SUMMARY:" << std::endl;
    std::cout << "  Disabling features above: ~130KB Flash savings!" << std::endl;
    std::cout << "  From 256KB Flash → 126KB available (49% → 98% usable!)" << std::endl;

```

```
    std::cout << "\n
        ==================================================================\n" <<
         std::endl;

    return 0;
}
```

23 Source Code: EmbeddedSystemsProgramming.cpp

File: `src/EmbeddedSystemsProgramming.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <array>
#include <cstdint>
#include <cstring>
#include <type_traits>
#include <utility>
#include <optional>
#include <cassert>

// ==================================================================
// EMBEDDED SYSTEMS PROGRAMMING IN MODERN C++
// ==================================================================
// This file demonstrates embedded C++ patterns and best practices:
// - No dynamic allocation (no new/delete)
// - Fixed-size buffers
// - Compile-time computation
// - RAII for hardware resources
// - Static polymorphism (no vtables)
// - Memory pools
// - Interrupt-safe techniques
// ==================================================================

// ==================================================================
// 1. MEMORY POOL (NO HEAP ALLOCATION)
// ==================================================================

template<typename T, size_t PoolSize>
class MemoryPool {
private:
    alignas(T) uint8_t storage[PoolSize * sizeof(T)];
    bool used[PoolSize] = {};

public:
    template<typename... Args>
    T* allocate(Args&&... args) {
        for (size_t i = 0; i < PoolSize; ++i) {
            if (!used[i]) {
                used[i] = true;
                T* ptr = reinterpret_cast<T*>(&storage[i * sizeof(T)]);
                new (ptr) T(std::forward<Args>(args)...);  // Placement new
                return ptr;
            }
        }
        return nullptr;  // Pool exhausted
    }

    void deallocate(T* ptr) {
        if (!ptr) return;

```

```
        size_t index = (reinterpret_cast<uint8_t*>(ptr) - storage) / sizeof(T);
        if (index < PoolSize && used[index]) {
            ptr->~T();
            used[index] = false;
        }
    }

    size_t available() const {
        size_t count = 0;
        for (size_t i = 0; i < PoolSize; ++i) {
            if (!used[i]) ++count;
        }
        return count;
    }

    size_t capacity() const { return PoolSize; }
};

void example_memory_pool() {
    std::cout << "\n=== 1. MEMORY POOL (NO HEAP) ===" << std::endl;

    struct Sensor {
        int id;
        float value;
        Sensor(int i, float v) : id(i), value(v) {
            std::cout << "    Sensor " << id << " created" << std::endl;
        }
        ~Sensor() {
            std::cout << "    Sensor " << id << " destroyed" << std::endl;
        }
    };

    MemoryPool<Sensor, 5> pool;
    std::cout << "  Pool capacity: " << pool.capacity() << std::endl;

    std::cout << "\nAllocating sensors:" << std::endl;
    Sensor* s1 = pool.allocate(1, 25.5f);
    Sensor* s2 = pool.allocate(2, 30.2f);
    Sensor* s3 = pool.allocate(3, 28.7f);

    std::cout << "  Available slots: " << pool.available() << std::endl;

    std::cout << "\nDeallocating sensor 2:" << std::endl;
    pool.deallocate(s2);
    std::cout << "  Available slots: " << pool.available() << std::endl;

    std::cout << "\nAllocating new sensor:" << std::endl;
    Sensor* s4 = pool.allocate(4, 27.1f);

    std::cout << "\nCleanup:" << std::endl;
    pool.deallocate(s1);
    pool.deallocate(s3);
    pool.deallocate(s4);
```

```

    std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
    std::cout << " •  No heap fragmentation" << std::endl;
    std::cout << " •  Deterministic allocation time" << std::endl;
    std::cout << " •  Fixed memory footprint" << std::endl;
    std::cout << " •  Suitable for real-time systems" << std::endl;
}

// ==================================================================
// 2. CIRCULAR BUFFER (RING BUFFER)
// ==================================================================

template<typename T, size_t Size>
class CircularBuffer {
private:
    std::array<T, Size> buffer;
    size_t head = 0;
    size_t tail = 0;
    size_t count = 0;

public:
    bool push(const T& item) {
        if (is_full()) return false;

        buffer[head] = item;
        head = (head + 1) % Size;
        ++count;
        return true;
    }

    bool pop(T& item) {
        if (is_empty()) return false;

        item = buffer[tail];
        tail = (tail + 1) % Size;
        --count;
        return true;
    }

    bool is_empty() const { return count == 0; }
    bool is_full() const { return count == Size; }
    size_t size() const { return count; }
    size_t capacity() const { return Size; }
};

void example_circular_buffer() {
    std::cout << "\n=== 2. CIRCULAR BUFFER (INTERRUPT-SAFE) ===" << std::endl;

    CircularBuffer<uint8_t, 8> uart_rx_buffer;

    std::cout << "\nSimulating UART receive (ISR):" << std::endl;
    for (uint8_t i = 0; i < 5; ++i) {
        if (uart_rx_buffer.push(0x30 + i)) {
            std::cout << "  [ISR] Received: 0x" << std::hex
```

```
                        << (int)(0x30 + i) << std::dec << std::endl;
        }
    }

    std::cout << "\nBuffer status: " << uart_rx_buffer.size()
              << "/" << uart_rx_buffer.capacity() << std::endl;

    std::cout << "\nMain loop processing:" << std::endl;
    uint8_t data;
    while (uart_rx_buffer.pop(data)) {
        std::cout << "  [Main] Processing: 0x" << std::hex
                  << (int)data << std::dec << std::endl;
    }

    std::cout << "\ n EMBEDDED BENEFITS:" << std::endl;
    std::cout << " •  Lock-free for single producer/consumer" << std::endl;
    std::cout << " •  Constant-time operations" << std::endl;
    std::cout << " •  ISR-safe data transfer" << std::endl;
    std::cout << " •  No dynamic allocation" << std::endl;
}

// ==================================================================
// 3. HARDWARE REGISTER ABSTRACTION (RAII)
// ==================================================================

// Simulate hardware registers
struct HardwareRegisters {
    static inline uint32_t GPIO_CONFIG = 0;
    static inline uint32_t GPIO_OUTPUT = 0;
    static inline uint32_t ADC_CONTROL = 0;
};

enum class PinMode : uint8_t {
    Input = 0,
    Output = 1,
    Analog = 2
};

class GpioPin {
private:
    uint8_t pin_number;
    PinMode original_mode;

    static constexpr uint32_t get_pin_mask(uint8_t pin) {
        return 1u << pin;
    }

public:
    GpioPin(uint8_t pin, PinMode mode)
        : pin_number(pin)
        , original_mode(static_cast<PinMode>(
            (HardwareRegisters::GPIO_CONFIG >> (pin * 2)) & 0x3)) {

        // Configure pin mode
```

```
        uint32_t mask = 0x3u << (pin * 2);
        HardwareRegisters::GPIO_CONFIG &= ~mask;
        HardwareRegisters::GPIO_CONFIG |= (static_cast<uint32_t>(mode) << (pin
            * 2));

        std::cout << "    GPIO Pin " << (int)pin_number
                  << " configured as " << (mode == PinMode::Output ? "Output"
                      : "Input")
                  << std::endl;
    }

    ~GpioPin() {
        // Restore original mode (RAII cleanup)
        uint32_t mask = 0x3u << (pin_number * 2);
        HardwareRegisters::GPIO_CONFIG &= ~mask;
        HardwareRegisters::GPIO_CONFIG |= (static_cast<uint32_t>(original_mode
            ) << (pin_number * 2));

        std::cout << "    GPIO Pin " << (int)pin_number
                  << " restored to original mode" << std::endl;
    }

    // Prevent copying (hardware resources are unique)
    GpioPin(const GpioPin&) = delete;
    GpioPin& operator=(const GpioPin&) = delete;

    // Allow moving
    GpioPin(GpioPin&& other) noexcept
        : pin_number(other.pin_number)
        , original_mode(other.original_mode) {
        other.pin_number = 0xFF;  // Mark as moved
    }

    void write(bool value) {
        if (value) {
            HardwareRegisters::GPIO_OUTPUT |= get_pin_mask(pin_number);
        } else {
            HardwareRegisters::GPIO_OUTPUT &= ~get_pin_mask(pin_number);
        }
    }

    bool read() const {
        return (HardwareRegisters::GPIO_OUTPUT & get_pin_mask(pin_number)) !=
            0;
    }
};

void example_hardware_raii() {
    std::cout << "\n=== 3. HARDWARE REGISTER ABSTRACTION (RAII) ===" << std::
        endl;

    std::cout << "\nConfiguring LED pin:" << std::endl;
    {
        GpioPin led(5, PinMode::Output);
```

```

        std::cout << "  Setting LED ON" << std::endl;
        led.write(true);

        std::cout << "  LED state: " << (led.read() ? "ON" : "OFF") << std::
            endl;

        // RAII: Pin automatically restored when going out of scope
    }
    std::cout << "  (Pin automatically restored)\n" << std::endl;

    std::cout << " EMBEDDED BENEFITS:" << std::endl;
    std::cout << " •  Automatic resource cleanup" << std::endl;
    std::cout << " •  Exception-safe (if exceptions enabled)" << std::endl;
    std::cout << " •  Can't forget to restore state" << std::endl;
    std::cout << " •  Move semantics for ownership transfer" << std::endl;
}

// ================================================================
// 4. STATIC POLYMORPHISM (NO VTABLES)
// ================================================================

// CRTP (Curiously Recurring Template Pattern)
template<typename Derived>
class SensorBase {
public:
    float read() {
        return static_cast<Derived*>(this)->read_impl();
    }

    const char* name() {
        return static_cast<Derived*>(this)->name_impl();
    }
};

class TemperatureSensor : public SensorBase<TemperatureSensor> {
public:
    float read_impl() { return 25.5f; }
    const char* name_impl() { return "Temperature"; }
};

class HumiditySensor : public SensorBase<HumiditySensor> {
public:
    float read_impl() { return 65.2f; }
    const char* name_impl() { return "Humidity"; }
};

template<typename Sensor>
void process_sensor(Sensor& sensor) {
    std::cout << "  Reading " << sensor.name() << " sensor: "
              << sensor.read() << std::endl;
}

void example_static_polymorphism() {
```

```
    std::cout << "\n=== 4. STATIC POLYMORPHISM (NO VTABLES) ===" << std::endl;

    TemperatureSensor temp;
    HumiditySensor humid;

    std::cout << "\nReading sensors (compile-time dispatch):" << std::endl;
    process_sensor(temp);
    process_sensor(humid);

    std::cout << "\ n EMBEDDED BENEFITS:" << std::endl;
    std::cout << " •  No vtable overhead (saves RAM)" << std::endl;
    std::cout << " •  No runtime indirection (faster)" << std::endl;
    std::cout << " •  Fully inlineable" << std::endl;
    std::cout << " •  Smaller code size" << std::endl;
}

// ==================================================================
// 5. COMPILE-TIME COMPUTATION (constexpr)
// ==================================================================

constexpr uint32_t calculate_baud_rate_register(uint32_t system_clock,
    uint32_t baud_rate) {
    return (system_clock / (16 * baud_rate)) - 1;
}

constexpr uint32_t calculate_pwm_period(uint32_t timer_clock, uint32_t
    frequency) {
    return timer_clock / frequency;
}

template<size_t N>
constexpr uint32_t fibonacci() {
    if constexpr (N <= 1) {
        return N;
    } else {
        return fibonacci<N-1>() + fibonacci<N-2>();
    }
}

void example_compile_time() {
    std::cout << "\n=== 5. COMPILE-TIME COMPUTATION (constexpr) ===" << std::
        endl;

    // Computed at compile time!
    constexpr uint32_t BAUD_115200 = calculate_baud_rate_register(16000000,
        115200);
    constexpr uint32_t PWM_1KHZ = calculate_pwm_period(1000000, 1000);
    constexpr uint32_t FIB_10 = fibonacci<10>();

    std::cout << "\nCompile-time computed values:" << std::endl;
    std::cout << "  UART baud rate register: 0x" << std::hex << BAUD_115200 <<
         std::dec << std::endl;
    std::cout << "  PWM period (1kHz): " << PWM_1KHZ << std::endl;
    std::cout << "  Fibonacci(10): " << FIB_10 << std::endl;
```

```

    std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
    std::cout << " • Zero runtime overhead" << std::endl;
    std::cout << " • Compile-time error checking" << std::endl;
    std::cout << " • Values in ROM, not computed" << std::endl;
    std::cout << " • No initialization code needed" << std::endl;
}

// ==================================================================
// 6. FIXED-SIZE STRING (NO STD::STRING)
// ==================================================================

template<size_t MaxSize>
class FixedString {
private:
    char data[MaxSize + 1] = {};  // +1 for null terminator
    size_t length = 0;

public:
    FixedString() = default;

    FixedString(const char* str) {
        if (str) {
            length = 0;
            while (str[length] && length < MaxSize) {
                data[length] = str[length];
                ++length;
            }
            data[length] = '\0';
        }
    }

    bool append(char c) {
        if (length < MaxSize) {
            data[length++] = c;
            data[length] = '\0';
            return true;
        }
        return false;
    }

    bool append(const char* str) {
        if (!str) return false;

        while (*str && length < MaxSize) {
            data[length++] = *str++;
        }
        data[length] = '\0';
        return *str == '\0';  // True if entire string was appended
    }

    void clear() {
        length = 0;
        data[0] = '\0';
```

```
    }

    const char* c_str() const { return data; }
    size_t size() const { return length; }
    size_t capacity() const { return MaxSize; }
    bool is_full() const { return length >= MaxSize; }
};

void example_fixed_string() {
    std::cout << "\n=== 6. FIXED-SIZE STRING (NO HEAP) ===" << std::endl;

    FixedString<32> msg("Sensor: ");

    std::cout << "\nBuilding message:" << std::endl;
    std::cout << "  Initial: \"" << msg.c_str() << "\"" << std::endl;

    msg.append("Temp=");
    msg.append("25.5");
    msg.append("C");

    std::cout << "  Final: \"" << msg.c_str() << "\"" << std::endl;
    std::cout << "  Size: " << msg.size() << "/" << msg.capacity() << std::
        endl;

    std::cout << "\ n EMBEDDED BENEFITS:" << std::endl;
    std::cout << " •  No heap allocation" << std::endl;
    std::cout << " •  Fixed memory footprint" << std::endl;
    std::cout << " •  Predictable behavior" << std::endl;
    std::cout << " •  Stack-based storage" << std::endl;
}

// ==================================================================
// 7. STATE MACHINE (EMBEDDED PATTERN)
// ==================================================================

enum class State {
    Idle,
    Reading,
    Processing,
    Sending,
    Error
};

enum class Event {
    StartRead,
    DataReady,
    ProcessComplete,
    SendComplete,
    ErrorOccurred,
    Reset
};

class StateMachine {
private:
```

```
    State current_state = State::Idle;

public:
    void process_event(Event event) {
        std::cout << "  [" << state_name(current_state) << "] -> Event: "
                  << event_name(event);

        State new_state = current_state;

        switch (current_state) {
            case State::Idle:
                if (event == Event::StartRead) {
                    new_state = State::Reading;
                }
                break;

            case State::Reading:
                if (event == Event::DataReady) {
                    new_state = State::Processing;
                } else if (event == Event::ErrorOccurred) {
                    new_state = State::Error;
                }
                break;

            case State::Processing:
                if (event == Event::ProcessComplete) {
                    new_state = State::Sending;
                } else if (event == Event::ErrorOccurred) {
                    new_state = State::Error;
                }
                break;

            case State::Sending:
                if (event == Event::SendComplete) {
                    new_state = State::Idle;
                } else if (event == Event::ErrorOccurred) {
                    new_state = State::Error;
                }
                break;

            case State::Error:
                if (event == Event::Reset) {
                    new_state = State::Idle;
                }
                break;
        }

        if (new_state != current_state) {
            current_state = new_state;
            std::cout << " -> [" << state_name(current_state) << "]" << std::
                endl;
        } else {
            std::cout << " (no transition)" << std::endl;
        }
```

```
    }

    State get_state() const { return current_state; }

private:
    static const char* state_name(State s) {
        switch (s) {
            case State::Idle: return "Idle";
            case State::Reading: return "Reading";
            case State::Processing: return "Processing";
            case State::Sending: return "Sending";
            case State::Error: return "Error";
            default: return "Unknown";
        }
    }

    static const char* event_name(Event e) {
        switch (e) {
            case Event::StartRead: return "StartRead";
            case Event::DataReady: return "DataReady";
            case Event::ProcessComplete: return "ProcessComplete";
            case Event::SendComplete: return "SendComplete";
            case Event::ErrorOccurred: return "ErrorOccurred";
            case Event::Reset: return "Reset";
            default: return "Unknown";
        }
    }
};

void example_state_machine() {
    std::cout << "\n=== 7. STATE MACHINE (EMBEDDED PATTERN) ===" << std::endl;

    StateMachine sm;

    std::cout << "\nNormal operation sequence:" << std::endl;
    sm.process_event(Event::StartRead);
    sm.process_event(Event::DataReady);
    sm.process_event(Event::ProcessComplete);
    sm.process_event(Event::SendComplete);

    std::cout << "\nError handling:" << std::endl;
    sm.process_event(Event::StartRead);
    sm.process_event(Event::ErrorOccurred);
    sm.process_event(Event::Reset);

    std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
    std::cout << " •  Clear, predictable behavior" << std::endl;
    std::cout << " •  Easy to validate and test" << std::endl;
    std::cout << " •  Common in embedded protocols" << std::endl;
    std::cout << " •  No dynamic dispatch overhead" << std::endl;
}

// ================================================================
// 8. BIT MANIPULATION UTILITIES
```

```
// ==================================================================

namespace BitOps {
    template<typename T>
    constexpr void set_bit(T& reg, uint8_t bit) {
        reg |= (static_cast<T>(1) << bit);
    }

    template<typename T>
    constexpr void clear_bit(T& reg, uint8_t bit) {
        reg &= ~(static_cast<T>(1) << bit);
    }

    template<typename T>
    constexpr void toggle_bit(T& reg, uint8_t bit) {
        reg ^= (static_cast<T>(1) << bit);
    }

    template<typename T>
    constexpr bool test_bit(T reg, uint8_t bit) {
        return (reg & (static_cast<T>(1) << bit)) != 0;
    }

    template<typename T>
    constexpr void write_bits(T& reg, uint8_t start_bit, uint8_t num_bits, T
        value) {
        T mask = ((static_cast<T>(1) << num_bits) - 1) << start_bit;
        reg = (reg & ~mask) | ((value << start_bit) & mask);
    }
}

void example_bit_manipulation() {
    std::cout << "\n=== 8. BIT MANIPULATION UTILITIES ===" << std::endl;

    uint32_t control_reg = 0;

    std::cout << "\nManipulating control register:" << std::endl;
    std::cout << "  Initial value: 0x" << std::hex << control_reg << std::dec
        << std::endl;

    BitOps::set_bit(control_reg, 0);  // Enable bit 0
    std::cout << "  After set bit 0: 0x" << std::hex << control_reg << std::
        dec << std::endl;

    BitOps::set_bit(control_reg, 5);  // Enable bit 5
    std::cout << "  After set bit 5: 0x" << std::hex << control_reg << std::
        dec << std::endl;

    BitOps::write_bits(control_reg, 2, 3, 0b101u);  // Write 3 bits starting
        at bit 2
    std::cout << "  After write bits 2-4: 0x" << std::hex << control_reg <<
        std::dec << std::endl;

    std::cout << "\nTesting bits:" << std::endl;
```

```
    std::cout << "  Bit 0 is " << (BitOps::test_bit(control_reg, 0) ? "SET" : "CLEAR") << std::endl;
    std::cout << "  Bit 1 is " << (BitOps::test_bit(control_reg, 1) ? "SET" : "CLEAR") << std::endl;

    std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
    std::cout << " •  Type-safe bit operations" << std::endl;
    std::cout << " •  Constexpr for compile-time use" << std::endl;
    std::cout << " •  Clear, readable code" << std::endl;
    std::cout << " •  No magic numbers" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  EMBEDDED SYSTEMS PROGRAMMING IN MODERN C++" << std::endl;
    std::cout << " ==================================================================" << std::endl;

    example_memory_pool();
    example_circular_buffer();
    example_hardware_raii();
    example_static_polymorphism();
    example_compile_time();
    example_fixed_string();
    example_state_machine();
    example_bit_manipulation();

    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  EMBEDDED C++ BEST PRACTICES SUMMARY" << std::endl;
    std::cout << " ==================================================================" << std::endl;

    std::cout << "\n KEY PRINCIPLES FOR EMBEDDED C++:" << std::endl;

    std::cout << "\n1. NO DYNAMIC ALLOCATION" << std::endl;
    std::cout << " •   Use memory pools instead of new/delete" << std::endl;
    std::cout << " •   Fixed-size containers (std::array)" << std::endl;
    std::cout << " •   Stack allocation preferred" << std::endl;
    std::cout << " •   Predictable memory usage" << std::endl;

    std::cout << "\n2. COMPILE-TIME COMPUTATION" << std::endl;
    std::cout << " •   constexpr functions for configuration" << std::endl;
    std::cout << " •   Template metaprogramming" << std::endl;
    std::cout << " •   if constexpr for conditional compilation" << std::endl;
```

```
    std::cout << " •   Zero runtime overhead" << std::endl;

    std::cout << "\n3. STATIC POLYMORPHISM" << std::endl;
    std::cout << " •   CRTP instead of virtual functions" << std::endl;
    std::cout << " •   No vtable overhead" << std::endl;
    std::cout << " •   Fully inlineable" << std::endl;
    std::cout << " •   Saves RAM and Flash" << std::endl;

    std::cout << "\n4. RAII FOR HARDWARE" << std::endl;
    std::cout << " •   Automatic resource management" << std::endl;
    std::cout << " •   Scope-based cleanup" << std::endl;
    std::cout << " •   Move semantics for ownership" << std::endl;
    std::cout << " •   Exception-safe (if enabled)" << std::endl;

    std::cout << "\n5. INTERRUPT-SAFE PATTERNS" << std::endl;
    std::cout << " •   Circular buffers for ISR->Main communication" << std::
        endl;
    std::cout << " •   Atomic operations" << std::endl;
    std::cout << " •   Lock-free data structures" << std::endl;
    std::cout << " •   Minimal ISR processing" << std::endl;

    std::cout << "\n6. DETERMINISTIC BEHAVIOR" << std::endl;
    std::cout << " •   No recursion (or limited)" << std::endl;
    std::cout << " •   Bounded loops" << std::endl;
    std::cout << " •   Fixed execution time" << std::endl;
    std::cout << " •   Real-time constraints" << std::endl;

    std::cout << "\ n MODERN C++ FEATURES FOR EMBEDDED:" << std::endl;
    std::cout << "     constexpr (C++11/14/17) - Compile-time computation" <<
        std::endl;
    std::cout << "     std::array (C++11) - Fixed-size arrays" << std::endl;
    std::cout << "     std::optional (C++17) - Error handling without
        exceptions" << std::endl;
    std::cout << "     if constexpr (C++17) - Conditional compilation" << std
        ::endl;
    std::cout << "     Templates - Static polymorphism" << std::endl;
    std::cout << "     Move semantics (C++11) - Efficient ownership transfer"
        << std::endl;
    std::cout << "     RAII - Automatic resource management" << std::endl;
    std::cout << "     Type traits - Compile-time type checking" << std::endl;

    std::cout << "\ n AVOID IN EMBEDDED:" << std::endl;
    std::cout << "     new/delete (use memory pools)" << std::endl;
    std::cout << "     std::vector, std::string (dynamic allocation)" << std::
        endl;
    std::cout << "     Virtual functions (unless justified)" << std::endl;
    std::cout << "     RTTI (enable only if needed)" << std::endl;
    std::cout << "     Exceptions (often disabled in embedded)" << std::endl;
    std::cout << "     iostream (large code size)" << std::endl;
    std::cout << "     Recursion (stack overflow risk)" << std::endl;

    std::cout << "\ n EMBEDDED C++ IDIOMS:" << std::endl;
    std::cout << " •   Singleton pattern for hardware peripherals" << std::
        endl;
```

```
    std::cout << " •   State machines for protocol handling" << std::endl;
    std::cout << " •   Circular buffers for data streaming" << std::endl;
    std::cout << " •   Memory pools for dynamic-like allocation" << std::endl;
    std::cout << " •   CRTP for zero-cost abstraction" << std::endl;
    std::cout << " •   Register classes with bit fields" << std::endl;

    std::cout << "\n TYPICAL MEMORY CONSTRAINTS:" << std::endl;
    std::cout << " •   Flash: 32KB - 2MB (code storage)" << std::endl;
    std::cout << " •   RAM: 4KB - 256KB (runtime data)" << std::endl;
    std::cout << " •   Stack: 1KB - 8KB (limited depth)" << std::endl;
    std::cout << " •   No heap or tiny heap (fragmentation risk)" << std::endl;

    std::cout << "\n REAL-TIME CONSIDERATIONS:" << std::endl;
    std::cout << " •   Interrupt latency: <10µs typical" << std::endl;
    std::cout << " •   Task scheduling: RTOS or bare-metal" << std::endl;
    std::cout << " •   Timing critical sections" << std::endl;
    std::cout << " •   Watchdog timer management" << std::endl;

    std::cout << "\n RECOMMENDED READING:" << std::endl;
    std::cout << " •   \"Embedded C++ Coding Standard\" by BARR Group" << std::endl;
    std::cout << " •   \"Real-Time C++\" by Christopher Kormanyos" << std::endl;
    std::cout << " •   \"Effective Modern C++\" (embedded-applicable patterns)" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

24 Source Code: ErrorHandling.cpp

File: `src/ErrorHandling.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <stdexcept>
#include <string>
#include <memory>
#include <vector>
#include <optional>
#include <variant>
#include <fstream>
#include <system_error>
#include <cassert>
#include <type_traits>

// ==================================================================
// C++ ERROR HANDLING: COMPREHENSIVE EXAMPLES
// ==================================================================
// Demonstrates:
// 1. Throw by value, catch by reference (BEST PRACTICE)
// 2. Standard exception hierarchy
// 3. Custom exceptions
// 4. Exception safety guarantees
// 5. Compile-time error detection
// 6. Alternative error handling patterns
// ==================================================================

// ==================================================================
// 1. STANDARD EXCEPTIONS - THROW BY VALUE, CATCH BY REFERENCE
// ==================================================================

void demonstrate_basic_exception_handling() {
    std::cout << "\n=== 1. BASIC EXCEPTION HANDLING ===" << std::endl;
    std::cout << "Rule: THROW BY VALUE, CATCH BY REFERENCE" << std::endl;

    // Example 1: std::runtime_error
    try {
        std::cout << "\nThrowing std::runtime_error..." << std::endl;
        throw std::runtime_error("Something went wrong at runtime!");  // Throw by value
    }
    catch (const std::runtime_error& e) {  // Catch by const reference
        std::cout << "Caught runtime_error: " << e.what() << std::endl;
    }

    // Example 2: std::logic_error
    try {
        std::cout << "\nThrowing std::logic_error..." << std::endl;
        throw std::logic_error("Logic error in the program!");  // Throw by value
    }
    catch (const std::logic_error& e) {  // Catch by const reference
```

```
        std::cout << "Caught logic_error: " << e.what() << std::endl;
    }

    // Example 3: std::invalid_argument
    try {
        std::cout << "\nThrowing std::invalid_argument..." << std::endl;
        throw std::invalid_argument("Invalid argument provided!");  // Throw by value
    }
    catch (const std::invalid_argument& e) {  // Catch by const reference
        std::cout << "Caught invalid_argument: " << e.what() << std::endl;
    }

    // Example 4: std::out_of_range
    try {
        std::cout << "\nThrowing std::out_of_range..." << std::endl;
        throw std::out_of_range("Index out of range!");  // Throw by value
    }
    catch (const std::out_of_range& e) {  // Catch by const reference
        std::cout << "Caught out_of_range: " << e.what() << std::endl;
    }
}

// ==================================================================
// 2. STANDARD EXCEPTION HIERARCHY
// ==================================================================

void demonstrate_exception_hierarchy() {
    std::cout << "\n=== 2. EXCEPTION HIERARCHY ===" << std::endl;
    std::cout << "std::exception (base)" << std::endl;
    std::cout << "   std::logic_error" << std::endl;
    std::cout << "       std::invalid_argument" << std::endl;
    std::cout << "       std::domain_error" << std::endl;
    std::cout << "       std::length_error" << std::endl;
    std::cout << "       std::out_of_range" << std::endl;
    std::cout << "   std::runtime_error" << std::endl;
    std::cout << "       std::range_error" << std::endl;
    std::cout << "       std::overflow_error" << std::endl;
    std::cout << "       std::underflow_error" << std::endl;

    // Catch base class to handle all derived exceptions
    try {
        std::cout << "\nThrowing derived exception..." << std::endl;
        throw std::overflow_error("Integer overflow occurred!");
    }
    catch (const std::runtime_error& e) {  // Catches std::overflow_error (derived)
        std::cout << "Caught via base class runtime_error: " << e.what() << std::endl;
    }

    // Multiple catch blocks - order matters!
    try {
        std::cout << "\nDemonstrating catch order..." << std::endl;
```

```
        throw std::invalid_argument("Bad argument");
    }
    catch (const std::invalid_argument& e) {  // Specific exception first
        std::cout << "Caught specific: invalid_argument: " << e.what() << std
            ::endl;
    }
    catch (const std::logic_error& e) {  // Base class second
        std::cout << "Caught base: logic_error: " << e.what() << std::endl;
    }
    catch (const std::exception& e) {  // Most general last
        std::cout << "Caught most general: exception: " << e.what() << std::
            endl;
    }
}

// ==================================================================
// 3. CUSTOM EXCEPTIONS
// ==================================================================

// Custom exception class - inherit from std::exception
class CameraException : public std::runtime_error {
private:
    int error_code;

public:
    explicit CameraException(const std::string& message, int code = 0)
        : std::runtime_error(message), error_code(code) {}

    int get_error_code() const noexcept { return error_code; }
};

class CameraNotConnectedException : public CameraException {
public:
    explicit CameraNotConnectedException(const std::string& camera_name)
        : CameraException("Camera not connected: " + camera_name, 100) {}
};

class CameraConfigurationException : public CameraException {
public:
    explicit CameraConfigurationException(const std::string& config_error)
        : CameraException("Configuration error: " + config_error, 200) {}
};

void demonstrate_custom_exceptions() {
    std::cout << "\n=== 3. CUSTOM EXCEPTIONS ===" << std::endl;

    // Throw custom exception by value
    try {
        std::cout << "\nThrowing CameraNotConnectedException..." << std::endl;
        throw CameraNotConnectedException("USB-CAM-001");  // Throw by value
    }
    catch (const CameraNotConnectedException& e) {  // Catch by const
        reference
        std::cout << "Caught: " << e.what() << std::endl;
```

```
        std::cout << "Error code: " << e.get_error_code() << std::endl;
    }

    // Catch hierarchy with custom exceptions
    try {
        std::cout << "\nThrowing CameraConfigurationException..." << std::endl;
        throw CameraConfigurationException("Invalid resolution: 9999x9999");
    }
    catch (const CameraException& e) {  // Catch base custom exception
        std::cout << "Caught via base CameraException: " << e.what() << std::endl;
        std::cout << "Error code: " << e.get_error_code() << std::endl;
    }
    catch (const std::exception& e) {  // Catch any std::exception
        std::cout << "Caught via std::exception: " << e.what() << std::endl;
    }
}

// ================================================================
// 4. EXCEPTION SAFETY GUARANTEES
// ================================================================

class SafeVector {
private:
    std::vector<int> data;

public:
    // Basic guarantee: No resources leaked, but object state may change
    void push_back_basic(int value) {
        data.push_back(value);  // May throw, but vector cleans up
    }

    // Strong guarantee: Operation succeeds or has no effect (rollback)
    void push_back_strong(int value) {
        std::vector<int> temp = data;  // Copy current state
        temp.push_back(value);         // Modify copy (if this throws, original unchanged)
        data = std::move(temp);        // Commit (noexcept move)
    }

    // No-throw guarantee: Marked noexcept
    size_t size() const noexcept {
        return data.size();
    }

    // No-throw guarantee: Never throws
    void clear() noexcept {
        data.clear();
    }

    int& at(size_t index) {
        return data.at(index);  // May throw std::out_of_range
    }
```

```
};

void demonstrate_exception_safety() {
    std::cout << "\n=== 4. EXCEPTION SAFETY GUARANTEES ===" << std::endl;

    std::cout << "\nException Safety Levels:" << std::endl;
    std::cout << "1. No-throw guarantee (noexcept): Never throws" << std::endl
        ;
    std::cout << "2. Strong guarantee: Rollback on exception" << std::endl;
    std::cout << "3. Basic guarantee: No resource leaks" << std::endl;
    std::cout << "4. No guarantee: May leak resources" << std::endl;

    SafeVector vec;
    vec.push_back_basic(1);
    vec.push_back_basic(2);

    try {
        std::cout << "\nAccessing valid index: vec.at(0)" << std::endl;
        std::cout << "Value: " << vec.at(0) << std::endl;

        std::cout << "\nAccessing invalid index: vec.at(100)" << std::endl;
        vec.at(100);  // Throws std::out_of_range
    }
    catch (const std::out_of_range& e) {
        std::cout << "Caught: " << e.what() << std::endl;
        std::cout << "Vector is still valid, size: " << vec.size() << std::
            endl;
    }
}

// ==================================================================
// 5. RAII AND EXCEPTION SAFETY
// ==================================================================

class FileHandle {
private:
    std::string filename;
    FILE* file;

public:
    explicit FileHandle(const std::string& fname, const char* mode)
        : filename(fname), file(nullptr) {
        file = fopen(fname.c_str(), mode);
        if (!file) {
            throw std::runtime_error("Failed to open file: " + fname);
        }
        std::cout << "File opened: " << filename << std::endl;
    }

    ~FileHandle() {
        if (file) {
            fclose(file);
            std::cout << "File closed: " << filename << std::endl;
        }
```

```
    }

    // Delete copy operations
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    // Move operations
    FileHandle(FileHandle&& other) noexcept
        : filename(std::move(other.filename)), file(other.file) {
        other.file = nullptr;
    }

    FILE* get() const noexcept { return file; }
};

void demonstrate_raii() {
    std::cout << "\n=== 5. RAII AND EXCEPTION SAFETY ===" << std::endl;

    try {
        std::cout << "\nOpening file (will throw)..." << std::endl;
        FileHandle handle("nonexistent_file_xyz.txt", "r");
        std::cout << "This won't print" << std::endl;
    }
    catch (const std::runtime_error& e) {
        std::cout << "Caught: " << e.what() << std::endl;
        std::cout << "No resource leak - file was never opened!" << std::endl;
    }

    std::cout << "\nDemonstrating automatic cleanup with scope:" << std::endl;
    try {
        // Create temporary file for demo
        {
            FileHandle handle1("temp_test.txt", "w");
            fprintf(handle1.get(), "Test data\n");
            // File automatically closed when handle1 goes out of scope
        }
        std::cout << "File closed automatically (RAII)" << std::endl;

        // Clean up temp file
        std::remove("temp_test.txt");
    }
    catch (const std::exception& e) {
        std::cout << "Exception: " << e.what() << std::endl;
    }
}

// ==================================================================
// 6. COMPILE-TIME ERROR DETECTION
// ==================================================================

// Static assertions - compile-time checks
template<typename T>
class Buffer {
private:
```

```
    std::vector<T> data;

public:
    // Compile-time check: Only allow trivially copyable types
    static_assert(std::is_trivially_copyable_v<T>,
                  "Buffer only works with trivially copyable types!");

    Buffer(size_t size) : data(size) {}

    T* get_data() { return data.data(); }
};

// constexpr for compile-time evaluation
constexpr int divide_compile_time(int a, int b) {
    // This will cause compile error if b is 0 at compile time
    return (b == 0) ? throw std::invalid_argument("Division by zero!") : a / b;
}

// C++20 Concepts for compile-time type checking
template<typename T>
concept Numeric = std::is_arithmetic_v<T>;

template<Numeric T>
T safe_divide(T a, T b) {
    if (b == T(0)) {
        throw std::invalid_argument("Division by zero!");
    }
    return a / b;
}

void demonstrate_compile_time_checks() {
    std::cout << "\n=== 6. COMPILE-TIME ERROR DETECTION ===" << std::endl;

    std::cout << "\n1. Static Assertions:" << std::endl;
    Buffer<int> int_buffer(10);  // OK: int is trivially copyable
    std::cout << "    Buffer<int> compiles (trivially copyable)" << std::endl;

    // This would NOT compile:
    // Buffer<std::string> string_buffer(10);  // ERROR: string not trivially copyable
    std::cout << "    Buffer<std::string> would fail to compile!" << std::endl;

    std::cout << "\n2. constexpr Functions:" << std::endl;
    constexpr int result = divide_compile_time(10, 2);  // OK at compile time
    std::cout << "   10 / 2 = " << result << " (evaluated at compile time)" << std::endl;

    // This would NOT compile:
    // constexpr int error = divide_compile_time(10, 0);  // Compile error!
    std::cout << "   10 / 0 would fail at compile time!" << std::endl;

```

```
    std::cout << "\n3. Concepts (C++20):" << std::endl;
    try {
        int a = safe_divide(10, 2);
        std::cout << "   10 / 2 = " << a << std::endl;

        double b = safe_divide(10.0, 0.0);  // Runtime error
        std::cout << "   This won't print: " << b << std::endl;
    }
    catch (const std::invalid_argument& e) {
        std::cout << "   Caught: " << e.what() << std::endl;
    }

    // This would NOT compile:
    // safe_divide(std::string("10"), std::string("2"));  // Concept constraint violated!
    std::cout << "   safe_divide with std::string would fail to compile!" << std::endl;
}

// ==================================================================
// 7. NOEXCEPT SPECIFICATION
// ==================================================================

class NoexceptDemo {
public:
    // Guaranteed not to throw
    int get_value() const noexcept {
        return 42;
    }

    // May throw
    int divide(int a, int b) {
        if (b == 0) {
            throw std::invalid_argument("Division by zero");
        }
        return a / b;
    }

    // Conditionally noexcept based on template parameter
    template<typename T>
    void swap(T& a, T& b) noexcept(std::is_nothrow_move_constructible_v<T>) {
        T temp = std::move(a);
        a = std::move(b);
        b = std::move(temp);
    }
};

void demonstrate_noexcept() {
    std::cout << "\n=== 7. NOEXCEPT SPECIFICATION ===" << std::endl;

    NoexceptDemo demo;

    std::cout << "\nnoexcept functions:" << std::endl;
    std::cout << "  get_value() is noexcept: "
```

```
              << noexcept(demo.get_value()) << std::endl;
    std::cout << "  divide() is noexcept: "
              << noexcept(demo.divide(10, 2)) << std::endl;

    std::cout << "\nBenefits of noexcept:" << std::endl;
    std::cout << " •  Compiler optimizations" << std::endl;
    std::cout << " •  Move constructors in std::vector" << std::endl;
    std::cout << " •  Enables certain optimizations in algorithms" << std::
        endl;
    std::cout << " •  Self-documenting code" << std::endl;

    std::cout << "\n  If noexcept function throws: std::terminate() is called
        !" << std::endl;
}

// ==================================================================
// 8. ALTERNATIVE ERROR HANDLING: std::optional
// ==================================================================

std::optional<int> safe_divide_optional(int a, int b) {
    if (b == 0) {
        return std::nullopt;  // Return empty optional instead of throwing
    }
    return a / b;
}

void demonstrate_optional() {
    std::cout << "\n=== 8. ERROR HANDLING WITH std::optional ===" << std::endl
        ;
    std::cout << "Alternative to exceptions for expected failures" << std::
        endl;

    auto result1 = safe_divide_optional(10, 2);
    if (result1) {
        std::cout << "\n10 / 2 = " << *result1 << "  " << std::endl;
    } else {
        std::cout << "\n10 / 2 failed  " << std::endl;
    }

    auto result2 = safe_divide_optional(10, 0);
    if (result2) {
        std::cout << "10 / 0 = " << *result2 << "  " << std::endl;
    } else {
        std::cout << "10 / 0 failed (expected)  " << std::endl;
    }

    std::cout << "\nBenefits:" << std::endl;
    std::cout << " •  No exception overhead" << std::endl;
    std::cout << " •  Makes errors explicit in return type" << std::endl;
    std::cout << " •  Good for expected failures" << std::endl;
}

// ==================================================================
// 9. ALTERNATIVE ERROR HANDLING: std::variant
```

```
// ==================================================================

struct Error {
    int code;
    std::string message;
};

std::variant<int, Error> safe_divide_variant(int a, int b) {
    if (b == 0) {
        return Error{1, "Division by zero"};
    }
    return a / b;
}

void demonstrate_variant() {
    std::cout << "\n=== 9. ERROR HANDLING WITH std::variant ===" << std::endl;
    std::cout << "Return either result or error" << std::endl;

    auto result1 = safe_divide_variant(10, 2);
    if (std::holds_alternative<int>(result1)) {
        std::cout << "\n10 / 2 = " << std::get<int>(result1) << "  " << std::
            endl;
    } else {
        const auto& err = std::get<Error>(result1);
        std::cout << "\nError " << err.code << ": " << err.message << "  " <<
            std::endl;
    }

    auto result2 = safe_divide_variant(10, 0);
    if (std::holds_alternative<int>(result2)) {
        std::cout << "10 / 0 = " << std::get<int>(result2) << "  " << std::
            endl;
    } else {
        const auto& err = std::get<Error>(result2);
        std::cout << "Error " << err.code << ": " << err.message << "  " <<
            std::endl;
    }

    std::cout << "\nBenefits:" << std::endl;
    std::cout << " •  Can return detailed error information" << std::endl;
    std::cout << " •  Type-safe error handling" << std::endl;
    std::cout << " •  No exception overhead" << std::endl;
}

// ==================================================================
// 10. RETHROWING AND NESTED EXCEPTIONS
// ==================================================================

void inner_function() {
    throw std::runtime_error("Error in inner function");
}

void middle_function() {
    try {
```

```
        inner_function();
    }
    catch (...) {
        std::cout << "middle_function: Caught exception, rethrowing..." << std::endl;
        throw;  // Rethrow the same exception
    }
}

void demonstrate_rethrowing() {
    std::cout << "\n=== 10. RETHROWING EXCEPTIONS ===" << std::endl;

    try {
        middle_function();
    }
    catch (const std::runtime_error& e) {
        std::cout << "Caught in demonstrate_rethrowing: " << e.what() << std::endl;
    }

    std::cout << "\nRethrow with throw; (not throw e;)" << std::endl;
    std::cout << "  throw;  - Rethrows original exception (correct)" << std::endl;
    std::cout << "  throw e; - Creates new exception (slicing!)" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n==================================================================" << std::endl;
    std::cout << "  C++ ERROR HANDLING: COMPREHENSIVE GUIDE" << std::endl;
    std::cout << "==================================================================" << std::endl;

    demonstrate_basic_exception_handling();
    demonstrate_exception_hierarchy();
    demonstrate_custom_exceptions();
    demonstrate_exception_safety();
    demonstrate_raii();
    demonstrate_compile_time_checks();
    demonstrate_noexcept();
    demonstrate_optional();
    demonstrate_variant();
    demonstrate_rethrowing();

    std::cout << "\n==================================================================" << std::endl;
    std::cout << "  BEST PRACTICES SUMMARY" << std::endl;
```

```
    std::cout << "
        ================================================================" <<
        std::endl;

    std::cout << "\ n EXCEPTION HANDLING RULES:" << std::endl;
    std::cout << "\n1. THROW BY VALUE, CATCH BY REFERENCE" << std::endl;
    std::cout << "     throw std::runtime_error(\"message\");" << std::endl;
    std::cout << "     catch (const std::runtime_error& e)" << std::endl;
    std::cout << "     throw new std::runtime_error(\"message\");  // NO!" <<
        std::endl;
    std::cout << "     catch (std::runtime_error e)  // Slicing!" << std::endl
        ;

    std::cout << "\n2. CATCH ORDER MATTERS" << std::endl;
    std::cout << " •   Catch specific exceptions first" << std::endl;
    std::cout << " •   Catch base classes last" << std::endl;
    std::cout << " •   catch(...) for unknown exceptions" << std::endl;

    std::cout << "\n3. USE STANDARD EXCEPTIONS" << std::endl;
    std::cout << " •   std::runtime_error - Runtime failures" << std::endl;
    std::cout << " •   std::logic_error - Programming errors" << std::endl;
    std::cout << " •   std::invalid_argument - Bad function arguments" << std
        ::endl;
    std::cout << " •   std::out_of_range - Index out of bounds" << std::endl;

    std::cout << "\n4. RAII FOR EXCEPTION SAFETY" << std::endl;
    std::cout << " •   Use smart pointers (std::unique_ptr, std::shared_ptr)"
        << std::endl;
    std::cout << " •   RAII wrappers for resources" << std::endl;
    std::cout << " •   Destructors never throw" << std::endl;

    std::cout << "\n5. NOEXCEPT FOR NON-THROWING FUNCTIONS" << std::endl;
    std::cout << " •   Mark functions that never throw" << std::endl;
    std::cout << " •   Enables compiler optimizations" << std::endl;
    std::cout << " •   Move constructors should be noexcept" << std::endl;

    std::cout << "\n6. COMPILE-TIME ERROR DETECTION" << std::endl;
    std::cout << " •   static_assert for compile-time checks" << std::endl;
    std::cout << " •   constexpr for compile-time evaluation" << std::endl;
    std::cout << " •   Concepts (C++20) for type constraints" << std::endl;
    std::cout << " •   Template SFINAE for type checking" << std::endl;

    std::cout << "\n7. ALTERNATIVES TO EXCEPTIONS" << std::endl;
    std::cout << " •   std::optional<T> - For expected failures" << std::endl;
    std::cout << " •   std::variant<T, Error> - Return result or error" << std
        ::endl;
    std::cout << " •   Error codes - For performance-critical code" << std::
        endl;
    std::cout << " •   std::expected<T, E> (C++23) - Best of both worlds" <<
        std::endl;

    std::cout << "\ n WHEN TO USE WHAT:" << std::endl;
    std::cout << "\nExceptions:" << std::endl;
    std::cout << "    Unexpected errors (file not found, network failure)" <<
```

```
        std::endl;
    std::cout << "    Constructor failures" << std::endl;
    std::cout << "    Deep call stacks (error propagation)" << std::endl;
    std::cout << "    Performance-critical code" << std::endl;
    std::cout << "    Expected failures (validation)" << std::endl;

    std::cout << "\nstd::optional:" << std::endl;
    std::cout << "    Expected failures (search not found)" << std::endl;
    std::cout << "    Optional return values" << std::endl;
    std::cout << "    Performance-critical code" << std::endl;
    std::cout << "    Need detailed error information" << std::endl;

    std::cout << "\nstd::variant<T, Error>:" << std::endl;
    std::cout << "    Need detailed error information" << std::endl;
    std::cout << "    Multiple error types" << std::endl;
    std::cout << "    Performance-critical code" << std::endl;
    std::cout << "    Simple success/failure cases" << std::endl;

    std::cout << "\nCompile-time checks:" << std::endl;
    std::cout << "    Type constraints" << std::endl;
    std::cout << "    API misuse prevention" << std::endl;
    std::cout << "    Zero runtime cost" << std::endl;
    std::cout << "    Early error detection" << std::endl;

    std::cout << "\ n PERFORMANCE NOTES:" << std::endl;
    std::cout << " •  Exceptions have zero cost if not thrown (modern
        compilers)" << std::endl;
    std::cout << " •  Throwing exception is expensive (~1000x slower than
        return)" << std::endl;
    std::cout << " •  noexcept enables optimizations (especially in std::
        vector)" << std::endl;
    std::cout << " •  Compile-time checks have zero runtime cost" << std::endl
        ;

    std::cout << "\n
        ================================================================\n" <<
         std::endl;

    return 0;
}
```

25 Source Code: ErrorHandlingStroustrup.cpp

File: `src/ErrorHandlingStroustrup.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// ERROR HANDLING STRATEGIES - Following Bjarne Stroustrup's Guidance
// Based on "A Tour of C++", 3rd Edition, Chapter 4
// ==================================================================
//
// Three approaches to error handling:
// 1. RETURN VALUES - When function can't complete the task
// 2. EXCEPTIONS - When constructor can't establish invariants
// 3. TERMINATION - When continuing would be dangerous
//
// Key Principles:
// - Constructors cannot return values → must throw or terminate
// - Partially constructed objects are dangerous
// - RAII ensures proper cleanup even with exceptions
// - Return values for expected/recoverable errors
// - Exceptions for unexpected/exceptional situations
// - Terminate for programming errors/invariant violations
// ==================================================================

#include <iostream>
#include <string>
#include <vector>
#include <optional>
#include <memory>
#include <fstream>
#include <cstdlib>
#include <stdexcept>
#include <cassert>

// ==================================================================
// 1. RETURN VALUES - For Expected/Recoverable Errors
// ==================================================================

// Use case: Function can't complete task, but caller can handle it
// Appropriate when: Error is expected and can be recovered

class ConfigParser {
public:
    // Return std::optional - error is expected (file might not exist)
    static std::optional<std::string> read_config_value(const std::string& key) {
        std::cout << "[1.1] Reading config key: " << key << std::endl;

        // Simulate config lookup
        if (key == "server_url") {
            return "http://localhost:8080";
        }
        if (key == "timeout") {
            return "30";
```

```
        }

        // Key not found - this is EXPECTED, not exceptional
        std::cout << "      Config key '" << key << "' not found (expected scenario)" << std::endl;
        return std::nullopt;  // Caller can provide default or retry
    }

    // Return int with special value - traditional C-style
    static int parse_integer(const std::string& str, int default_value = -1) {
        std::cout << "[1.2] Parsing integer from: " << str << std::endl;

        try {
            return std::stoi(str);
        } catch (...) {
            // Parsing failed - return sentinel value
            std::cout << "      Parse failed, returning default: " << default_value << std::endl;
            return default_value;
        }
    }

    // Return bool with output parameter - when you need to return value too
    static bool try_connect(const std::string& url, int& out_status_code) {
        std::cout << "[1.3] Attempting connection to: " << url << std::endl;

        // Simulate connection attempt
        if (url.find("localhost") != std::string::npos) {
            out_status_code = 200;
            std::cout << "      Connected successfully, status: " << out_status_code << std::endl;
            return true;  // Success
        }

        out_status_code = 404;
        std::cout << "      Connection failed, status: " << out_status_code << std::endl;
        return false;  // Expected failure - caller can retry
    }
};

void demonstrate_return_values() {
    std::cout << "\n=== 1. RETURN VALUES - Expected/Recoverable Errors ===" << std::endl;
    std::cout << "Use when: Function can't complete, but caller can handle it" << std::endl;

    // Example 1: Optional value
    auto url = ConfigParser::read_config_value("server_url");
    if (url) {
        std::cout << "  Found URL: " << *url << std::endl;
    } else {
        std::cout << "   Using default URL" << std::endl;
    }
```

```

    // Missing key - caller provides default
    auto missing = ConfigParser::read_config_value("missing_key");
    std::string final_value = missing.value_or("default_value");
    std::cout << "  Using value: " << final_value << std::endl;

    // Example 2: Integer parsing with default
    int timeout = ConfigParser::parse_integer("abc", 10);
    std::cout << "  Timeout value: " << timeout << " seconds" << std::endl;

    // Example 3: Try pattern
    int status_code;
    if (ConfigParser::try_connect("http://invalid.url", status_code)) {
        std::cout << "  Connected" << std::endl;
    } else {
        std::cout << "   Connection failed with code: " << status_code << " (retry possible)" << std::endl;
    }

    std::cout << "\n When to use RETURN VALUES:" << std::endl;
    std::cout << " •   Error is EXPECTED (file not found, network timeout)" << std::endl;
    std::cout << " •   Caller can RECOVER (retry, use default, ask user)" << std::endl;
    std::cout << " •   Normal flow of control" << std::endl;
    std::cout << " •   Performance-critical code" << std::endl;
}

// ==================================================================
// 2. EXCEPTIONS - For Constructors and Unexpected Errors
// ==================================================================

// Problem: Partially constructed objects are dangerous!

class FileHandler_BAD {
    std::string filename;
    FILE* file_ptr;  // Raw pointer - dangerous!
    bool is_valid;   // Flag to track construction success

public:
    //  BAD: Constructor can't return error, uses flag instead
    FileHandler_BAD(const std::string& fname)
        : filename(fname), file_ptr(nullptr), is_valid(false) {

        std::cout << "\n[2.1 BAD] Attempting to open file: " << fname << std::endl;

        file_ptr = fopen(fname.c_str(), "r");
        if (file_ptr == nullptr) {
            //  PROBLEM: Can't return error from constructor
            // Object is partially constructed with invalid state!
            std::cout << "       File open failed, but object still created!" << std::endl;
            is_valid = false;  // Caller must check this flag
```

```
            return;
        }

        is_valid = true;
    }

    // Every member function must check is_valid!
    bool read_line(std::string& line) {
        if (!is_valid || !file_ptr) {
            return false;  // Undefined behavior if caller forgets to check!
        }
        // ... read logic
        return true;
    }

    ~FileHandler_BAD() {
        if (file_ptr) {
            fclose(file_ptr);
        }
    }

    bool valid() const { return is_valid; }
};

// Solution: Use RAII and throw from constructor

class FileHandler_GOOD {
    std::string filename;
    std::unique_ptr<FILE, decltype(&fclose)> file_ptr;  // RAII wrapper

public:
    //  GOOD: Constructor either succeeds completely or throws
    FileHandler_GOOD(const std::string& fname)
        : filename(fname), file_ptr(nullptr, &fclose) {

        std::cout << "\n[2.2 GOOD] Attempting to open file: " << fname << std
            ::endl;

        FILE* raw_ptr = fopen(fname.c_str(), "r");
        if (raw_ptr == nullptr) {
            //  THROW: Constructor can't establish invariant
            std::cout << "       Throwing exception - no partial object!" <<
                std::endl;
            throw std::runtime_error("Failed to open file: " + fname);
        }

        file_ptr.reset(raw_ptr);
        std::cout << "       File opened successfully, object fully
            constructed" << std::endl;

        // Invariant established: file_ptr is valid
        // No need for is_valid flag - if object exists, it's valid!
    }

```

```
    // No need to check validity - if object exists, it's valid
    std::string read_line() {
        // Can safely use file_ptr - invariant guaranteed by constructor
        char buffer[256];
        if (fgets(buffer, sizeof(buffer), file_ptr.get())) {
            return std::string(buffer);
        }
        return "";
    }

    // Destructor automatically called even if constructor throws
    // unique_ptr ensures file is closed
    ~FileHandler_GOOD() {
        std::cout << "      Closing file: " << filename << std::endl;
        // file_ptr's destructor automatically closes file
    }
};

// Another example: Vector with invariants

class BoundedVector_BAD {
    std::vector<int> data;
    size_t max_size;
    bool is_valid;

public:
    //  BAD: Doesn't enforce invariant in constructor
    BoundedVector_BAD(size_t max_sz, const std::vector<int>& initial)
        : max_size(max_sz), is_valid(false) {

        std::cout << "\n[2.3 BAD] Creating bounded vector, max_size="
                  << max_sz << ", initial.size()=" << initial.size() << std::endl;

        if (initial.size() > max_sz) {
            //  PROBLEM: Partial construction!
            std::cout << "       Initial size exceeds max, but object created anyway!" << std::endl;
            is_valid = false;
            return;
        }

        data = initial;
        is_valid = true;
    }

    bool push_back(int value) {
        if (!is_valid) return false;  // Must check everywhere!
        if (data.size() >= max_size) return false;
        data.push_back(value);
        return true;
    }

    bool valid() const { return is_valid; }
```

```
};

class BoundedVector_GOOD {
    std::vector<int> data;
    size_t max_size;

public:
    //  GOOD: Enforces invariant, throws if violated
    BoundedVector_GOOD(size_t max_sz, const std::vector<int>& initial)
        : max_size(max_sz) {

        std::cout << "\n[2.4 GOOD] Creating bounded vector, max_size="
                  << max_sz << ", initial.size()=" << initial.size() << std::endl;

        if (initial.size() > max_sz) {
            //  THROW: Invariant cannot be established
            std::cout << "       Throwing - initial size exceeds maximum!" << std::endl;
            throw std::invalid_argument(
                "Initial size " + std::to_string(initial.size()) +
                " exceeds maximum " + std::to_string(max_sz)
            );
        }

        data = initial;
        std::cout << "       Bounded vector created, invariant satisfied" << std::endl;

        // Invariant: data.size() <= max_size
        // Guaranteed for object's entire lifetime!
    }

    void push_back(int value) {
        // Invariant guaranteed - no need to check is_valid
        if (data.size() >= max_size) {
            throw std::length_error("Vector at maximum capacity");
        }
        data.push_back(value);
    }

    size_t size() const { return data.size(); }
};

// RAII Example: Exception safety with resource management

class ResourceManager {
    std::unique_ptr<int[]> buffer;
    FILE* log_file;
    bool initialized;

public:
    //  RAII: Resources acquired in constructor
    ResourceManager(size_t buffer_size, const std::string& log_path)
```

```
        : buffer(nullptr), log_file(nullptr), initialized(false) {

        std::cout << "\n[2.5] Constructing ResourceManager..." << std::endl;

        // Step 1: Allocate buffer (RAII with unique_ptr)
        buffer = std::make_unique<int[]>(buffer_size);
        std::cout << "        Buffer allocated" << std::endl;

        // Step 2: Open log file
        log_file = fopen(log_path.c_str(), "a");
        if (!log_file) {
            std::cout << "        Log file open failed - throwing..." << std::endl;
            // buffer automatically deallocated by unique_ptr destructor!
            throw std::runtime_error("Failed to open log file");
        }
        std::cout << "        Log file opened" << std::endl;

        initialized = true;
        std::cout << "        ResourceManager fully constructed" << std::endl;
    }

    ~ResourceManager() {
        std::cout << "        Cleaning up ResourceManager..." << std::endl;

        if (log_file) {
            fclose(log_file);
            std::cout << "        Log file closed" << std::endl;
        }

        // buffer automatically deallocated by unique_ptr
        std::cout << "        Buffer deallocated" << std::endl;
    }

    // Disable copy to prevent resource issues
    ResourceManager(const ResourceManager&) = delete;
    ResourceManager& operator=(const ResourceManager&) = delete;

    // Enable move for transfer of ownership
    ResourceManager(ResourceManager&& other) noexcept
        : buffer(std::move(other.buffer))
        , log_file(other.log_file)
        , initialized(other.initialized) {
        other.log_file = nullptr;
        other.initialized = false;
    }
};

void demonstrate_exceptions() {
    std::cout << "\n=== 2. EXCEPTIONS - Constructors and Invariants ===" << std::endl;
    std::cout << "Use when: Constructor can't establish object invariants" << std::endl;

```

```
    // Example 2.1: BAD - Partially constructed object
    std::cout << "\n--- Demonstrating BAD approach ---" << std::endl;
    {
        FileHandler_BAD bad_handler("nonexistent_file.txt");
        if (!bad_handler.valid()) {
            std::cout << "  Object exists but is invalid - DANGEROUS!" << std::endl;
            std::cout << "   Every function must check valid() flag" << std::endl;
        }
    }

    // Example 2.2: GOOD - Exception prevents partial construction
    std::cout << "\n--- Demonstrating GOOD approach ---" << std::endl;
    try {
        FileHandler_GOOD good_handler("nonexistent_file.txt");
        // This line never executes if file doesn't exist
        std::cout << "  Object constructed, guaranteed valid" << std::endl;
    } catch (const std::exception& e) {
        std::cout << "  Exception caught: " << e.what() << std::endl;
        std::cout << "   No object exists - no invalid state possible!" << std::endl;
    }

    // Example 2.3: BAD - Invariant violation
    std::cout << "\n--- Bounded Vector: BAD approach ---" << std::endl;
    {
        std::vector<int> too_large = {1, 2, 3, 4, 5};
        BoundedVector_BAD bad_vec(3, too_large);  // max_size=3, but initial has 5
        if (!bad_vec.valid()) {
            std::cout << "  Invalid object exists with violated invariant!" << std::endl;
        }
    }

    // Example 2.4: GOOD - Exception prevents invariant violation
    std::cout << "\n--- Bounded Vector: GOOD approach ---" << std::endl;
    try {
        std::vector<int> too_large = {1, 2, 3, 4, 5};
        BoundedVector_GOOD good_vec(3, too_large);  // Will throw
    } catch (const std::invalid_argument& e) {
        std::cout << "  Exception caught: " << e.what() << std::endl;
        std::cout << "   Invariant violation prevented!" << std::endl;
    }

    // Example 2.5: RAII ensures cleanup even with exceptions
    std::cout << "\n--- RAII with Exception ---" << std::endl;
    try {
        ResourceManager mgr(1024, "/invalid/path/log.txt");  // Will fail
    } catch (const std::exception& e) {
        std::cout << "  Exception caught: " << e.what() << std::endl;
        std::cout << "  Buffer was automatically cleaned up (RAII)!" << std::endl;
```

```
    }

    std::cout << "\n When to use EXCEPTIONS:" << std::endl;
    std::cout << " •   Constructor can't establish invariants" << std::endl;
    std::cout << " •   Partially constructed object would be dangerous" << std::endl;
    std::cout << " •   Error is UNEXPECTED (programming error, resource exhaustion)" << std::endl;
    std::cout << " •   Can't return error value (constructors, operators)" << std::endl;
    std::cout << " •   RAII ensures cleanup even if exception thrown" << std::endl;
}

// ==================================================================
// 3. TERMINATION - For Unrecoverable Errors
// ==================================================================

// Use case: Programming error or invariant violation where continuing is dangerous

class CriticalSystem {
    int* data_ptr;
    size_t size;

    // Internal invariant check
    void check_invariant() const {
        if (data_ptr == nullptr && size > 0) {
            // PROGRAMMING ERROR: Inconsistent state!
            std::cerr << "\n FATAL: Invariant violated - nullptr with size > 0" << std::endl;
            std::cerr << "   This should never happen - terminating!" << std::endl;
            std::terminate();  // Cannot continue safely
        }
    }

public:
    CriticalSystem(size_t sz) : data_ptr(nullptr), size(sz) {
        if (sz == 0) {
            std::cerr << "\n FATAL: Zero-size system not allowed" << std::endl;
            std::terminate();  // Precondition violation
        }

        data_ptr = new int[sz];
        std::cout << "[3.1] CriticalSystem constructed with size " << sz << std::endl;
    }

    void set_value(size_t index, int value) {
        check_invariant();  // Verify internal consistency

        if (index >= size) {
```

```
            // PROGRAMMING ERROR: Out of bounds access
            std::cerr << "\n FATAL: Index " << index << " out of bounds [0, "
                      << size << ")" << std::endl;
            std::cerr << "   This indicates a bug - terminating!" << std::endl;
            std::abort();  // Cannot recover from logic error
        }

        data_ptr[index] = value;
    }

    int get_value(size_t index) const {
        // Use assert for debug builds (removed in release with NDEBUG)
        assert(index < size && "Index out of bounds");

        if (index >= size) {
            std::cerr << "\n FATAL: Index out of bounds in release build" << std::endl;
            std::terminate();
        }

        return data_ptr[index];
    }

    ~CriticalSystem() {
        delete[] data_ptr;
    }
};

// Comparison: When to use each approach

class SafetyLevel {
public:
    enum class Level {
        INFO,       // Normal operation
        WARNING,    // Something unexpected but recoverable
        ERROR,      // Error that can be handled
        CRITICAL,   // Error requiring immediate attention
        FATAL       // Cannot continue
    };

    // Different error handling based on severity
    static void handle_error(Level level, const std::string& message) {
        switch (level) {
            case Level::INFO:
                // Just log - continue normally
                std::cout << "\n  INFO: " << message << std::endl;
                break;

            case Level::WARNING:
                // Log and return - caller handles it
                std::cout << "\n  WARNING: " << message << std::endl;
                // Return value or optional
                break;
```

```

            case Level::ERROR:
                // Throw exception - caller can catch and recover
                std::cout << "\n ERROR: " << message << std::endl;
                throw std::runtime_error(message);

            case Level::CRITICAL:
                // Log critical error, attempt emergency cleanup
                std::cerr << "\n CRITICAL: " << message << std::endl;
                std::cerr << "   Attempting emergency save..." << std::endl;
                // ... emergency cleanup ...
                throw std::runtime_error("Critical: " + message);

            case Level::FATAL:
                // Cannot continue - terminate immediately
                std::cerr << "\n FATAL: " << message << std::endl;
                std::cerr << "   Cannot continue safely - terminating!" << std
                    ::endl;
                std::terminate();
        }
    }
};

void demonstrate_termination() {
    std::cout << "\n=== 3. TERMINATION - Unrecoverable Errors ===" << std::
        endl;
    std::cout << "Use when: Continuing would be dangerous or impossible" <<
        std::endl;

    // Example: Normal operation
    std::cout << "\n--- Normal operation ---" << std::endl;
    {
        CriticalSystem sys(10);
        sys.set_value(0, 42);
        sys.set_value(5, 100);
        std::cout << "  Values set successfully" << std::endl;
    }

    // Example: Safety levels
    std::cout << "\n--- Different safety levels ---" << std::endl;

    // INFO - just logging
    SafetyLevel::handle_error(SafetyLevel::Level::INFO, "System started");

    // WARNING - caller can ignore or handle
    SafetyLevel::handle_error(SafetyLevel::Level::WARNING, "Cache miss (using
        fallback)");

    // ERROR - throw exception
    try {
        SafetyLevel::handle_error(SafetyLevel::Level::ERROR, "Network timeout"
            );
    } catch (const std::exception& e) {
        std::cout << "  Caught and recovered from ERROR: " << e.what() << std
```

```
            ::endl;
    }

    std::cout << "\n When to use TERMINATION:" << std::endl;
    std::cout << " •   PROGRAMMING ERROR detected (invariant violated)" << std::endl;
    std::cout << " •   Precondition violated (contract broken)" << std::endl;
    std::cout << " •   Continuing would corrupt data or cause undefined behavior" << std::endl;
    std::cout << " •   Use assert() in debug builds (removed in release)" << std::endl;
    std::cout << " •   Use std::terminate() or std::abort() for fatal errors" << std::endl;

    // Note: The examples below would terminate if uncommented
    std::cout << "\n  The following would terminate (commented out):" << std::endl;
    std::cout << "   // CriticalSystem sys(0);  // Zero size - would terminate" << std::endl;
    std::cout << "   // sys.set_value(999, 42);  // Out of bounds - would terminate" << std::endl;
}

// ================================================================
// SUMMARY: Decision Guide
// ================================================================

void print_decision_guide() {
    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  DECISION GUIDE: Which Error Handling Strategy?" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\n 1. RETURN VALUES (std::optional, bool, error codes)" << std::endl;
    std::cout << "     Use when:" << std::endl;
    std::cout << " •      Error is EXPECTED and part of normal operation" << std::endl;
    std::cout << " •      Caller can and should handle the error" << std::endl;
    std::cout << " •      Checking for availability (file exists, key found)" << std::endl;
    std::cout << " •      Performance-critical code" << std::endl;
    std::cout << " •      Parsing/validation where failure is common" << std::endl;
    std::cout << "\n     Examples:" << std::endl;
    std::cout << " •      std::optional<T> find_user(int id)" << std::endl;
    std::cout << " •      bool try_connect(const string& url, int& status)" << std::endl;
    std::cout << " •      std::expected<Value, Error> parse_config()" << std::
```

```
        endl;

    std::cout << "\n\n 2. EXCEPTIONS (throw/catch)" << std::endl;
    std::cout << "     Use when:" << std::endl;
    std::cout << " •      Constructor cannot establish invariants" << std::endl;
    std::cout << " •      Partially constructed object would be dangerous" << std::endl;
    std::cout << " •      Error is UNEXPECTED (resource exhaustion, logic error)" << std::endl;
    std::cout << " •      Cannot return error value (constructors, operators)" << std::endl;
    std::cout << " •      Need to skip multiple stack frames to handler" << std::endl;
    std::cout << " •      RAII ensures automatic cleanup" << std::endl;
    std::cout << "\n     Examples:" << std::endl;
    std::cout << " •      FileHandle(const string& path)  // Constructor" << std::endl;
    std::cout << " •      BoundedVector(size_t max, vector<T> initial)  // Invariant" << std::endl;
    std::cout << " •      operator[](size_t idx)  // Can't return error" << std::endl;

    std::cout << "\n\n 3. TERMINATION (std::terminate, std::abort, assert)" << std::endl;
    std::cout << "     Use when:" << std::endl;
    std::cout << " •      PROGRAMMING ERROR (bug in code)" << std::endl;
    std::cout << " •      Invariant violated (internal consistency broken)" << std::endl;
    std::cout << " •      Precondition not met (contract violated)" << std::endl;
    std::cout << " •      Continuing would cause data corruption" << std::endl;
    std::cout << " •      Undefined behavior would result" << std::endl;
    std::cout << "\n     Examples:" << std::endl;
    std::cout << " •      assert(ptr != nullptr);  // Debug check" << std::endl;
    std::cout << " •      if (invariant_broken()) std::terminate();" << std::endl;
    std::cout << " •      if (out_of_bounds) std::abort();" << std::endl;

    std::cout << "\n\n COMPARISON TABLE:" << std::endl;
    std::cout << "                                " << std::endl;
    std::cout << "  Aspect            Return Value   Exception      Termination" << std::endl;
    std::cout << "                                " << std::endl;
    std::cout << "  Error frequency   Common         Rare           Never*" << std::endl;
    std::cout << "  Recoverability    Recoverable    May recover    Fatal" << std::endl;
    std::cout << "  Caller control    Explicit       Can catch      None" << std::endl;
    std::cout << "  Performance       Fast           Slow           N/A" << std::endl;
```

```
    std::cout << "  Constructor use   No             Yes            Yes (rare)
          " << std::endl;
    std::cout << "  RAII cleanup      Manual         Automatic      No
                  " << std::endl;
    std::cout << "                                 " << std::endl;
    std::cout << "* \"Never\" = should never happen in correct program" << std
        ::endl;

    std::cout << "\n\n BJARNE'S KEY INSIGHTS:" << std::endl;
    std::cout << "\n    1  \"A constructor establishes the invariant for a
        class\"" << std::endl;
    std::cout << " →      Must throw if invariant cannot be established" <<
        std::endl;
    std::cout << " →      No partially constructed objects" << std::endl;

    std::cout << "\n    2  \"Use exceptions for exceptional circumstances\""
        << std::endl;
    std::cout << " →      Not for normal control flow" << std::endl;
    std::cout << " →      When caller can't be expected to check every call"
        << std::endl;

    std::cout << "\n    3  \"Use error codes when errors are expected\"" <<
        std::endl;
    std::cout << " →      File not found, network timeout, parse error" << std
        ::endl;
    std::cout << " →      Caller should handle explicitly" << std::endl;

    std::cout << "\n    4  \"Terminate when continuing would be dangerous\""
        << std::endl;
    std::cout << " →      Programming errors (bugs)" << std::endl;
    std::cout << " →      Invariant violations" << std::endl;

    std::cout << "\n    5  \"RAII is fundamental to exception safety\"" << std
        ::endl;
    std::cout << " →      Resources acquired in constructor" << std::endl;
    std::cout << " →      Released in destructor" << std::endl;
    std::cout << " →      Cleanup happens even with exceptions" << std::endl;

    std::cout << "\n
        ================================================================" <<
        std::endl;
}

// ==================================================================
// MAIN
// ==================================================================

int main() {
    std::cout << "\n
        ================================================================" <<
        std::endl;
    std::cout << "  ERROR HANDLING STRATEGIES" << std::endl;
    std::cout << "  Following Bjarne Stroustrup's Philosophy" << std::endl;
    std::cout << "
```

```
        ================================================================" <<
        std::endl;

    try {
        demonstrate_return_values();
        demonstrate_exceptions();
        demonstrate_termination();
        print_decision_guide();

        std::cout << "\n All demonstrations completed successfully!" << std::
            endl;

    } catch (const std::exception& e) {
        std::cerr << "\n Unexpected exception: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

26 Source Code: EventDrivenProgramming_Inheritance.cpp

File: `src/EventDrivenProgramming_Inheritance.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <algorithm>
#include <queue>

// ====================================================================
// EVENT-DRIVEN PROGRAMMING: TRADITIONAL INHERITANCE-BASED APPROACH
// ====================================================================
// This file demonstrates traditional event-driven programming using:
// - Abstract base classes
// - Virtual functions
// - Inheritance hierarchies
// - Classic OOP patterns
// ====================================================================

// ====================================================================
// 1. OBSERVER PATTERN (INHERITANCE-BASED)
// ====================================================================

// Abstract observer interface
class IObserver {
public:
    virtual ~IObserver() = default;
    virtual void on_notify(const std::string& event) = 0;
};

// Concrete observer implementations
class ConsoleObserver : public IObserver {
private:
    std::string name;

public:
    ConsoleObserver(std::string n) : name(std::move(n)) {}

    void on_notify(const std::string& event) override {
        std::cout << "    [" << name << "] Received: " << event << std::endl;
    }
};

class CountingObserver : public IObserver {
private:
    int count = 0;

public:
    void on_notify(const std::string& event) override {
        count++;
        std::cout << "    [Counter] Event #" << count << ": " << event << std
```

```
            ::endl;
    }

    int get_count() const { return count; }
};

class Subject {
private:
    std::vector<IObserver*> observers;  // Raw pointers or shared_ptr

public:
    void attach(IObserver* observer) {
        observers.push_back(observer);
    }

    void detach(IObserver* observer) {
        observers.erase(
            std::remove(observers.begin(), observers.end(), observer),
            observers.end()
        );
    }

    void notify(const std::string& event) {
        std::cout << "  [Subject] Notifying " << observers.size() << "
            observers" << std::endl;
        for (auto* observer : observers) {
            observer->on_notify(event);
        }
    }
};

void example_inheritance_observer() {
    std::cout << "\n=== 1. OBSERVER PATTERN (INHERITANCE-BASED) ===" << std::
        endl;

    Subject subject;

    // Need to create concrete observer objects
    ConsoleObserver obs1("Observer1");
    ConsoleObserver obs2("Observer2");
    CountingObserver counter;

    subject.attach(&obs1);
    subject.attach(&obs2);
    subject.attach(&counter);

    subject.notify("User logged in");
    subject.notify("Data updated");

    std::cout << "\n DISADVANTAGES:" << std::endl;
    std::cout << " •  Must define observer classes" << std::endl;
    std::cout << " •  Need inheritance hierarchy" << std::endl;
    std::cout << " •  More boilerplate code" << std::endl;
    std::cout << " •  Lifetime management issues (pointers)" << std::endl;
```

```
}

// ==================================================================
// 2. EVENT HIERARCHY (INHERITANCE-BASED)
// ==================================================================

// Abstract event base class
class Event {
public:
    virtual ~Event() = default;
    virtual std::string get_name() const = 0;
    virtual void print() const = 0;
};

class MouseClickEvent : public Event {
private:
    int x, y;
    std::string button;

public:
    MouseClickEvent(int x, int y, std::string btn)
        : x(x), y(y), button(std::move(btn)) {}

    std::string get_name() const override { return "MouseClick"; }

    void print() const override {
        std::cout << "    Mouse clicked: " << button
                  << " at (" << x << "," << y << ")" << std::endl;
    }

    int get_x() const { return x; }
    int get_y() const { return y; }
    const std::string& get_button() const { return button; }
};

class KeyPressEvent : public Event {
private:
    char key;
    bool ctrl;

public:
    KeyPressEvent(char k, bool c) : key(k), ctrl(c) {}

    std::string get_name() const override { return "KeyPress"; }

    void print() const override {
        std::cout << "    Key pressed: '" << key << "'"
                  << (ctrl ? " (Ctrl)" : "") << std::endl;
    }

    char get_key() const { return key; }
    bool has_ctrl() const { return ctrl; }
};

```

```cpp
// Abstract event handler
class IEventHandler {
public:
    virtual ~IEventHandler() = default;
    virtual void handle_event(const Event& event) = 0;
};

class EventLogger : public IEventHandler {
public:
    void handle_event(const Event& event) override {
        std::cout << "    [Logger] Handling " << event.get_name() << std::endl;
        event.print();
    }
};

class MouseHandler : public IEventHandler {
public:
    void handle_event(const Event& event) override {
        // Need to use dynamic_cast to access specific event data
        if (auto* mouse_event = dynamic_cast<const MouseClickEvent*>(&event)) {
            std::cout << "    [MouseHandler] Processing mouse at ("
                      << mouse_event->get_x() << "," << mouse_event->get_y() << ")" << std::endl;
        }
        // Ignore other event types
    }
};

class EventDispatcher {
private:
    std::vector<IEventHandler*> handlers;

public:
    void subscribe(IEventHandler* handler) {
        handlers.push_back(handler);
    }

    void dispatch(const Event& event) {
        for (auto* handler : handlers) {
            handler->handle_event(event);
        }
    }
};

void example_event_hierarchy() {
    std::cout << "\n=== 2. EVENT HIERARCHY (INHERITANCE-BASED) ===" << std::endl;

    EventDispatcher dispatcher;

    EventLogger logger;
    MouseHandler mouse_handler;
```

```

    dispatcher.subscribe(&logger);
    dispatcher.subscribe(&mouse_handler);

    std::cout << "\nDispatching events:" << std::endl;
    MouseClickEvent click(100, 200, "left");
    dispatcher.dispatch(click);

    KeyPressEvent key('A', true);
    dispatcher.dispatch(key);

    std::cout << "\n DISADVANTAGES:" << std::endl;
    std::cout << " •  Need event class hierarchy" << std::endl;
    std::cout << " •  Must use dynamic_cast (RTTI)" << std::endl;
    std::cout << " •  Pointer semantics required" << std::endl;
    std::cout << " •  More complex type checking" << std::endl;
}

// ================================================================
// 3. LISTENER PATTERN (CLASSIC GUI APPROACH)
// ================================================================

class IButtonListener {
public:
    virtual ~IButtonListener() = default;
    virtual void on_button_clicked(const std::string& button_id) = 0;
};

class Button {
private:
    std::string id;
    std::vector<IButtonListener*> listeners;

public:
    Button(std::string button_id) : id(std::move(button_id)) {}

    void add_listener(IButtonListener* listener) {
        listeners.push_back(listener);
    }

    void click() {
        std::cout << "  [Button '" << id << "'] Clicked!" << std::endl;
        for (auto* listener : listeners) {
            listener->on_button_clicked(id);
        }
    }
};

class SaveHandler : public IButtonListener {
public:
    void on_button_clicked(const std::string& button_id) override {
        std::cout << "    [SaveHandler] Saving file..." << std::endl;
    }
};
```

```

class LogHandler : public IButtonListener {
public:
    void on_button_clicked(const std::string& button_id) override {
        std::cout << "    [LogHandler] Logging click on " << button_id << std
            ::endl;
    }
};

void example_listener_pattern() {
    std::cout << "\n=== 3. LISTENER PATTERN (CLASSIC GUI) ===" << std::endl;

    Button save_btn("SaveButton");

    SaveHandler save_handler;
    LogHandler log_handler;

    save_btn.add_listener(&save_handler);
    save_btn.add_listener(&log_handler);

    save_btn.click();

    std::cout << "\n DISADVANTAGES:" << std::endl;
    std::cout << " •  Need listener interface classes" << std::endl;
    std::cout << " •  Can't define behavior inline" << std::endl;
    std::cout << " •  More classes to maintain" << std::endl;
    std::cout << " •  Harder to share state" << std::endl;
}

// ==================================================================
// 4. COMMAND PATTERN (INHERITANCE-BASED)
// ==================================================================

class ICommand {
public:
    virtual ~ICommand() = default;
    virtual void execute() = 0;
    virtual void undo() = 0;
    virtual std::string get_description() const = 0;
};

class TextCommand : public ICommand {
private:
    std::string& text_ref;
    std::string new_text;
    std::string old_text;

public:
    TextCommand(std::string& text, std::string new_val)
        : text_ref(text), new_text(std::move(new_val)), old_text(text) {}

    void execute() override {
        std::cout << "    Execute: Set text to '" << new_text << "'" << std::
            endl;
```

```
        text_ref = new_text;
    }

    void undo() override {
        std::cout << "    Undo: Restore text to '" << old_text << "'" << std::endl;
        text_ref = old_text;
    }

    std::string get_description() const override {
        return "Change text to '" + new_text + "'";
    }
};

class CommandManager {
private:
    std::vector<std::unique_ptr<ICommand>> history;
    size_t current = 0;

public:
    void execute(std::unique_ptr<ICommand> cmd) {
        cmd->execute();

        // Clear redo history
        history.erase(history.begin() + current, history.end());

        history.push_back(std::move(cmd));
        current = history.size();
    }

    void undo() {
        if (current > 0) {
            current--;
            history[current]->undo();
        }
    }

    void redo() {
        if (current < history.size()) {
            history[current]->execute();
            current++;
        }
    }
};

void example_command_pattern() {
    std::cout << "\n=== 4. COMMAND PATTERN (INHERITANCE-BASED) ===" << std::endl;

    std::string document = "Original";
    CommandManager manager;

    std::cout << "\nExecuting commands:" << std::endl;
    manager.execute(std::make_unique<TextCommand>(document, "Modified 1"));
```

```
    std::cout << "      Text: " << document << std::endl;

    manager.execute(std::make_unique<TextCommand>(document, "Modified 2"));
    std::cout << "      Text: " << document << std::endl;

    std::cout << "\nUndoing commands:" << std::endl;
    manager.undo();
    std::cout << "      Text: " << document << std::endl;

    manager.undo();
    std::cout << "      Text: " << document << std::endl;

    std::cout << "\n DISADVANTAGES:" << std::endl;
    std::cout << " •  Need command class for each operation" << std::endl;
    std::cout << " •  More verbose class definitions" << std::endl;
    std::cout << " •  State management in constructors" << std::endl;
    std::cout << " •  Can't define commands inline" << std::endl;
}

// ================================================================
// 5. CALLBACK INTERFACE (OLD-SCHOOL)
// ================================================================

class ICallback {
public:
    virtual ~ICallback() = default;
    virtual void on_success(int result) = 0;
    virtual void on_error(const std::string& error) = 0;
};

class AsyncOperation {
private:
    ICallback* callback = nullptr;

public:
    void set_callback(ICallback* cb) {
        callback = cb;
    }

    void execute(bool should_fail = false) {
        if (callback) {
            if (should_fail) {
                callback->on_error("Operation failed!");
            } else {
                callback->on_success(42);
            }
        }
    }
};

class MyCallback : public ICallback {
public:
    void on_success(int result) override {
        std::cout << "    Success! Result: " << result << std::endl;
```

```
    }

    void on_error(const std::string& error) override {
        std::cout << "    Error: " << error << std::endl;
    }
};

void example_callback_interface() {
    std::cout << "\n=== 5. CALLBACK INTERFACE (OLD-SCHOOL) ===" << std::endl;

    AsyncOperation op;
    MyCallback callback;

    op.set_callback(&callback);

    std::cout << "\n--- Success case ---" << std::endl;
    op.execute(false);

    std::cout << "\n--- Error case ---" << std::endl;
    op.execute(true);

    std::cout << "\ n DISADVANTAGES:" << std::endl;
    std::cout << " •  Must define callback class" << std::endl;
    std::cout << " •  Can't chain callbacks easily" << std::endl;
    std::cout << " •  Only one callback per operation" << std::endl;
    std::cout << " •  Less readable than fluent API" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n
        ==================================================================" <<
        std::endl;
    std::cout << "  EVENT-DRIVEN PROGRAMMING: TRADITIONAL INHERITANCE APPROACH
        " << std::endl;
    std::cout << "
        ==================================================================" <<
        std::endl;

    example_inheritance_observer();
    example_event_hierarchy();
    example_listener_pattern();
    example_command_pattern();
    example_callback_interface();

    std::cout << "\n
        ==================================================================" <<
        std::endl;
    std::cout << "  SUMMARY: PROBLEMS WITH INHERITANCE-BASED APPROACH" << std
        ::endl;
    std::cout << "
```

```
        ================================================================" << std::endl;

    std::cout << "\n DISADVANTAGES OF INHERITANCE-BASED APPROACH:" << std::endl;

    std::cout << "\n1. BOILERPLATE CODE" << std::endl;
    std::cout << " •   Must define interface classes (IObserver, IListener, etc.)" << std::endl;
    std::cout << " •   Must define concrete implementations" << std::endl;
    std::cout << " •   Each behavior needs a new class" << std::endl;
    std::cout << " •   Lots of virtual functions" << std::endl;

    std::cout << "\n2. INFLEXIBILITY" << std::endl;
    std::cout << " •   Can't define behavior inline" << std::endl;
    std::cout << " •   Hard to share state between callbacks" << std::endl;
    std::cout << " •   Difficult to compose behaviors" << std::endl;
    std::cout << " •   Rigid class hierarchies" << std::endl;

    std::cout << "\n3. LIFETIME MANAGEMENT" << std::endl;
    std::cout << " •   Raw pointers cause dangling reference bugs" << std::endl;
    std::cout << " •   shared_ptr adds complexity and overhead" << std::endl;
    std::cout << " •   Must manage attach/detach carefully" << std::endl;
    std::cout << " •   Who owns the observer objects?" << std::endl;

    std::cout << "\n4. POOR READABILITY" << std::endl;
    std::cout << " •   Behavior scattered across multiple files" << std::endl;
    std::cout << " •   Must jump between class definitions" << std::endl;
    std::cout << " •   Harder to see full picture" << std::endl;
    std::cout << " •   Intent hidden in class names" << std::endl;

    std::cout << "\n5. RTTI AND CASTING" << std::endl;
    std::cout << " •   Need dynamic_cast for event hierarchies" << std::endl;
    std::cout << " •   Runtime type checking (slower)" << std::endl;
    std::cout << " •   Can fail at runtime" << std::endl;
    std::cout << " •   Not as type-safe" << std::endl;

    std::cout << "\n6. MAINTENANCE BURDEN" << std::endl;
    std::cout << " •   More files to maintain" << std::endl;
    std::cout << " •   More classes in codebase" << std::endl;
    std::cout << " •   Harder to refactor" << std::endl;
    std::cout << " •   Changes ripple through hierarchy" << std::endl;

    std::cout << "\n WHEN INHERITANCE IS JUSTIFIED:" << std::endl;
    std::cout << " •   Polymorphic object collections" << std::endl;
    std::cout << " •   Complex state machines" << std::endl;
    std::cout << " •   Need object identity (not just behavior)" << std::endl;
    std::cout << " •   Multiple related virtual methods" << std::endl;
    std::cout << " •   Performance critical (no std::function overhead)" << std::endl;
    std::cout << " •   Plugin architectures" << std::endl;

    std::cout << "\n MODERN ALTERNATIVE:" << std::endl;
```

```
    std::cout << "   Use EventDrivenProgramming_Lambdas.cpp for:" << std::endl;
    std::cout << "     std::function + lambdas (no inheritance)" << std::endl;
    std::cout << "     std::variant (no event hierarchy)" << std::endl;
    std::cout << "     Inline behavior definitions" << std::endl;
    std::cout << "     Automatic state capture" << std::endl;
    std::cout << "     Much less boilerplate" << std::endl;

    std::cout << "\n COMPARISON:" << std::endl;
    std::cout << "   Inheritance approach: ~200 lines for basic observer" << std::endl;
    std::cout << "   Lambda approach:      ~50 lines for same functionality" << std::endl;
    std::cout << "   Code reduction:       75% less code!" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

27 Source Code: EventDrivenProgramming_Lambdas.cpp

File: `src/EventDrivenProgramming_Lambdas.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <functional>
#include <string>
#include <vector>
#include <map>
#include <memory>
#include <variant>
#include <algorithm>
#include <queue>
#include <typeindex>

// ==================================================================
// EVENT-DRIVEN PROGRAMMING: MODERN LAMBDA-BASED APPROACH
// ==================================================================
// This file demonstrates event-driven programming using:
// - std::function
// - Lambdas with captures
// - std::variant
// - No inheritance hierarchies needed!
// ==================================================================

// ==================================================================
// 1. OBSERVER PATTERN (LAMBDA-BASED)
// ==================================================================

class Subject {
private:
    std::vector<std::function<void(const std::string&)>> observers;

public:
    void subscribe(std::function<void(const std::string&)> observer) {
        observers.push_back(std::move(observer));
    }

    void notify(const std::string& event) {
        std::cout << "  [Subject] Notifying " << observers.size() << " observers" << std::endl;
        for (auto& observer : observers) {
            observer(event);
        }
    }
};

void example_lambda_observer() {
    std::cout << "\n=== 1. OBSERVER PATTERN (LAMBDA-BASED) ===" << std::endl;

    Subject subject;

    // No observer class needed! Just pass lambdas
```

```
    subject.subscribe([](const std::string& event) {
        std::cout << "    Lambda Observer 1: " << event << std::endl;
    });

    // Can capture state inline
    int count = 0;
    subject.subscribe([&count](const std::string& event) {
        std::cout << "    Lambda Observer 2 (count=" << ++count << "): " <<
            event << std::endl;
    });

    // Can define behavior at subscription time
    std::string prefix = "CustomPrefix";
    subject.subscribe([prefix](const std::string& event) {
        std::cout << "    [" << prefix << "] " << event << std::endl;
    });

    subject.notify("User logged in");
    subject.notify("Data updated");

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " •  No observer interface/base class needed" << std::endl;
    std::cout << " •  Inline behavior definition" << std::endl;
    std::cout << " •  Can capture local state" << std::endl;
    std::cout << " •  Less boilerplate code" << std::endl;
}

// ==================================================================
// 2. SIGNAL/SLOT PATTERN (MODERN)
// ==================================================================

template<typename... Args>
class Signal {
private:
    std::vector<std::function<void(Args...)>> slots;

public:
    void connect(std::function<void(Args...)> slot) {
        slots.push_back(std::move(slot));
    }

    void emit(Args... args) {
        for (auto& slot : slots) {
            slot(args...);
        }
    }

    void operator()(Args... args) { emit(args...); }
};

class Button {
public:
    Signal<const std::string&> clicked;
    Signal<int, int> positionChanged;
```

```

    std::string id;

    Button(std::string id) : id(std::move(id)) {}
};

void example_signal_slot() {
    std::cout << "\n=== 2. SIGNAL/SLOT PATTERN (MODERN) ===" << std::endl;

    Button btn("SaveButton");

    // Connect signals to lambdas (no slot classes needed!)
    btn.clicked.connect([](const std::string& id) {
        std::cout << "    Handler 1: Button " << id << " was clicked" << std::endl;
    });

    btn.clicked.connect([](const std::string& id) {
        std::cout << "    Handler 2: Logging click on " << id << std::endl;
    });

    btn.positionChanged.connect([](int x, int y) {
        std::cout << "    Position handler: Moved to (" << x << "," << y << ")" << std::endl;
    });

    std::cout << "\nTriggering signals:" << std::endl;
    btn.clicked.emit("SaveButton");
    btn.positionChanged.emit(150, 200);

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " • Type-safe (template-based)" << std::endl;
    std::cout << " • No slot base class needed" << std::endl;
    std::cout << " • Multiple handlers per signal" << std::endl;
    std::cout << " • Clean, declarative syntax" << std::endl;
}

// ================================================================
// 3. EVENT DISPATCHER WITH std::variant
// ================================================================

struct MouseClick { int x; int y; std::string button; };
struct KeyPress { char key; bool ctrl; bool shift; };
struct WindowResize { int width; int height; };

using UIEvent = std::variant<MouseClick, KeyPress, WindowResize>;

class EventDispatcher {
private:
    std::vector<std::function<void(const UIEvent&)>> handlers;

public:
    void subscribe(std::function<void(const UIEvent&)> handler) {
        handlers.push_back(std::move(handler));
```

```
    }

    void dispatch(const UIEvent& event) {
        for (auto& handler : handlers) {
            handler(event);
        }
    }
};

void example_variant_dispatcher() {
    std::cout << "\n=== 3. EVENT DISPATCHER WITH std::variant ===" << std::
        endl;

    EventDispatcher dispatcher;

    // Subscribe with lambda that uses std::visit
    dispatcher.subscribe([](const UIEvent& event) {
        std::visit([](const auto& e) {
            using T = std::decay_t<decltype(e)>;

            if constexpr (std::is_same_v<T, MouseClick>) {
                std::cout << "    Mouse clicked: " << e.button
                          << " at (" << e.x << "," << e.y << ")" << std::endl;
            } else if constexpr (std::is_same_v<T, KeyPress>) {
                std::cout << "    Key pressed: '" << e.key << "'"
                          << (e.ctrl ? " (Ctrl)" : "")
                          << (e.shift ? " (Shift)" : "") << std::endl;
            } else if constexpr (std::is_same_v<T, WindowResize>) {
                std::cout << "    Window resized: " << e.width << "x" << e.
                    height << std::endl;
            }
        }, event);
    });

    // Another handler that only cares about mouse clicks
    dispatcher.subscribe([](const UIEvent& event) {
        if (auto* click = std::get_if<MouseClick>(&event)) {
            std::cout << "    [Logger] Mouse event at (" << click->x << "," <<
                click->y << ")" << std::endl;
        }
    });

    std::cout << "\nDispatching events:" << std::endl;
    dispatcher.dispatch(MouseClick{100, 200, "left"});
    dispatcher.dispatch(KeyPress{'A', true, false});
    dispatcher.dispatch(WindowResize{1920, 1080});

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " •  No event hierarchy needed" << std::endl;
    std::cout << " •  Type-safe with std::variant" << std::endl;
    std::cout << " •  Compile-time dispatch with std::visit" << std::endl;
    std::cout << " •  Value semantics (no pointers)" << std::endl;
}
```

```
// ==================================================================
// 4. CALLBACK COMPOSITION (FLUENT API)
// ==================================================================

class AsyncTask {
private:
    std::function<void(int)> success_callback;
    std::function<void(const std::string&)> error_callback;
    std::function<void()> finally_callback;

public:
    AsyncTask& on_success(std::function<void(int)> callback) {
        success_callback = std::move(callback);
        return *this;
    }

    AsyncTask& on_error(std::function<void(const std::string&)> callback) {
        error_callback = std::move(callback);
        return *this;
    }

    AsyncTask& finally(std::function<void()> callback) {
        finally_callback = std::move(callback);
        return *this;
    }

    void execute(bool should_fail = false) {
        try {
            if (should_fail) {
                if (error_callback) error_callback("Task failed!");
            } else {
                if (success_callback) success_callback(42);
            }
        } catch (...) {
            if (error_callback) error_callback("Exception occurred");
        }

        if (finally_callback) finally_callback();
    }
};

void example_callback_composition() {
    std::cout << "\n=== 4. CALLBACK COMPOSITION (FLUENT API) ===" << std::endl
        ;

    std::cout << "\n--- Success case ---" << std::endl;
    AsyncTask task1;
    task1.on_success([](int result) {
            std::cout << "    Success! Result: " << result << std::endl;
        })
        .on_error([](const std::string& error) {
            std::cout << "    Error: " << error << std::endl;
        })
        .finally([]() {
```

```
            std::cout << "    Cleanup completed" << std::endl;
        })
        .execute(false);

    std::cout << "\n--- Error case ---" << std::endl;
    AsyncTask task2;
    task2.on_success([](int result) {
            std::cout << "    Success! Result: " << result << std::endl;
        })
        .on_error([](const std::string& error) {
            std::cout << "    Error: " << error << std::endl;
        })
        .finally([]() {
            std::cout << "    Cleanup completed" << std::endl;
        })
        .execute(true);

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " •  No callback interface classes" << std::endl;
    std::cout << " •  Readable, sequential setup" << std::endl;
    std::cout << " •  Similar to JavaScript Promises" << std::endl;
    std::cout << " •  Fluent API (method chaining)" << std::endl;
}

// ==================================================================
// 5. REACTIVE OBSERVABLE
// ==================================================================

template<typename T>
class Observable {
private:
    T value;
    std::vector<std::function<void(const T&)>> subscribers;

public:
    Observable(T initial) : value(std::move(initial)) {}

    void subscribe(std::function<void(const T&)> subscriber) {
        subscriber(value);  // Immediate notification
        subscribers.push_back(std::move(subscriber));
    }

    void set(T new_value) {
        if (value != new_value) {
            value = std::move(new_value);
            for (auto& subscriber : subscribers) {
                subscriber(value);
            }
        }
    }

    const T& get() const { return value; }
};

```

```
void example_reactive_observable() {
    std::cout << "\n=== 5. REACTIVE OBSERVABLE ===" << std::endl;

    Observable<int> counter(0);

    // Subscribe with lambdas
    counter.subscribe([](int value) {
        std::cout << "    Subscriber 1: Counter = " << value << std::endl;
    });

    counter.subscribe([](int value) {
        if (value > 0 && value % 2 == 0) {
            std::cout << "    Subscriber 2: Even number " << value << std::
                endl;
        }
    });

    std::cout << "\nUpdating counter:" << std::endl;
    counter.set(1);
    counter.set(2);
    counter.set(3);
    counter.set(4);

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " •  Automatic propagation" << std::endl;
    std::cout << " •  No observer base class" << std::endl;
    std::cout << " •  Declarative subscriptions" << std::endl;
    std::cout << " •  Great for UI data binding" << std::endl;
}

// ==================================================================
// 6. EVENT QUEUE WITH LAMBDAS
// ==================================================================

class EventQueue {
private:
    std::queue<std::function<void()>> events;

public:
    void post(std::function<void()> event) {
        events.push(std::move(event));
    }

    void process_all() {
        while (!events.empty()) {
            auto event = std::move(events.front());
            events.pop();
            event();
        }
    }

    size_t size() const { return events.size(); }
};

```

```
void example_event_queue() {
    std::cout << "\n=== 6. EVENT QUEUE WITH LAMBDAS ===" << std::endl;

    EventQueue queue;

    // Post events as lambdas (no event classes needed!)
    queue.post([]() {
        std::cout << "    Event 1: Initialize" << std::endl;
    });

    queue.post([]() {
        std::cout << "    Event 2: Load config" << std::endl;
    });

    int data = 42;
    queue.post([data]() {
        std::cout << "    Event 3: Process data: " << data << std::endl;
    });

    std::cout << "\nProcessing " << queue.size() << " events:" << std::endl;
    queue.process_all();

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " •  No event class hierarchy" << std::endl;
    std::cout << " •  Can capture context in closure" << std::endl;
    std::cout << " •  Extremely flexible" << std::endl;
    std::cout << " •  Minimal boilerplate" << std::endl;
}

// ==================================================================
// 7. COMMAND PATTERN WITH LAMBDAS
// ==================================================================

class Command {
private:
    std::function<void()> execute_func;
    std::function<void()> undo_func;
    std::string description;

public:
    Command(std::function<void()> exec, std::function<void()> undo, std::
        string desc)
        : execute_func(std::move(exec))
        , undo_func(std::move(undo))
        , description(std::move(desc)) {}

    void execute() {
        std::cout << "    Execute: " << description << std::endl;
        if (execute_func) execute_func();
    }

    void undo() {
        std::cout << "    Undo: " << description << std::endl;
        if (undo_func) undo_func();
```

```
    }
};

void example_lambda_command() {
    std::cout << "\n=== 7. COMMAND PATTERN WITH LAMBDAS ===" << std::endl;

    std::string text = "Original";
    std::vector<Command> history;

    // Create commands with lambdas (no command classes!)
    history.emplace_back(
        [&text]() { text = "Modified 1"; },
        [&text]() { text = "Original"; },
        "Change to 'Modified 1'"
    );

    history.emplace_back(
        [&text]() { text = "Modified 2"; },
        [&text]() { text = "Modified 1"; },
        "Change to 'Modified 2'"
    );

    std::cout << "\nExecuting commands:" << std::endl;
    for (auto& cmd : history) {
        cmd.execute();
        std::cout << "      Text: " << text << std::endl;
    }

    std::cout << "\nUndoing commands:" << std::endl;
    for (auto it = history.rbegin(); it != history.rend(); ++it) {
        it->undo();
        std::cout << "      Text: " << text << std::endl;
    }

    std::cout << "\n ADVANTAGES:" << std::endl;
    std::cout << " •  No command class hierarchy" << std::endl;
    std::cout << " •  Define behavior inline" << std::endl;
    std::cout << " •  Capture state automatically" << std::endl;
    std::cout << " •  Very concise" << std::endl;
}

// ================================================================
// MAIN FUNCTION
// ================================================================

int main() {
    std::cout << "\n
        ================================================================" <<
        std::endl;
    std::cout << "  EVENT-DRIVEN PROGRAMMING: MODERN LAMBDA-BASED APPROACH" <<
         std::endl;
    std::cout << "
        ================================================================" <<
        std::endl;
```

```

    example_lambda_observer();
    example_signal_slot();
    example_variant_dispatcher();
    example_callback_composition();
    example_reactive_observable();
    example_event_queue();
    example_lambda_command();

    std::cout << "\n
        ================================================================" <<
        std::endl;
    std::cout << "  SUMMARY: WHY LAMBDAS ARE BETTER" << std::endl;
    std::cout << "
        ================================================================" <<
        std::endl;

    std::cout << "\ n ADVANTAGES OF LAMBDA-BASED APPROACH:" << std::endl;
    std::cout << "\n1. LESS BOILERPLATE" << std::endl;
    std::cout << " •   No need to define observer/listener classes" << std::
        endl;
    std::cout << " •   No inheritance hierarchies" << std::endl;
    std::cout << " •   Behavior defined at point of use" << std::endl;

    std::cout << "\n2. MORE FLEXIBLE" << std::endl;
    std::cout << " •   Can capture local state [=], [&]" << std::endl;
    std::cout << " •   Different behavior per subscription" << std::endl;
    std::cout << " •   Easy to compose and chain" << std::endl;

    std::cout << "\n3. BETTER READABILITY" << std::endl;
    std::cout << " •   Callback logic near subscription point" << std::endl;
    std::cout << " •   No jumping between class definitions" << std::endl;
    std::cout << " •   Intent is clear and local" << std::endl;

    std::cout << "\n4. TYPE SAFETY" << std::endl;
    std::cout << " •   std::function provides type checking" << std::endl;
    std::cout << " •   Template parameters enforce signatures" << std::endl;
    std::cout << " •   Compile-time errors for mismatches" << std::endl;

    std::cout << "\n5. MODERN C++ FEATURES" << std::endl;
    std::cout << " •   std::function (C++11)" << std::endl;
    std::cout << " •   Lambda expressions (C++11)" << std::endl;
    std::cout << " •   std::variant (C++17)" << std::endl;
    std::cout << " •   if constexpr (C++17)" << std::endl;
    std::cout << " •   Move semantics" << std::endl;

    std::cout << "\ n THINGS TO WATCH OUT FOR:" << std::endl;
    std::cout << " •   Lambda lifetime: Don't capture [&] if object may be
        destroyed" << std::endl;
    std::cout << " •   std::function has overhead: Consider templates for hot
        paths" << std::endl;
    std::cout << " •   Circular references: Can cause memory leaks" << std::
        endl;
    std::cout << " •   Check if std::function is empty before calling" << std
```

```
        ::endl;

    std::cout << "\n WHEN TO USE INHERITANCE INSTEAD:" << std::endl;
    std::cout << " •   Need polymorphic object identity" << std::endl;
    std::cout << " •   Complex state management" << std::endl;
    std::cout << " •   Multiple virtual methods needed" << std::endl;
    std::cout << " •   Performance-critical (avoid std::function overhead)" <<
         std::endl;

    std::cout << "\n
        ================================================================\n" <<
         std::endl;

    return 0;
}
```

28 Source Code: ExceptionWithSourceLocation.cpp

File: `src/ExceptionWithSourceLocation.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <stdexcept>
#include <string>
#include <sstream>
#include <vector>
#include <memory>

// ====================================================================
// EXCEPTION WITH SOURCE LOCATION TRACKING
// ====================================================================
// Demonstrates how to capture and display source file name and line
// number when exceptions are thrown.
// Format: "ModuleName->Line #123: Error message"
// ====================================================================

// ====================================================================
// 1. BASIC EXCEPTION WITH SOURCE LOCATION
// ====================================================================

class SourceLocationException : public std::runtime_error {
private:
    std::string source_file;
    int source_line;
    std::string formatted_message;

    std::string format_message(const std::string& file, int line, const std::string& msg) {
        // Extract just the filename without path
        size_t last_slash = file.find_last_of("/\\");
        std::string filename = (last_slash != std::string::npos)
            ? file.substr(last_slash + 1)
            : file;

        std::ostringstream oss;
        oss << filename << "->Line #" << line << ": " << msg;
        return oss.str();
    }

public:
    SourceLocationException(const std::string& file, int line, const std::string& message)
        : std::runtime_error(format_message(file, line, message)),
          source_file(file),
          source_line(line),
          formatted_message(format_message(file, line, message)) {}

    const std::string& get_source_file() const noexcept { return source_file; }
    int get_source_line() const noexcept { return source_line; }
```

```
    const char* what() const noexcept override { return formatted_message.c_str(); }
};

// Macro to automatically capture __FILE__ and __LINE__
#define THROW_WITH_LOCATION(message) \
    throw SourceLocationException(__FILE__, __LINE__, message)

void demonstrate_basic_location() {
    std::cout << "\n=== 1. BASIC SOURCE LOCATION TRACKING ===" << std::endl;

    try {
        std::cout << "\nAbout to throw exception with location..." << std::endl;
        THROW_WITH_LOCATION("Something went wrong!");
    }
    catch (const SourceLocationException& e) {
        std::cout << "Caught exception: " << e.what() << std::endl;
        std::cout << "  Source file: " << e.get_source_file() << std::endl;
        std::cout << "  Source line: " << e.get_source_line() << std::endl;
    }
}

// ==================================================================
// 2. ENHANCED EXCEPTION WITH FUNCTION NAME
// ==================================================================

class DetailedException : public std::runtime_error {
private:
    std::string source_file;
    int source_line;
    std::string function_name;
    std::string error_message;
    std::string formatted_message;

    std::string format_message() {
        size_t last_slash = source_file.find_last_of("/\\");
        std::string filename = (last_slash != std::string::npos)
            ? source_file.substr(last_slash + 1)
            : source_file;

        std::ostringstream oss;
        oss << filename << "->Line #" << source_line;
        if (!function_name.empty()) {
            oss << " [" << function_name << "]";
        }
        oss << ": " << error_message;
        return oss.str();
    }

public:
    DetailedException(const std::string& file, int line, const std::string& function,
                      const std::string& message)
```

```
        : std::runtime_error(""),
          source_file(file),
          source_line(line),
          function_name(function),
          error_message(message) {
        formatted_message = format_message();
    }

    const std::string& get_source_file() const noexcept { return source_file; }
    int get_source_line() const noexcept { return source_line; }
    const std::string& get_function_name() const noexcept { return function_name; }
    const char* what() const noexcept override { return formatted_message.c_str(); }
};

// Macros for different detail levels
#define THROW_DETAILED(message) \
    throw DetailedException(__FILE__, __LINE__, __FUNCTION__, message)

#define THROW_DETAILED_CUSTOM(message, function) \
    throw DetailedException(__FILE__, __LINE__, function, message)

void some_function_that_fails() {
    THROW_DETAILED("Failed to process data");
}

void demonstrate_detailed_location() {
    std::cout << "\n=== 2. DETAILED EXCEPTION WITH FUNCTION NAME ===" << std::endl;

    try {
        std::cout << "\nCalling function that will throw..." << std::endl;
        some_function_that_fails();
    }
    catch (const DetailedException& e) {
        std::cout << "Caught exception: " << e.what() << std::endl;
        std::cout << "  Module: " << e.get_source_file() << std::endl;
        std::cout << "  Line: " << e.get_source_line() << std::endl;
        std::cout << "  Function: " << e.get_function_name() << std::endl;
    }
}

// ==================================================================
// 3. EXCEPTION HIERARCHY WITH SOURCE LOCATION
// ==================================================================

// Base exception with source location
class BaseLocationException : public std::runtime_error {
protected:
    std::string source_file;
    int source_line;
    std::string function_name;
```

```
    std::string error_type;
    std::string error_message;

    std::string format_message() const {
        size_t last_slash = source_file.find_last_of("/\\");
        std::string filename = (last_slash != std::string::npos)
            ? source_file.substr(last_slash + 1)
            : source_file;

        std::ostringstream oss;
        oss << "[" << error_type << "] ";
        oss << filename << "->Line #" << source_line;
        if (!function_name.empty()) {
            oss << " in " << function_name << "()";
        }
        oss << ": " << error_message;
        return oss.str();
    }

public:
    BaseLocationException(const std::string& file, int line, const std::string
        & function,
                          const std::string& type, const std::string& message)
        : std::runtime_error(message),
          source_file(file),
          source_line(line),
          function_name(function),
          error_type(type),
          error_message(message) {}

    virtual const char* what() const noexcept override {
        static std::string formatted;
        formatted = format_message();
        return formatted.c_str();
    }

    const std::string& get_source_file() const noexcept { return source_file;
        }
    int get_source_line() const noexcept { return source_line; }
    const std::string& get_function_name() const noexcept { return
        function_name; }
    const std::string& get_error_type() const noexcept { return error_type; }
};

// Specific exception types
class CameraException : public BaseLocationException {
public:
    CameraException(const std::string& file, int line, const std::string&
        function,
                    const std::string& message)
        : BaseLocationException(file, line, function, "CameraError", message)
            {}
};

```

```
class NetworkException : public BaseLocationException {
public:
    NetworkException(const std::string& file, int line, const std::string& function,
                     const std::string& message)
        : BaseLocationException(file, line, function, "NetworkError", message) {}
};

class ValidationException : public BaseLocationException {
public:
    ValidationException(const std::string& file, int line, const std::string& function,
                        const std::string& message)
        : BaseLocationException(file, line, function, "ValidationError", message) {}
};

// Macros for specific exception types
#define THROW_CAMERA_ERROR(message) \
    throw CameraException(__FILE__, __LINE__, __FUNCTION__, message)

#define THROW_NETWORK_ERROR(message) \
    throw NetworkException(__FILE__, __LINE__, __FUNCTION__, message)

#define THROW_VALIDATION_ERROR(message) \
    throw ValidationException(__FILE__, __LINE__, __FUNCTION__, message)

void camera_capture() {
    THROW_CAMERA_ERROR("Failed to capture image - device not responding");
}

void network_send_data() {
    THROW_NETWORK_ERROR("Connection timeout after 30 seconds");
}

void validate_input(int value) {
    if (value < 0) {
        THROW_VALIDATION_ERROR("Value must be non-negative");
    }
}

void demonstrate_exception_hierarchy() {
    std::cout << "\n=== 3. EXCEPTION HIERARCHY WITH SOURCE LOCATION ===" << std::endl;

    // Camera exception
    try {
        std::cout << "\nTesting camera exception..." << std::endl;
        camera_capture();
    }
    catch (const CameraException& e) {
        std::cout << "Caught: " << e.what() << std::endl;
    }
```

```

    // Network exception
    try {
        std::cout << "\nTesting network exception..." << std::endl;
        network_send_data();
    }
    catch (const NetworkException& e) {
        std::cout << "Caught: " << e.what() << std::endl;
    }

    // Validation exception
    try {
        std::cout << "\nTesting validation exception..." << std::endl;
        validate_input(-5);
    }
    catch (const ValidationException& e) {
        std::cout << "Caught: " << e.what() << std::endl;
    }
}

// ==================================================================
// 4. STACK TRACE SIMULATION
// ==================================================================

class StackTraceException : public std::runtime_error {
private:
    struct StackFrame {
        std::string file;
        int line;
        std::string function;

        std::string format() const {
            size_t last_slash = file.find_last_of("/\\");
            std::string filename = (last_slash != std::string::npos)
                ? file.substr(last_slash + 1)
                : file;
            return filename + "->Line #" + std::to_string(line) + " in " +
                function + "()";
        }
    };

    std::vector<StackFrame> stack_trace;
    std::string error_message;

    std::string format_full_message() const {
        std::ostringstream oss;
        oss << "Exception: " << error_message << "\n";
        oss << "Stack trace:\n";
        for (size_t i = 0; i < stack_trace.size(); ++i) {
            oss << "  #" << i << " " << stack_trace[i].format() << "\n";
        }
        return oss.str();
    }

```

```
public:
    StackTraceException(const std::string& message)
        : std::runtime_error(message), error_message(message) {}

    void add_frame(const std::string& file, int line, const std::string&
        function) {
        stack_trace.push_back({file, line, function});
    }

    const char* what() const noexcept override {
        static std::string formatted;
        formatted = format_full_message();
        return formatted.c_str();
    }

    const std::vector<StackFrame>& get_stack_trace() const { return
        stack_trace; }
};

// Macro to add current location to stack trace
#define ADD_STACK_FRAME(exception) \
    exception.add_frame(__FILE__, __LINE__, __FUNCTION__)

void level3_function() {
    throw StackTraceException("Critical error in deepest function");
}

void level2_function() {
    try {
        level3_function();
    }
    catch (StackTraceException& e) {
        ADD_STACK_FRAME(e);
        throw;  // Rethrow
    }
}

void level1_function() {
    try {
        level2_function();
    }
    catch (StackTraceException& e) {
        ADD_STACK_FRAME(e);
        throw;  // Rethrow
    }
}

void demonstrate_stack_trace() {
    std::cout << "\n=== 4. STACK TRACE SIMULATION ===" << std::endl;

    try {
        std::cout << "\nCalling nested functions..." << std::endl;
        level1_function();
    }
```

```
    catch (const StackTraceException& e) {
        std::cout << "\nCaught exception with stack trace:" << std::endl;
        std::cout << e.what() << std::endl;
    }
}

// ==================================================================
// 5. C++20 std::source_location (MODERN APPROACH)
// ==================================================================

#if __cpp_lib_source_location >= 201907L
#include <source_location>

class ModernLocationException : public std::runtime_error {
private:
    std::source_location location;
    std::string error_message;

    std::string format_message() const {
        std::string filename = location.file_name();
        size_t last_slash = filename.find_last_of("/\\");
        if (last_slash != std::string::npos) {
            filename = filename.substr(last_slash + 1);
        }

        std::ostringstream oss;
        oss << filename << "->Line #" << location.line()
            << " in " << location.function_name()
            << ": " << error_message;
        return oss.str();
    }

public:
    ModernLocationException(const std::string& message,
                            std::source_location loc = std::source_location::
                                current())
        : std::runtime_error(message),
          location(loc),
          error_message(message) {}

    const char* what() const noexcept override {
        static std::string formatted;
        formatted = format_message();
        return formatted.c_str();
    }

    const std::source_location& get_location() const { return location; }
};

void modern_function_that_fails() {
    throw ModernLocationException("Modern exception with automatic location
        capture");
}

```

```
void demonstrate_modern_location() {
    std::cout << "\n=== 5. C++20 std::source_location ===" << std::endl;
    std::cout << "Automatic location capture without macros!" << std::endl;

    try {
        std::cout << "\nThrowing modern exception..." << std::endl;
        modern_function_that_fails();
    }
    catch (const ModernLocationException& e) {
        std::cout << "Caught: " << e.what() << std::endl;
        auto loc = e.get_location();
        std::cout << "  File: " << loc.file_name() << std::endl;
        std::cout << "  Line: " << loc.line() << std::endl;
        std::cout << "  Column: " << loc.column() << std::endl;
        std::cout << "  Function: " << loc.function_name() << std::endl;
    }
}
#else
void demonstrate_modern_location() {
    std::cout << "\n=== 5. C++20 std::source_location ===" << std::endl;
    std::cout << "   std::source_location not available (requires C++20)" << std::endl;
    std::cout << "Compiler support needed: GCC 11+, Clang 16+, MSVC 2019+" << std::endl;
}
#endif

// ==================================================================
// 6. PRACTICAL EXAMPLE: IMAGE PROCESSING WITH LOCATION TRACKING
// ==================================================================

class ImageProcessingException : public std::runtime_error {
private:
    std::string module;
    int line;
    std::string operation;
    std::string details;

public:
    ImageProcessingException(const std::string& file, int ln,
                             const std::string& op, const std::string& det)
        : std::runtime_error(""),
          module(extract_filename(file)),
          line(ln),
          operation(op),
          details(det) {}

    const char* what() const noexcept override {
        static std::string msg;
        msg = module + "->Line #" + std::to_string(line) +
              " [" + operation + "]: " + details;
        return msg.c_str();
    }

```

```
    static std::string extract_filename(const std::string& path) {
        size_t pos = path.find_last_of("/\\");
        return (pos != std::string::npos) ? path.substr(pos + 1) : path;
    }
};

#define THROW_IMAGE_ERROR(operation, details) \
    throw ImageProcessingException(__FILE__, __LINE__, operation, details)

class ImageProcessor {
public:
    void load_image(const std::string& filename) {
        if (filename.empty()) {
            THROW_IMAGE_ERROR("LoadImage", "Filename cannot be empty");
        }
        // Simulate file not found
        THROW_IMAGE_ERROR("LoadImage", "File not found: " + filename);
    }

    void resize_image(int width, int height) {
        if (width <= 0 || height <= 0) {
            THROW_IMAGE_ERROR("ResizeImage",
                "Invalid dimensions: " + std::to_string(width) + "x" + std::
                    to_string(height));
        }
    }

    void apply_filter(const std::string& filter_name) {
        std::vector<std::string> valid_filters = {"blur", "sharpen", "
            edge_detect"};
        bool found = false;
        for (const auto& f : valid_filters) {
            if (f == filter_name) {
                found = true;
                break;
            }
        }
        if (!found) {
            THROW_IMAGE_ERROR("ApplyFilter", "Unknown filter: " + filter_name)
                ;
        }
    }
};

void demonstrate_practical_example() {
    std::cout << "\n=== 6. PRACTICAL EXAMPLE: IMAGE PROCESSING ===" << std::
        endl;

    ImageProcessor processor;

    // Test 1: Load with empty filename
    try {
        std::cout << "\nTest 1: Loading image with empty filename..." << std::
            endl;
```

```
        processor.load_image("");
    }
    catch (const ImageProcessingException& e) {
        std::cout << "  Error: " << e.what() << std::endl;
    }

    // Test 2: Invalid resize dimensions
    try {
        std::cout << "\nTest 2: Resizing with invalid dimensions..." << std::
            endl;
        processor.resize_image(-100, 200);
    }
    catch (const ImageProcessingException& e) {
        std::cout << "  Error: " << e.what() << std::endl;
    }

    // Test 3: Unknown filter
    try {
        std::cout << "\nTest 3: Applying unknown filter..." << std::endl;
        processor.apply_filter("invalid_filter");
    }
    catch (const ImageProcessingException& e) {
        std::cout << "  Error: " << e.what() << std::endl;
    }

    // Test 4: File not found
    try {
        std::cout << "\nTest 4: Loading non-existent file..." << std::endl;
        processor.load_image("nonexistent_image.png");
    }
    catch (const ImageProcessingException& e) {
        std::cout << "  Error: " << e.what() << std::endl;
    }
}

// ==================================================================
// 7. DEBUGGING HELPER: EXCEPTION LOGGING
// ==================================================================

class LoggedException : public std::runtime_error {
private:
    std::string module;
    int line;
    std::string function;
    std::string message;

public:
    LoggedException(const std::string& file, int ln, const std::string& func,
                    const std::string& msg)
        : std::runtime_error(msg),
          module(extract_filename(file)),
          line(ln),
          function(func),
          message(msg) {
```

```
        log_exception();
    }

    void log_exception() const {
        std::cerr << "\                              n" << std::endl;
        std::cerr << "  EXCEPTION THROWN" << std::endl;
        std::cerr << "                              " << std::endl;
        std::cerr << "  Module:   " << module << std::endl;
        std::cerr << "  Line:     #" << line << std::endl;
        std::cerr << "  Function: " << function << "()" << std::endl;
        std::cerr << "  Message:  " << message << std::endl;
        std::cerr << "                              " << std::endl;
    }

    const char* what() const noexcept override {
        static std::string formatted;
        formatted = module + "->Line #" + std::to_string(line) + ": " +
            message;
        return formatted.c_str();
    }

    static std::string extract_filename(const std::string& path) {
        size_t pos = path.find_last_of("/\\");
        return (pos != std::string::npos) ? path.substr(pos + 1) : path;
    }
};

#define THROW_AND_LOG(message) \
    throw LoggedException(__FILE__, __LINE__, __FUNCTION__, message)

void function_with_logging() {
    THROW_AND_LOG("Critical error detected - system state corrupted");
}

void demonstrate_logging() {
    std::cout << "\n=== 7. EXCEPTION LOGGING ===" << std::endl;

    try {
        std::cout << "\nCalling function that logs exception details..." <<
            std::endl;
        function_with_logging();
    }
    catch (const LoggedException& e) {
        std::cout << "\nException caught in main handler" << std::endl;
        std::cout << "Formatted: " << e.what() << std::endl;
    }
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n
```

```
        ================================================================" << std::endl;
    std::cout << "  EXCEPTION SOURCE LOCATION TRACKING" << std::endl;
    std::cout << "
        ================================================================" << std::endl;
    std::cout << "Format: ModuleName->Line #XXX: Error message" << std::endl;

    demonstrate_basic_location();
    demonstrate_detailed_location();
    demonstrate_exception_hierarchy();
    demonstrate_stack_trace();
    demonstrate_modern_location();
    demonstrate_practical_example();
    demonstrate_logging();

    std::cout << "\n
        ================================================================" << std::endl;
    std::cout << "  IMPLEMENTATION TECHNIQUES" << std::endl;
    std::cout << "
        ================================================================" << std::endl;

    std::cout << "\n1  PREPROCESSOR MACROS (C++11/14/17):" << std::endl;
    std::cout << "   __FILE__      - Source file path" << std::endl;
    std::cout << "   __LINE__      - Line number" << std::endl;
    std::cout << "   __FUNCTION__ - Function name (compiler extension)" << std::endl;
    std::cout << "\n   Usage:" << std::endl;
    std::cout << "   #define THROW_WITH_LOC(msg) \\" << std::endl;
    std::cout << "       throw MyException(__FILE__, __LINE__, msg)" << std::endl;

    std::cout << "\n2  std::source_location (C++20):" << std::endl;
    std::cout << " •   Automatic location capture" << std::endl;
    std::cout << " •   No macros needed" << std::endl;
    std::cout << " •   Default parameter trick" << std::endl;
    std::cout << "\n   Usage:" << std::endl;
    std::cout << "   throw MyException(message, std::source_location::current());" << std::endl;

    std::cout << "\n3  CUSTOM EXCEPTION CLASS:" << std::endl;
    std::cout << "   class MyException : public std::runtime_error {" << std::endl;
    std::cout << "       std::string file;" << std::endl;
    std::cout << "       int line;" << std::endl;
    std::cout << "   public:" << std::endl;
    std::cout << "       MyException(const string& f, int l, const string& msg);" << std::endl;
    std::cout << "       const char* what() const noexcept override;" << std::endl;
    std::cout << "   };" << std::endl;

```

```
    std::cout << "\n4  MESSAGE FORMATTING:" << std::endl;
    std::cout << "   Format: \"ModuleName->Line #123: Error message\"" << std::endl;
    std::cout << " •   Extract filename from path" << std::endl;
    std::cout << " •   Use std::ostringstream for formatting" << std::endl;
    std::cout << " •   Store formatted message in exception" << std::endl;

    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  BEST PRACTICES" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\n DO:" << std::endl;
    std::cout << " •   Use macros to automatically capture location" << std::endl;
    std::cout << " •   Extract filename only (not full path)" << std::endl;
    std::cout << " •   Include function name for better context" << std::endl;
    std::cout << " •   Format consistently: Module->Line #XXX" << std::endl;
    std::cout << " •   Store location info in exception members" << std::endl;
    std::cout << " •   Use std::source_location in C++20 (no macros!)" << std::endl;

    std::cout << "\n DON'T:" << std::endl;
    std::cout << " •   Don't include full file paths (too verbose)" << std::endl;
    std::cout << " •   Don't hardcode file/line (will be wrong)" << std::endl;
    std::cout << " •   Don't forget to extract filename from path" << std::endl;
    std::cout << " •   Don't use exceptions for control flow" << std::endl;

    std::cout << "\n COMPARISON:" << std::endl;
    std::cout << "\nMacro Approach (C++11-17):" << std::endl;
    std::cout << "     Works with older compilers" << std::endl;
    std::cout << "     Full control over formatting" << std::endl;
    std::cout << "     Requires macros (ugly)" << std::endl;
    std::cout << "     Manual __FILE__, __LINE__ passing" << std::endl;

    std::cout << "\nstd::source_location (C++20):" << std::endl;
    std::cout << "     No macros needed" << std::endl;
    std::cout << "     Automatic capture via default parameter" << std::endl;
    std::cout << "     Type-safe and elegant" << std::endl;
    std::cout << "     Requires C++20 compiler support" << std::endl;

    std::cout << "\n REAL-WORLD USAGE:" << std::endl;
    std::cout << "\nDefensive Programming:" << std::endl;
    std::cout << "   if (!validate(data)) {" << std::endl;
    std::cout << "       THROW_WITH_LOCATION(\"Invalid data format\");" << std::endl;
    std::cout << "   }" << std::endl;

    std::cout << "\nResource Management:" << std::endl;
```

```
    std::cout << "    if (!file.open()) {" << std::endl;
    std::cout << "        THROW_WITH_LOCATION(\"Failed to open: \" + filename);" << std::endl;
    std::cout << "    }" << std::endl;

    std::cout << "\nAPI Boundary Checking:" << std::endl;
    std::cout << "    if (index >= size()) {" << std::endl;
    std::cout << "        THROW_WITH_LOCATION(\"Index out of range\");" << std::endl;
    std::cout << "    }" << std::endl;

    std::cout << "\n==================================================================\n" << std::endl;

    return 0;
}
```

29 Source Code: FindCountOfCommonNodes.cpp

File: `src/FindCountOfCommonNodes.cpp`
Repository: View on GitHub

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

using namespace std;

struct ListNode {
       int val;
       ListNode *next;
       ListNode(int x = 0) : val(x), next(NULL) {}
};

class Solution {
public:
    void PrintList(ListNode *pListNode) {
         if (pListNode != NULL) {
            cout << pListNode->val << endl;
            PrintList(pListNode->next);
         } else {
            cout << "NULL\n" << endl;
         }
    }

    int getLengthOfList(const ListNode *pListNode) {
        int length = 0;

        while (pListNode != NULL) {
              pListNode = pListNode->next;
              length ++;
        }

        return length;
    }

    int FindNumberOfCommonNodes(ListNode *pHead1, ListNode *pHead2) {

        ListNode* current1 = pHead1;
        ListNode* current2 = pHead2;

        int count = 0;

        // traverse list A till the end of list
        while (current1 != NULL) {

            // traverse list B till the end of list
            while (current2 != NULL) {

            // if data is match then count increase
            if (current1->val == current2->val) {
```

```
                count++;
            }

            // increase current pointer for next node
            current2 = current2->next;
        }

        // increase current pointer of first list
        current1 = current1->next;

        // initialize starting point for second list
        current2 = pHead2;
        }

        return count;
    }
};

int main () {
    Solution testSolution;
    std::array<int, 12> intArray1 {{1, 3, 4, 6, 7, 10, 12, 13, 14, 15, 16,
        17}};
    std::array<int, 10> intArray2 = {2, 4, 5, 6, 7, 8, 11, 13 , 14, 16};

    ListNode *pMHead = NULL;
    ListNode *pCurrent   = NULL;
    ListNode *pNHead = NULL;
    auto i = 0;

    for (i = 0; i < intArray1.size(); i ++) {
        ListNode *pTemp = new ListNode(intArray1[i]);

        if (i == 0) {
            pMHead = pCurrent = pTemp;
        } else {
            pCurrent->next = pTemp;
            pCurrent = pCurrent->next; // pCurrent->next == pTemp
        }
    }

    for (i = 0; i < intArray2.size(); i ++) {
        ListNode *pTemp = new ListNode(intArray2[i]);

        if (i == 0) {
            pNHead = pCurrent = pTemp;
        } else {
            pCurrent->next = pTemp;
            pCurrent = pCurrent->next; // pCurrent->next == pTemp
        }
    }

    // Print Linkedlist
    cout << "PrintList(pMHead)" << endl;
    testSolution.PrintList(pMHead);
```

```

    cout << "PrintList(pNHead)" << endl;
    testSolution.PrintList(pNHead);

    int commonNodesCount = testSolution.FindNumberOfCommonNodes(pMHead, pNHead
        );


    cout << "commonNodesCount = " << commonNodesCount << endl;


    return 0;
}
```

30 Source Code: FindFirstCommonNode.cpp

File: `src/FindFirstCommonNode.cpp`
Repository: View on GitHub

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

using namespace std;

struct ListNode {
       int val;
       ListNode *next;
       ListNode(int x = 0) : val(x), next(NULL) {}
};

class Solution {
public:
    void PrintList(ListNode *pListNode) {
         if (pListNode != NULL) {
            cout << pListNode->val << endl;
            PrintList(pListNode->next);
         } else {
            cout << "NULL\n" << endl;
         }
    }

    int getLengthOfList(const ListNode *pListNode) {
        int length = 0;

        while (pListNode != NULL) {
              pListNode = pListNode->next;
              length ++;
        }

        return length;
    }

    ListNode* FindFirstCommonNode(ListNode *pHead1, ListNode *pHead2) {

        ListNode* current1 = pHead1;
        ListNode* current2 = pHead2;

        // traverse list A till the end of list
        while (current1 != NULL) {

            // traverse list B till the end of list
            while (current2 != NULL) {

            // if data is match then count increase
            if (current1->val == current2->val) {

                return current1;
```

```
            }

            // increase current pointer for next node
            current2 = current2->next;
        }

        // increase current pointer of first list
        current1 = current1->next;

        // initialize starting point for second list
        current2 = pHead2;
        }
    }
};

int main () {
    Solution testSolution;
    std::array<int, 12> intArray1 {{1, 3, 4, 6, 7, 10, 12, 13, 14, 15, 16,
        17}};
    std::array<int, 10> intArray2 = {2, 4, 5, 6, 7, 8, 11, 13 , 14, 16};

    ListNode *pMHead = NULL;
    ListNode *pCurrent   = NULL;
    ListNode *pNHead = NULL;
    auto i = 0;

    for (i = 0; i < intArray1.size(); i ++) {
        ListNode *pTemp = new ListNode(intArray1[i]);

        if (i == 0) {
            pMHead = pCurrent = pTemp;
        } else {
            pCurrent->next = pTemp;
            pCurrent = pCurrent->next; // pCurrent->next == pTemp
        }
    }

    for (i = 0; i < intArray2.size(); i ++) {
        ListNode *pTemp = new ListNode(intArray2[i]);

        if (i == 0) {
            pNHead = pCurrent = pTemp;
        } else {
            pCurrent->next = pTemp;
            pCurrent = pCurrent->next; // pCurrent->next == pTemp
        }
    }

    // Print Linkedlist
    cout << "PrintList(pMHead)" << endl;
    testSolution.PrintList(pMHead);

    cout << "PrintList(pNHead)" << endl;
    testSolution.PrintList(pNHead);
```

```

    ListNode *pTarget = testSolution.FindFirstCommonNode(pMHead, pNHead);

    // Print Returned Node
    if (pTarget != NULL)
       cout << "pTarget->val = " << pTarget->val << endl;
    cout << endl;

    return 0;
}
```

31 Source Code: FindMToLastElement.cpp

File: `src/FindMToLastElement.cpp`
Repository: View on GitHub

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

using namespace std;

struct ListNode {
       int val;
       ListNode *next;
       ListNode(int x = 0) : val(x), next(NULL) {}
};

class Solution {
public:
    void PrintList(ListNode *pListNode) {
         if (pListNode != NULL) {
            cout << pListNode->val << endl;
            PrintList(pListNode->next);
         } else {
            cout << "NULL\n" << endl;
         }
    }

    int getLengthOfList(const ListNode *pListNode) {
        int length = 0;

        while (pListNode != NULL) {
              pListNode = pListNode->next;
              length ++;
        }

        return length;
    }

    ListNode* FindMthToLastElement(ListNode *pHead1, int valueM) {

        ListNode* current = pHead1;

        // traverse list A till the end of list
        while (current != NULL) {

            // if data is match then count increase
            if (current->val == valueM) {
                return current;
            }


            // increase current pointer of first list
            current = current->next;
```

```
        }

         return NULL;
    }
};

int main () {
    Solution testSolution;
    std::array<int, 12> intArray1 {{1, 3, 4, 6, 7, 10, 12, 13, 14, 15, 16,
        17}};

    ListNode *pMHead = NULL;
    ListNode *pCurrent   = NULL;
    ListNode *pNHead = NULL;
    auto i = 0;

    for (i = 0; i < intArray1.size(); i ++) {
        ListNode *pTemp = new ListNode(intArray1[i]);

        if (i == 0) {
            pMHead = pCurrent = pTemp;
        } else {
            pCurrent->next = pTemp;
            pCurrent = pCurrent->next; // pCurrent->next == pTemp
        }
    }

    // Print Linkedlist
    cout << "PrintList(pMHead)" << endl;
    testSolution.PrintList(pMHead);

    int valueOfM = 4;

    ListNode *pTarget = testSolution.FindMthToLastElement(pMHead, valueOfM);

    // Print Returned Node
    if (pTarget != NULL) {
       testSolution.PrintList(pMHead);
    } else {
        cout << "There is no element that has value equal to " << valueOfM <<
            endl;
    }

    valueOfM = 5;

    pTarget = testSolution.FindMthToLastElement(pMHead, valueOfM);

    // Print Returned Node
    if (pTarget != NULL) {
       testSolution.PrintList(pMHead);
    } else {
        cout << "There is no element that has value equal to " << valueOfM <<
            endl;
    }
```

```


    return 0;
}
```

32 Source Code: FindMaxNoOfConsecutiveOnesFromIntArray.cpp

File: `src/FindMaxNoOfConsecutiveOnesFromIntArray.cpp`
Repository: View on GitHub

```cpp
/*
Instructions:

Write a function that returns the maximum number of consecutive 'ones' in an integer array of length N.

You can assume the input is an array containing integers 0 and 1 only.

For example, for the following input the function should return 7.

    0101011110110111011111000000010

This can be written in pseudo-code, Java, or C/C++. Syntax mistakes are not important.
*/

// Example program
#include <iostream>
#include <string>
#include <array>
#include <algorithm>

int main()
{
    int cArray[] = {0,1,0,0,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,0};

    const int sizeArray = sizeof(cArray)/sizeof(cArray[0]);

    std::array<int, sizeArray> cppArray;

    std::move(std::begin(cArray), std::end(cArray), cppArray.begin());

    auto maxiConsecutiveOnes = 0;
    auto runningMaximumOnes= 0;

    for(const auto& i : cppArray) {

      if(i == 1) {  // Fixed: use i directly, not cppArray[i]
          runningMaximumOnes++;
      } else {

          if(maxiConsecutiveOnes < runningMaximumOnes) {
              maxiConsecutiveOnes = runningMaximumOnes;
          }
          runningMaximumOnes = 0;
      }
   }
```

```

    std::cout << " maximum consecutive ones = " << maxiConsecutiveOnes << std::endl;
}
```

33 Source Code: FragileBaseClass.cpp

File: `src/FragileBaseClass.cpp`
Repository: View on GitHub

```cpp
// FragileBaseClass.cpp
// Demonstrates the Fragile Base Class Problem in C++ and its solutions
//
// The Fragile Base Class problem occurs when changes to a base class,
// which seem safe in isolation, inadvertently break the functionality
// of its derived (child) classes.
//
// KEY CONCEPTS:
// 1. Changes in Method Calls - base class refactoring breaks derived classes
// 2. Altering Data Layout - ABI compatibility issues
// 3. Virtual Function Overriding - unexpected behavior changes
//
// SOLUTIONS:
// 1. Composition over Inheritance
// 2. Hide Implementation Details (Encapsulation)
// 3. Use 'final' keyword
// 4. Design for Inheritance Explicitly
// 5. Prefer Interfaces/Abstract Classes

#include <iostream>
#include <string>
#include <vector>
#include <memory>

// ==========================================================================
// SECTION 1: THE PROBLEM - Fragile Base Class
// ==========================================================================

namespace fragile_example {

// Version 1: Original base class
class Counter_V1 {
protected:
    int count_ = 0;

public:
    virtual void add(int value) {
        count_ += value;
        std::cout << "    [Base] Added " << value << ", count = " << count_ << "\n";
    }

    void addMultiple(const std::vector<int>& values) {
        for (int val : values) {
            count_ += val;  // Direct implementation
```

```
        }
        std::cout << "    [Base] Added multiple values, count = " << count_ << "\n";
    }

    int getCount() const { return count_; }
};

// Derived class works fine with V1
class LoggingCounter_V1 : public Counter_V1 {
private:
    std::vector<std::string> log_;

public:
    void add(int value) override {
        log_.push_back("Adding " + std::to_string(value));
        Counter_V1::add(value);
    }

    void printLog() const {
        std::cout << "    Log entries: " << log_.size() << "\n";
        for (const auto& entry : log_) {
            std::cout << "      - " << entry << "\n";
        }
    }
};

// ============================================================================
// Now developer "optimizes" the base class by refactoring
// ============================================================================

class Counter_V2 {
protected:
    int count_ = 0;

public:
    virtual void add(int value) {
        count_ += value;
        std::cout << "    [Base] Added " << value << ", count = " << count_ << "\n";
    }

    // FRAGILE CHANGE: Refactored to use add() for "code reuse"
    void addMultiple(const std::vector<int>& values) {
        for (int val : values) {
            add(val);  // Now calls virtual add()!
        }
        std::cout << "    [Base] Added multiple values, count = " << count_ << "\n";
    }
```

```

    int getCount() const { return count_; }
};

// Same derived class, now BROKEN!
class LoggingCounter_V2 : public Counter_V2 {
private:
    std::vector<std::string> log_;

public:
    void add(int value) override {
        log_.push_back("Adding " + std::to_string(value));
        Counter_V2::add(value);
    }

    void printLog() const {
        std::cout << "    Log entries: " << log_.size() << "\n";
        for (const auto& entry : log_) {
            std::cout << "      - " << entry << "\n";
        }
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1: The Fragile Base Class Problem ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Scenario: Developer refactors base class to call virtual
        method\n\n";

    std::cout << "1. Original Version (V1) - Works as expected:\n";
    LoggingCounter_V1 counter1;
    counter1.addMultiple({1, 2, 3});
    counter1.printLog();
    std::cout << "   Expected: No log entries (addMultiple bypasses add())\n";
    std::cout << "   Actual: " << (counter1.printLog(), "  Works!\n");

    std::cout << "\n2. After Refactoring (V2) - BROKEN:\n";
    LoggingCounter_V2 counter2;
    counter2.addMultiple({1, 2, 3});
    counter2.printLog();
    std::cout << "   Problem: Now creates log entries unexpectedly!\n";
    std::cout << "   The derived class behavior changed without any
        modification!\n";
}

} // namespace fragile_example

//
    ==========================================================================
// SECTION 2: SOLUTION 1 - Composition Over Inheritance
//
```

```
    ==========================================================================

namespace composition_solution {

// Internal implementation (not exposed for inheritance)
class CounterImpl {
private:
    int count_ = 0;

public:
    void add(int value) {
        count_ += value;
    }

    void addMultiple(const std::vector<int>& values) {
        for (int val : values) {
            add(val);  // Safe - not virtual
        }
    }

    int getCount() const { return count_; }
};

// Use composition instead of inheritance
class LoggingCounter {
private:
    CounterImpl counter_;  // Composition
    std::vector<std::string> log_;

public:
    void add(int value) {
        log_.push_back("Adding " + std::to_string(value));
        counter_.add(value);
    }

    void addMultiple(const std::vector<int>& values) {
        for (int val : values) {
            counter_.add(val);  // Direct delegation, no logging
        }
    }

    int getCount() const { return counter_.getCount(); }

    void printLog() const {
        std::cout << "    Log entries: " << log_.size() << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 2: Solution 1 - Composition Over Inheritance
        ===\n";
    std::cout << std::string(70, '=') << "\n\n";
```

```

    std::cout << "  Using composition - behavior is explicit and controlled\n";
    LoggingCounter counter;
    counter.add(5);
    counter.addMultiple({1, 2, 3});
    std::cout << "  Count: " << counter.getCount() << "\n";
    counter.printLog();
    std::cout << "  Benefit: No fragility - implementation changes don't break us\n";
}

} // namespace composition_solution

// ============================================================================
// SECTION 3: SOLUTION 2 - Hide Implementation Details
// ============================================================================

namespace encapsulation_solution {

class Counter {
private:
    int count_ = 0;

    // Private helper - derived classes cannot rely on this
    void incrementInternal(int value) {
        count_ += value;
    }

protected:
    // Protected interface for derived classes
    void setCount(int value) { count_ = value; }
    int getCountInternal() const { return count_; }

public:
    virtual void add(int value) {
        incrementInternal(value);
    }

    // Non-virtual - cannot be overridden
    void addMultiple(const std::vector<int>& values) {
        for (int val : values) {
            incrementInternal(val);  // Calls private method
        }
    }

    int getCount() const { return count_; }
};

class LoggingCounter : public Counter {
```

```
private:
    std::vector<std::string> log_;

public:
    void add(int value) override {
        log_.push_back("Adding " + std::to_string(value));
        Counter::add(value);
    }

    void printLog() const {
        std::cout << "    Log entries: " << log_.size() << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 3: Solution 2 - Hide Implementation Details ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  Private implementation, public/protected interface only\n";
    LoggingCounter counter;
    counter.add(5);
    counter.addMultiple({1, 2, 3});
    std::cout << "  Count: " << counter.getCount() << "\n";
    counter.printLog();
    std::cout << "  Benefit: addMultiple() is non-virtual and uses private method\n";
}

} // namespace encapsulation_solution

// ============================================================================
// SECTION 4: SOLUTION 3 - Use 'final' Keyword
// ============================================================================

namespace final_solution {

// Concrete class marked as final - cannot be inherited
class Counter final {
private:
    int count_ = 0;

public:
    void add(int value) {
        count_ += value;
    }

    void addMultiple(const std::vector<int>& values) {
```

```
        for (int val : values) {
            add(val);
        }
    }

    int getCount() const { return count_; }
};

// Base class with final methods
class ExtensibleCounter {
protected:
    int count_ = 0;

public:
    virtual ~ExtensibleCounter() = default;

    // This method can be overridden
    virtual void add(int value) {
        count_ += value;
    }

    // This method CANNOT be overridden - marked final
    virtual void addMultiple(const std::vector<int>& values) final {
        for (int val : values) {
            add(val);
        }
    }

    int getCount() const { return count_; }
};

class LoggingCounter : public ExtensibleCounter {
private:
    std::vector<std::string> log_;

public:
    void add(int value) override {
        log_.push_back("Adding " + std::to_string(value));
        ExtensibleCounter::add(value);
    }

    // Cannot override addMultiple() - it's final
    // void addMultiple(...) { } // ERROR!

    void printLog() const {
        std::cout << "    Log entries: " << log_.size() << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 4: Solution 3 - Use 'final' Keyword ===\n";
    std::cout << std::string(70, '=') << "\n\n";

```

```
    std::cout << "  Using 'final' to prevent inheritance or method override\n\n";

    std::cout << "1. Final class (Counter) - cannot be inherited\n";
    Counter counter1;
    counter1.add(5);
    std::cout << "  Count: " << counter1.getCount() << "\n";
    // class Derived : public Counter { }; // ERROR: cannot inherit from final class

    std::cout << "\n2. Final method (addMultiple) - cannot be overridden\n";
    LoggingCounter counter2;
    counter2.add(5);
    counter2.addMultiple({1, 2, 3});
    std::cout << "  Count: " << counter2.getCount() << "\n";
    counter2.printLog();
    std::cout << "  Benefit: Base class controls critical behavior\n";
}

} // namespace final_solution

// ==============================================================================
// SECTION 5: SOLUTION 4 - Template Method Pattern
// ==============================================================================

namespace template_method_solution {

// Base class designed for inheritance with clear extension points
class Counter {
protected:
    int count_ = 0;

    // Hook for derived classes - well-documented extension point
    virtual void onBeforeAdd([[maybe_unused]] int value) {
        // Default: do nothing
    }

    virtual void onAfterAdd([[maybe_unused]] int value) {
        // Default: do nothing
    }

private:
    // Template method - defines the algorithm structure
    void addImpl(int value) {
        onBeforeAdd(value);
        count_ += value;
        onAfterAdd(value);
    }

public:
```

```
    virtual ~Counter() = default;

    // Public interface - calls template method
    void add(int value) {
        addImpl(value);
    }

    // Non-virtual - uses template method internally
    virtual void addMultiple(const std::vector<int>& values) final {
        for (int val : values) {
            addImpl(val);  // Each add goes through hooks
        }
    }

    int getCount() const { return count_; }
};

class LoggingCounter : public Counter {
private:
    std::vector<std::string> log_;

protected:
    void onBeforeAdd(int value) override {
        log_.push_back("Adding " + std::to_string(value));
    }

public:
    void printLog() const {
        std::cout << "    Log entries: " << log_.size() << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 5: Solution 4 - Template Method Pattern ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  Clear extension points with template method pattern\n";
    LoggingCounter counter;
    counter.add(5);
    counter.addMultiple({1, 2, 3});
    std::cout << "  Count: " << counter.getCount() << "\n";
    counter.printLog();
    std::cout << "  Benefit: Hooks are called for both add() and addMultiple()
        \n";
    std::cout << "  Base class controls algorithm, derived class controls
        hooks\n";
}

} // namespace template_method_solution

//
    ==========================================================================
```

```cpp
// SECTION 6: SOLUTION 5 - Prefer Interfaces (Pure Abstract Classes)
// ==========================================================================

namespace interface_solution {

// Pure interface - no implementation details to break
class ICounter {
public:
    virtual ~ICounter() = default;
    virtual void add(int value) = 0;
    virtual void addMultiple(const std::vector<int>& values) = 0;
    virtual int getCount() const = 0;
};

// Concrete implementation
class BasicCounter : public ICounter {
private:
    int count_ = 0;

public:
    void add(int value) override {
        count_ += value;
    }

    void addMultiple(const std::vector<int>& values) override {
        for (int val : values) {
            count_ += val;  // Direct implementation
        }
    }

    int getCount() const override { return count_; }
};

// Logging decorator using composition
class LoggingCounter : public ICounter {
private:
    std::unique_ptr<ICounter> wrapped_;
    std::vector<std::string> log_;

public:
    explicit LoggingCounter(std::unique_ptr<ICounter> counter)
        : wrapped_(std::move(counter)) {}

    void add(int value) override {
        log_.push_back("Adding " + std::to_string(value));
        wrapped_->add(value);
    }

    void addMultiple(const std::vector<int>& values) override {
        // No logging for batch operations
        wrapped_->addMultiple(values);
    }
```

```

    int getCount() const override { return wrapped_->getCount(); }

    void printLog() const {
        std::cout << "    Log entries: " << log_.size() << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 6: Solution 5 - Prefer Interfaces ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  Pure interface + composition (Decorator pattern)\n";
    auto counter = std::make_unique<LoggingCounter>(
        std::make_unique<BasicCounter>()
    );
    counter->add(5);
    counter->addMultiple({1, 2, 3});
    std::cout << "  Count: " << counter->getCount() << "\n";
    counter->printLog();
    std::cout << "  Benefit: No implementation inheritance - no fragility\n";
}

} // namespace interface_solution

// ==========================================================================
// SECTION 7: ABI Stability Example
// ==========================================================================

namespace abi_example {

// Version 1: Library shipped to customers
struct Device_V1 {
    int id;
    std::string name;

    virtual ~Device_V1() = default;
    virtual void process() { std::cout << "    Processing " << name << "\n"; }
};

struct Sensor_V1 : public Device_V1 {
    double reading;

    void process() override {
        std::cout << "    Sensor " << name << " reading: " << reading << "\n";
    }
};

// Version 2: Developer adds a field (BREAKS ABI)
```

```
struct Device_V2 {
    int id;
    std::string name;
    bool enabled;  // NEW FIELD - changes memory layout!

    virtual ~Device_V2() = default;
    virtual void process() {
        if (enabled) std::cout << "    Processing " << name << "\n";
    }
};

// Customer code (compiled against V1) now has wrong offsets!

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 7: ABI Stability and Memory Layout ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Problem: Adding fields to base class changes memory layout\n\n";

    std::cout << "Device_V1 layout:\n";
    Device_V1 d1;
    d1.id = 1;
    d1.name = "Device1";
    std::cout << "  sizeof(Device_V1) = " << sizeof(Device_V1) << " bytes\n";
    std::cout << "  &id offset   = " << (void*)&d1.id << "\n";
    std::cout << "  &name offset = " << (void*)&d1.name << "\n";

    std::cout << "\nSensor_V1 (derived) layout:\n";
    Sensor_V1 s1;
    s1.id = 2;
    s1.name = "Sensor1";
    s1.reading = 42.5;
    std::cout << "  sizeof(Sensor_V1) = " << sizeof(Sensor_V1) << " bytes\n";
    std::cout << "  &reading offset = " << (void*)&s1.reading << "\n";

    std::cout << "\nDevice_V2 layout (after adding 'enabled' field):\n";
    Device_V2 d2;
    std::cout << "  sizeof(Device_V2) = " << sizeof(Device_V2) << " bytes\n";
    std::cout << "  Problem: All derived class member offsets have shifted!\n";
    std::cout << "  Solution: Never add non-static data members to base classes\n";
    std::cout << "            Use Pimpl idiom or version interfaces\n";
}

} // namespace abi_example

// ============================================================================
// SECTION 8: Best Practices Summary
//
```

```
    ==========================================================================

void show_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Best Practices to Avoid Fragile Base Class ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. PREFER COMPOSITION OVER INHERITANCE\n";
    std::cout << "     Use 'has-a' instead of 'is-a' when possible\n";
    std::cout << "     Decouple implementation details\n";
    std::cout << "     More flexible and testable\n\n";

    std::cout << "2. DESIGN FOR INHERITANCE OR PROHIBIT IT\n";
    std::cout << "     Make classes 'final' by default\n";
    std::cout << "     Only allow inheritance for classes explicitly designed for it\n";
    std::cout << "     Document extension points clearly\n\n";

    std::cout << "3. USE THE 'final' KEYWORD\n";
    std::cout << "     Mark concrete classes as 'final'\n";
    std::cout << "     Mark critical methods as 'final' to prevent override\n";
    std::cout << "     Prevents accidental misuse\n\n";

    std::cout << "4. HIDE IMPLEMENTATION DETAILS\n";
    std::cout << "     Make data members private\n";
    std::cout << "     Use private helper methods\n";
    std::cout << "     Expose minimal protected interface\n\n";

    std::cout << "5. PREFER INTERFACES (PURE ABSTRACT CLASSES)\n";
    std::cout << "     Define contracts, not implementations\n";
    std::cout << "     Use composition + decorator pattern\n";
    std::cout << "     Eliminates implementation inheritance fragility\n\n";

    std::cout << "6. USE TEMPLATE METHOD PATTERN\n";
    std::cout << "     Base class controls algorithm structure\n";
    std::cout << "     Derived classes override hooks only\n";
    std::cout << "     Clear extension points\n\n";

    std::cout << "7. FOR LIBRARY DEVELOPERS - MAINTAIN ABI STABILITY\n";
    std::cout << "     Never add non-static data members to base classes\n";
    std::cout << "     Use Pimpl idiom for implementation hiding\n";
    std::cout << "     Version your interfaces\n\n";

    std::cout << "REMEMBER: \"Prefer composition over inheritance\" - Gang of Four\n";
}

//
    ==========================================================================
// MAIN FUNCTION
//
```

```
    ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "      Fragile Base Class Problem in C++ and Solutions
                  \n";
    std::cout << "                                    \n";

    // Demonstrate the problem
    fragile_example::demonstrate();

    // Show solutions
    composition_solution::demonstrate();
    encapsulation_solution::demonstrate();
    final_solution::demonstrate();
    template_method_solution::demonstrate();
    interface_solution::demonstrate();

    // ABI considerations
    abi_example::demonstrate();

    // Best practices
    show_best_practices();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

34 Source Code: FunctionalSafetyISO26262.cpp

File: `src/FunctionalSafetyISO26262.cpp`
Repository: View on GitHub

```cpp
// FunctionalSafetyISO26262.cpp
// Comprehensive demonstration of ISO 26262 Functional Safety principles in Modern C++
// Covers ASIL levels, MISRA C++, redundancy, watchdogs, safe data types, and deterministic patterns

#include <iostream>
#include <cstdint>
#include <array>
#include <limits>
#include <chrono>
#include <functional>
#include <algorithm>
#include <cstring>

// ============================================================================
// SECTION 1: ASIL Levels and Safety Requirements
// ============================================================================
// ISO 26262 defines 4 Automotive Safety Integrity Levels (ASIL A-D)
// ASIL D = highest safety requirements (e.g., braking systems)
// ASIL A = lowest safety requirements (e.g., rear lights)

enum class ASILLevel : uint8_t {
    QM = 0,  // Quality Management (no ASIL)
    A = 1,   // Lowest safety integrity
    B = 2,
    C = 3,
    D = 4    // Highest safety integrity
};

// Safety state for system degradation
enum class SafetyState : uint8_t {
    NORMAL_OPERATION = 0,
    DEGRADED_MODE = 1,
    SAFE_STATE = 2,
    EMERGENCY_STOP = 3
};

// Safety-critical function return type (no exceptions!)
enum class SafetyResult : uint8_t {
    OK = 0,
    WARNING = 1,
    ERROR = 2,
    CRITICAL_FAILURE = 3
};
```

```cpp

void demonstrate_asil_levels() {
    std::cout << "\n=== 1. ASIL Levels and Safety Requirements ===\n";

    struct SafetyRequirement {
        const char* component;
        ASILLevel asil;
        const char* requirement;
    };

    constexpr std::array<SafetyRequirement, 5> requirements = {{
        {"Brake-by-wire", ASILLevel::D, "Dual redundancy + voting"},
        {"Electric power steering", ASILLevel::C, "Plausibility checks + monitoring"},
        {"Airbag control", ASILLevel::D, "Watchdog + memory protection"},
        {"Engine control", ASILLevel::B, "Runtime checks + diagnostics"},
        {"Interior lighting", ASILLevel::QM, "Basic error handling"}
    }};

    std::cout << "\nASIL Classification Examples:\n";
    for (const auto& req : requirements) {
        std::cout << "  " << req.component << " [ASIL-";
        switch (req.asil) {
            case ASILLevel::QM: std::cout << "QM"; break;
            case ASILLevel::A: std::cout << "A"; break;
            case ASILLevel::B: std::cout << "B"; break;
            case ASILLevel::C: std::cout << "C"; break;
            case ASILLevel::D: std::cout << "D"; break;
        }
        std::cout << "]: " << req.requirement << "\n";
    }

    std::cout << "\nKey ISO 26262 Principles:\n";
    std::cout << "    No dynamic memory allocation in safety functions\n";
    std::cout << "    No exceptions in safety-critical paths (use error codes)\n";
    std::cout << "    Deterministic timing (WCET - Worst Case Execution Time)\n";
    std::cout << "    Redundancy and voting for ASIL-D\n";
    std::cout << "    Comprehensive diagnostics and fault detection\n";
}

// ==========================================================================
// SECTION 2: Safe Data Types with Overflow Protection
// ==========================================================================
// MISRA C++ requires explicit bounds checking and overflow protection

template<typename T>
class SafeInteger {
private:
```

```
    T value_;
    bool valid_;

public:
    constexpr SafeInteger() noexcept : value_(0), valid_(true) {}
    constexpr explicit SafeInteger(T val) noexcept : value_(val), valid_(true)
        {}

    // Safe addition with overflow detection
    SafeInteger add(T other) const noexcept {
        SafeInteger result;

        // Check for overflow before operation (MISRA requirement)
        if (other > 0 && value_ > std::numeric_limits<T>::max() - other) {
            result.valid_ = false;  // Overflow detected
            result.value_ = std::numeric_limits<T>::max();
        } else if (other < 0 && value_ < std::numeric_limits<T>::min() - other
            ) {
            result.valid_ = false;  // Underflow detected
            result.value_ = std::numeric_limits<T>::min();
        } else {
            result.value_ = value_ + other;
            result.valid_ = true;
        }

        return result;
    }

    // Safe multiplication with overflow detection
    SafeInteger multiply(T other) const noexcept {
        SafeInteger result;

        if (other == 0 || value_ == 0) {
            result.value_ = 0;
            result.valid_ = true;
        } else if (value_ > std::numeric_limits<T>::max() / other) {
            result.valid_ = false;  // Overflow
            result.value_ = std::numeric_limits<T>::max();
        } else {
            result.value_ = value_ * other;
            result.valid_ = true;
        }

        return result;
    }

    constexpr T get() const noexcept { return value_; }
    constexpr bool is_valid() const noexcept { return valid_; }
};

void demonstrate_safe_data_types() {
    std::cout << "\n=== 2. Safe Data Types with Overflow Protection ===\n";

    SafeInteger<int32_t> speed(100);
```

```
    SafeInteger<int32_t> delta(50);

    auto new_speed = speed.add(delta.get());
    std::cout << "\nSafe Addition: 100 + 50 = " << new_speed.get()
              << " (valid: " << (new_speed.is_valid() ? " " : " ") << ")\n";

    // Demonstrate overflow detection
    SafeInteger<int32_t> large_val(std::numeric_limits<int32_t>::max() - 10);
    auto overflow_result = large_val.add(100);
    std::cout << "\nOverflow Detection: MAX-10 + 100 = " << overflow_result.get()
              << " (valid: " << (overflow_result.is_valid() ? " " : " OVERFLOW DETECTED") << ")\n";

    std::cout << "\nMISRA C++ Guidelines:\n";
    std::cout << "    All arithmetic checked for overflow/underflow\n";
    std::cout << "    Explicit error flags instead of exceptions\n";
    std::cout << "    constexpr for compile-time safety verification\n";
    std::cout << "    noexcept guarantees for safety-critical functions\n";
}

// ==========================================================================
// SECTION 3: Redundancy and Voting Mechanisms (ASIL-D Requirement)
// ==========================================================================
// Dual-channel or triple-modular redundancy for critical computations

template<typename T>
class DualChannelComputation {
private:
    std::function<T()> channel1_;
    std::function<T()> channel2_;
    T tolerance_;

public:
    DualChannelComputation(std::function<T()> ch1, std::function<T()> ch2, T tol) noexcept
        : channel1_(ch1), channel2_(ch2), tolerance_(tol) {}

    // Execute both channels and compare results
    SafetyResult compute(T& result) noexcept {
        T result1 = channel1_();
        T result2 = channel2_();

        // Calculate absolute difference
        T diff = (result1 > result2) ? (result1 - result2) : (result2 - result1);

        if (diff <= tolerance_) {
            // Results agree within tolerance
            result = (result1 + result2) / 2;  // Use average
```

```
            return SafetyResult::OK;
        } else {
            // Mismatch detected - critical failure
            result = 0;  // Safe default
            return SafetyResult::CRITICAL_FAILURE;
        }
    }
};

// Triple Modular Redundancy (TMR) with voting
template<typename T>
class TripleModularRedundancy {
private:
    std::array<std::function<T()>, 3> channels_;
    T tolerance_;

public:
    TripleModularRedundancy(std::function<T()> ch1, std::function<T()> ch2,
                            std::function<T()> ch3, T tol) noexcept
        : channels_{{ch1, ch2, ch3}}, tolerance_(tol) {}

    // 2-out-of-3 voting
    SafetyResult compute(T& result) noexcept {
        std::array<T, 3> results;
        for (size_t i = 0; i < 3; ++i) {
            results[i] = channels_[i]();
        }

        // Check if any two results agree within tolerance
        for (size_t i = 0; i < 3; ++i) {
            for (size_t j = i + 1; j < 3; ++j) {
                T diff = (results[i] > results[j]) ?
                         (results[i] - results[j]) : (results[j] - results[i])
                             ;

                if (diff <= tolerance_) {
                    // Two channels agree
                    result = (results[i] + results[j]) / 2;
                    return (i == 2 || j == 2) ? SafetyResult::WARNING :
                        SafetyResult::OK;
                }
            }
        }

        // No agreement - critical failure
        result = 0;
        return SafetyResult::CRITICAL_FAILURE;
    }
};

void demonstrate_redundancy_voting() {
    std::cout << "\n=== 3. Redundancy and Voting Mechanisms ===\n";

    // Simulate brake pressure sensor readings
```

```
    auto sensor_channel_1 = []() -> int32_t { return 850; };  // 850 kPa
    auto sensor_channel_2 = []() -> int32_t { return 852; };  // 852 kPa (slight variation)

    DualChannelComputation<int32_t> brake_pressure(sensor_channel_1, sensor_channel_2, 5);

    int32_t pressure = 0;
    SafetyResult status = brake_pressure.compute(pressure);

    std::cout << "\nDual-Channel Brake Pressure Sensor:\n";
    std::cout << "  Channel 1: 850 kPa\n";
    std::cout << "  Channel 2: 852 kPa\n";
    std::cout << "  Result: " << pressure << " kPa (avg)\n";
    std::cout << "  Status: " << (status == SafetyResult::OK ? "  OK" : "  FAILURE") << "\n";

    // Triple modular redundancy example
    auto tmr_ch1 = []() -> int32_t { return 100; };
    auto tmr_ch2 = []() -> int32_t { return 101; };
    auto tmr_ch3 = []() -> int32_t { return 99; };  // All agree within tolerance

    TripleModularRedundancy<int32_t> steering_angle(tmr_ch1, tmr_ch2, tmr_ch3, 5);

    int32_t angle = 0;
    SafetyResult tmr_status = steering_angle.compute(angle);

    std::cout << "\nTriple Modular Redundancy (Steering Angle):\n";
    std::cout << "  Channel 1: 100°\n";
    std::cout << "  Channel 2: 101°\n";
    std::cout << "  Channel 3: 99°\n";
    std::cout << "  Voted Result: " << angle << "°\n";
    std::cout << "  Status: " << (tmr_status == SafetyResult::OK ? "  OK (2-of-3 agree)" : "  FAILURE") << "\n";

    std::cout << "\nRedundancy Patterns:\n";
    std::cout << "    Dual-channel for ASIL-C/D\n";
    std::cout << "    Triple modular redundancy (TMR) for highest safety\n";
    std::cout << "    2-out-of-3 voting tolerates single-point failures\n";
    std::cout << "    Tolerance bands for real-world sensor variations\n";
}

// ==========================================================================
// SECTION 4: Watchdog Timer and Heartbeat Monitoring
// ==========================================================================
// Detect software hangs and stuck states

class WatchdogTimer {
```

```cpp
private:
    uint32_t timeout_ms_;
    uint32_t last_kick_time_;
    bool expired_;

public:
    explicit WatchdogTimer(uint32_t timeout_ms) noexcept
        : timeout_ms_(timeout_ms), last_kick_time_(0), expired_(false) {}

    // Kick the watchdog (reset timer)
    void kick(uint32_t current_time_ms) noexcept {
        last_kick_time_ = current_time_ms;
        expired_ = false;
    }

    // Check if watchdog expired
    bool check_expired(uint32_t current_time_ms) noexcept {
        if (current_time_ms - last_kick_time_ > timeout_ms_) {
            expired_ = true;
        }
        return expired_;
    }

    bool is_expired() const noexcept { return expired_; }
};

class SafetyMonitor {
private:
    WatchdogTimer watchdog_;
    uint32_t heartbeat_count_;
    SafetyState state_;

public:
    explicit SafetyMonitor(uint32_t watchdog_timeout) noexcept
        : watchdog_(watchdog_timeout), heartbeat_count_(0), state_(SafetyState
            ::NORMAL_OPERATION) {}

    void heartbeat(uint32_t current_time) noexcept {
        watchdog_.kick(current_time);
        ++heartbeat_count_;
    }

    SafetyResult check_safety(uint32_t current_time) noexcept {
        if (watchdog_.check_expired(current_time)) {
            state_ = SafetyState::EMERGENCY_STOP;
            return SafetyResult::CRITICAL_FAILURE;
        }
        return SafetyResult::OK;
    }

    SafetyState get_state() const noexcept { return state_; }
    uint32_t get_heartbeat_count() const noexcept { return heartbeat_count_; }
};

```

```cpp
void demonstrate_watchdog_monitoring() {
    std::cout << "\n=== 4. Watchdog Timer and Heartbeat Monitoring ===\n";

    SafetyMonitor monitor(100);  // 100ms timeout

    std::cout << "\nNormal Operation:\n";
    for (uint32_t time = 0; time <= 250; time += 50) {
        monitor.heartbeat(time);
        SafetyResult status = monitor.check_safety(time);
        std::cout << "  Time " << time << "ms: Heartbeat #" << monitor.get_heartbeat_count()
                  << " - Status: " << (status == SafetyResult::OK ? "  OK" : "  FAILURE") << "\n";
    }

    // Simulate watchdog expiration (no heartbeat for >100ms)
    std::cout << "\nSimulating Watchdog Timeout (no heartbeat):\n";
    SafetyMonitor timeout_monitor(100);
    timeout_monitor.heartbeat(0);

    for (uint32_t time = 50; time <= 200; time += 50) {
        // Skip heartbeat to simulate hang
        SafetyResult status = timeout_monitor.check_safety(time);
        std::cout << "  Time " << time << "ms: ";
        if (status == SafetyResult::CRITICAL_FAILURE) {
            std::cout << "  WATCHDOG EXPIRED - EMERGENCY STOP\n";
            break;
        } else {
            std::cout << "  OK\n";
        }
    }

    std::cout << "\nWatchdog Best Practices:\n";
    std::cout << "    Timeout based on WCET (Worst Case Execution Time)\n";
    std::cout << "    Hardware watchdog preferred for fail-safe\n";
    std::cout << "    Heartbeat every control cycle\n";
    std::cout << "    Transition to safe state on expiration\n";
}

// ==========================================================================
// SECTION 5: Memory Safety and Bounds Checking
// ==========================================================================
// MISRA C++ requires explicit bounds checking, no buffer overruns

template<typename T, size_t N>
class SafeArray {
private:
    std::array<T, N> data_;

public:
```

```
    constexpr SafeArray() noexcept : data_{} {}

    // Safe access with bounds checking (no exceptions!)
    SafetyResult get(size_t index, T& value) const noexcept {
        if (index >= N) {
            return SafetyResult::ERROR;
        }
        value = data_[index];
        return SafetyResult::OK;
    }

    SafetyResult set(size_t index, T value) noexcept {
        if (index >= N) {
            return SafetyResult::ERROR;
        }
        data_[index] = value;
        return SafetyResult::OK;
    }

    constexpr size_t size() const noexcept { return N; }

    // Safe iteration
    T* begin() noexcept { return data_.data(); }
    T* end() noexcept { return data_.data() + N; }
    const T* begin() const noexcept { return data_.data(); }
    const T* end() const noexcept { return data_.data() + N; }
};

// Safe string operations (fixed-size, no dynamic allocation)
template<size_t N>
class SafeString {
private:
    std::array<char, N> buffer_;
    size_t length_;

public:
    constexpr SafeString() noexcept : buffer_{}, length_(0) {
        buffer_[0] = '\0';
    }

    SafetyResult copy_from(const char* src) noexcept {
        if (src == nullptr) {
            return SafetyResult::ERROR;
        }

        size_t i = 0;
        while (i < N - 1 && src[i] != '\0') {
            buffer_[i] = src[i];
            ++i;
        }
        buffer_[i] = '\0';
        length_ = i;

        return (src[i] == '\0') ? SafetyResult::OK : SafetyResult::WARNING;
```

```
            // Truncated
    }

    SafetyResult append(const char* src) noexcept {
        if (src == nullptr) {
            return SafetyResult::ERROR;
        }

        size_t i = 0;
        while (length_ < N - 1 && src[i] != '\0') {
            buffer_[length_++] = src[i++];
        }
        buffer_[length_] = '\0';

        return (src[i] == '\0') ? SafetyResult::OK : SafetyResult::WARNING;
    }

    const char* c_str() const noexcept { return buffer_.data(); }
    size_t length() const noexcept { return length_; }
    constexpr size_t capacity() const noexcept { return N - 1; }
};

void demonstrate_memory_safety() {
    std::cout << "\n=== 5. Memory Safety and Bounds Checking ===\n";

    // Safe array with compile-time size
    SafeArray<int32_t, 10> sensor_readings;

    std::cout << "\nSafe Array Operations:\n";
    for (size_t i = 0; i < sensor_readings.size(); ++i) {
        sensor_readings.set(i, static_cast<int32_t>(i * 100));
    }

    int32_t value = 0;
    SafetyResult result = sensor_readings.get(5, value);
    std::cout << "  Reading[5] = " << value << " (status: "
              << (result == SafetyResult::OK ? "  OK" : "  ERROR") << ")\n";

    // Attempt out-of-bounds access
    result = sensor_readings.get(15, value);
    std::cout << "  Reading[15] = " << (result == SafetyResult::OK ? "Valid" :
        "  OUT OF BOUNDS") << "\n";

    // Safe string operations
    SafeString<32> device_id;
    device_id.copy_from("BRAKE_CTRL_");
    device_id.append("ECU001");

    std::cout << "\nSafe String Operations:\n";
    std::cout << "  Device ID: \"" << device_id.c_str() << "\"\n";
    std::cout << "  Length: " << device_id.length() << "/" << device_id.
       capacity() << "\n";

    std::cout << "\nMemory Safety Rules:\n";
```

```
    std::cout << "    No dynamic allocation (new/malloc) in safety functions\n";
    std::cout << "    Fixed-size buffers with compile-time bounds\n";
    std::cout << "    All array accesses bounds-checked\n";
    std::cout << "    No pointer arithmetic without validation\n";
    std::cout << "    Stack-allocated containers only\n";
}

// ==========================================================================
// SECTION 6: Runtime Diagnostics and Self-Test
// ==========================================================================
// Continuous self-monitoring and fault detection

class DiagnosticMonitor {
private:
    struct DiagnosticCounter {
        uint32_t total_checks;
        uint32_t failures;
        uint32_t warnings;
    };

    DiagnosticCounter counters_;
    bool system_healthy_;

public:
    DiagnosticMonitor() noexcept
        : counters_{0, 0, 0}, system_healthy_(true) {}

    // RAM test (simple pattern check)
    SafetyResult test_ram(uint32_t* test_area, size_t size) noexcept {
        ++counters_.total_checks;

        // Write pattern
        constexpr uint32_t PATTERN1 = 0x55555555;
        constexpr uint32_t PATTERN2 = 0xAAAAAAAA;

        for (size_t i = 0; i < size; ++i) {
            test_area[i] = PATTERN1;
        }

        // Verify pattern 1
        for (size_t i = 0; i < size; ++i) {
            if (test_area[i] != PATTERN1) {
                ++counters_.failures;
                system_healthy_ = false;
                return SafetyResult::CRITICAL_FAILURE;
            }
        }

        // Write and verify pattern 2
```

```
        for (size_t i = 0; i < size; ++i) {
            test_area[i] = PATTERN2;
        }

        for (size_t i = 0; i < size; ++i) {
            if (test_area[i] != PATTERN2) {
                ++counters_.failures;
                system_healthy_ = false;
                return SafetyResult::CRITICAL_FAILURE;
            }
        }

        return SafetyResult::OK;
    }

    // Plausibility check for sensor values
    SafetyResult check_plausibility(int32_t value, int32_t min, int32_t max)
        noexcept {
        ++counters_.total_checks;

        if (value < min || value > max) {
            ++counters_.failures;
            return SafetyResult::ERROR;
        }

        // Warning range (10% from limits)
        int32_t warning_margin = (max - min) / 10;
        if (value < min + warning_margin || value > max - warning_margin) {
            ++counters_.warnings;
            return SafetyResult::WARNING;
        }

        return SafetyResult::OK;
    }

    void get_diagnostics(uint32_t& total, uint32_t& failures, uint32_t&
        warnings) const noexcept {
        total = counters_.total_checks;
        failures = counters_.failures;
        warnings = counters_.warnings;
    }

    bool is_healthy() const noexcept { return system_healthy_; }
};

void demonstrate_runtime_diagnostics() {
    std::cout << "\n=== 6. Runtime Diagnostics and Self-Test ===\n";

    DiagnosticMonitor diagnostics;

    // RAM self-test
    std::array<uint32_t, 256> test_ram;
    std::cout << "\nRAM Self-Test:\n";
    SafetyResult ram_result = diagnostics.test_ram(test_ram.data(), test_ram.
```

```
        size());
    std::cout << "  Pattern Test (0x55555555/0xAAAAAAAA): "
              << (ram_result == SafetyResult::OK ? "  PASS" : "  FAIL") << "\n"
                  ;

    // Plausibility checks
    std::cout << "\nSensor Plausibility Checks:\n";

    struct SensorTest {
        const char* name;
        int32_t value;
        int32_t min;
        int32_t max;
    };

    constexpr std::array<SensorTest, 4> tests = {{
        {"Wheel speed", 1200, 0, 3000},      // OK
        {"Brake pressure", 850, 0, 1000},     // OK
        {"Steering angle", 950, -900, 900},   // WARNING (near limit)
        {"Throttle position", 1500, 0, 1000}  // ERROR (out of range)
    }};

    for (const auto& test : tests) {
        SafetyResult result = diagnostics.check_plausibility(test.value, test.
            min, test.max);
        std::cout << "  " << test.name << " = " << test.value
                  << " [" << test.min << ".." << test.max << "]: ";

        switch (result) {
            case SafetyResult::OK:
                std::cout << "  OK\n";
                break;
            case SafetyResult::WARNING:
                std::cout << "  WARNING (near limit)\n";
                break;
            case SafetyResult::ERROR:
                std::cout << "  OUT OF RANGE\n";
                break;
            default:
                std::cout << "  CRITICAL\n";
                break;
        }
    }

    // Diagnostic summary
    uint32_t total, failures, warnings;
    diagnostics.get_diagnostics(total, failures, warnings);

    std::cout << "\nDiagnostic Summary:\n";
    std::cout << "  Total checks: " << total << "\n";
    std::cout << "  Failures: " << failures << "\n";
    std::cout << "  Warnings: " << warnings << "\n";
    std::cout << "  System Health: " << (diagnostics.is_healthy() ? "  HEALTHY
        " : "  DEGRADED") << "\n";
```

```

    std::cout << "\nDiagnostic Coverage:\n";
    std::cout << "    RAM pattern test (startup + periodic)\n";
    std::cout << "    Plausibility checks (range validation)\n";
    std::cout << "    Stuck-at fault detection\n";
    std::cout << "    Diagnostic Trouble Code (DTC) logging\n";
}

// ==============================================================================
// SECTION 7: Safe State Transitions and Fault Handling
// ==============================================================================
// Deterministic state machine with safe fallback

class BrakingController {
private:
    enum class BrakeState : uint8_t {
        INIT = 0,
        NORMAL = 1,
        DEGRADED = 2,
        FAILSAFE = 3,
        EMERGENCY = 4
    };

    BrakeState current_state_;
    BrakeState previous_state_;
    uint32_t fault_count_;

public:
    BrakingController() noexcept
        : current_state_(BrakeState::INIT),
          previous_state_(BrakeState::INIT),
          fault_count_(0) {}

    SafetyResult initialize() noexcept {
        if (current_state_ != BrakeState::INIT) {
            return SafetyResult::ERROR;
        }

        // Perform self-tests
        // ... (RAM test, sensor check, actuator test)

        previous_state_ = current_state_;
        current_state_ = BrakeState::NORMAL;
        return SafetyResult::OK;
    }

    SafetyResult handle_fault(SafetyResult fault_severity) noexcept {
        ++fault_count_;
        previous_state_ = current_state_;

```

```
        switch (fault_severity) {
            case SafetyResult::WARNING:
                // Stay in current state but log warning
                return SafetyResult::WARNING;

            case SafetyResult::ERROR:
                // Transition to degraded mode
                if (current_state_ == BrakeState::NORMAL) {
                    current_state_ = BrakeState::DEGRADED;
                }
                return SafetyResult::ERROR;

            case SafetyResult::CRITICAL_FAILURE:
                // Immediate transition to emergency
                current_state_ = BrakeState::EMERGENCY;
                return SafetyResult::CRITICAL_FAILURE;

            default:
                return SafetyResult::OK;
        }
    }

    SafetyResult apply_brakes(int32_t pressure) noexcept {
        switch (current_state_) {
            case BrakeState::INIT:
                return SafetyResult::ERROR;  // Not initialized

            case BrakeState::NORMAL:
                // Full functionality
                return apply_normal_braking(pressure);

            case BrakeState::DEGRADED:
                // Limited functionality (e.g., one channel failed)
                return apply_degraded_braking(pressure / 2);

            case BrakeState::FAILSAFE:
            case BrakeState::EMERGENCY:
                // Maximum braking regardless of input
                return apply_emergency_braking();
        }

        return SafetyResult::ERROR;
    }

    const char* get_state_name() const noexcept {
        switch (current_state_) {
            case BrakeState::INIT: return "INIT";
            case BrakeState::NORMAL: return "NORMAL";
            case BrakeState::DEGRADED: return "DEGRADED";
            case BrakeState::FAILSAFE: return "FAILSAFE";
            case BrakeState::EMERGENCY: return "EMERGENCY";
            default: return "UNKNOWN";
        }
    }
```

```

    uint32_t get_fault_count() const noexcept { return fault_count_; }

private:
    SafetyResult apply_normal_braking(int32_t pressure) noexcept {
        // Normal braking logic
        (void)pressure;  // Suppress unused warning
        return SafetyResult::OK;
    }

    SafetyResult apply_degraded_braking(int32_t pressure) noexcept {
        // Degraded mode (e.g., mechanical backup)
        (void)pressure;
        return SafetyResult::WARNING;
    }

    SafetyResult apply_emergency_braking() noexcept {
        // Maximum braking effort
        return SafetyResult::CRITICAL_FAILURE;
    }
};

void demonstrate_safe_state_transitions() {
    std::cout << "\n=== 7. Safe State Transitions and Fault Handling ===\n";

    BrakingController brake_system;

    std::cout << "\nBraking System State Machine:\n";
    std::cout << "  Initial State: " << brake_system.get_state_name() << "\n";

    // Initialize
    brake_system.initialize();
    std::cout << "  After Init: " << brake_system.get_state_name() << "\n";

    // Normal operation
    SafetyResult result = brake_system.apply_brakes(500);
    std::cout << "  Apply 500 kPa: " << (result == SafetyResult::OK ? "  OK" :
        "  ERROR") << "\n";

    // Simulate fault
    std::cout << "\n  Simulating sensor fault...\n";
    brake_system.handle_fault(SafetyResult::ERROR);
    std::cout << "  State after fault: " << brake_system.get_state_name() << "
        \n";

    result = brake_system.apply_brakes(500);
    std::cout << "  Apply 500 kPa in degraded: "
              << (result == SafetyResult::WARNING ? "  LIMITED" : "  ERROR") <<
                  "\n";

    // Critical fault
    std::cout << "\n  Simulating critical failure...\n";
    brake_system.handle_fault(SafetyResult::CRITICAL_FAILURE);
    std::cout << "  State after critical: " << brake_system.get_state_name()
```

```
        << "\n";
    std::cout << "  Total faults handled: " << brake_system.get_fault_count()
        << "\n";

    std::cout << "\nState Machine Principles:\n";
    std::cout << "    Deterministic transitions (no race conditions)\n";
    std::cout << "    Always transition to safe state on fault\n";
    std::cout << "    No invalid states reachable\n";
    std::cout << "    Fail-safe defaults (e.g., emergency braking)\n";
}

// ==========================================================================
// SECTION 8: Deterministic Timing and WCET
// ==========================================================================
// Predictable execution time for real-time safety functions

class ExecutionTimeMonitor {
private:
    std::chrono::steady_clock::time_point start_time_;
    uint32_t wcet_us_;  // Worst-case execution time in microseconds
    bool timing_violation_;

public:
    explicit ExecutionTimeMonitor(uint32_t wcet_us) noexcept
        : wcet_us_(wcet_us), timing_violation_(false) {}

    void start() noexcept {
        start_time_ = std::chrono::steady_clock::now();
        timing_violation_ = false;
    }

    SafetyResult check() noexcept {
        auto end_time = std::chrono::steady_clock::now();
        auto elapsed_us = std::chrono::duration_cast<std::chrono::microseconds
            >(
            end_time - start_time_).count();

        if (elapsed_us > wcet_us_) {
            timing_violation_ = true;
            return SafetyResult::ERROR;
        }

        return SafetyResult::OK;
    }

    bool has_violation() const noexcept { return timing_violation_; }
};

// Safety-critical control loop with deterministic timing
SafetyResult safety_control_cycle(ExecutionTimeMonitor& timer) noexcept {
```

```
    timer.start();

    // Read sensors (deterministic)
    volatile int32_t sensor1 = 100;
    volatile int32_t sensor2 = 200;

    // Compute control output (no branches, no loops with variable iterations)
    volatile int32_t output = (sensor1 + sensor2) / 2;

    // Write actuators
    (void)output;  // Suppress unused warning

    // Check timing constraint
    return timer.check();
}

void demonstrate_deterministic_timing() {
    std::cout << "\n=== 8. Deterministic Timing and WCET ===\n";

    // Set WCET budget to 100 microseconds
    ExecutionTimeMonitor timer(100);

    std::cout << "\nControl Cycle Timing (WCET = 100 µs):\n";

    for (int i = 0; i < 5; ++i) {
        SafetyResult result = safety_control_cycle(timer);
        std::cout << "  Cycle " << (i + 1) << ": "
                  << (result == SafetyResult::OK ? "  Within WCET" : "  TIMING
                      VIOLATION") << "\n";
    }

    std::cout << "\nDeterminism Requirements:\n";
    std::cout << "    No dynamic memory allocation (non-deterministic)\n";
    std::cout << "    No unbounded loops (use fixed iteration count)\n";
    std::cout << "    No recursive functions (stack usage)\n";
    std::cout << "    Disable interrupts in critical sections\n";
    std::cout << "    Priority-based scheduling (preemptive RTOS)\n";
    std::cout << "    WCET analysis tools (e.g., aiT, RapiTime)\n";
}

//
    ==========================================================================
// SECTION 9: MISRA C++ Compliance Patterns
//
    ==========================================================================

void demonstrate_misra_compliance() {
    std::cout << "\n=== 9. MISRA C++ Compliance Patterns ===\n";

    std::cout << "\nKey MISRA C++ 2023 Rules:\n\n";

    std::cout << "Rule 5-0-3: Prohibited types and features:\n";
```

```
    std::cout << "    Avoid: exceptions, RTTI (typeid), dynamic_cast in safety code\n";
    std::cout << "    Use: Error codes, static polymorphism (templates)\n\n";

    std::cout << "Rule 5-2-12: No dynamic memory allocation:\n";
    std::cout << "    Avoid: new, delete, malloc, free, std::vector (dynamic)\n";
    std::cout << "    Use: std::array, static buffers, placement new (if needed)\n\n";

    std::cout << "Rule 6-4-5: Unconditional throw/goto prohibited:\n";
    std::cout << "    Avoid: throw exceptions in safety functions\n";
    std::cout << "    Use: Return error codes (SafetyResult enum)\n\n";

    std::cout << "Rule 8-0-1: All functions have one exit point:\n";
    std::cout << "    Avoid: Multiple return statements\n";
    std::cout << "    Use: Single return with result variable\n\n";

    std::cout << "Rule 18-0-3: No library functions with undefined behavior:\n";
    std::cout << "    Avoid: atoi, gets, sprintf (unsafe)\n";
    std::cout << "    Use: strtol with error checking, snprintf\n\n";

    std::cout << "Rule 27-0-1: All includes have include guards:\n";
    std::cout << "    Use: #pragma once or #ifndef guards\n\n";

    std::cout << "Additional Safety Guidelines:\n";
    std::cout << "    All arithmetic operations overflow-checked\n";
    std::cout << "    All array accesses bounds-checked\n";
    std::cout << "    All pointers validated before dereference\n";
    std::cout << "    All functions declared noexcept if they don't throw\n";
    std::cout << "    constexpr for compile-time evaluation\n";
    std::cout << "    Static analysis tools (e.g., LDRA, PC-lint Plus)\n";
}

// ==========================================================================
// SECTION 10: Safety Checklist and Best Practices Summary
// ==========================================================================

void demonstrate_safety_checklist() {
    std::cout << "\n=== 10. ISO 26262 Safety Checklist ===\n";

    std::cout << "\ n DESIGN PHASE:\n";
    std::cout << "   [] ASIL classification completed\n";
    std::cout << "   [] Hazard analysis and risk assessment (HARA)\n";
    std::cout << "   [] Safety goals defined (fail-safe behavior)\n";
    std::cout << "   [] Redundancy strategy selected (dual/TMR)\n";
    std::cout << "   [] WCET analysis performed\n";
    std::cout << "   [] Diagnostic coverage targets set\n\n";

```

```
    std::cout << "  IMPLEMENTATION PHASE:\n";
    std::cout << "   [] MISRA C++ guidelines followed\n";
    std::cout << "   [] No dynamic memory allocation\n";
    std::cout << "   [] No exceptions in safety functions\n";
    std::cout << "   [] All arithmetic overflow-checked\n";
    std::cout << "   [] All arrays bounds-checked\n";
    std::cout << "   [] Watchdog timer implemented\n";
    std::cout << "   [] Safe state transitions verified\n";
    std::cout << "   [] Deterministic timing validated\n\n";

    std::cout << "  VERIFICATION PHASE:\n";
    std::cout << "   [] Static analysis (PC-lint, Coverity, PVS-Studio)\n";
    std::cout << "   [] Dynamic testing (functional + fault injection)\n";
    std::cout << "   [] Code reviews (multiple reviewers)\n";
    std::cout << "   [] Requirements traceability matrix\n";
    std::cout << "   [] Coverage analysis (MC/DC for ASIL-D)\n";
    std::cout << "   [] Safety case documentation\n\n";

    std::cout << "  VALIDATION PHASE:\n";
    std::cout << "   [] Hardware-in-the-loop (HIL) testing\n";
    std::cout << "   [] Environmental stress testing\n";
    std::cout << "   [] EMC/EMI validation\n";
    std::cout << "   [] Long-term reliability testing\n";
    std::cout << "   [] Safety audit completed\n\n";

    std::cout << "KEY TAKEAWAYS:\n";
    std::cout << "  1. Safety is a process, not just code\n";
    std::cout << "  2. Redundancy and diagnostics are essential for ASIL-C/D\n";
    std::cout << "  3. Deterministic behavior is mandatory\n";
    std::cout << "  4. Always transition to safe state on fault\n";
    std::cout << "  5. Document everything (traceability is critical)\n";
    std::cout << "  6. Use certified tools and compilers for final builds\n";
    std::cout << "  7. Continuous monitoring in production (field data)\n";
}

// ============================================================================
// MAIN FUNCTION
// ============================================================================

int main() {
    std::cout << "                                    \n";
    std::cout << "      ISO 26262 Functional Safety - Modern C++ Demonstration\n";
    std::cout << "                                                            \n";
    std::cout << "   Comprehensive examples of safety-critical automotive code\n";
    std::cout << "   covering ASIL levels, MISRA C++, redundancy, and more\n";
```

```
    std::cout << "                                    \n";

    demonstrate_asil_levels();
    demonstrate_safe_data_types();
    demonstrate_redundancy_voting();
    demonstrate_watchdog_monitoring();
    demonstrate_memory_safety();
    demonstrate_runtime_diagnostics();
    demonstrate_safe_state_transitions();
    demonstrate_deterministic_timing();
    demonstrate_misra_compliance();
    demonstrate_safety_checklist();

    std::cout << "\n" << std::string(68, '=') << "\n";
    std::cout << "All safety demonstrations completed successfully!\n";
    std::cout << "Remember: Safety certification requires formal verification
        ,\n";
    std::cout << "          extensive testing, and compliance with ISO 26262
        process.\n";
    std::cout << std::string(68, '=') << "\n\n";

    return 0;
}
```

35 Source Code: FuturePromiseAsync.cpp

File: `src/FuturePromiseAsync.cpp`
Repository: View on GitHub

```cpp
// FuturePromiseAsync.cpp
// Comprehensive guide to C++ concurrency primitives:
// - std::future and std::promise
// - std::packaged_task
// - std::async
// And how they relate to ASIO
//
// KEY CONCEPTS:
// 1. These are NOT alternatives to ASIO - they serve different purposes
// 2. future/promise/async are for TASK-BASED concurrency
// 3. ASIO is for ASYNCHRONOUS I/O and event-driven programming
// 4. They can be used together!
//
// WHEN TO USE WHAT:
// - std::async: Simplest, for fire-and-forget async tasks
// - std::promise/future: Manual control, producer-consumer pattern
// - std::packaged_task: Wrap callable, control execution timing
// - ASIO: Network I/O, timers, serial ports, event-driven architecture

#include <iostream>
#include <future>
#include <thread>
#include <chrono>
#include <functional>
#include <vector>
#include <queue>
#include <random>
#include <iomanip>

using namespace std::chrono_literals;

// ==========================================================================
// SECTION 1: std::async - The Simplest Way
// ==========================================================================

namespace async_examples {

// Simple async task
int calculate_sum(int a, int b) {
    std::cout << "  [async] Calculating " << a << " + " << b
              << " on thread " << std::this_thread::get_id() << "\n";
    std::this_thread::sleep_for(500ms);
    return a + b;
}

```

```
// Async task with exception
int divide(int a, int b) {
    std::cout << "  [async] Dividing " << a << " / " << b << "\n";
    std::this_thread::sleep_for(300ms);
    if (b == 0) {
        throw std::invalid_argument("Division by zero!");
    }
    return a / b;
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1: std::async - The Simplest Way ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Main thread ID: " << std::this_thread::get_id() << "\n\n";

    // 1. Launch async task (may run in new thread or deferred)
    std::cout << "1. Basic async (default policy):\n";
    auto future1 = std::async(calculate_sum, 10, 20);
    std::cout << "  Task launched, doing other work...\n";
    std::this_thread::sleep_for(200ms);
    std::cout << "  Getting result: " << future1.get() << "\n\n";

    // 2. Force async execution in new thread
    std::cout << "2. Force async (std::launch::async):\n";
    auto future2 = std::async(std::launch::async, calculate_sum, 15, 25);
    std::cout << "  Task running in parallel...\n";
    std::cout << "  Result: " << future2.get() << "\n\n";

    // 3. Deferred execution (lazy evaluation)
    std::cout << "3. Deferred execution (std::launch::deferred):\n";
    std::cout << "  Launching deferred task...\n";
    auto future3 = std::async(std::launch::deferred, calculate_sum, 5, 10);
    std::cout << "  Task not started yet!\n";
    std::cout << "  Calling get()... (task runs NOW on this thread)\n";
    std::cout << "  Result: " << future3.get() << "\n\n";

    // 4. Exception handling
    std::cout << "4. Exception handling with async:\n";
    auto future4 = std::async(std::launch::async, divide, 100, 0);
    try {
        std::cout << "  Getting result...\n";
        int result = future4.get();
        std::cout << "  Result: " << result << "\n";
    } catch (const std::exception& e) {
        std::cout << "   Caught exception: " << e.what() << "\n";
    }

    std::cout << "\n WHEN TO USE std::async:\n";
    std::cout << " •  Quick fire-and-forget async operations\n";
    std::cout << " •  Simple parallel computations\n";
    std::cout << " •  Don't need fine control over thread management\n";
    std::cout << " •  Want automatic exception propagation\n";
```

```
}

} // namespace async_examples

// ==========================================================================
// SECTION 2: std::promise and std::future - Producer-Consumer Pattern
// ==========================================================================

namespace promise_future_examples {

// Producer thread sets value via promise
void produce_value(std::promise<int> promise, int value) {
    std::cout << "  [Producer] Starting work...\n";
    std::this_thread::sleep_for(1s);
    std::cout << "  [Producer] Producing value: " << value << "\n";
    promise.set_value(value);  // Set the value
    std::cout << "  [Producer] Value set, exiting\n";
}

// Producer that fails
void produce_with_error(std::promise<int> promise) {
    std::cout << "  [Producer] Starting work...\n";
    std::this_thread::sleep_for(500ms);
    try {
        throw std::runtime_error("Production failed!");
    } catch (...) {
        std::cout << "  [Producer] Error occurred, setting exception\n";
        promise.set_exception(std::current_exception());
    }
}

// Multiple consumers waiting for same value
void consume_value(std::shared_future<int> future, int consumer_id) {
    std::cout << "  [Consumer " << consumer_id << "] Waiting for value...\n";
    int value = future.get();  // Blocks until value is ready
    std::cout << "  [Consumer " << consumer_id << "] Got value: " << value <<
        "\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 2: std::promise and std::future ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    // 1. Basic promise-future pair
    std::cout << "1. Basic producer-consumer with promise/future:\n";
    std::promise<int> promise1;
    std::future<int> future1 = promise1.get_future();

    std::thread producer1(produce_value, std::move(promise1), 42);
```

```

    std::cout << "  [Consumer] Waiting for result...\n";
    int result1 = future1.get();
    std::cout << "  [Consumer] Received: " << result1 << "\n\n";
    producer1.join();

    // 2. Exception propagation
    std::cout << "2. Exception handling with promise/future:\n";
    std::promise<int> promise2;
    std::future<int> future2 = promise2.get_future();

    std::thread producer2(produce_with_error, std::move(promise2));

    try {
        std::cout << "  [Consumer] Waiting for result...\n";
        int result2 = future2.get();
        std::cout << "  [Consumer] Received: " << result2 << "\n";
    } catch (const std::exception& e) {
        std::cout << "  [Consumer]  Caught exception: " << e.what() << "\n";
    }
    producer2.join();
    std::cout << "\n";

    // 3. Multiple consumers with shared_future
    std::cout << "3. Multiple consumers with shared_future:\n";
    std::promise<int> promise3;
    std::shared_future<int> shared_future = promise3.get_future().share();

    // Launch multiple consumers
    std::vector<std::thread> consumers;
    for (int i = 1; i <= 3; ++i) {
        consumers.emplace_back(consume_value, shared_future, i);
    }

    std::this_thread::sleep_for(500ms);
    std::cout << "  [Main] Setting value 100\n";
    promise3.set_value(100);

    for (auto& t : consumers) {
        t.join();
    }

    std::cout << "\n WHEN TO USE std::promise/future:\n";
    std::cout << " •  Need manual control over when value is set\n";
    std::cout << " •  Producer-consumer pattern across threads\n";
    std::cout << " •  Complex synchronization scenarios\n";
    std::cout << " •  Want to set value from a different location than task
        creation\n";
    std::cout << " •  Need shared_future for multiple consumers\n";
}

} // namespace promise_future_examples

//
```

```
    ==============================================================================

// SECTION 3: std::packaged_task - Wrap Callable, Control Execution
//
    ==============================================================================

namespace packaged_task_examples {

// Function to be wrapped
int compute_factorial(int n) {
    std::cout << "  [Task] Computing factorial of " << n << "\n";
    std::this_thread::sleep_for(500ms);
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}

// Task queue for thread pool simulation
class SimpleThreadPool {
    std::vector<std::thread> threads_;
    std::queue<std::function<void()>> tasks_;
    std::mutex mutex_;
    std::condition_variable cv_;
    bool stop_ = false;

public:
    SimpleThreadPool(size_t num_threads) {
        for (size_t i = 0; i < num_threads; ++i) {
            threads_.emplace_back([this, i] {
                std::cout << "  [Worker " << i << "] Started\n";
                while (true) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(mutex_);
                        cv_.wait(lock, [this] { return stop_ || !tasks_.empty
                            (); });

                        if (stop_ && tasks_.empty()) {
                            std::cout << "  [Worker " << i << "] Stopping\n";
                            return;
                        }

                        task = std::move(tasks_.front());
                        tasks_.pop();
                    }
                    std::cout << "  [Worker " << i << "] Executing task\n";
                    task();
                }
            });
        }
    }
```

```

    ~SimpleThreadPool() {
        {
            std::unique_lock<std::mutex> lock(mutex_);
            stop_ = true;
        }
        cv_.notify_all();
        for (auto& t : threads_) {
            t.join();
        }
    }

    template<typename F>
    void enqueue(F&& task) {
        {
            std::unique_lock<std::mutex> lock(mutex_);
            tasks_.push(std::forward<F>(task));
        }
        cv_.notify_one();
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 3: std::packaged_task ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    // 1. Basic packaged_task
    std::cout << "1. Basic packaged_task:\n";
    std::packaged_task<int(int)> task1(compute_factorial);
    std::future<int> future1 = task1.get_future();

    std::cout << "  Task created but not executed yet\n";
    std::cout << "  Launching task on new thread...\n";
    std::thread t1(std::move(task1), 5);

    std::cout << "  Waiting for result...\n";
    std::cout << "  Factorial(5) = " << future1.get() << "\n";
    t1.join();
    std::cout << "\n";

    // 2. Using packaged_task with thread pool
    std::cout << "2. Thread pool with packaged_task:\n";
    SimpleThreadPool pool(2);

    std::vector<std::future<int>> futures;

    for (int i = 3; i <= 6; ++i) {
        std::packaged_task<int(int)> task(compute_factorial);
        futures.push_back(task.get_future());

        // Enqueue task to thread pool - use shared_ptr for copy-ability
        auto task_ptr = std::make_shared<std::packaged_task<int(int)>>(std::
            move(task));
```

```
        int value = i;
        pool.enqueue([task_ptr, value]() {
            (*task_ptr)(value);
        });
    }

    std::cout << "\n  All tasks enqueued, waiting for results...\n\n";

    for (size_t i = 0; i < futures.size(); ++i) {
        int result = futures[i].get();
        std::cout << "  Result " << (i + 3) << ": factorial = " << result << "
            \n";
    }

    std::cout << "\ n WHEN TO USE std::packaged_task:\n";
    std::cout << " •  Wrapping callable objects for later execution\n";
    std::cout << " •  Implementing thread pools or task queues\n";
    std::cout << " •  Need to separate task creation from execution\n";
    std::cout << " •  Want to store tasks in containers\n";
    std::cout << " •  Building custom scheduling systems\n";
}

} // namespace packaged_task_examples

//
    ==========================================================================
// SECTION 4: Comparison Summary
//
    ==========================================================================

void show_comparison() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 4: When to Use What? ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "                                  \n";
    std::cout << "   MECHANISM         USE CASE
                                        \n";
    std::cout << "                                  \n";
    std::cout << "   std::async  •       Simplest way to run async tasks
                  \n";
    std::cout << "  •                    Fire-and-forget operations
                      \n";
    std::cout << "  •                    Quick parallel computations
                     \n";
    std::cout << "  •                    Don't need thread control
                        \n";
    std::cout << "                      Best for: Simple async operations
                \n";
    std::cout << "                                  \n";
    std::cout << "   promise/future  •   Producer-consumer pattern
                        \n";
```

```
    std::cout << "  •                    Set value from different location \n";
    std::cout << "  •                    Complex synchronization \n";
    std::cout << "  •                    Multiple consumers (shared_future) \n";
    std::cout << "                       Best for: Manual control over value \n";
    std::cout << "                                 \n";
    std::cout << "   packaged_task  •    Thread pools / task queues \n";
    std::cout << "  •                    Separate creation from execution \n";
    std::cout << "  •                    Store tasks in containers \n";
    std::cout << "  •                    Custom scheduling systems \n";
    std::cout << "                       Best for: Building task systems \n";
    std::cout << "                                 \n";
}

// ============================================================================
// SECTION 5: Relationship with ASIO
// ============================================================================

void explain_asio_relationship() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 5: Are They Alternatives to ASIO? ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "SHORT ANSWER: NO - They serve DIFFERENT purposes!\n\n";

    std::cout << "                                 \n";
    std::cout << "   future/promise/async (Standard Library Concurrency) \n";
    std::cout << "                                 \n";
    std::cout << "   PURPOSE:                                                \n";
    std::cout << "  •    Task-based concurrency                             \n";
    std::cout << "  •    CPU-bound computations                             \n";
    std::cout << "  •    Running functions asynchronously                   \n";
    std::cout << "  •    Getting results from other threads                 \n";
    std::cout << "                                                          \n";
    std::cout << "   EXAMPLES:
```

```
                                                                    \n";
    std::cout << "  •    Parallel calculations                                  \n";
    std::cout << "  •    Image processing in background                         \n";
    std::cout << "  •    File compression                                       \n";
    std::cout << "  •    Any CPU-intensive work                                 \n";
    std::cout << "                                                                                    \n";
    std::cout << "   LIMITATIONS:                                                                     \n";
    std::cout << "      Not designed for I/O operations                         \n";
    std::cout << "      No built-in event loop                                  \n";
    std::cout << "      No socket/network abstractions                          \n";
    std::cout << "      Limited scalability for many I/O operations             \n";
    std::cout << "                                                  \n\n";

    std::cout << "                                                  \n";
    std::cout << "   ASIO (Asynchronous I/O Library)                            \n";
    std::cout << "                                                  \n";
    std::cout << "   PURPOSE:                                                                         \n";
    std::cout << "  •    Asynchronous I/O operations                            \n";
    std::cout << "  •    Network programming (sockets, TCP/UDP)                 \n";
    std::cout << "  •    Event-driven architecture                              \n";
    std::cout << "  •    Timers and signals                                     \n";
    std::cout << "  •    Serial ports                                           \n";
    std::cout << "                                                                                    \n";
    std::cout << "   EXAMPLES:                                                                        \n";
    std::cout << "  •    HTTP/REST servers and clients                          \n";
    std::cout << "  •    WebSocket servers                                      \n";
    std::cout << "  •    Chat applications                                      \n";
    std::cout << "  •    Real-time data streaming                               \n";
    std::cout << "  •    Timers and periodic tasks                              \n";
    std::cout << "
```

```
                                                                    \n";
    std::cout << "   STRENGTHS:                                          \n";
    std::cout << "      Optimized for I/O-bound operations                \n";
    std::cout << "      Event loop / io_context                           \n";
    std::cout << "      Handles thousands of connections efficiently      \n";
    std::cout << "      Cross-platform I/O abstractions                   \n";
    std::cout << "      Proactor pattern                                  \n";
    std::cout << "                                                        \n\n";

    std::cout << "                                                        \n";
    std::cout << "   THEY ARE COMPLEMENTARY - Use Together!               \n";
    std::cout << "                                                        \n";
    std::cout << "                                                                    \n";
    std::cout << "   EXAMPLE 1: ASIO for I/O + async for CPU work         \n";
    std::cout << " •    Use ASIO to handle HTTP requests                 \n";
    std::cout << " •    Use std::async to process images in parallel     \n";
    std::cout << " •    Return result via ASIO response                  \n";
    std::cout << "                                                                    \n";
    std::cout << "   EXAMPLE 2: ASIO for networking + packaged_task for jobs \n";
    std::cout << " •    ASIO receives network messages                   \n";
    std::cout << " •    Queue CPU-intensive tasks with packaged_task     \n";
    std::cout << " •    Thread pool processes tasks                      \n";
    std::cout << " •    ASIO sends results back over network             \n";
    std::cout << "                                                                    \n";
    std::cout << "   EXAMPLE 3: ASIO timers + promise/future for results  \n";
    std::cout << " •    ASIO timer for periodic checks                   \n";
    std::cout << " •    Use promise/future to coordinate between timers  \n";
    std::cout << " •    Combine I/O events with task results             \n";
    std::cout << "                                                                    \n";
    std::cout << "                                                        \n\n";
```

```

    std::cout << "DECISION TREE:\n";
    std::cout << "    Need async I/O (network, files, timers)?  → Use ASIO\n";
    std::cout << "   \n";
    std::cout << "    Need CPU-bound parallel computation?    → Use std::async\n";
    std::cout << "   \n";
    std::cout << "    Need producer-consumer pattern?      → Use promise/future\n";
    std::cout << "   \n";
    std::cout << "    Building thread pool / task queue?     → Use packaged_task\n";
    std::cout << "   \n";
    std::cout << "    Need both I/O AND computation?      → Use ASIO + future/async\n";
}

// ============================================================================
// SECTION 6: Practical Example - Combining Them
// ============================================================================

namespace combined_example {

// Simulate a web server scenario
class RequestProcessor {
public:
    // Simulate receiving HTTP request (would use ASIO in real app)
    static std::string receive_request() {
        std::cout << "  [ASIO would be here] Received HTTP request\n";
        return "process_image:photo.jpg";
    }

    // CPU-intensive image processing (use std::async)
    static std::string process_image(const std::string& filename) {
        std::cout << "  [Worker] Processing image: " << filename << "\n";
        std::this_thread::sleep_for(1s);  // Simulate heavy processing
        return "processed_" + filename;
    }

    // Send response (would use ASIO in real app)
    static void send_response(const std::string& result) {
        std::cout << "  [ASIO would be here] Sending response: " << result << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 6: Practical Example - ASIO + async ===\n";
    std::cout << std::string(70, '=') << "\n\n";
```

```

    std::cout << "Scenario: Web server handling image processing requests\n\n";

    // 1. ASIO would receive request (simulated)
    std::string request = RequestProcessor::receive_request();

    // 2. Parse request
    auto pos = request.find(':');
    std::string filename = request.substr(pos + 1);

    // 3. Use std::async for CPU-intensive work (doesn't block I/O thread)
    std::cout << "  Launching async image processing...\n";
    auto future = std::async(std::launch::async,
                             RequestProcessor::process_image,
                             filename);

    std::cout << "  ASIO thread is free to handle other requests!\n";
    std::this_thread::sleep_for(200ms);
    std::cout << "  (Handling other requests...)\n";
    std::this_thread::sleep_for(300ms);

    // 4. Get result and send response
    std::cout << "  Waiting for processing to complete...\n";
    std::string result = future.get();

    // 5. ASIO would send response (simulated)
    RequestProcessor::send_response(result);

    std::cout << "\n This is how you combine them:\n";
    std::cout << " •  ASIO handles network I/O (non-blocking)\n";
    std::cout << " •  std::async handles CPU work (parallel)\n";
    std::cout << " •  Best of both worlds!\n";
}

} // namespace combined_example

// ============================================================================
// MAIN FUNCTION
// ============================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "      C++ Concurrency: future, promise, async, packaged_task \n";
    std::cout << "             And their relationship with ASIO \n";
    std::cout << "                                    \n";

```

```
    // Section 1: std::async
    async_examples::demonstrate();

    // Section 2: promise/future
    promise_future_examples::demonstrate();

    // Section 3: packaged_task
    packaged_task_examples::demonstrate();

    // Section 4: Comparison
    show_comparison();

    // Section 5: ASIO relationship
    explain_asio_relationship();

    // Section 6: Combined example
    combined_example::demonstrate();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "KEY TAKEAWAYS:\n";
    std::cout << "1. future/promise/async are for TASK-BASED concurrency\n";
    std::cout << "2. ASIO is for ASYNCHRONOUS I/O and event-driven programming
        \n";
    std::cout << "3. They are NOT alternatives - use them TOGETHER!\n";
    std::cout << "4. std::async: simplest for parallel tasks\n";
    std::cout << "5. promise/future: manual control, producer-consumer\n";
    std::cout << "6. packaged_task: thread pools, task queues\n";
    std::cout << "7. ASIO: network I/O, timers, scalable servers\n\n";

    return 0;
}
```

36 Source Code: GenericLambdas.cpp

File: `src/GenericLambdas.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
#include <map>
#include <memory>

// ==================================================================
// 1. BASIC GENERIC LAMBDA (C++14)
// ==================================================================
void example_basic_generic_lambda() {
    std::cout << "\n=== 1. BASIC GENERIC LAMBDA (C++14) ===" << std::endl;

    // Generic lambda that works with any type
    auto identity = [](auto x) {
        return x;
    };

    std::cout << "Identity int: " << identity(42) << std::endl;
    std::cout << "Identity double: " << identity(3.14) << std::endl;
    std::cout << "Identity string: " << identity(std::string("Hello")) << std::endl;
    std::cout << "Identity char: " << identity('A') << std::endl;
}

// ==================================================================
// 2. GENERIC LAMBDA WITH TYPE OPERATIONS
// ==================================================================
void example_generic_lambda_operations() {
    std::cout << "\n=== 2. GENERIC LAMBDA WITH TYPE OPERATIONS ===" << std::endl;

    // Generic lambda that adds two values
    auto add = [](auto a, auto b) {
        return a + b;
    };

    std::cout << "Add ints: " << add(10, 20) << std::endl;
    std::cout << "Add doubles: " << add(3.14, 2.86) << std::endl;
    std::cout << "Add strings: " << add(std::string("Hello "), std::string("World")) << std::endl;

    // Generic lambda for multiplication
    auto multiply = [](auto a, auto b) {
        return a * b;
    };

```

```
    std::cout << "Multiply ints: " << multiply(5, 6) << std::endl;
    std::cout << "Multiply doubles: " << multiply(2.5, 4.0) << std::endl;
}

// ==================================================================
// 3. GENERIC LAMBDA WITH CONTAINERS
// ==================================================================
void example_generic_lambda_containers() {
    std::cout << "\n=== 3. GENERIC LAMBDA WITH CONTAINERS ===" << std::endl;

    // Generic lambda to print container elements
    auto print_container = [](const auto& container, const std::string& name)
        {
        std::cout << name << ": ";
        for (const auto& elem : container) {
            std::cout << elem << " ";
        }
        std::cout << std::endl;
    };

    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::vector<std::string> vec_str = {"Hello", "World", "C++14"};

    print_container(vec, "Vector<int>");
    print_container(vec_str, "Vector<string>");
}

// ==================================================================
// 4. GENERIC LAMBDA WITH STL ALGORITHMS
// ==================================================================
void example_generic_lambda_algorithms() {
    std::cout << "\n=== 4. GENERIC LAMBDA WITH STL ALGORITHMS ===" << std::
        endl;

    std::vector<int> numbers = {5, 2, 8, 1, 9, 3};

    // Generic lambda for comparison
    auto greater_than = [](auto a, auto b) {
        return a > b;
    };

    std::sort(numbers.begin(), numbers.end(), greater_than);

    std::cout << "Sorted (descending): ";
    for (auto n : numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Generic lambda for transformation
    auto square = [](auto x) {
        return x * x;
    };

```

```
    std::vector<int> squared(numbers.size());
    std::transform(numbers.begin(), numbers.end(), squared.begin(), square);

    std::cout << "Squared: ";
    for (auto n : squared) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 5. GENERIC LAMBDA WITH MULTIPLE AUTO PARAMETERS
// ==================================================================
void example_multiple_auto_parameters() {
    std::cout << "\n=== 5. GENERIC LAMBDA WITH MULTIPLE AUTO PARAMETERS ==="
        << std::endl;

    // Generic lambda with three different types
    auto combine = [](auto a, auto b, auto c) {
        std::cout << "Types can differ: " << a << ", " << b << ", " << c <<
            std::endl;
        return true;
    };

    combine(42, 3.14, "Hello");
    combine(std::string("World"), 100, 'X');
}

// ==================================================================
// 6. GENERIC LAMBDA WITH VARIADIC PARAMETERS
// ==================================================================
void example_generic_lambda_variadic() {
    std::cout << "\n=== 6. GENERIC LAMBDA WITH VARIADIC PARAMETERS ===" << std
        ::endl;

    // Generic lambda that sums any number of arguments
    auto sum_all = [](auto... args) {
        return (args + ...);  // C++17 fold expression
    };

    std::cout << "Sum of ints: " << sum_all(1, 2, 3, 4, 5) << std::endl;
    std::cout << "Sum of doubles: " << sum_all(1.1, 2.2, 3.3) << std::endl;
    std::cout << "Sum mixed: " << sum_all(1, 2.5, 3) << std::endl;
}

// ==================================================================
// 7. GENERIC LAMBDA FOR FACTORY PATTERN
// ==================================================================
void example_generic_lambda_factory() {
    std::cout << "\n=== 7. GENERIC LAMBDA FOR FACTORY PATTERN ===" << std::
        endl;

    // Generic factory lambda
    auto make_unique_ptr = [](auto value) {
```

```
        using T = decltype(value);
        return std::make_unique<T>(value);
    };

    auto int_ptr = make_unique_ptr(42);
    auto str_ptr = make_unique_ptr(std::string("Hello"));
    auto double_ptr = make_unique_ptr(3.14);

    std::cout << "Int unique_ptr: " << *int_ptr << std::endl;
    std::cout << "String unique_ptr: " << *str_ptr << std::endl;
    std::cout << "Double unique_ptr: " << *double_ptr << std::endl;
}

// ==================================================================
// 8. GENERIC LAMBDA WITH MAP OPERATIONS
// ==================================================================
void example_generic_lambda_map() {
    std::cout << "\n=== 8. GENERIC LAMBDA WITH MAP OPERATIONS ===" << std::
        endl;

    std::map<std::string, int> age_map = {
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35}
    };

    // Generic lambda to print key-value pairs
    auto print_map = [](const auto& map, const std::string& name) {
        std::cout << name << ":" << std::endl;
        for (const auto& [key, value] : map) {
            std::cout << "  " << key << " -> " << value << std::endl;
        }
    };

    print_map(age_map, "Age Map");

    // Generic lambda for filtering
    auto filter_by_value = [](const auto& map, auto threshold) {
        std::vector<typename std::decay<decltype(map)>::type::key_type> result
            ;
        for (const auto& [key, value] : map) {
            if (value > threshold) {
                result.push_back(key);
            }
        }
        return result;
    };

    auto filtered = filter_by_value(age_map, 28);
    std::cout << "People older than 28: ";
    for (const auto& name : filtered) {
        std::cout << name << " ";
    }
    std::cout << std::endl;
```

```
}

// ==================================================================
// 9. GENERIC LAMBDA AS PREDICATE
// ==================================================================
void example_generic_lambda_predicate() {
    std::cout << "\n=== 9. GENERIC LAMBDA AS PREDICATE ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Generic lambda predicate
    auto is_even = [](auto n) {
        return n % 2 == 0;
    };

    auto count = std::count_if(numbers.begin(), numbers.end(), is_even);
    std::cout << "Even numbers count: " << count << std::endl;

    // Find first even number
    auto it = std::find_if(numbers.begin(), numbers.end(), is_even);
    if (it != numbers.end()) {
        std::cout << "First even number: " << *it << std::endl;
    }
}

// ==================================================================
// 10. GENERIC LAMBDA WITH RETURN TYPE DEDUCTION
// ==================================================================
void example_return_type_deduction() {
    std::cout << "\n=== 10. GENERIC LAMBDA WITH RETURN TYPE DEDUCTION ===" <<
        std::endl;

    // Lambda with auto return type - deduced from expression
    auto double_value = [](auto x) {
        return x * 2;
    };

    // Lambda with explicit return type
    auto to_string_value = [](auto x) -> std::string {
        return std::to_string(x);
    };

    std::cout << "Double int: " << double_value(21) << std::endl;
    std::cout << "Double double: " << double_value(3.14) << std::endl;
    std::cout << "To string (int): " << to_string_value(42) << std::endl;
    std::cout << "To string (double): " << to_string_value(3.14159) << std::
        endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n==============================================" << std::
```

```
        endl;
    std::cout << "    C++14 GENERIC LAMBDAS EXAMPLES" << std::endl;
    std::cout << "================================================" << std::
        endl;

    example_basic_generic_lambda();
    example_generic_lambda_operations();
    example_generic_lambda_containers();
    example_generic_lambda_algorithms();
    example_multiple_auto_parameters();
    example_generic_lambda_variadic();
    example_generic_lambda_factory();
    example_generic_lambda_map();
    example_generic_lambda_predicate();
    example_return_type_deduction();

    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    ALL EXAMPLES COMPLETED" << std::endl;
    std::cout << "================================================\n" << std::
        endl;

    return 0;
}
```

37 Source Code: InheritanceTypes.cpp

File: `src/InheritanceTypes.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <algorithm>

// ====================================================================
// PRIVATE, PROTECTED, AND PUBLIC INHERITANCE IN C++
// ====================================================================

// ====================================================================
// 1. PUBLIC INHERITANCE - "IS-A" RELATIONSHIP
// ====================================================================

class Animal {
protected:
    std::string name;

public:
    Animal(const std::string& n) : name(n) {}
    virtual ~Animal() = default;

    virtual void makeSound() const {
        std::cout << name << " makes a sound" << std::endl;
    }

    void eat() const {
        std::cout << name << " is eating" << std::endl;
    }
};

// Public inheritance: Dog IS-A Animal
// All public members of Animal remain public in Dog
class Dog : public Animal {
public:
    Dog(const std::string& n) : Animal(n) {}

    void makeSound() const override {
        std::cout << name << " barks: Woof!" << std::endl;
    }
};

void example_public_inheritance() {
    std::cout << "\n=== 1. PUBLIC INHERITANCE (IS-A) ===" << std::endl;
    std::cout << "Use case: Dog IS-A Animal (polymorphism, substitutability)\n
        " << std::endl;

    Dog dog("Buddy");
    dog.makeSound();  //  Can call inherited method
```

```
    dog.eat();        //  Can call inherited method

    //  Polymorphism works with public inheritance
    Animal* animal_ptr = &dog;
    animal_ptr->makeSound();  // Calls Dog::makeSound()

    std::cout << "\n PUBLIC INHERITANCE:" << std::endl;
    std::cout << " •   Dog IS-A Animal" << std::endl;
    std::cout << " •   Public members stay public" << std::endl;
    std::cout << " •   Protected members stay protected" << std::endl;
    std::cout << " •   Polymorphism works (can upcast to base)" << std::endl;
    std::cout << " •   Use for: Substitutability, \"is-a\" relationships" <<
        std::endl;
}

// ==================================================================
// 2. PRIVATE INHERITANCE - "IMPLEMENTED-IN-TERMS-OF"
// ==================================================================

// Base class providing timer functionality
class Timer {
private:
    int ticks = 0;

public:
    void tick() { ++ticks; }
    void reset() { ticks = 0; }
    int getTicks() const { return ticks; }

    void display() const {
        std::cout << "Timer: " << ticks << " ticks" << std::endl;
    }
};

// Private inheritance: Widget is IMPLEMENTED using Timer
// Timer's public interface becomes private in Widget
class Widget : private Timer {  // "implemented-in-terms-of"
private:
    std::string id;

public:
    Widget(const std::string& i) : id(i) {}

    // Widget controls which Timer methods to expose
    void update() {
        tick();  //  Can call Timer::tick() internally
        std::cout << "Widget " << id << " updated" << std::endl;
    }

    void resetState() {
        reset();  //  Can call Timer::reset() internally
    }

    // Expose only specific functionality
```

```
    void showUpdateCount() const {
        std::cout << "Widget " << id << " has been updated "
                  << getTicks() << " times" << std::endl;
    }

    // Note: Timer::display() is NOT exposed
};

void example_private_inheritance() {
    std::cout << "\n=== 2. PRIVATE INHERITANCE (IMPLEMENTED-IN-TERMS-OF) ===" << std::endl;
    std::cout << "Use case: Widget is IMPLEMENTED using Timer (code reuse, not substitutability)\n" << std::endl;

    Widget widget("W1");
    widget.update();
    widget.update();
    widget.update();
    widget.showUpdateCount();

    //  Cannot access Timer's public methods from outside
    // widget.tick();     // Compile error! tick() is private in Widget
    // widget.display();  // Compile error! display() is private in Widget

    //  Cannot upcast to Timer (no polymorphism)
    // Timer* timer_ptr = &widget;  // Compile error! Private inheritance breaks is-a

    std::cout << "\n PRIVATE INHERITANCE:" << std::endl;
    std::cout << " •   Widget is IMPLEMENTED using Timer" << std::endl;
    std::cout << " •   All base members become private" << std::endl;
    std::cout << " •   No polymorphism (cannot upcast)" << std::endl;
    std::cout << " •   Base class is an implementation detail" << std::endl;
    std::cout << " •   Use for: Code reuse without exposing interface" << std::endl;
}

// ==================================================================
// 3. PROTECTED INHERITANCE - "IMPLEMENTED-IN-TERMS-OF" FOR FURTHER DERIVATION
// ==================================================================

class Logger {
public:
    void log(const std::string& msg) const {
        std::cout << "[LOG] " << msg << std::endl;
    }

    void debug(const std::string& msg) const {
        std::cout << "[DEBUG] " << msg << std::endl;
    }
};

// Protected inheritance: Component uses Logger, allows derived classes to access it
```

```cpp
class Component : protected Logger {
protected:
    std::string name;

public:
    Component(const std::string& n) : name(n) {
        log("Component " + name + " created");  //  Can use Logger internally
    }

    void doWork() {
        debug("Component " + name + " is working");
    }
};

// Derived class can access Logger through protected inheritance
class AdvancedComponent : public Component {
public:
    AdvancedComponent(const std::string& n) : Component(n) {
        //  Can access Logger methods because of protected inheritance
        log("Advanced features initialized");
    }

    void advancedWork() {
        debug("Advanced component processing");  //  Logger accessible here
        doWork();
    }
};

void example_protected_inheritance() {
    std::cout << "\n=== 3. PROTECTED INHERITANCE (IMPLEMENTED-IN-TERMS-OF + DERIVATION) ===" << std::endl;
    std::cout << "Use case: Component uses Logger, derived classes can also use it\n" << std::endl;

    AdvancedComponent comp("AC1");
    comp.advancedWork();

    //  Cannot access Logger from outside
    // comp.log("test");   // Compile error! log() is protected in Component
    // comp.debug("test"); // Compile error! debug() is protected in Component

    //  Cannot upcast to Logger
    // Logger* logger_ptr = &comp;  // Compile error! Protected inheritance

    std::cout << "\n PROTECTED INHERITANCE:" << std::endl;
    std::cout << " •  Component is IMPLEMENTED using Logger" << std::endl;
    std::cout << " •  All base public members become protected" << std::endl;
    std::cout << " •  Derived classes can access base interface" << std::endl;
    std::cout << " •  No polymorphism (cannot upcast)" << std::endl;
    std::cout << " •  Use for: Implementation detail shared with derived classes" << std::endl;
}
```

```cpp
// ==================================================================
// 4. REAL-WORLD EXAMPLE: STACK IMPLEMENTED USING VECTOR
// ==================================================================

// Private inheritance: Stack is implemented using vector
template<typename T>
class Stack : private std::vector<T> {
    using Base = std::vector<T>;

public:
    // Expose only stack operations
    void push(const T& value) {
        Base::push_back(value);
    }

    void push(T&& value) {
        Base::push_back(std::move(value));
    }

    void pop() {
        if (!Base::empty()) {
            Base::pop_back();
        }
    }

    const T& top() const {
        return Base::back();
    }

    bool empty() const {
        return Base::empty();
    }

    size_t size() const {
        return Base::size();
    }

    // Note: We DON'T expose vector's random access, insert, erase, etc.
    // This enforces stack semantics (LIFO)
};

void example_stack_private_inheritance() {
    std::cout << "\n=== 4. REAL-WORLD: STACK USING PRIVATE INHERITANCE ===" << std::endl;
    std::cout << "Use case: Implement Stack using vector, but hide vector's interface\n" << std::endl;

    Stack<int> stack;
    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Stack top: " << stack.top() << std::endl;
    std::cout << "Stack size: " << stack.size() << std::endl;
```

```

    stack.pop();
    std::cout << "After pop, top: " << stack.top() << std::endl;

    //  Vector operations are hidden
    // stack[0] = 100;        // Compile error! operator[] not accessible
    // stack.insert(...);     // Compile error! insert() not accessible
    // stack.at(0);           // Compile error! at() not accessible

    std::cout << "\n BENEFIT:" << std::endl;
    std::cout << " •   Reuses vector's implementation" << std::endl;
    std::cout << " •   Enforces stack semantics (LIFO only)" << std::endl;
    std::cout << " •   Users cannot break stack invariants" << std::endl;
}

// ==================================================================
// 5. COMPARISON: PRIVATE INHERITANCE VS COMPOSITION
// ==================================================================

// Option 1: Private inheritance
class WindowPrivate : private Timer {
public:
    void refresh() {
        tick();
        std::cout << "Window refreshed " << getTicks() << " times" << std::
            endl;
    }
};

// Option 2: Composition (often preferred)
class WindowComposition {
private:
    Timer timer;  // Has-a Timer

public:
    void refresh() {
        timer.tick();
        std::cout << "Window refreshed " << timer.getTicks() << " times" <<
            std::endl;
    }
};

void example_private_vs_composition() {
    std::cout << "\n=== 5. PRIVATE INHERITANCE VS COMPOSITION ===" << std::
        endl;

    std::cout << "\nPrivate inheritance:" << std::endl;
    WindowPrivate win1;
    win1.refresh();
    win1.refresh();

    std::cout << "\nComposition:" << std::endl;
    WindowComposition win2;
    win2.refresh();
```

```
    win2.refresh();

    std::cout << "\n WHEN TO USE PRIVATE INHERITANCE:" << std::endl;
    std::cout << "     Need to override virtual functions from base" << std::endl;
    std::cout << "     Need access to protected members of base" << std::endl;
    std::cout << "     Want Empty Base Optimization (EBO) for zero-size bases" << std::endl;

    std::cout << "\n WHEN TO USE COMPOSITION (USUALLY PREFERRED):" << std::endl;
    std::cout << "     Don't need to override virtual functions" << std::endl;
    std::cout << "     Don't need access to protected members" << std::endl;
    std::cout << "     Want to hold multiple instances" << std::endl;
    std::cout << "     Want more explicit \"has-a\" relationship" << std::endl;
    std::cout << "     Better encapsulation and flexibility" << std::endl;
}

// ==================================================================
// 6. OVERRIDING VIRTUAL FUNCTIONS WITH PRIVATE INHERITANCE
// ==================================================================

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

    virtual void onEvent() {
        std::cout << "Observable: Event occurred" << std::endl;
    }

    void triggerEvent() {
        std::cout << "Triggering event..." << std::endl;
        onEvent();
    }
};

// Private inheritance to customize behavior
class Sensor : private Observable {
private:
    std::string sensorId;
    int reading = 0;

    // Override virtual function from Observable
    void onEvent() override {
        reading++;
        std::cout << "Sensor " << sensorId << " received event, reading: " << reading << std::endl;
    }

public:
    Sensor(const std::string& id) : sensorId(id) {}

    void simulate() {
```

```
        // Call base class method that will call our overridden onEvent()
        triggerEvent();
    }
};

void example_override_with_private() {
    std::cout << "\n=== 6. OVERRIDING VIRTUAL FUNCTIONS (PRIVATE INHERITANCE) ===" << std::endl;
    std::cout << "Use case: Customize base class behavior without exposing base interface\n" << std::endl;

    Sensor sensor("TEMP-01");
    sensor.simulate();
    sensor.simulate();
    sensor.simulate();

    std::cout << "\n KEY POINT:" << std::endl;
    std::cout << " •   Can override virtual functions even with private inheritance" << std::endl;
    std::cout << " •   Base class doesn't need to know about derived class" << std::endl;
    std::cout << " •   This is the main reason to use private inheritance over composition" << std::endl;
}

// ==================================================================
// 7. USING DECLARATION TO SELECTIVELY EXPOSE MEMBERS
// ==================================================================

class Engine {
public:
    void start() { std::cout << "Engine started" << std::endl; }
    void stop() { std::cout << "Engine stopped" << std::endl; }
    void diagnose() { std::cout << "Running diagnostics..." << std::endl; }
    void internalMaintenance() { std::cout << "Internal maintenance" << std::endl; }
};

class Car : private Engine {
public:
    // Selectively expose only specific Engine methods
    using Engine::start;  // Make start() public
    using Engine::stop;   // Make stop() public

    // Note: diagnose() and internalMaintenance() remain private

    void drive() {
        start();
        std::cout << "Car is driving" << std::endl;
    }
};

void example_using_declaration() {
    std::cout << "\n=== 7. USING DECLARATION - SELECTIVE EXPOSURE ===" << std
```

```
    std::cout << "Use case: Expose only specific base class methods\n" << std::endl;

    Car car;
    car.start();     //  Works - exposed via using declaration
    car.drive();
    car.stop();      //  Works - exposed via using declaration

    //  These are not exposed
    // car.diagnose();            // Compile error!
    // car.internalMaintenance(); // Compile error!

    std::cout << "\n BENEFIT:" << std::endl;
    std::cout << " •   Fine-grained control over interface" << std::endl;
    std::cout << " •   Can expose some methods while hiding others" << std::endl;
    std::cout << " •   More maintainable than forwarding each method manually" << std::endl;
}

// ==================================================================
// 8. ACCESS LEVEL SUMMARY TABLE
// ==================================================================

void example_access_summary() {
    std::cout << "\n=== 8. ACCESS LEVEL SUMMARY ===" << std::endl;

    std::cout << "\                                n" << std::endl;
    std::cout << "  Base Class Member         Public       Protected       Private    " << std::endl;
    std::cout << "  Access Level            Inheritance   Inheritance    Inheritance  " << std::endl;
    std::cout << "                                  " << std::endl;
    std::cout << "  public                    public       protected       private    " << std::endl;
    std::cout << "  protected                protected     protected       private    " << std::endl;
    std::cout << "  private                   (hidden)      (hidden)      (hidden)   " << std::endl;
    std::cout << "                                  " << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  PRIVATE, PROTECTED, AND PUBLIC INHERITANCE IN C++" << std::endl;
    std::cout << "
```

```
        ================================================================" << std::endl;

    example_public_inheritance();
    example_private_inheritance();
    example_protected_inheritance();
    example_stack_private_inheritance();
    example_private_vs_composition();
    example_override_with_private();
    example_using_declaration();
    example_access_summary();

    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  DECISION GUIDE: WHICH INHERITANCE TYPE TO USE?" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\n USE PUBLIC INHERITANCE WHEN:" << std::endl;
    std::cout << " •   Derived class IS-A base class" << std::endl;
    std::cout << " •   You need polymorphism and substitutability" << std::endl;
    std::cout << " •   Liskov Substitution Principle applies" << std::endl;
    std::cout << " •   Example: Dog is-a Animal, Circle is-a Shape" << std::endl;
    std::cout << " •   Usage: ~95% of all inheritance cases" << std::endl;

    std::cout << "\n USE PRIVATE INHERITANCE WHEN:" << std::endl;
    std::cout << " •   Derived class is IMPLEMENTED using base class" << std::endl;
    std::cout << " •   Need to override virtual functions from base" << std::endl;
    std::cout << " •   Need access to protected members of base" << std::endl;
    std::cout << " •   Want Empty Base Optimization (EBO)" << std::endl;
    std::cout << " •   Example: Stack implemented using vector" << std::endl;
    std::cout << " •   Usage: <5% of inheritance cases" << std::endl;
    std::cout << " •   Alternative: Prefer composition if possible" << std::endl;

    std::cout << "\n USE PROTECTED INHERITANCE WHEN:" << std::endl;
    std::cout << " •   Same as private inheritance, but..." << std::endl;
    std::cout << " •   Further derived classes need access to base interface" << std::endl;
    std::cout << " •   Creating a hierarchy of implementation details" << std::endl;
    std::cout << " •   Example: Component hierarchy sharing Logger" << std::endl;
    std::cout << " •   Usage: <1% of inheritance cases (very rare)" << std::endl;
    std::cout << " •   Alternative: Usually better to use composition" << std::endl;
```

```

    std::cout << "\n COMPOSITION VS PRIVATE INHERITANCE:" << std::endl;
    std::cout << " •   Default to COMPOSITION (has-a relationship)" << std::endl;
    std::cout << " •   Use private inheritance ONLY if:" << std::endl;
    std::cout << "     - Need to override virtual functions, OR" << std::endl;
    std::cout << "     - Need access to protected members, OR" << std::endl;
    std::cout << "     - Need Empty Base Optimization" << std::endl;
    std::cout << " •   Composition is more explicit and flexible" << std::endl;

    std::cout << "\n MODERN C++ BEST PRACTICES:" << std::endl;
    std::cout << " •   Public inheritance: Use for polymorphism" << std::endl;
    std::cout << " •   Private inheritance: Rare, prefer composition" << std::endl;
    std::cout << " •   Protected inheritance: Almost never use" << std::endl;
    std::cout << " •   If unsure: Choose composition over private inheritance" << std::endl;
    std::cout << " •   Scott Meyers: \"Prefer composition to private inheritance\"" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

38 Source Code: InsertAndDeleteNodes.cpp

File: `src/InsertAndDeleteNodes.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
using namespace std;

struct ListNode {
       int val;
       ListNode *next;
       ListNode(int x) : val(x), next(NULL) {}
};

class Solution
{
public:
    void PrintListNode(ListNode* inListNode)  {
         if (inListNode != NULL) {
            cout << inListNode->val << " -> ";
            PrintListNode(inListNode->next);
         } else {
            cout << "NULL" << endl;
         }
    }

    void Test(ListNode* inListNode) {
         delete inListNode;
         inListNode = NULL;
         cout << inListNode << endl;

         PrintListNode(inListNode);
    }

    // Insert node at the beginning of the list
    ListNode* InsertNodeAtHead(ListNode* pHead, int value) {
         cout << "Entry: InsertNodeAtHead(value=" << value << ")" << endl;

         ListNode* pNewNode = new ListNode(value);
         pNewNode->next = pHead;

         cout << "Exit: InsertNodeAtHead" << endl;
         return pNewNode;  // New head
    }

    // Insert node at the end of the list
    ListNode* InsertNodeAtTail(ListNode* pHead, int value) {
         cout << "Entry: InsertNodeAtTail(value=" << value << ")" << endl;

         ListNode* pNewNode = new ListNode(value);

         // Empty list case
         if (pHead == NULL) {
            cout << "Exit: InsertNodeAtTail (empty list)" << endl;
```

```
            return pNewNode;
        }

        // Traverse to the end
        ListNode* pCur = pHead;
        while (pCur->next != NULL) {
            pCur = pCur->next;
        }

        pCur->next = pNewNode;

        cout << "Exit: InsertNodeAtTail" << endl;
        return pHead;
    }

    // Insert node after a specific node
    void InsertNodeAfter(ListNode* pPrevNode, int value) {
        cout << "Entry: InsertNodeAfter(value=" << value << ")" << endl;

        if (pPrevNode == NULL) {
            cout << "Error: Previous node cannot be NULL" << endl;
            return;
        }

        ListNode* pNewNode = new ListNode(value);
        pNewNode->next = pPrevNode->next;
        pPrevNode->next = pNewNode;

        cout << "Exit: InsertNodeAfter" << endl;
    }

    // Insert node at a specific position (0-indexed)
    ListNode* InsertNodeAtPosition(ListNode* pHead, int position, int value) {
        cout << "Entry: InsertNodeAtPosition(pos=" << position << ", value="
            << value << ")" << endl;

        // Insert at head
        if (position == 0) {
            return InsertNodeAtHead(pHead, value);
        }

        // Traverse to position-1
        ListNode* pCur = pHead;
        for (int i = 0; i < position - 1 && pCur != NULL; i++) {
            pCur = pCur->next;
        }

        if (pCur == NULL) {
            cout << "Error: Position out of bounds" << endl;
            return pHead;
        }

        InsertNodeAfter(pCur, value);

```

```
        cout << "Exit: InsertNodeAtPosition" << endl;
        return pHead;
    }

    // Insert node in sorted list (ascending order)
    ListNode* InsertNodeSorted(ListNode* pHead, int value) {
        cout << "Entry: InsertNodeSorted(value=" << value << ")" << endl;

        ListNode* pNewNode = new ListNode(value);

        // Empty list or insert at head
        if (pHead == NULL || pHead->val >= value) {
            pNewNode->next = pHead;
            cout << "Exit: InsertNodeSorted (at head)" << endl;
            return pNewNode;
        }

        // Find insertion point
        ListNode* pCur = pHead;
        while (pCur->next != NULL && pCur->next->val < value) {
            pCur = pCur->next;
        }

        pNewNode->next = pCur->next;
        pCur->next = pNewNode;

        cout << "Exit: InsertNodeSorted" << endl;
        return pHead;
    }

    void DeleteNode(ListNode* pInHead, ListNode* pToBeDeleted)    {
        cout << "Entry void DeleteNode(ListNode* pInHead, ListNode*
            pToBeDeleted)\n" << endl;

        // Check NULL always
        if (pInHead == NULL || pToBeDeleted == NULL) {
            cout << "Abort void DeleteNode(ListNode* pInHead, ListNode*
                pToBeDeleted)\n" << endl;
            return;
        }

//          PrintListNode(pInHead);

        // Delete non-tail node including head node
        if (pToBeDeleted->next != NULL) {
            ListNode* pNext = pToBeDeleted->next;
            pToBeDeleted->val = pNext->val;
            pToBeDeleted->next = pNext->next;

            delete pNext;
            pNext = NULL;
        } else { // Delete tail
            ListNode* pPre = pInHead;

```

```
            while (pPre->next != pToBeDeleted && pPre != NULL) {
                  pPre = pPre->next;
            }
            if (pPre == NULL)
               return;

            pPre->next = NULL;
            delete pToBeDeleted;
            pToBeDeleted = NULL;
         }

          cout << "Exit void DeleteNode(ListNode* pInHead, ListNode*
             pToBeDeleted)\n" << endl;
    }

    // Delete node by value
    ListNode* DeleteNodeByValue(ListNode* pHead, int value) {
         cout << "Entry: DeleteNodeByValue(value=" << value << ")" << endl;

         if (pHead == NULL) {
            cout << "Error: Empty list" << endl;
            return NULL;
         }

         // Delete head node
         if (pHead->val == value) {
            ListNode* pTemp = pHead;
            pHead = pHead->next;
            delete pTemp;
            cout << "Exit: DeleteNodeByValue (deleted head)" << endl;
            return pHead;
         }

         // Find and delete node
         ListNode* pCur = pHead;
         while (pCur->next != NULL && pCur->next->val != value) {
            pCur = pCur->next;
         }

         if (pCur->next == NULL) {
            cout << "Error: Value not found" << endl;
            return pHead;
         }

         ListNode* pToDelete = pCur->next;
         pCur->next = pCur->next->next;
         delete pToDelete;

         cout << "Exit: DeleteNodeByValue" << endl;
         return pHead;
    }

};

```

```
void TestInsertOperations() {
    cout << "\n" << string(70, '=') << endl;
    cout << "TESTING INSERT OPERATIONS\n";
    cout << string(70, '=') << endl;

    Solution solution;
    ListNode* pHead = NULL;

    // Test 1: Insert at head (empty list)
    cout << "\n--- Test 1: Insert at head (empty list) ---" << endl;
    pHead = solution.InsertNodeAtHead(pHead, 10);
    solution.PrintListNode(pHead);

    // Test 2: Insert at head (non-empty list)
    cout << "\n--- Test 2: Insert at head (non-empty list) ---" << endl;
    pHead = solution.InsertNodeAtHead(pHead, 5);
    solution.PrintListNode(pHead);

    // Test 3: Insert at tail
    cout << "\n--- Test 3: Insert at tail ---" << endl;
    pHead = solution.InsertNodeAtTail(pHead, 20);
    pHead = solution.InsertNodeAtTail(pHead, 25);
    solution.PrintListNode(pHead);

    // Test 4: Insert after specific node
    cout << "\n--- Test 4: Insert after second node ---" << endl;
    solution.InsertNodeAfter(pHead->next, 15);
    solution.PrintListNode(pHead);

    // Test 5: Insert at position
    cout << "\n--- Test 5: Insert at position 2 ---" << endl;
    pHead = solution.InsertNodeAtPosition(pHead, 2, 12);
    solution.PrintListNode(pHead);

    // Clean up
    while (pHead != NULL) {
        ListNode* temp = pHead;
        pHead = pHead->next;
        delete temp;
    }
}

void TestInsertSorted() {
    cout << "\n" << string(70, '=') << endl;
    cout << "TESTING INSERT IN SORTED LIST\n";
    cout << string(70, '=') << endl;

    Solution solution;
    ListNode* pHead = NULL;

    // Insert values in random order
    int values[] = {30, 10, 50, 20, 40};

    for (int val : values) {
```

```
        cout << "\nInserting " << val << " into sorted list:" << endl;
        pHead = solution.InsertNodeSorted(pHead, val);
        solution.PrintListNode(pHead);
    }

    // Clean up
    while (pHead != NULL) {
        ListNode* temp = pHead;
        pHead = pHead->next;
        delete temp;
    }
}

void TestDeleteOperations() {
    cout << "\n" << string(70, '=') << endl;
    cout << "TESTING DELETE OPERATIONS\n";
    cout << string(70, '=') << endl;

    Solution testSolution;
    int count = 5;

    for (int k = 0; k <= count; k ++) {
        ListNode* pHead = NULL;
        ListNode* pCur = NULL;
        ListNode* pDel = NULL;

        cout << "\n--- Creating list and deleting node at position " << k << "
            ---" << endl;

        for (int i = 0; i < count; i ++) {
            ListNode* pTemp = new ListNode(i);

            if (i == 0) {
                pHead = pCur = pTemp;
            }
            else {
                pCur->next = pTemp;
                pCur = pCur->next; // pCur->next == pTemp
            }

            if (i == k)
               pDel = pTemp;
        }

        cout << "Original list: ";
        testSolution.PrintListNode(pHead);

        testSolution.DeleteNode(pHead, pDel);

        cout << "After deletion: ";
        testSolution.PrintListNode(pHead);

        // Clean up remaining nodes
        while (pHead != NULL) {
```

```
            ListNode* temp = pHead;
            pHead = pHead->next;
            delete temp;
        }
    }
}

void TestDeleteByValue() {
    cout << "\n" << string(70, '=') << endl;
    cout << "TESTING DELETE BY VALUE\n";
    cout << string(70, '=') << endl;

    Solution solution;
    ListNode* pHead = NULL;

    // Create list: 10 -> 20 -> 30 -> 40 -> 50
    pHead = solution.InsertNodeAtTail(pHead, 10);
    pHead = solution.InsertNodeAtTail(pHead, 20);
    pHead = solution.InsertNodeAtTail(pHead, 30);
    pHead = solution.InsertNodeAtTail(pHead, 40);
    pHead = solution.InsertNodeAtTail(pHead, 50);

    cout << "\nOriginal list: ";
    solution.PrintListNode(pHead);

    // Delete middle node
    cout << "\nDeleting 30:" << endl;
    pHead = solution.DeleteNodeByValue(pHead, 30);
    solution.PrintListNode(pHead);

    // Delete head node
    cout << "\nDeleting 10 (head):" << endl;
    pHead = solution.DeleteNodeByValue(pHead, 10);
    solution.PrintListNode(pHead);

    // Delete tail node
    cout << "\nDeleting 50 (tail):" << endl;
    pHead = solution.DeleteNodeByValue(pHead, 50);
    solution.PrintListNode(pHead);

    // Clean up
    while (pHead != NULL) {
        ListNode* temp = pHead;
        pHead = pHead->next;
        delete temp;
    }
}

int main () {
    cout << "\n";
    cout << "                                    \n";
    cout << "            LINKED LIST INSERT AND DELETE OPERATIONS
                    \n";
    cout << "
```

```
                                                                    \n";
    cout << "   Demonstrates various insertion and deletion techniques
                 \n";
    cout << "                                \n";

    TestInsertOperations();
    TestInsertSorted();
    TestDeleteOperations();
    TestDeleteByValue();

    cout << "\n" << string(70, '=') << endl;
    cout << "ALL TESTS COMPLETED\n";
    cout << string(70, '=') << endl;
    cout << endl;

    return 0;
}
```

39 Source Code: LambdaCaptures.cpp

File: `src/LambdaCaptures.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <memory>
#include <functional>
#include <vector>
#include <future>
#include <thread>

// ==================================================================
// 1. NO CAPTURE []
// ==================================================================
void example_no_capture() {
    std::cout << "\n=== 1. NO CAPTURE [] ===" << std::endl;

    int global_counter = 100;  // Global-like variable

    // Lambda can only access its parameters and static/global variables
    auto lambda = []() {
        static int static_var = 42;
        std::cout << "Static variable: " << static_var++ << std::endl;
        return static_var;
    };

    std::cout << "Call 1: " << lambda() << std::endl;  // 43
    std::cout << "Call 2: " << lambda() << std::endl;  // 44
}

// ==================================================================
// 2. CAPTURE ALL BY VALUE [=]
// ==================================================================
void example_capture_all_by_value() {
    std::cout << "\n=== 2. CAPTURE ALL BY VALUE [=] ===" << std::endl;

    int x = 10;
    int y = 20;
    int z = 30;  // Not referenced, so not captured

    auto lambda = [=]() {  // Captures x and y by value
        std::cout << "x = " << x << ", y = " << y << std::endl;
        // Cannot modify: x and y are const copies
        // x = 100;  // ERROR!
        return x + y;
    };

    x = 100;  // Change original
    std::cout << "Lambda result: " << lambda() << std::endl;  // Still 30 (10+20)
    std::cout << "Original x after: " << x << std::endl;      // 100
}
```

```

// ==================================================================
// 3. CAPTURE ALL BY REFERENCE [&]
// ==================================================================
void example_capture_all_by_reference() {
    std::cout << "\n=== 3. CAPTURE ALL BY REFERENCE [&] ===" << std::endl;

    int x = 10;
    int y = 20;

    auto lambda = [&]() {  // Captures x and y by reference
        std::cout << "Before modify: x = " << x << ", y = " << y << std::endl;
        x = 100;  // Modifies original
        y = 200;  // Modifies original
        return x + y;
    };

    std::cout << "Before lambda: x=" << x << ", y=" << y << std::endl;  // 10, 20
    std::cout << "Lambda result: " << lambda() << std::endl;            // 300
    std::cout << "After lambda: x=" << x << ", y=" << y << std::endl;   // 100, 200
}

// ==================================================================
// 4. CAPTURE SPECIFIC BY VALUE [var]
// ==================================================================
void example_specific_capture_by_value() {
    std::cout << "\n=== 4. CAPTURE SPECIFIC BY VALUE [var] ===" << std::endl;

    int a = 1, b = 2, c = 3;

    auto lambda = [a, c]() {  // Only captures a and c by value
        std::cout << "a = " << a << ", c = " << c << std::endl;
        // b is not accessible here
        return a + c;
    };

    a = 100;  // Change original
    std::cout << "Lambda result: " << lambda() << std::endl;  // 4 (1+3)
    std::cout << "Original a after: " << a << std::endl;      // 100
}

// ==================================================================
// 5. CAPTURE SPECIFIC BY REFERENCE [&var]
// ==================================================================
void example_specific_capture_by_reference() {
    std::cout << "\n=== 5. CAPTURE SPECIFIC BY REFERENCE [&var] ===" << std::endl;

    int a = 1, b = 2, c = 3;

    auto lambda = [&a, &c]() {  // Only captures a and c by reference
        std::cout << "Before modify: a = " << a << ", c = " << c << std::endl;
```

```
        a = 10;  // Modifies original a
        c = 30;  // Modifies original c
        return a + c;
    };

    std::cout << "Before lambda: a=" << a << ", c=" << c << std::endl;  // 1, 3
    std::cout << "Lambda result: " << lambda() << std::endl;            // 40
    std::cout << "After lambda: a=" << a << ", c=" << c << std::endl;   // 10, 30
}

// ==================================================================
// 6. MIXED CAPTURE [=, &var]
// ==================================================================
void example_mixed_capture_value_default() {
    std::cout << "\n=== 6. MIXED CAPTURE [=, &var] ===" << std::endl;

    int x = 10, y = 20, z = 30;

    auto lambda = [=, &z]() {  // x,y by value, z by reference
        std::cout << "x=" << x << ", y=" << y << ", z=" << z << std::endl;
        // x = 100;  // ERROR: captured by value
        z = 300;     // OK: captured by reference
        return x + y + z;
    };

    std::cout << "Before lambda: z=" << z << std::endl;    // 30
    std::cout << "Lambda result: " << lambda() << std::endl;  // 330
    std::cout << "After lambda: z=" << z << std::endl;     // 300
}

// ==================================================================
// 7. MIXED CAPTURE [&, var]
// ==================================================================
void example_mixed_capture_reference_default() {
    std::cout << "\n=== 7. MIXED CAPTURE [&, var] ===" << std::endl;

    int x = 10, y = 20, z = 30;

    auto lambda = [&, y]() {  // x,z by reference, y by value
        std::cout << "x=" << x << ", y=" << y << ", z=" << z << std::endl;
        x = 100;  // OK: captured by reference
        // y = 200;  // ERROR: captured by value
        z = 300;  // OK: captured by reference
        return x + y + z;
    };

    std::cout << "Before lambda: x=" << x << ", y=" << y << ", z=" << z << std::endl;  // 10,20,30
    std::cout << "Lambda result: " << lambda() << std::endl;  // 420
    std::cout << "After lambda: x=" << x << ", y=" << y << ", z=" << z << std::endl;   // 100,20,300
}
```

```

// ==================================================================
// 8. CAPTURE THIS POINTER [this]
// ==================================================================
class ExampleClass {
private:
    int member_var = 42;
    std::string name = "TestClass";

public:
    void example_capture_this() {
        std::cout << "\n=== 8. CAPTURE THIS POINTER [this] ===" << std::endl;

        int local_var = 100;

        auto lambda = [this, local_var]() {
            std::cout << "Member var: " << member_var << std::endl;
            std::cout << "Name: " << name << std::endl;
            std::cout << "Local var: " << local_var << std::endl;
            member_var = 999;  // Can modify member through this
            return member_var + local_var;
        };

        std::cout << "Before lambda: member_var=" << member_var << std::endl;
            // 42
        std::cout << "Lambda result: " << lambda() << std::endl;
                          // 1099
        std::cout << "After lambda: member_var=" << member_var << std::endl;
             // 999
    }
};

// ==================================================================
// 9. CAPTURE *THIS [*this] (C++17)
// ==================================================================
class ExampleClass2 {
private:
    int value = 42;

public:
    void example_capture_star_this() {
        std::cout << "\n=== 9. CAPTURE *THIS [*this] (C++17) ===" << std::endl
            ;

        std::cout << "Original object value: " << value << std::endl;

        // Capture a copy of the entire object
        auto lambda = [*this]() mutable {
            std::cout << "Copied object value: " << value << std::endl;
            value = 100;  // Modifies the copy
            return value;
        };

        // Simulate async usage
```

```
        auto future = std::async(std::launch::async, lambda);

        std::cout << "Original unchanged during async: " << value << std::endl;  // 42
        std::cout << "Async result (copy modified): " << future.get() << std::endl;  // 100
        std::cout << "Original still: " << value << std::endl;  // 42
    }
};

// ==================================================================
// 10. MUTABLE LAMBDA
// ==================================================================
void example_mutable_lambda() {
    std::cout << "\n=== 10. MUTABLE LAMBDA ===" << std::endl;

    int x = 10;
    int counter = 0;

    // Without mutable - cannot modify captured values
    // auto lambda1 = [x, counter]() {
    //     counter++;  // ERROR
    //     return x + counter;
    // };

    // With mutable - can modify captured copies
    auto lambda2 = [x, counter]() mutable {
        std::cout << "Before increment: x=" << x << ", counter=" << counter << std::endl;
        x += 5;      // Can modify captured copy
        counter++;   // Can modify captured copy
        std::cout << "After increment: x=" << x << ", counter=" << counter << std::endl;
        return x + counter;
    };

    std::cout << "Before lambda: counter=" << counter << std::endl;  // 0
    std::cout << "Lambda result 1: " << lambda2() << std::endl;      // 16 (15+1)
    std::cout << "Lambda result 2: " << lambda2() << std::endl;      // 22 (20+2)
    std::cout << "Original unchanged: counter=" << counter << std::endl;  // 0
}

// ==================================================================
// 11. CAPTURE WITH INITIALIZATION (C++14)
// ==================================================================
void example_capture_with_initialization() {
    std::cout << "\n=== 11. CAPTURE WITH INITIALIZATION (C++14) ===" << std::endl;

    int x = 10;
    std::string str = "Hello";

```

```cpp
    // C++14: Capture with initialization expressions
    auto lambda1 = [value = x * 2, message = str + " World"]() {
        std::cout << "value = " << value << std::endl;      // 20
        std::cout << "message = " << message << std::endl;  // "Hello World"
        return value;
    };

    auto lambda2 = [&ref = x]() {  // Capture reference with custom name
        ref = 100;  // Modifies original x
        return ref;
    };

    std::cout << "Lambda1 result: " << lambda1() << std::endl;  // 20
    std::cout << "Before lambda2: x=" << x << std::endl;        // 10
    std::cout << "Lambda2 result: " << lambda2() << std::endl;  // 100
    std::cout << "After lambda2: x=" << x << std::endl;         // 100

    // Capture unique_ptr
    auto lambda3 = [ptr = std::make_unique<int>(42)]() {
        return *ptr;
    };
    std::cout << "Lambda3 with unique_ptr: " << lambda3() << std::endl;  // 42
}

// ==================================================================
// 12. MULTIPLE SPECIFIC CAPTURES
// ==================================================================
void example_multiple_specific_captures() {
    std::cout << "\n=== 12. MULTIPLE SPECIFIC CAPTURES ===" << std::endl;

    int a = 1, b = 2, c = 3, d = 4, e = 5;

    auto lambda = [a, &b, c, &d]() {  // Mix of value and reference
        std::cout << "Captured: a=" << a << ", b=" << b
                  << ", c=" << c << ", d=" << d << std::endl;
        b = 20;  // OK: reference
        d = 40;  // OK: reference
        // a = 10;  // ERROR: value capture
        return a + b + c + d;
    };

    std::cout << "Before lambda: a=" << a << ", b=" << b
              << ", c=" << c << ", d=" << d << std::endl;  // 1,2,3,4
    std::cout << "Lambda result: " << lambda() << std::endl;           // 66
        (1+20+3+40)
    std::cout << "After lambda: a=" << a << ", b=" << b
              << ", c=" << c << ", d=" << d << std::endl;   // 1,20,3,40
}

// ==================================================================
// 13. PRACTICAL EXAMPLE: EVENT HANDLER
// ==================================================================
void example_event_handler() {
    std::cout << "\n=== 13. PRACTICAL EXAMPLE: EVENT HANDLER ===" << std::endl
```

```
        ;

    class Button {
    private:
        std::vector<std::function<void()>> click_handlers;
        std::string label;

    public:
        Button(const std::string& lbl) : label(lbl) {}

        void add_handler(std::function<void()> handler) {
            click_handlers.push_back(handler);
        }

        void click() {
            std::cout << "Button '" << label << "' clicked!" << std::endl;
            for (auto& handler : click_handlers) {
                handler();
            }
        }
    };

    std::string username = "Alice";
    int click_count = 0;

    Button button("Submit");

    // Capture username by value, click_count by reference
    button.add_handler([username, &click_count]() {
        std::cout << "Hello " << username << "!" << std::endl;
        click_count++;
        std::cout << "Click count: " << click_count << std::endl;
    });

    button.click();  // Hello Alice! Click count: 1
    button.click();  // Hello Alice! Click count: 2

    username = "Bob";  // Change doesn't affect captured value

    button.click();  // Still: Hello Alice! Click count: 3
}

// ==================================================================
// 14. DANGEROUS EXAMPLE: DANGLING REFERENCE
// ==================================================================
void example_dangling_reference() {
    std::cout << "\n=== 14. DANGEROUS: DANGLING REFERENCE ===" << std::endl;

    std::function<void()> dangerous_lambda;

    {
        int local_variable = 42;

        // CAPTURE BY REFERENCE - DANGER!
```

```
        dangerous_lambda = [&local_variable]() {
            std::cout << "Local variable: " << local_variable << std::endl;
                // UNDEFINED!
        };

        // local_variable will be destroyed when scope ends
    }

    std::cout << "WARNING: Calling lambda with dangling reference..." << std::
        endl;
    // dangerous_lambda();  // UNDEFINED BEHAVIOR - might crash

    // SAFE ALTERNATIVE
    std::function<void()> safe_lambda;
    {
        int local_variable = 42;
        safe_lambda = [local_variable]() {  // Capture by value
            std::cout << "Safe: captured value = " << local_variable << std::
                endl;
        };
    }

    safe_lambda();  // Safe! Outputs: 42
}

// ==================================================================
// 15. COMPREHENSIVE EXAMPLE: ALL IN ONE
// ==================================================================
void comprehensive_example() {
    std::cout << "\n=== 15. COMPREHENSIVE EXAMPLE ===" << std::endl;

    int x = 1, y = 2, z = 3;
    static int static_var = 100;

    std::cout << "Creating multiple lambdas with different captures:" << std::
        endl;

    // 1. No capture
    auto lambda1 = []() {
        std::cout << "  Lambda1: No capture, static=" << static_var++ << std::
            endl;
    };

    // 2. Capture all by value
    auto lambda2 = [=]() {
        std::cout << "  Lambda2: All by value, x=" << x << ", y=" << y << std
            ::endl;
    };

    // 3. Capture all by reference
    auto lambda3 = [&]() {
        std::cout << "  Lambda3: All by ref, before x=" << x;
        x = 10;
        std::cout << ", after x=" << x << std::endl;
```

```
    };

    // 4. Specific captures
    auto lambda4 = [x, &y]() {
        std::cout << "  Lambda4: x by value=" << x << ", y by ref=" << y << std::endl;
        y = 20;
    };

    // 5. Mixed capture
    auto lambda5 = [=, &z]() {
        std::cout << "  Lambda5: x,y by value, z by ref, z=" << z << std::endl;
        z = 30;
    };

    // 6. Mutable
    auto lambda6 = [x]() mutable {
        std::cout << "  Lambda6: Mutable, x before=" << x;
        x = 100;  // Modifies copy
        std::cout << ", x after=" << x << std::endl;
    };

    // Execute all
    lambda1();
    lambda2();
    lambda3();
    lambda4();
    lambda5();
    lambda6();

    std::cout << "\nFinal values: x=" << x << ", y=" << y << ", z=" << z << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "==================================================================" << std::endl;
    std::cout << "COMPLETE C++ LAMBDA CAPTURE EXAMPLES IN ONE FILE" << std::endl;
    std::cout << "==================================================================" << std::endl;

    // Run all examples
    example_no_capture();
    example_capture_all_by_value();
    example_capture_all_by_reference();
    example_specific_capture_by_value();
    example_specific_capture_by_reference();
```

```
    example_mixed_capture_value_default();
    example_mixed_capture_reference_default();

    ExampleClass obj1;
    obj1.example_capture_this();

    ExampleClass2 obj2;
    obj2.example_capture_star_this();

    example_mutable_lambda();
    example_capture_with_initialization();
    example_multiple_specific_captures();
    example_event_handler();
    example_dangling_reference();
    comprehensive_example();

    std::cout << "\n
        ================================================================"
        << std::endl;
    std::cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!" << std::endl;
    std::cout << "
        ================================================================"
        << std::endl;

    return 0;
}
```

40 Source Code: MISRACppDemo.cpp

File: `src/MISRACppDemo.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// MISRA C++ CODING GUIDELINES DEMONSTRATION
// ==================================================================
// Demonstrates key MISRA C++ rules for safety-critical systems
// MISRA C++:2008 and MISRA C++:2023 guidelines
//
// MISRA C++ focuses on:
// - Avoiding undefined behavior
// - Avoiding implementation-defined behavior
// - Defensive programming practices
// - Code clarity and maintainability
// - Safety in automotive, aerospace, medical devices
//
// Build: g++ -std=c++20 -Wall -Wextra -Wpedantic -O2 -o MISRACppDemo MISRACppDemo.cpp
// ==================================================================

#include <iostream>
#include <cstdint>
#include <array>
#include <limits>
#include <memory>
#include <string>
#include <algorithm>  // for std::copy_n

// ==================================================================
// RULE CATEGORY 1: TYPES AND DECLARATIONS
// ==================================================================

namespace types_and_declarations {

// MISRA Rule 3-9-1: Use fixed-width integer types from <cstdint>
// Rationale: Ensures portability across platforms

//  BAD: Platform-dependent types
void bad_types() {
    std::cout << "\n BAD: Platform-dependent types\n";

    int x = 100;           // Size undefined (16, 32, or 64 bits?)
    long y = 1000;         // Size undefined (32 or 64 bits?)
    unsigned int z = 500;  // Size undefined

    std::cout << "   int size: " << sizeof(x) << " bytes (undefined!)\n";
    std::cout << "   long size: " << sizeof(y) << " bytes (undefined!)\n";
}

//  GOOD: Fixed-width types
void good_types() {
    std::cout << "\n GOOD: Fixed-width types\n";
```

```

    int32_t x = 100;        // Always 32 bits
    int64_t y = 1000;       // Always 64 bits
    uint32_t z = 500;       // Always unsigned 32 bits

    std::cout << "   int32_t: " << x << " (always 32 bits)\n";
    std::cout << "   int64_t: " << y << " (always 64 bits)\n";
    std::cout << "   uint32_t: " << z << " (always unsigned 32 bits)\n";
}

// MISRA Rule 4-5-1: Expressions with type bool shall not be used with operators other than
// =, ==, !=, !, &&, ||

//   BAD: Arithmetic on bool
void bad_bool_usage() {
    std::cout << "\n BAD: Arithmetic on bool\n";

    bool flag1 = true;
    bool flag2 = false;

    // int result = flag1 + flag2;  // MISRA violation!
    // int value = flag1 * 5;       // MISRA violation!

    std::cout << "   Arithmetic on bool is prohibited by MISRA\n";
}

//   GOOD: Proper bool usage
void good_bool_usage() {
    std::cout << "\n GOOD: Proper bool usage\n";

    bool flag1 = true;
    bool flag2 = false;

    bool result = flag1 && flag2;  // OK: logical operator
    bool equal = (flag1 == flag2); // OK: comparison
    bool negate = !flag1;          // OK: logical NOT

    std::cout << "   flag1 && flag2 = " << result << "\n";
    std::cout << "   flag1 == flag2 = " << equal << "\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "MISRA CATEGORY 1: TYPES AND DECLARATIONS\n";
    std::cout << std::string(70, '=') << "\n";

    bad_types();
    good_types();
    bad_bool_usage();
    good_bool_usage();
}

} // namespace types_and_declarations
```

```

// ==================================================================
// RULE CATEGORY 2: EXPRESSIONS AND OPERATORS
// ==================================================================

namespace expressions_and_operators {

// MISRA Rule 5-0-21: Bitwise operators shall only be applied to operands of unsigned type

//   BAD: Bitwise operations on signed integers
void bad_bitwise() {
    std::cout << "\n BAD: Bitwise operations on signed integers\n";

    int32_t signed_val = -1;
    // int32_t result = signed_val << 2;  // MISRA violation! Undefined behavior
    // int32_t mask = signed_val & 0xFF;  // MISRA violation!

    std::cout << "   Bitwise ops on signed integers can cause undefined behavior\n";
}

//   GOOD: Bitwise operations on unsigned integers
void good_bitwise() {
    std::cout << "\n GOOD: Bitwise operations on unsigned integers\n";

    uint32_t unsigned_val = 0xF0;
    uint32_t result = unsigned_val << 2;  // OK: unsigned
    uint32_t mask = unsigned_val & 0xFF;  // OK: unsigned

    std::cout << "   unsigned_val << 2 = 0x" << std::hex << result << "\n";
    std::cout << "   unsigned_val & 0xFF = 0x" << mask << std::dec << "\n";
}

// MISRA Rule 5-0-5: There shall be no implicit floating-integral conversions

//   BAD: Implicit conversions
void bad_conversions() {
    std::cout << "\n BAD: Implicit floating-integral conversions\n";

    float f = 3.14f;
    // int32_t i = f;        // MISRA violation! Implicit conversion
    // float result = 5 + f; // MISRA violation! Mixed type arithmetic

    std::cout << "   Implicit float-int conversions lose precision\n";
}

//   GOOD: Explicit conversions
void good_conversions() {
    std::cout << "\n GOOD: Explicit conversions\n";

    float f = 3.14f;
    int32_t i = static_cast<int32_t>(f);  // OK: explicit cast
```

```
    float result = static_cast<float>(5) + f;  // OK: explicit cast

    std::cout << "   static_cast<int32_t>(3.14f) = " << i << "\n";
    std::cout << "   Explicit casts make intent clear\n";
}

// MISRA Rule 5-0-6: An implicit integral or floating-point conversion shall not reduce
// the size of the underlying type

//   BAD: Narrowing conversions
void bad_narrowing() {
    std::cout << "\n BAD: Narrowing conversions\n";

    int64_t large = 1000000;
    // int16_t small = large;  // MISRA violation! Data loss possible

    std::cout << "   Narrowing conversions can lose data\n";
}

//   GOOD: Safe conversions or explicit narrowing
void good_narrowing() {
    std::cout << "\n GOOD: Safe conversions with range checks\n";

    int64_t large = 1000;

    // Check before narrowing
    if (large <= std::numeric_limits<int16_t>::max() &&
        large >= std::numeric_limits<int16_t>::min()) {
        int16_t small = static_cast<int16_t>(large);  // OK: checked and explicit
        std::cout << "   Safe narrowing: " << small << "\n";
    } else {
        std::cout << "   Value out of range for int16_t\n";
    }
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "MISRA CATEGORY 2: EXPRESSIONS AND OPERATORS\n";
    std::cout << std::string(70, '=') << "\n";

    bad_bitwise();
    good_bitwise();
    bad_conversions();
    good_conversions();
    bad_narrowing();
    good_narrowing();
}

} // namespace expressions_and_operators

// ==================================================================
// RULE CATEGORY 3: STATEMENTS
```

```
// ==================================================================

namespace statements {

// MISRA Rule 6-4-1: An if (condition) construct shall be followed by a compound statement
// MISRA Rule 6-4-4: A switch statement shall have at least two case clauses

//   BAD: Single-statement if without braces
void bad_if_statements() {
    std::cout << "\n BAD: Single-statement if without braces\n";

    int32_t value = 5;

    // if (value > 0)
    //     std::cout << "Positive\n";  // MISRA violation! No braces

    std::cout << "   Missing braces can lead to errors during maintenance\n";
}

//   GOOD: Always use braces
void good_if_statements() {
    std::cout << "\n GOOD: Always use braces\n";

    int32_t value = 5;

    if (value > 0) {  // OK: compound statement
        std::cout << "   Positive\n";
    } else {
        std::cout << "   Non-positive\n";
    }
}

// MISRA Rule 6-4-5: An unconditional throw or break shall terminate every non-empty
// switch clause

//   BAD: Fall-through in switch
void bad_switch(int32_t value) {
    std::cout << "\n BAD: Fall-through in switch\n";

    // switch (value) {
    //     case 1:
    //         std::cout << "One\n";
    //         // Fall-through! MISRA violation
    //     case 2:
    //         std::cout << "Two\n";
    //         break;
    //     default:
    //         std::cout << "Other\n";
    // }

    std::cout << "   Fall-through is error-prone\n";
}
```

```

//   GOOD: Explicit break in every case
void good_switch(int32_t value) {
    std::cout << "\n GOOD: Explicit break in every case\n";

    switch (value) {
        case 1: {
            std::cout << "   One\n";
            break;
        }
        case 2: {
            std::cout << "   Two\n";
            break;
        }
        default: {
            std::cout << "   Other\n";
            break;
        }
    }
}

// MISRA Rule 6-6-1, 6-6-2: Any label referenced by a goto shall be in the
    same block
// or enclosing block

//   BAD: goto usage (generally prohibited)
void bad_goto() {
    std::cout << "\n BAD: goto usage\n";
    std::cout << "   MISRA strongly discourages or prohibits goto\n";
    std::cout << "   Use structured programming instead\n";
}

//   GOOD: Structured programming
void good_structured() {
    std::cout << "\n GOOD: Structured programming\n";

    bool error_occurred = false;

    // Instead of goto for error handling, use early return
    if (error_occurred) {
        std::cout << "   Error handled with early return\n";
        return;
    }

    std::cout << "   Normal execution path\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "MISRA CATEGORY 3: STATEMENTS\n";
    std::cout << std::string(70, '=') << "\n";

    bad_if_statements();
    good_if_statements();
```

```
    bad_switch(1);
    good_switch(1);
    bad_goto();
    good_structured();
}

} // namespace statements

// ==================================================================
// RULE CATEGORY 4: FUNCTIONS
// ==================================================================

namespace functions {

// MISRA Rule 8-4-2: All exit paths from a function with non-void return type shall have
// an explicit return statement

//   BAD: Missing return in some paths
// int32_t bad_function(int32_t value) {
//     if (value > 0) {
//         return value;
//     }
//     // MISRA violation! Missing return for value <= 0
// }

//   GOOD: All paths have explicit return
int32_t good_function(int32_t value) {
    if (value > 0) {
        return value;
    } else {
        return 0;
    }
}

// MISRA Rule 8-4-4: A function with no parameters shall be declared with parameter (void)

//   BAD: Empty parameter list in C++
// void bad_no_params();  // In C++, this is OK, but MISRA prefers explicit void

//   GOOD: Explicit void for clarity (C-style, but clearer)
void good_no_params(void) {
    std::cout << "   Function with explicit void parameter list\n";
}

// MISRA Rule 7-1-1: A variable that is not modified shall be const qualified

//   BAD: Non-const variable that isn't modified
void bad_const_usage() {
    std::cout << "\n BAD: Non-const variable that isn't modified\n";

    int32_t value = 100;  // MISRA violation! Should be const
```

```
    std::cout << "   Value: " << value << "\n";
}

//   GOOD: const-qualified for immutable data
void good_const_usage() {
    std::cout << "\n GOOD: const-qualified for immutable data\n";

    const int32_t value = 100;  // OK: const qualifier
    std::cout << "   Value: " << value << "\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "MISRA CATEGORY 4: FUNCTIONS\n";
    std::cout << std::string(70, '=') << "\n";

    std::cout << "\n GOOD: All paths return a value\n";
    std::cout << "   good_function(5) = " << good_function(5) << "\n";
    std::cout << "   good_function(-5) = " << good_function(-5) << "\n";

    good_no_params();

    bad_const_usage();
    good_const_usage();
}

} // namespace functions

// ==================================================================
// RULE CATEGORY 5: ARRAYS AND POINTERS
// ==================================================================

namespace arrays_and_pointers {

// MISRA Rule 5-0-15: Array indexing shall be the only form of pointer
    arithmetic

//   BAD: Pointer arithmetic
void bad_pointer_arithmetic() {
    std::cout << "\n BAD: Pointer arithmetic\n";

    int32_t arr[5] = {1, 2, 3, 4, 5};
    int32_t* ptr = arr;

    // ptr++;            // MISRA violation!
    // int32_t val = *(ptr + 2);  // MISRA violation!

    std::cout << "   Pointer arithmetic is error-prone\n";
}

//   GOOD: Array indexing
void good_array_indexing() {
    std::cout << "\n GOOD: Array indexing\n";

```

```
    std::array<int32_t, 5> arr = {1, 2, 3, 4, 5};

    for (size_t i = 0; i < arr.size(); ++i) {
        std::cout << "   arr[" << i << "] = " << arr[i] << "\n";
    }
}

// MISRA Rule 5-0-16: The result of pointer subtraction shall not be used

//  BAD: Pointer subtraction
void bad_pointer_subtraction() {
    std::cout << "\n BAD: Pointer subtraction\n";

    int32_t arr[5] = {1, 2, 3, 4, 5};
    int32_t* p1 = &arr[0];
    int32_t* p2 = &arr[3];

    // ptrdiff_t diff = p2 - p1;  // MISRA violation!

    std::cout << "   Pointer subtraction should be avoided\n";
}

//  GOOD: Use indices instead
void good_index_difference() {
    std::cout << "\n GOOD: Use indices instead\n";

    std::array<int32_t, 5> arr = {1, 2, 3, 4, 5};
    size_t idx1 = 0;
    size_t idx2 = 3;

    size_t diff = idx2 - idx1;
    std::cout << "   Index difference: " << diff << "\n";
}

// MISRA Rule: Use std::array instead of C-style arrays

//  BAD: C-style arrays
void bad_c_arrays() {
    std::cout << "\n BAD: C-style arrays\n";

    int32_t arr[10];  // MISRA prefers std::array
    arr[0] = 1;

    std::cout << "   C-style arrays lack bounds checking\n";
}

//  GOOD: std::array with bounds checking
void good_std_array() {
    std::cout << "\n GOOD: std::array with bounds checking\n";

    std::array<int32_t, 10> arr{};
    arr.at(0) = 1;  // Bounds-checked access

    std::cout << "   arr.at(0) = " << arr.at(0) << "\n";
```

```
    std::cout << "   std::array provides bounds checking with at()\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "MISRA CATEGORY 5: ARRAYS AND POINTERS\n";
    std::cout << std::string(70, '=') << "\n";

    bad_pointer_arithmetic();
    good_array_indexing();
    bad_pointer_subtraction();
    good_index_difference();
    bad_c_arrays();
    good_std_array();
}

} // namespace arrays_and_pointers

// ==================================================================
// RULE CATEGORY 6: CLASSES
// ==================================================================

namespace classes {

// MISRA Rule 12-8-1: A copy constructor shall only initialize its base classes and the
// non-static members of the class of which it is a member

//   GOOD: Proper class design
class Resource {
private:
    int32_t* data_;
    size_t size_;

public:
    // Constructor
    explicit Resource(size_t size)
        : data_(new int32_t[size]), size_(size) {
        std::cout << "   [Resource] Constructor: allocated " << size_ << " ints\n";
    }

    // Destructor
    ~Resource() {
        delete[] data_;
        std::cout << "   [Resource] Destructor: freed memory\n";
    }

    // MISRA Rule: Define all special members or none (Rule of Five)
    Resource(const Resource& other)
        : data_(new int32_t[other.size_]), size_(other.size_) {
        std::copy_n(other.data_, size_, data_);
        std::cout << "   [Resource] Copy constructor\n";
    }
```

```

    Resource& operator=(const Resource& other) {
        if (this != &other) {
            delete[] data_;
            size_ = other.size_;
            data_ = new int32_t[size_];
            std::copy_n(other.data_, size_, data_);
        }
        std::cout << "   [Resource] Copy assignment\n";
        return *this;
    }

    Resource(Resource&& other) noexcept
        : data_(other.data_), size_(other.size_) {
        other.data_ = nullptr;
        other.size_ = 0;
        std::cout << "   [Resource] Move constructor\n";
    }

    Resource& operator=(Resource&& other) noexcept {
        if (this != &other) {
            delete[] data_;
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
        }
        std::cout << "   [Resource] Move assignment\n";
        return *this;
    }

    size_t size() const { return size_; }
};

// MISRA Rule: Virtual destructor for base classes

//   BAD: Base class without virtual destructor
class BadBase {
public:
    ~BadBase() {  // MISRA violation! Should be virtual
        std::cout << "   [BadBase] Destructor\n";
    }
};

//   GOOD: Virtual destructor for polymorphic base
class GoodBase {
public:
    virtual ~GoodBase() {
        std::cout << "   [GoodBase] Virtual destructor\n";
    }

    virtual void process() = 0;
};

```

```
class Derived : public GoodBase {
public:
    void process() override {
        std::cout << "   [Derived] Processing\n";
    }

    ~Derived() override {
        std::cout << "   [Derived] Destructor\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "MISRA CATEGORY 6: CLASSES\n";
    std::cout << std::string(70, '=') << "\n";

    std::cout << "\n GOOD: Rule of Five implementation\n";
    {
        Resource r1(5);
        Resource r2 = r1;  // Copy
        Resource r3 = std::move(r1);  // Move
    }

    std::cout << "\n GOOD: Virtual destructor for polymorphism\n";
    {
        std::unique_ptr<GoodBase> ptr = std::make_unique<Derived>();
        ptr->process();
    }
}

} // namespace classes

// ==================================================================
// RULE CATEGORY 7: EXCEPTION HANDLING
// ==================================================================

namespace exception_handling {

// MISRA Rule 15-1-1: Only throw exceptions for exceptional conditions
// MISRA Rule 15-3-1: Exceptions shall be used only for error handling

//  BAD: Using exceptions for control flow
void bad_exception_usage() {
    std::cout << "\n BAD: Using exceptions for control flow\n";

    // try {
    //     // Don't use exceptions for normal flow!
    //     throw 42;
    // } catch (int value) {
    //     // Control flow via exception (BAD!)
    // }

    std::cout << "   Exceptions should only be used for errors\n";
}
```

```cpp

//   GOOD: Exceptions for error handling only
void good_exception_usage() {
    std::cout << "\n GOOD: Exceptions for error handling\n";

    try {
        // Simulate error condition
        throw std::runtime_error("Genuine error occurred");
    } catch (const std::exception& e) {
        std::cout << "   Caught exception: " << e.what() << "\n";
    }
}

// MISRA Note: Many safety-critical systems disable exceptions entirely
void no_exceptions_approach() {
    std::cout << "\n ALTERNATIVE: Error codes (no exceptions)\n";
    std::cout << "   Many MISRA-compliant projects compile with -fno-exceptions\n";
    std::cout << "   Use error codes, std::optional, or std::expected instead\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "MISRA CATEGORY 7: EXCEPTION HANDLING\n";
    std::cout << std::string(70, '=') << "\n";

    bad_exception_usage();
    good_exception_usage();
    no_exceptions_approach();
}

} // namespace exception_handling

// ==================================================================
// MAIN - Demonstrate All MISRA C++ Categories
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "                    MISRA C++ CODING GUIDELINES                    \n";
    std::cout << "            Safety-Critical Software Development            \n";
    std::cout << "                                    \n";

    try {
        types_and_declarations::demonstrate();
        expressions_and_operators::demonstrate();
        statements::demonstrate();
        functions::demonstrate();
        arrays_and_pointers::demonstrate();
        classes::demonstrate();
```

```
        exception_handling::demonstrate();

        // Summary
        std::cout << "\n" << std::string(70, '=') << "\n";
        std::cout << "SUMMARY: KEY MISRA C++ PRINCIPLES\n";
        std::cout << std::string(70, '=') << "\n\n";

        std::cout << "  MISRA C++ OBJECTIVES:\n";
        std::cout << " •   Avoid undefined behavior\n";
        std::cout << " •   Avoid implementation-defined behavior\n";
        std::cout << " •   Maximize portability\n";
        std::cout << " •   Enhance code clarity and maintainability\n";
        std::cout << " •   Enable thorough code review and testing\n\n";

        std::cout << "  KEY PRACTICES:\n";
        std::cout << "   1. Use fixed-width types (int32_t, uint32_t)\n";
        std::cout << "   2. Always use braces for if/while/for\n";
        std::cout << "   3. No pointer arithmetic (use array indexing)\n";
        std::cout << "   4. Explicit conversions (avoid implicit casts)\n";
        std::cout << "   5. const-qualify immutable data\n";
        std::cout << "   6. Virtual destructors for polymorphic classes\n";
        std::cout << "   7. Rule of Five for resource-managing classes\n";
        std::cout << "   8. Prefer std::array over C-style arrays\n\n";

        std::cout << "  APPLICATION DOMAINS:\n";
        std::cout << " •   Automotive (ISO 26262)\n";
        std::cout << " •   Aerospace (DO-178C)\n";
        std::cout << " •   Medical devices (IEC 62304)\n";
        std::cout << " •   Industrial control systems\n";
        std::cout << " •   Railway (EN 50128)\n\n";

        std::cout << "   NOTE:\n";
        std::cout << "   Many MISRA-compliant projects also:\n";
        std::cout << " •   Disable exceptions (-fno-exceptions)\n";
        std::cout << " •   Disable RTTI (-fno-rtti)\n";
        std::cout << " •   Limit or prohibit dynamic memory allocation\n";
        std::cout << " •   Use static analysis tools (PC-lint, Coverity, etc.)
            \n\n";

        std::cout << "                                    \n";
        std::cout << "             ALL MISRA C++ CATEGORIES DEMONSTRATED!
                        \n";
        std::cout << "                                    \n\n";

    } catch (const std::exception& e) {
        std::cerr << "  Error: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

41 Source Code: MockInterview.cpp

File: `src/MockInterview.cpp`
Repository: View on GitHub

```cpp
// ================================================================
// COMPLETE C++ INTERVIEW QUESTIONS & ANSWERS
// ================================================================
// Comprehensive collection of C++ interview questions with detailed answers
// covering fundamentals, memory management, OOP, templates, concurrency,
// and optimization techniques.
//
// Topics covered:
// - C++ Fundamentals (pointers, references, const)
// - Memory Management (smart pointers, allocators, alignment)
// - OOP & Design Patterns (virtual dispatch, CRTP, PIMPL)
// - Templates & Metaprogramming (SFINAE, variadic, compile-time)
// - Concurrency (lock-free, thread pools, atomics)
// - Performance & Optimization (cache-friendly, branch prediction)
//
// Build: g++ -std=c++20 -pthread -O2 -o MockInterview MockInterview.cpp
// Run: ./MockInterview
// ================================================================

#include <iostream>
#include <memory>
#include <thread>
#include <mutex>
#include <atomic>
#include <vector>
#include <list>
#include <unordered_map>
#include <optional>
#include <functional>
#include <queue>
#include <condition_variable>
#include <shared_mutex>
#include <future>
#include <array>
#include <type_traits>
#include <variant>
#include <any>
#include <tuple>
#include <string>
#include <algorithm>
#include <numeric>
#include <random>
#include <chrono>
#include <cstdio>
#include <stdexcept>

using namespace std;

// ================================================================
```

```
// SECTION 1: C++ FUNDAMENTALS
// ==================================================================

// ============ Q1.1: Pointers vs References ============
void pointers_vs_references() {
    cout << "\n=== POINTERS VS REFERENCES ===\n";

    int x = 10;
    int y = 20;

    // Reference - alias, must be initialized, cannot be null
    int& ref = x;
    ref = 30;  // Changes x

    // Pointer - stores address, can be null, can be reassigned
    int* ptr = &x;
    *ptr = 40;  // Changes x
    ptr = &y;   // Now points to y

    cout << "x = " << x << ", y = " << y << endl;
    cout << "ref = " << ref << ", *ptr = " << *ptr << endl;

    // Key differences:
    cout << "\nKey Differences:" << endl;
    cout << "1. References must be initialized, pointers can be null" << endl;
    cout << "2. References cannot be reassigned, pointers can" << endl;
    cout << "3. sizeof(reference) = sizeof(object), sizeof(pointer) = platform dependent" << endl;
    cout << "4. References have cleaner syntax, pointers more flexible" << endl;
}

// ============ Q1.2: Const Correctness ============
void const_correctness() {
    cout << "\n=== CONST CORRECTNESS ===\n";

    class Data {
        mutable int counter;  // Can be modified even in const methods
        int value;

    public:
        Data(int v) : counter(0), value(v) {}

        // 1. Non-const getter - can only be called on non-const objects
        int getValue() {
            cout << "Non-const getValue" << endl;
            return value;
        }

        // 2. Const getter - can be called on both const and non-const objects
        int getValue() const {
            cout << "Const getValue" << endl;
            counter++;  // OK because counter is mutable
            return value;
```

```
        }

        // 3. Const reference return - prevents modification
        const int& getValueRef() const {
            cout << "Const reference getter" << endl;
            return value;
        }
    };

    Data d1(42);
    const Data d2(100);

    d1.getValue();     // Calls non-const version
    d2.getValue();     // Calls const version
    // d2.getValueRef() = 50;  // ERROR: Cannot modify through const reference

    cout << "\nConst Rules:" << endl;
    cout << "1. const methods can be called on const objects" << endl;
    cout << "2. const methods cannot modify non-mutable members" << endl;
    cout << "3. Return const references to prevent modification" << endl;
    cout << "4. Use mutable for members that don't affect logical state" <<
        endl;
}

// ============ Q1.3: RAII Pattern ============
class FileHandler {
    FILE* file;

public:
    FileHandler(const char* filename, const char* mode) {
        file = fopen(filename, mode);
        if (!file) {
            throw runtime_error("Failed to open file");
        }
        cout << "File opened: " << filename << endl;
    }

    ~FileHandler() {
        if (file) {
            fclose(file);
            cout << "File closed" << endl;
        }
    }

    // Delete copy operations
    FileHandler(const FileHandler&) = delete;
    FileHandler& operator=(const FileHandler&) = delete;

    // Allow move operations
    FileHandler(FileHandler&& other) noexcept : file(other.file) {
        other.file = nullptr;
    }

    FileHandler& operator=(FileHandler&& other) noexcept {
```

```
        if (this != &other) {
            if (file) fclose(file);
            file = other.file;
            other.file = nullptr;
        }
        return *this;
    }

    void write(const string& data) {
        if (file) {
            fputs(data.c_str(), file);
        }
    }
};

void raii_demo() {
    cout << "\n=== RAII (Resource Acquisition Is Initialization) ===\n";

    try {
        FileHandler file("test_interview.txt", "w");
        file.write("Hello RAII!\n");
        // File automatically closed when scope ends
    } catch (const exception& e) {
        cout << "Exception: " << e.what() << endl;
    }

    cout << "\nRAII Principles:" << endl;
    cout << "1. Acquire resource in constructor" << endl;
    cout << "2. Release resource in destructor" << endl;
    cout << "3. Use stack unwinding for exception safety" << endl;
    cout << "4. Smart pointers, locks, containers use RAII" << endl;
}

// ==================================================================
// SECTION 2: MEMORY MANAGEMENT
// ==================================================================

// ============ Q2.1: Smart Pointers Deep Dive ============
void smart_pointers_advanced() {
    cout << "\n=== SMART POINTERS ADVANCED ===\n";

    // 1. Custom deleters
    {
        cout << "\n1. Custom Deleters:" << endl;
        auto deleter = [](int* p) {
            cout << "Custom delete: " << *p << endl;
            delete p;
        };

        unique_ptr<int, decltype(deleter)> ptr1(new int(42), deleter);
        shared_ptr<int> ptr2(new int(100), [](int* p) {
            cout << "Lambda deleter: " << *p << endl;
            delete p;
        });
```

```
    }

    // 2. Aliasing constructor
    {
        cout << "\n2. Aliasing Constructor:" << endl;
        struct Data {
            int x = 10;
            int y = 20;
        };

        auto data_ptr = make_shared<Data>();
        auto x_ptr = shared_ptr<int>(data_ptr, &data_ptr->x);
        auto y_ptr = shared_ptr<int>(data_ptr, &data_ptr->y);

        cout << "Use count: " << data_ptr.use_count() << endl;
        cout << "x_ptr points to x but shares ownership of Data" << endl;
    }

    // 3. enable_shared_from_this
    {
        cout << "\n3. enable_shared_from_this:" << endl;
        class Widget : public enable_shared_from_this<Widget> {
        public:
            shared_ptr<Widget> get_shared() {
                return shared_from_this();  // Safe even if multiple
                    shared_ptrs exist
            }
        };

        auto widget = make_shared<Widget>();
        auto another_ref = widget->get_shared();
        cout << "Use count: " << widget.use_count() << endl;
    }
}

// ============ Q2.2: Memory Alignment ============
void memory_alignment() {
    cout << "\n=== MEMORY ALIGNMENT ===\n";

    struct BadLayout {
        char c;      // 1 byte
        int i;       // 4 bytes (may need 3 bytes padding after c)
        double d;    // 8 bytes
        char c2;     // 1 byte (7 bytes padding at end)
        // Total: 1 + 3(pad) + 4 + 8 + 1 + 7(pad) = 24 bytes
    };

    struct GoodLayout {
        double d;    // 8 bytes
        int i;       // 4 bytes
        char c;      // 1 byte
        char c2;     // 1 byte (2 bytes padding at end)
        // Total: 8 + 4 + 1 + 1 + 2(pad) = 16 bytes
    };
```

```

    cout << "sizeof(BadLayout): " << sizeof(BadLayout) << " bytes" << endl;
    cout << "sizeof(GoodLayout): " << sizeof(GoodLayout) << " bytes" << endl;
    cout << "Savings: " << (sizeof(BadLayout) - sizeof(GoodLayout)) << " bytes
        " << endl;

    cout << "\nAlignment Rules:" << endl;
    cout << "1. Data type must be aligned to its size boundary" << endl;
    cout << "2. struct alignment = largest member alignment" << endl;
    cout << "3. Reorder members from largest to smallest" << endl;
    cout << "4. Use alignas() for custom alignment" << endl;
    cout << "5. Cache line size typically 64 bytes" << endl;
}

// ==================================================================
// SECTION 3: OOP & DESIGN PATTERNS
// ==================================================================

// ============ Q3.1: Virtual Dispatch Mechanics ============
void virtual_dispatch_details() {
    cout << "\n=== VIRTUAL DISPATCH MECHANICS ===\n";

    class Base {
    public:
        virtual void func1() { cout << "Base::func1" << endl; }
        virtual void func2() { cout << "Base::func2" << endl; }
        virtual ~Base() {}
    };

    class Derived : public Base {
    public:
        void func1() override { cout << "Derived::func1" << endl; }
        void func3() { cout << "Derived::func3" << endl; }  // Not in vtable
    };

    // How virtual call works:
    Base* b = new Derived();
    b->func1();

    cout << "\nVirtual call process:" << endl;
    cout << "1. Object contains vptr (pointer to vtable)" << endl;
    cout << "2. vtable contains function pointers" << endl;
    cout << "3. b->func1() becomes: (*b->vptr[0])()" << endl;
    cout << "4. vptr[0] points to Derived::func1" << endl;

    // Multiple inheritance
    class Base2 {
    public:
        virtual void func4() { cout << "Base2::func4" << endl; }
        virtual ~Base2() {}
    };

    class MultiDerived : public Base, public Base2 {
    public:
```

```
        void func1() override { cout << "MultiDerived::func1" << endl; }
        void func4() override { cout << "MultiDerived::func4" << endl; }
    };

    MultiDerived md;
    Base* b1 = &md;
    Base2* b2 = &md;

    cout << "\nMultiple inheritance layout:" << endl;
    cout << "MultiDerived object has TWO vptrs" << endl;
    cout << "b1 and b2 point to different subobjects" << endl;
    cout << "Address difference: " << (long long)b2 - (long long)b1 << " bytes
        " << endl;

    delete b;
}

// ============ Q3.2: CRTP with Mixins ============
template<typename Derived>
class Printable {
public:
    void print() const {
        cout << static_cast<const Derived*>(this)->to_string() << endl;
    }
};

template<typename Derived>
class Comparable {
public:
    bool operator==(const Comparable& other) const {
        return static_cast<const Derived*>(this)->get_id() ==
               static_cast<const Derived*>(&other)->get_id();
    }
};

class Widget : public Printable<Widget>, public Comparable<Widget> {
    int id;
    string name;

public:
    Widget(int i, string n) : id(i), name(move(n)) {}

    string to_string() const {
        return "Widget[" + std::to_string(id) + ", " + name + "]";
    }

    int get_id() const { return id; }
};

void crtp_mixins_demo() {
    cout << "\n=== CRTP WITH MIXINS ===\n";

    Widget w1(1, "First");
    Widget w2(2, "Second");
```

```
    Widget w3(1, "Another");

    w1.print();
    w2.print();

    cout << "w1 == w2: " << (w1 == w2) << endl;
    cout << "w1 == w3: " << (w1 == w3) << endl;

    cout << "\nCRTP Mixin Benefits:" << endl;
    cout << "1. Compile-time polymorphism" << endl;
    cout << "2. No vtable overhead" << endl;
    cout << "3. Can combine multiple behaviors" << endl;
    cout << "4. Type-safe at compile time" << endl;
}

// ==================================================================
// SECTION 4: TEMPLATES & METAPROGRAMMING
// ==================================================================

// ============ Q4.1: SFINAE Techniques ============
template<typename T>
auto print_value_impl(const T& value, int)
    -> decltype(cout << value, void()) {
    cout << "Printable: " << value << endl;
}

template<typename T>
void print_value_impl(const T&, long) {
    cout << "Not printable" << endl;
}

template<typename T>
void print_value(const T& value) {
    print_value_impl(value, 0);
}

// SFINAE with enable_if
template<typename T>
typename enable_if<is_integral<T>::value, void>::type
process(T value) {
    cout << "Processing integral: " << value << endl;
}

template<typename T>
typename enable_if<is_floating_point<T>::value, void>::type
process(T value) {
    cout << "Processing float: " << value << endl;
}

void sfinae_demo() {
    cout << "\n=== SFINAE TECHNIQUES ===\n";

    print_value(42);
    print_value(vector<int>{1, 2, 3});
```

```

    process(10);     // Integral version
    process(3.14);   // Float version
}

// ============ Q4.2: Variadic Templates ============
template<typename... Args>
void print_all(Args&&... args) {
    (cout << ... << args) << endl;  // C++17 fold expression
}

void variadic_demo() {
    cout << "\n=== VARIADIC TEMPLATES ===\n";

    print_all("Hello", " ", "World", " ", 2024, "!");

    cout << "\nVariadic Benefits:" << endl;
    cout << "1. Type-safe variadic functions" << endl;
    cout << "2. Compile-time expansion" << endl;
    cout << "3. Perfect forwarding support" << endl;
}

// ============ Q4.3: Compile-Time Computation ============
constexpr int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

template<int N>
struct Factorial {
    static constexpr int value = N * Factorial<N-1>::value;
};

template<>
struct Factorial<0> {
    static constexpr int value = 1;
};

void compile_time_demo() {
    cout << "\n=== COMPILE-TIME COMPUTATION ===\n";

    constexpr int fact = factorial(5);
    cout << "Factorial(5) at compile time: " << fact << endl;

    cout << "Template factorial(5): " << Factorial<5>::value << endl;

    // Static asserts
    static_assert(factorial(5) == 120, "Compile-time factorial failed");
    static_assert(Factorial<5>::value == 120, "Template factorial failed");

    cout << "\nCompile-Time Benefits:" << endl;
    cout << "1. Zero runtime overhead" << endl;
    cout << "2. Computed during compilation" << endl;
    cout << "3. Can be used in constant expressions" << endl;
```

```
}

// ==================================================================
// SECTION 5: CONCURRENCY & MULTITHREADING
// ==================================================================

// ============ Q5.1: Thread-Safe Singleton ============
class ThreadSafeSingleton {
    static mutex instance_mutex;
    static unique_ptr<ThreadSafeSingleton> instance;

    ThreadSafeSingleton() {
        cout << "Singleton created" << endl;
    }

public:
    static ThreadSafeSingleton& getInstance() {
        // C++11 guarantees thread-safe static initialization
        static ThreadSafeSingleton instance;
        return instance;
    }

    // Better: use call_once
    static ThreadSafeSingleton& getInstanceCallOnce() {
        static once_flag flag;
        call_once(flag, []() {
            cout << "Initialized with call_once" << endl;
        });
        static ThreadSafeSingleton instance;
        return instance;
    }

    void doSomething() {
        cout << "Singleton method called" << endl;
    }

    // Delete copy/move
    ThreadSafeSingleton(const ThreadSafeSingleton&) = delete;
    ThreadSafeSingleton& operator=(const ThreadSafeSingleton&) = delete;
};

void singleton_demo() {
    cout << "\n=== THREAD-SAFE SINGLETON ===\n";

    vector<thread> threads;
    for (int i = 0; i < 5; ++i) {
        threads.emplace_back([]() {
            ThreadSafeSingleton::getInstance().doSomething();
        });
    }

    for (auto& t : threads) {
        t.join();
    }
```

```

    cout << "\nSingleton Patterns:" << endl;
    cout << "1. Meyer's Singleton (C++11 static init)" << endl;
    cout << "2. std::call_once for initialization" << endl;
    cout << "3. Double-checked locking (avoid!)" << endl;
    cout << "4. Thread-local storage (per-thread instances)" << endl;
}

// ============ Q5.2: Producer-Consumer with Condition Variables ============
class ProducerConsumer {
    queue<int> buffer;
    mutex mtx;
    condition_variable cv_producer, cv_consumer;
    const size_t max_size = 10;
    bool done = false;

public:
    void produce(int id, int count) {
        for (int i = 0; i < count; ++i) {
            unique_lock lock(mtx);
            cv_producer.wait(lock, [this] {
                return buffer.size() < max_size || done;
            });

            if (done) break;

            buffer.push(id * 100 + i);
            cout << "Produced: " << (id * 100 + i) << " (buffer size: " <<
                buffer.size() << ")" << endl;

            cv_consumer.notify_one();
            this_thread::sleep_for(chrono::milliseconds(50));
        }
    }

    void consume(int id, int count) {
        for (int i = 0; i < count; ++i) {
            unique_lock lock(mtx);
            cv_consumer.wait(lock, [this] {
                return !buffer.empty() || done;
            });

            if (done && buffer.empty()) break;

            int value = buffer.front();
            buffer.pop();
            cout << "Consumed: " << value << " by consumer " << id << " (
                buffer size: " << buffer.size() << ")" << endl;

            cv_producer.notify_one();
            this_thread::sleep_for(chrono::milliseconds(100));
        }
    }

```

```
    void stop() {
        {
            lock_guard lock(mtx);
            done = true;
        }
        cv_producer.notify_all();
        cv_consumer.notify_all();
    }
};

void producer_consumer_demo() {
    cout << "\n=== PRODUCER-CONSUMER PATTERN ===\n";

    ProducerConsumer pc;

    thread producer1([&pc]() { pc.produce(1, 5); });
    thread producer2([&pc]() { pc.produce(2, 5); });
    thread consumer1([&pc]() { pc.consume(1, 5); });
    thread consumer2([&pc]() { pc.consume(2, 5); });

    producer1.join();
    producer2.join();
    consumer1.join();
    consumer2.join();

    pc.stop();

    cout << "\nCondition Variable Best Practices:" << endl;
    cout << "1. Always use with a mutex" << endl;
    cout << "2. Use wait() with predicate to avoid spurious wakeups" << endl;
    cout << "3. notify_one() vs notify_all()" << endl;
    cout << "4. Consider separate CVs for different conditions" << endl;
}

// ============ Q5.3: Atomic Operations ============
void atomic_operations_demo() {
    cout << "\n=== ATOMIC OPERATIONS ===\n";

    atomic<int> counter{0};
    atomic<bool> ready{false};

    vector<thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back([&counter, &ready]() {
            while (!ready.load(memory_order_acquire)) {
                this_thread::yield();
            }

            for (int j = 0; j < 1000; ++j) {
                counter.fetch_add(1, memory_order_relaxed);
            }
        });
    }

```

```
    ready.store(true, memory_order_release);

    for (auto& t : threads) {
        t.join();
    }

    cout << "Final counter value: " << counter.load() << " (expected: 10000)"
        << endl;

    cout << "\nAtomic Operation Types:" << endl;
    cout << "1. load/store - read/write atomic value" << endl;
    cout << "2. fetch_add/fetch_sub - atomic arithmetic" << endl;
    cout << "3. compare_exchange_weak/strong - CAS operations" << endl;
    cout << "4. Memory ordering: relaxed, acquire, release, seq_cst" << endl;
}

// ==================================================================
// SECTION 6: PERFORMANCE & OPTIMIZATION
// ==================================================================

// ============ Q6.1: Move Semantics Performance ============
class LargeObject {
    vector<int> data;

public:
    LargeObject(size_t size) : data(size, 42) {
        cout << "Constructor: " << size << " elements" << endl;
    }

    // Copy constructor (expensive)
    LargeObject(const LargeObject& other) : data(other.data) {
        cout << "Copy constructor: " << data.size() << " elements" << endl;
    }

    // Move constructor (cheap)
    LargeObject(LargeObject&& other) noexcept : data(move(other.data)) {
        cout << "Move constructor" << endl;
    }

    // Copy assignment
    LargeObject& operator=(const LargeObject& other) {
        if (this != &other) {
            data = other.data;
            cout << "Copy assignment: " << data.size() << " elements" << endl;
        }
        return *this;
    }

    // Move assignment
    LargeObject& operator=(LargeObject&& other) noexcept {
        if (this != &other) {
            data = move(other.data);
            cout << "Move assignment" << endl;
        }
```

```
        return *this;
    }
};

void move_semantics_performance() {
    cout << "\n=== MOVE SEMANTICS PERFORMANCE ===\n";

    {
        cout << "\nCopy version:" << endl;
        LargeObject obj1(1000000);
        LargeObject obj2 = obj1;  // Copy
    }

    {
        cout << "\nMove version:" << endl;
        LargeObject obj1(1000000);
        LargeObject obj2 = move(obj1);  // Move
    }

    cout << "\nMove Semantics Guidelines:" << endl;
    cout << "1. Return by value (compiler uses RVO/NRVO)" << endl;
    cout << "2. Use std::move() for rvalues" << endl;
    cout << "3. Mark move constructors noexcept" << endl;
    cout << "4. Don't move from const objects" << endl;
    cout << "5. Perfect forwarding with std::forward" << endl;
}

// ============ Q6.2: RVO and Copy Elision ============
class Expensive {
public:
    Expensive() {
        cout << "Default constructor" << endl;
    }

    Expensive(const Expensive&) {
        cout << "Copy constructor" << endl;
    }

    Expensive(Expensive&&) noexcept {
        cout << "Move constructor" << endl;
    }
};

Expensive createExpensive() {
    return Expensive{};  // RVO applies
}

Expensive createExpensiveNamed() {
    Expensive obj;
    return obj;  // NRVO applies
}

void copy_elision_demo() {
    cout << "\n=== COPY ELISION (RVO/NRVO) ===\n";
```

```

    cout << "Creating with RVO:" << endl;
    Expensive e1 = createExpensive();

    cout << "\nCreating with NRVO:" << endl;
    Expensive e2 = createExpensiveNamed();

    cout << "\nCopy Elision Rules:" << endl;
    cout << "1. RVO: Return Value Optimization" << endl;
    cout << "2. NRVO: Named Return Value Optimization" << endl;
    cout << "3. Guaranteed in C++17 for prvalue" << endl;
    cout << "4. Compiler may apply in other cases" << endl;
    cout << "5. Don't use std::move() on return values" << endl;
}

// ==================================================================
// MAIN - Demonstrate All Interview Topics
// ==================================================================

int main() {
    cout << "\n";
    cout << "                                    \n";
    cout << "          COMPLETE C++ INTERVIEW QUESTIONS & ANSWERS
                      \n";
    cout << "               Mock Interview Preparation Guide
                         \n";
    cout << "                                    \n";

    try {
        // Section 1: C++ Fundamentals
        cout << "\n" << string(70, '=') << "\n";
        cout << "SECTION 1: C++ FUNDAMENTALS\n";
        cout << string(70, '=') << "\n";

        pointers_vs_references();
        const_correctness();
        raii_demo();

        // Section 2: Memory Management
        cout << "\n" << string(70, '=') << "\n";
        cout << "SECTION 2: MEMORY MANAGEMENT\n";
        cout << string(70, '=') << "\n";

        smart_pointers_advanced();
        memory_alignment();

        // Section 3: OOP & Design Patterns
        cout << "\n" << string(70, '=') << "\n";
        cout << "SECTION 3: OOP & DESIGN PATTERNS\n";
        cout << string(70, '=') << "\n";

        virtual_dispatch_details();
        crtp_mixins_demo();

```

```
        // Section 4: Templates & Metaprogramming
        cout << "\n" << string(70, '=') << "\n";
        cout << "SECTION 4: TEMPLATES & METAPROGRAMMING\n";
        cout << string(70, '=') << "\n";

        sfinae_demo();
        variadic_demo();
        compile_time_demo();

        // Section 5: Concurrency & Multithreading
        cout << "\n" << string(70, '=') << "\n";
        cout << "SECTION 5: CONCURRENCY & MULTITHREADING\n";
        cout << string(70, '=') << "\n";

        singleton_demo();
        producer_consumer_demo();
        atomic_operations_demo();

        // Section 6: Performance & Optimization
        cout << "\n" << string(70, '=') << "\n";
        cout << "SECTION 6: PERFORMANCE & OPTIMIZATION\n";
        cout << string(70, '=') << "\n";

        move_semantics_performance();
        copy_elision_demo();

        // Summary
        cout << "\n" << string(70, '=') << "\n";
        cout << "SUMMARY: KEY INTERVIEW TOPICS COVERED\n";
        cout << string(70, '=') << "\n\n";

        cout << "  FUNDAMENTALS:\n";
        cout << " •   Pointers vs References\n";
        cout << " •   Const correctness\n";
        cout << " •   RAII pattern\n\n";

        cout << "  MEMORY MANAGEMENT:\n";
        cout << " •   Smart pointers (unique_ptr, shared_ptr, weak_ptr)\n";
        cout << " •   Memory alignment and padding\n";
        cout << " •   Custom allocators\n\n";

        cout << "  OOP & DESIGN:\n";
        cout << " •   Virtual dispatch mechanics\n";
        cout << " •   CRTP with mixins\n";
        cout << " •   PIMPL idiom\n\n";

        cout << "  TEMPLATES:\n";
        cout << " •   SFINAE techniques\n";
        cout << " •   Variadic templates\n";
        cout << " •   Compile-time computation\n\n";

        cout << "  CONCURRENCY:\n";
        cout << " •   Thread-safe singleton\n";
        cout << " •   Producer-consumer pattern\n";
```

```
        cout << " •   Atomic operations\n\n";

        cout << "  PERFORMANCE:\n";
        cout << " •   Move semantics\n";
        cout << " •   Copy elision (RVO/NRVO)\n";
        cout << " •   Cache-friendly design\n\n";

        cout << "  COMMON INTERVIEW QUESTIONS ANSWERED:\n";
        cout << "   1. What's the difference between pointers and references?\n";
        cout << "   2. Explain smart pointers and when to use each\n";
        cout << "   3. How does virtual dispatch work internally?\n";
        cout << "   4. What is SFINAE and how is it used?\n";
        cout << "   5. How do you implement thread-safe singleton?\n";
        cout << "   6. What's the difference between RVO and move semantics?\n";
        cout << "   7. Explain memory ordering in atomic operations\n";
        cout << "   8. How does CRTP provide compile-time polymorphism?\n\n";

        cout << "  ADDITIONAL TOPICS TO STUDY:\n";
        cout << " •   Lambda expressions and closures\n";
        cout << " •   Concepts (C++20)\n";
        cout << " •   Coroutines (C++20)\n";
        cout << " •   Ranges library (C++20)\n";
        cout << " •   std::expected (C++23)\n";
        cout << " •   Design patterns (Factory, Observer, Strategy)\n\n";

        cout << "                                    \n";
        cout << "          ALL INTERVIEW TOPICS SUCCESSFULLY DEMONSTRATED!\n";
        cout << "                                    \n\n";

    } catch (const exception& e) {
        cerr << "  Error: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

42 Source Code: MoveSemantics.cpp

File: `src/MoveSemantics.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// std::move AND MOVE SEMANTICS - COMPREHENSIVE GUIDE
// ==================================================================
// Topics covered:
// 1. Lvalues vs Rvalues - What they are and why they matter
// 2. std::move - What it actually does (cast to rvalue reference)
// 3. Move constructor vs Copy constructor - Performance comparison
// 4. Why move semantics are important - Real-world scenarios
// 5. Common pitfalls and best practices
// 6. Perfect forwarding with std::forward
// ==================================================================

#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <chrono>
#include <utility>  // for std::move, std::forward

// ==================================================================
// 1. LVALUES vs RVALUES - Understanding the Basics
// ==================================================================

void demonstrate_lvalue_rvalue() {
    std::cout << "\n=== 1. LVALUES vs RVALUES ===" << std::endl;

    std::cout << "\n Theory:" << std::endl;
    std::cout << "   LVALUE = 'Locator value' - has a name, addressable, persists" << std::endl;
    std::cout << "   RVALUE = 'Right-hand value' - temporary, about to expire" << std::endl;

    // LVALUES - have names, can take address
    int x = 42;           // x is an lvalue
    int* ptr = &x;        // Can take address of x
    std::string name = "John";  // name is an lvalue

    std::cout << "\n LVALUES (have names, addressable):" << std::endl;
    std::cout << "   int x = 42;           // x is lvalue, address: " << &x << std::endl;
    std::cout << "   string name = \"John\"; // name is lvalue, address: " << &name << std::endl;

    // RVALUES - temporaries, no name, about to be destroyed
    std::cout << "\n RVALUES (temporaries, no name):" << std::endl;
    std::cout << "   42              // literal rvalue" << std::endl;
    std::cout << "   x + 5           // expression result is rvalue" << std::endl;
    std::cout << "   string(\"hi\")   // temporary object is rvalue" << std::
```

```
        endl;

    // x = 42;         // OK: x is lvalue, 42 is rvalue
    // 42 = x;         // ERROR: 42 is rvalue, can't assign to rvalue
    // &42;            // ERROR: can't take address of rvalue

    std::cout << "\n Key insight:" << std::endl;
    std::cout << " •   Lvalues appear on LEFT of assignment: x = ..." << std::endl;
    std::cout << " •   Rvalues appear on RIGHT of assignment: ... = 42" << std::endl;
    std::cout << " •   Rvalues are about to die → safe to 'steal' their resources!" << std::endl;
}

// ==================================================================
// 2. std::move - What It Actually Does
// ==================================================================

void demonstrate_std_move() {
    std::cout << "\n=== 2. WHAT IS std::move? ===" << std::endl;

    std::cout << "\n std::move is NOT:" << std::endl;
    std::cout << "     Moving anything" << std::endl;
    std::cout << "     Transferring ownership" << std::endl;
    std::cout << "     Doing any work" << std::endl;

    std::cout << "\n std::move IS:" << std::endl;
    std::cout << " •   A CAST from lvalue to rvalue reference" << std::endl;
    std::cout << " •   Tells compiler: 'This object is about to expire'" << std::endl;
    std::cout << " •   Enables move constructor/assignment to be called" << std::endl;

    std::cout << "\n Conceptual implementation:" << std::endl;
    std::cout << "   template<typename T>" << std::endl;
    std::cout << "   typename remove_reference<T>::type&& move(T&& arg) {" << std::endl;
    std::cout << "       return static_cast<typename remove_reference<T>::type&&>(arg);" << std::endl;
    std::cout << "   }" << std::endl;

    std::string str1 = "Hello";

    std::cout << "\n Example:" << std::endl;
    std::cout << "   string str1 = \"Hello\";  // str1 is lvalue" << std::endl;
    std::cout << "   string str2 = str1;      // COPY: str1 is lvalue → copy constructor" << std::endl;
    std::cout << "   string str3 = std::move(str1); // MOVE: std::move(str1) is rvalue → move constructor" << std::endl;

    std::string str2 = str1;           // Copy
    std::string str3 = std::move(str1); // Move (str1 now in valid but
```

```
        unspecified state)

    std::cout << "\n   After move:" << std::endl;
    std::cout << "   str1 = \"" << str1 << "\" (moved-from, valid but empty)" << std::endl;
    std::cout << "   str2 = \"" << str2 << "\" (copy, unchanged)" << std::endl;
    std::cout << "   str3 = \"" << str3 << "\" (move, got str1's data)" << std::endl;
}

// ==================================================================
// 3. COPY vs MOVE - Performance Comparison
// ==================================================================

class LargeObject {
private:
    int* data;
    size_t size;
    std::string name;

public:
    // Constructor
    LargeObject(const std::string& n, size_t s)
        : size(s), name(n) {
        data = new int[size];
        for (size_t i = 0; i < size; ++i) {
            data[i] = static_cast<int>(i);
        }
        std::cout << "     Constructor: " << name
                  << " (" << size << " elements)" << std::endl;
    }

    // Destructor
    ~LargeObject() {
        delete[] data;
        std::cout << "    Destructor: " << name << std::endl;
    }

    // Copy constructor (EXPENSIVE)
    LargeObject(const LargeObject& other)
        : size(other.size), name(other.name + "_copy") {
        data = new int[size];
        // Copy every element - EXPENSIVE!
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
        std::cout << "    COPY constructor: " << name
                  << " (" << size << " elements copied)" << std::endl;
    }

    // Move constructor (CHEAP)
    LargeObject(LargeObject&& other) noexcept
        : data(other.data), size(other.size), name(other.name + "_moved") {
```

```
        // Steal the data pointer - CHEAP!
        other.data = nullptr;
        other.size = 0;
        std::cout << "     MOVE constructor: " << name
                  << " (pointer stolen, zero copy!)" << std::endl;
    }

    // Copy assignment (EXPENSIVE)
    LargeObject& operator=(const LargeObject& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            name = other.name + "_copy_assigned";
            data = new int[size];
            for (size_t i = 0; i < size; ++i) {
                data[i] = other.data[i];
            }
            std::cout << "     COPY assignment: " << name << std::endl;
        }
        return *this;
    }

    // Move assignment (CHEAP)
    LargeObject& operator=(LargeObject&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            size = other.size;
            name = other.name + "_move_assigned";
            other.data = nullptr;
            other.size = 0;
            std::cout << "     MOVE assignment: " << name << std::endl;
        }
        return *this;
    }

    size_t get_size() const { return size; }
};

void demonstrate_copy_vs_move() {
    std::cout << "\n=== 3. COPY vs MOVE CONSTRUCTORS ===" << std::endl;

    std::cout << "\n--- Copy Constructor (expensive) ---" << std::endl;
    {
        LargeObject obj1("obj1", 1000);
        LargeObject obj2 = obj1;  // COPY: obj1 is lvalue
        std::cout << "   Result: obj1 still has data, obj2 copied it" << std::
            endl;
    }

    std::cout << "\n--- Move Constructor (cheap) ---" << std::endl;
    {
        LargeObject obj3("obj3", 1000);
        LargeObject obj4 = std::move(obj3);  // MOVE: std::move(obj3) is
```

```
            rvalue
        std::cout << "   Result: obj3 empty (moved-from), obj4 stole its data" << std::endl;
    }

    std::cout << "\n--- Assignment operators ---" << std::endl;
    {
        LargeObject obj5("obj5", 500);
        LargeObject obj6("obj6", 500);
        LargeObject obj7("obj7", 500);

        std::cout << "\nCopy assignment:" << std::endl;
        obj6 = obj5;  // Copy assignment

        std::cout << "\nMove assignment:" << std::endl;
        obj7 = std::move(obj5);  // Move assignment
    }

    std::cout << "\n Performance impact:" << std::endl;
    std::cout << " •   Copy: O(n) - copies every element" << std::endl;
    std::cout << " •   Move: O(1) - just copies pointers" << std::endl;
    std::cout << " •   For 1,000,000 elements: Copy = 1M ops, Move = 3 ops!" << std::endl;
}

// ================================================================
// 4. WHY MOVE SEMANTICS ARE IMPORTANT
// ================================================================

// Scenario 1: Returning large objects from functions

std::vector<int> create_large_vector_without_move() {
    std::vector<int> vec(1000000, 42);
    // Before C++11: Copy entire vector on return (SLOW!)
    // With C++11: RVO or move (FAST!)
    return vec;  // Move constructor called (or RVO)
}

// Scenario 2: Storing move-only types in containers

class MoveOnlyType {
    std::unique_ptr<int> ptr;

public:
    MoveOnlyType(int value) : ptr(std::make_unique<int>(value)) {
        std::cout << "   MoveOnlyType created with value: " << *ptr << std::endl;
    }

    // Delete copy constructor and copy assignment
    MoveOnlyType(const MoveOnlyType&) = delete;
    MoveOnlyType& operator=(const MoveOnlyType&) = delete;

    // Enable move constructor and move assignment
```

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

    int get_value() const { return *ptr; }
};

void demonstrate_why_move_important() {
    std::cout << "\n=== 4. WHY MOVE SEMANTICS ARE IMPORTANT ===" << std::endl;

    // Scenario 1: Return values
    std::cout << "\n Scenario 1: Returning large objects" << std::endl;
    std::cout << "   Before C++11: Expensive copy on return" << std::endl;
    std::cout << "   With C++11: Move or RVO (Return Value Optimization)" << std::endl;

    auto start = std::chrono::high_resolution_clock::now();
    auto vec = create_large_vector_without_move();
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    std::cout << "   Created vector with " << vec.size()
              << " elements in " << duration.count() << "  s" << std::endl;
    std::cout << "   (Would be much slower without move/RVO)" << std::endl;

    // Scenario 2: Move-only types
    std::cout << "\n Scenario 2: Move-only types (unique_ptr, thread, etc.)" << std::endl;
    {
        MoveOnlyType obj1(42);
        // MoveOnlyType obj2 = obj1;  // ERROR: Copy deleted
        MoveOnlyType obj2 = std::move(obj1);  // OK: Move
        std::cout << "     Moved unique ownership from obj1 to obj2" << std::endl;
    }

    // Scenario 3: Container operations
    std::cout << "\n Scenario 3: Container optimizations" << std::endl;
    {
        std::vector<std::string> names;
        names.reserve(3);

        std::string name1 = "Alice";
        std::string name2 = "Bob";

        std::cout << "   push_back(name1): COPY (name1 still needed)" << std::endl;
        names.push_back(name1);  // Copy

        std::cout << "   push_back(std::move(name2)): MOVE (name2 not needed)" << std::endl;
        names.push_back(std::move(name2));  // Move

        std::cout << "   push_back(\"Charlie\"): MOVE (temporary rvalue)" <<
```

```
            std::endl;
        names.push_back("Charlie");  // Move from temporary

        std::cout << "\n   After operations:" << std::endl;
        std::cout << "   name1 = \"" << name1 << "\" (still valid, was copied)" << std::endl;
        std::cout << "   name2 = \"" << name2 << "\" (moved-from, empty)" << std::endl;
    }

    // Scenario 4: Swapping
    std::cout << "\n Scenario 4: Efficient swapping" << std::endl;
    {
        std::vector<int> vec1(1000, 1);
        std::vector<int> vec2(2000, 2);

        std::cout << "   Before swap: vec1.size()=" << vec1.size()
                  << ", vec2.size()=" << vec2.size() << std::endl;

        // std::swap uses move semantics internally
        std::swap(vec1, vec2);  // O(1) with moves, not O(n) with copies!

        std::cout << "   After swap:  vec1.size()=" << vec1.size()
                  << ", vec2.size()=" << vec2.size() << std::endl;
        std::cout << "   (Swap is O(1) thanks to move semantics)" << std::endl;
    }
}

// ==================================================================
// 5. COMMON PITFALLS AND BEST PRACTICES
// ==================================================================

void demonstrate_pitfalls() {
    std::cout << "\n=== 5. COMMON PITFALLS ===" << std::endl;

    // Pitfall 1: Using moved-from object
    std::cout << "\n Pitfall 1: Using moved-from object" << std::endl;
    {
        std::string str1 = "Hello";
        std::string str2 = std::move(str1);

        std::cout << "   str1 after move: \"" << str1 << "\"" << std::endl;
        std::cout << "     str1 is in valid but unspecified state!" << std::endl;
        std::cout << "    Safe: Check if empty or reassign" << std::endl;
        std::cout << "    Unsafe: Assume it has specific content" << std::endl;

        // Safe operations after move:
        str1.clear();        // OK
        str1 = "New value";  // OK
        if (str1.empty()) {} // OK
        // str1[0];          // UNSAFE! May be undefined behavior
```

```
    }

    // Pitfall 2: Moving const objects
    std::cout << "\n Pitfall 2: Can't move from const" << std::endl;
    {
        const std::string str1 = "Hello";
        std::string str2 = std::move(str1);  // Calls COPY, not move!

        std::cout << "   const objects can't be moved" << std::endl;
        std::cout << "   std::move on const → copy constructor called" << std::endl;
    }

    // Pitfall 3: Unnecessary moves
    std::cout << "\n Pitfall 3: Unnecessary std::move on return" << std::endl;
    std::cout << R"(
   // BAD - prevents RVO (Return Value Optimization)
   string bad_function() {
       string result = "value";
       return std::move(result);  // Don't do this!
   }

   // GOOD - enables RVO
   string good_function() {
       string result = "value";
       return result;  // Compiler automatically moves/optimizes
   }
)" << std::endl;

    // Best practices
    std::cout << "\n BEST PRACTICES:" << std::endl;
    std::cout << "   1. Mark move constructors 'noexcept'" << std::endl;
    std::cout << " →      Enables optimizations (vector resize)" << std::endl;
    std::cout << "   2. Don't std::move on return values" << std::endl;
    std::cout << " →      Compiler does RVO or automatic move" << std::endl;
    std::cout << "   3. Move when you know object won't be used again" << std::endl;
    std::cout << " →      push_back(std::move(obj))" << std::endl;
    std::cout << "   4. Don't use moved-from objects" << std::endl;
    std::cout << " →      Unless reassigning or destroying" << std::endl;
    std::cout << "   5. Move-only types (unique_ptr) can't be copied" << std::endl;
    std::cout << " →      Use std::move to transfer ownership" << std::endl;
}

// ================================================================
// 6. std::forward - PERFECT FORWARDING
// ================================================================

void process(int& x) {
    std::cout << "   process(int&): lvalue " << x << std::endl;
}

```

```
void process(int&& x) {
    std::cout << "   process(int&&): rvalue " << x << std::endl;
}

// Without perfect forwarding - loses rvalue-ness
template<typename T>
void wrapper_bad(T&& arg) {
    // arg is always an lvalue inside the function (even if T&& is rvalue ref)
    process(arg);  // Always calls process(int&)
}

// With perfect forwarding - preserves value category
template<typename T>
void wrapper_good(T&& arg) {
    // std::forward preserves lvalue/rvalue-ness
    process(std::forward<T>(arg));  // Calls correct overload
}

void demonstrate_perfect_forwarding() {
    std::cout << "\n=== 6. std::forward - PERFECT FORWARDING ===" << std::endl;

    std::cout << "\n Problem: Template functions lose value category" << std::endl;
    std::cout << "   Template parameter T&& is 'forwarding reference'" << std::endl;
    std::cout << "   But inside function, arg is always lvalue!" << std::endl;

    int x = 42;

    std::cout << "\n Without std::forward:" << std::endl;
    wrapper_bad(x);           // lvalue
    wrapper_bad(100);         // rvalue, but treated as lvalue!

    std::cout << "\n With std::forward:" << std::endl;
    wrapper_good(x);          // Forwards as lvalue
    wrapper_good(200);        // Forwards as rvalue

    std::cout << "\n Key differences:" << std::endl;
    std::cout << " •   std::move:    ALWAYS casts to rvalue" << std::endl;
    std::cout << " •   std::forward: Casts to rvalue ONLY if original was rvalue" << std::endl;
    std::cout << "                    (preserves value category)" << std::endl;

    std::cout << "\n When to use:" << std::endl;
    std::cout << " •   std::move:    When you know object is done being used" << std::endl;
    std::cout << " •   std::forward: In template functions forwarding arguments" << std::endl;
}

// ==================================================================
// 7. REAL-WORLD EXAMPLE: Factory Pattern
// ==================================================================
```

```

class Resource {
    std::string name;
    std::vector<int> data;

public:
    Resource(const std::string& n) : name(n) {
        data.resize(1000, 42);
        std::cout << "   Resource created: " << name << std::endl;
    }

    Resource(const Resource& other)
        : name(other.name + "_copy"), data(other.data) {
        std::cout << "   Resource COPIED: " << name << std::endl;
    }

    Resource(Resource&& other) noexcept
        : name(std::move(other.name)), data(std::move(other.data)) {
        std::cout << "   Resource MOVED: " << name << std::endl;
    }

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

class ResourceFactory {
public:
    // Return by value - move semantics makes this efficient
    static Resource create_resource(const std::string& name) {
        Resource res(name);
        // Return local object - move or RVO kicks in
        return res;  // Don't use std::move here!
    }

    // Store in container - use move to avoid copy
    static void store_in_container(std::vector<Resource>& container, Resource
        && res) {
        std::cout << "   Storing in container..." << std::endl;
        container.push_back(std::move(res));  // Move into container
    }
};

void demonstrate_real_world() {
    std::cout << "\n=== 7. REAL-WORLD: Factory Pattern ===" << std::endl;

    std::vector<Resource> resources;
    resources.reserve(3);

    std::cout << "\n--- Creating and storing resources ---" << std::endl;

    // Create and store - efficient with move semantics
    auto res1 = ResourceFactory::create_resource("Resource1");
    ResourceFactory::store_in_container(resources, std::move(res1));

    // Direct temporary - even more efficient
```

```
    ResourceFactory::store_in_container(
        resources,
        ResourceFactory::create_resource("Resource2")
    );

    std::cout << "\n Container now has " << resources.size()
              << " resources" << std::endl;
    std::cout << "   All stored efficiently with move semantics!" << std::endl;
}

// ==================================================================
// MAIN
// ==================================================================

int main() {
    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  std::move AND MOVE SEMANTICS - COMPREHENSIVE GUIDE" << std::endl;
    std::cout << " ================================================================" << std::endl;

    demonstrate_lvalue_rvalue();
    demonstrate_std_move();
    demonstrate_copy_vs_move();
    demonstrate_why_move_important();
    demonstrate_pitfalls();
    demonstrate_perfect_forwarding();
    demonstrate_real_world();

    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  SUMMARY: std::move and Move Semantics" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\n KEY CONCEPTS:" << std::endl;
    std::cout << "\n 1  Lvalue vs Rvalue:" << std::endl;
    std::cout << " •   Lvalue: Has name, addressable, persists" << std::endl;
    std::cout << " •   Rvalue: Temporary, about to expire, no name" << std::endl;

    std::cout << "\n 2  std::move:" << std::endl;
    std::cout << " •   Just a CAST from lvalue to rvalue reference" << std::endl;
    std::cout << " •   Doesn't actually move anything" << std::endl;
    std::cout << " •   Tells compiler 'safe to steal resources'" << std::endl;

    std::cout << "\ n3  Move Constructor/Assignment:" << std::endl;
```

```
    std::cout << " •   Steals resources from source object" << std::endl;
    std::cout << " •   O(1) instead of O(n) for large objects" << std::endl;
    std::cout << " •   Should be marked 'noexcept'" << std::endl;

    std::cout << "\n4  Why Important:" << std::endl;
    std::cout << " •   Performance: O(1) vs O(n) for copies" << std::endl;
    std::cout << " •   Enables move-only types (unique_ptr, thread)" << std::endl;
    std::cout << " •   Efficient return values and container ops" << std::endl;
    std::cout << " •   Essential for modern C++" << std::endl;

    std::cout << "\n5  Best Practices:" << std::endl;
    std::cout << "     DO: Mark move operations noexcept" << std::endl;
    std::cout << "     DO: Move when object not needed anymore" << std::endl;
    std::cout << "     DO: Use std::forward in template forwarding" << std::endl;
    std::cout << "     DON'T: std::move on return values (blocks RVO)" << std::endl;
    std::cout << "     DON'T: Use moved-from objects (unless reassigning)" << std::endl;
    std::cout << "     DON'T: Move const objects (silently copies)" << std::endl;

    std::cout << "\n PERFORMANCE IMPACT:" << std::endl;
    std::cout << "   vector<int>(1,000,000):" << std::endl;
    std::cout << " •   Copy:  ~4,000,000 bytes copied, ~1ms" << std::endl;
    std::cout << " •   Move:  ~24 bytes copied, ~1s (1000x faster!)" << std::endl;

    std::cout << "\n WHEN TO USE:" << std::endl;
    std::cout << " •   Returning local objects from functions" << std::endl;
    std::cout << " •   Inserting into containers (push_back)" << std::endl;
    std::cout << " •   Transferring unique ownership (unique_ptr)" << std::endl;
    std::cout << " •   Implementing swap operations" << std::endl;
    std::cout << " •   Last use of a variable before destruction" << std::endl;

    std::cout << "\n==================================================================\n" << std::endl;

    return 0;
}
```

43 Source Code: MoveSematicsPerfectForwarding.cpp

File: `src/MoveSematicsPerfectForwarding.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// IMPLEMENTING MOVE SEMANTICS AND PERFECT FORWARDING
// ==================================================================
// A comprehensive guide to C++11/14/17/20 move semantics and perfect forwarding
//
// Topics Covered:
// 1. Lvalues vs Rvalues - Understanding value categories
// 2. Move Semantics - Efficient resource transfer
// 3. std::move - What it really does
// 4. Perfect Forwarding - Preserving value categories
// 5. Rule of Zero/Three/Five
// 6. Real-World Examples
// 7. Performance Comparison
//
// Build: g++ -std=c++20 -Wall -Wextra -O2 -o MoveSematicsPerfectForwarding MoveSematicsPerfectForwarding.cpp
// ==================================================================

#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <utility>  // std::move, std::forward
#include <chrono>

// ==================================================================
// SECTION 1: LVALUES vs RVALUES
// ==================================================================

namespace lvalues_rvalues {

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 1: LVALUES vs RVALUES\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  THEORY:\n";
    std::cout << "    Lvalue = Has a NAME and PERSISTENT ADDRESS\n";
    std::cout << "    Rvalue = TEMPORARY object without persistent address\n\n";

    // LVALUES - have names, can take address
    int x = 42;
    std::string name = "Hello";

    std::cout << "  LVALUES (have names, addressable):\n";
    std::cout << "    int x = 42;             // x is lvalue, address: " << &x << "\n";
```

```
    std::cout << "   string name = \"Hello\"; // name is lvalue, address: " << &name << "\n\n";

    // RVALUES - temporaries
    std::cout << "  RVALUES (temporaries, no persistent address):\n";
    std::cout << "   42                      // Literal\n";
    std::cout << "   x + 1                   // Expression result\n";
    std::cout << "   string(\"World\")         // Temporary object\n\n";

    // REFERENCE BINDING
    std::cout << "  REFERENCE BINDING:\n\n";

    int y = 10;

    // Lvalue reference - binds to lvalue
    int& lref = y;
    std::cout << "   int& lref = y;          //  OK: lvalue ref binds to lvalue\n";
    // int& lref2 = 42;    //  ERROR: lvalue ref cannot bind to rvalue

    // Rvalue reference - binds to rvalue
    int&& rref = 42;
    std::cout << "   int&& rref = 42;        //  OK: rvalue ref binds to rvalue\n";
    // int&& rref2 = y;    //  ERROR: rvalue ref cannot bind to lvalue

    // Special case: const lvalue reference can bind to both!
    const int& cref = 42;
    std::cout << "   const int& cref = 42;   //  OK: const lvalue ref extends lifetime\n\n";

    std::cout << "  WHY const& accepts rvalues:\n";
    std::cout << "   This is why function parameters use 'const T&' for read-only access.\n";
    std::cout << "   They can accept BOTH lvalues and rvalues!\n";
}

} // namespace lvalues_rvalues

// ==================================================================
// SECTION 2: MOVE SEMANTICS - Basic Implementation
// ==================================================================

namespace move_semantics {

class Resource {
private:
    int* data_;
    size_t size_;

public:
    // Constructor
    explicit Resource(size_t size = 0)
        : data_(size > 0 ? new int[size] : nullptr), size_(size) {
```

```
        std::cout << "   [Resource] Constructor: allocated " << size_ << " ints\n";
    }

    // Destructor
    ~Resource() {
        if (data_) {
            std::cout << "   [Resource] Destructor: freeing " << size_ << " ints\n";
            delete[] data_;
        }
    }

    // Copy Constructor - EXPENSIVE! (O(n))
    Resource(const Resource& other) : size_(other.size_) {
        std::cout << "   [Resource] Copy Constructor - EXPENSIVE! Copying "
                  << size_ << " ints\n";
        if (size_ > 0) {
            data_ = new int[size_];
            std::copy_n(other.data_, size_, data_);
        } else {
            data_ = nullptr;
        }
    }

    // Copy Assignment - EXPENSIVE! (O(n))
    Resource& operator=(const Resource& other) {
        std::cout << "   [Resource] Copy Assignment - EXPENSIVE! Copying "
                  << other.size_ << " ints\n";
        if (this != &other) {
            delete[] data_;
            size_ = other.size_;
            if (size_ > 0) {
                data_ = new int[size_];
                std::copy_n(other.data_, size_, data_);
            } else {
                data_ = nullptr;
            }
        }
        return *this;
    }

    // Move Constructor - FAST! (O(1))
    Resource(Resource&& other) noexcept
        : data_(other.data_), size_(other.size_) {
        std::cout << "   [Resource] Move Constructor - FAST! O(1) pointer transfer\n";

        // Leave source in valid state
        other.data_ = nullptr;
        other.size_ = 0;
    }

    // Move Assignment - FAST! (O(1))
```

```
    Resource& operator=(Resource&& other) noexcept {
        std::cout << "   [Resource] Move Assignment - FAST! O(1) pointer transfer\n";
        if (this != &other) {
            // Clean up existing resource
            delete[] data_;

            // Transfer ownership
            data_ = other.data_;
            size_ = other.size_;

            // Leave source in valid state
            other.data_ = nullptr;
            other.size_ = 0;
        }
        return *this;
    }

    size_t size() const { return size_; }

    int* get_data() { return data_; }
    const int* get_data() const { return data_; }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 2: MOVE SEMANTICS\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  THE PROBLEM: Expensive Copies\n";
    std::cout << "    Copy constructor/assignment performs deep copy (O(n))\n";
    std::cout << "    For large objects, this is 10x-1000x slower than necessary!\n\n";

    std::cout << "  Demonstrating COPY vs MOVE:\n\n";

    // Copy Constructor
    std::cout << " 1 COPY Constructor (expensive):\n";
    Resource r1(5);
    Resource r2 = r1;  // Copy constructor
    std::cout << "    Result: r1 still valid with " << r1.size() << " ints\n\n";

    // Move Constructor
    std::cout << " 2 MOVE Constructor (fast):\n";
    Resource r3(5);
    Resource r4 = std::move(r3);  // Move constructor
    std::cout << "    Result: r3 moved-from (size=" << r3.size()
              << "), r4 owns data (" << r4.size() << " ints)\n\n";

    // Copy Assignment
    std::cout << " 3 COPY Assignment (expensive):\n";
    Resource r5(3);
    Resource r6(7);
```

```
    r6 = r5;  // Copy assignment
    std::cout << "   Result: Both valid, r6 has " << r6.size() << " ints\n\n";

    // Move Assignment
    std::cout << " 4 MOVE Assignment (fast):\n";
    Resource r7(3);
    Resource r8(7);
    r8 = std::move(r7);  // Move assignment
    std::cout << "   Result: r7 moved-from (size=" << r7.size()
              << "), r8 owns data (" << r8.size() << " ints)\n\n";

    std::cout << "  KEY INSIGHT:\n";
    std::cout << "   Move operations transfer ownership in O(1) time!\n";
    std::cout << "   Copy operations duplicate data in O(n) time.\n";
    std::cout << "   Always mark move operations 'noexcept' for std::vector
        optimization!\n";
}

} // namespace move_semantics

// ==================================================================
// SECTION 3: UNDERSTANDING std::move
// ==================================================================

namespace understanding_move {

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 3: UNDERSTANDING std::move\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  WHAT IS std::move?\n";
    std::cout << "   std::move does NOT move anything!\n";
    std::cout << "   It's just a CAST to rvalue reference (T&&)\n";
    std::cout << "   It tells compiler: 'this object can be moved from'\n\n";

    std::cout << "   Simplified implementation:\n";
    std::cout << "   template<typename T>\n";
    std::cout << "   T&& move(T&& t) noexcept {\n";
    std::cout << "       return static_cast<T&&>(t);\n";
    std::cout << "   }\n\n";

    std::cout << "  DEMONSTRATION:\n\n";

    // std::move with string
    std::cout << " 1 std::move with std::string:\n";
    std::string s1 = "Hello, World!";
    std::cout << "   Before: s1 = \"" << s1 << "\"\n";

    std::string s2 = std::move(s1);  // Move constructor called
    std::cout << "   After move:\n";
    std::cout << "     s1 = \"" << s1 << "\" (moved-from state)\n";
    std::cout << "     s2 = \"" << s2 << "\" (owns the data)\n\n";

```

```
    // Moved-from state
    std::cout << " 2 MOVED-FROM STATE:\n";
    std::cout << "   After std::move, object is in 'valid but unspecified' state\n";
    std::cout << "     SAFE: Destroy it, assign to it\n";
    std::cout << "     UNSAFE: Don't use its value!\n\n";

    s1 = "New Value";  // Safe: reassignment
    std::cout << "   s1 reassigned: \"" << s1 << "\" (valid again)\n\n";

    // Common mistakes
    std::cout << "  COMMON MISTAKES:\n\n";

    std::cout << "  MISTAKE 1: Using object after move\n";
    std::cout << "   string s3 = \"test\";\n";
    std::cout << "   string s4 = std::move(s3);\n";
    std::cout << "   cout << s3;  //  WRONG! Undefined behavior!\n\n";

    std::cout << "  MISTAKE 2: Returning std::move(local)\n";
    std::cout << "   string func() {\n";
    std::cout << "       string s = \"hello\";\n";
    std::cout << "       return std::move(s);  //  Prevents RVO!\n";
    std::cout << "   }\n";
    std::cout << "     Correct: return s;  // RVO or move, NOT copy\n\n";

    std::cout << "  MISTAKE 3: Moving from const\n";
    std::cout << "   const string cs = \"test\";\n";
    std::cout << "   string s = std::move(cs);  //  Calls COPY, not move!\n\n";

    std::cout << "  KEY TAKEAWAY:\n";
    std::cout << "   std::move is a PERMISSION to move, not a command.\n";
    std::cout << "   The actual move happens in the move constructor/assignment.\n";
}

} // namespace understanding_move

// ==================================================================
// SECTION 4: PERFECT FORWARDING
// ==================================================================

namespace perfect_forwarding {

class Widget {
public:
    Widget(const std::string& name) {
        std::cout << "   [Widget] Constructed with LVALUE: \"" << name << "\" (copied)\n";
    }

    Widget(std::string&& name) {
        std::cout << "   [Widget] Constructed with RVALUE: \"" << name << "\" (moved)\n";
```

```
    }
};

// Bad: Always copies
template<typename T>
void bad_wrapper(T arg) {
    // arg is always an lvalue here, even if rvalue was passed!
    Widget w(arg);  // Always calls Widget(const string&)
}

// Good: Perfect forwarding
template<typename T>
void good_wrapper(T&& arg) {  // Universal/Forwarding reference
    // std::forward preserves the value category
    Widget w(std::forward<T>(arg));
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 4: PERFECT FORWARDING\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  THE PROBLEM:\n";
    std::cout << "    When passing arguments through wrapper functions,\n";
    std::cout << "    we lose information about whether they were lvalues or rvalues.\n\n";

    std::cout << "  DEMONSTRATION:\n\n";

    std::cout << " 1 WITHOUT Perfect Forwarding (bad_wrapper):\n";
    std::string s1 = "lvalue";
    bad_wrapper(s1);                      // Copies (expected)
    bad_wrapper(std::string("rvalue")); // Still COPIES (not desired!)
    std::cout << "\n";

    std::cout << " 2 WITH Perfect Forwarding (good_wrapper):\n";
    std::string s2 = "lvalue";
    good_wrapper(s2);                      // Copies (as expected)
    good_wrapper(std::string("rvalue")); // MOVES (optimized!)
    std::cout << "\n";

    std::cout << "  UNIVERSAL REFERENCE (T&&):\n";
    std::cout << "    template<typename T>\n";
    std::cout << "    void func(T&& arg);  // NOT an rvalue reference!\n\n";
    std::cout << "    It's a UNIVERSAL REFERENCE (also called forwarding reference)\n";
    std::cout << "    Can bind to BOTH lvalues and rvalues\n\n";

    std::cout << "    Reference Collapsing Rules:\n";
    std::cout << "    T&  &&  ->  T&    (lvalue reference)\n";
    std::cout << "    T&& &&  ->  T&&   (rvalue reference)\n\n";

    std::cout << "  WHEN TO USE:\n";
    std::cout << "    std::move    - UNCONDITIONAL cast to rvalue\n";
```

```
    std::cout << "                    Use when you KNOW you have an lvalue to transfer\n";
    std::cout << "   std::forward - CONDITIONAL forwarding preserving value category\n";
    std::cout << "                    Use with T&& (universal reference) in templates\n";
}

// Example: Factory function
template<typename T, typename... Args>
std::unique_ptr<T> make_unique_custom(Args&&... args) {
    std::cout << "   [Factory] Creating object with perfect forwarding\n";
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

void demonstrate_factory() {
    std::cout << "\ n3 Factory Pattern with Perfect Forwarding:\n";

    std::string name = "Gadget";
    auto w1 = make_unique_custom<Widget>(name);           // Forwards as lvalue
    auto w2 = make_unique_custom<Widget>("Gizmo");        // Forwards as rvalue
    auto w3 = make_unique_custom<Widget>(std::move(name)); // Forwards as rvalue
}

} // namespace perfect_forwarding

// ==================================================================
// SECTION 5: RULE OF ZERO/THREE/FIVE
// ==================================================================

namespace rule_of_five {

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 5: RULE OF ZERO/THREE/FIVE\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  RULE OF ZERO (PREFERRED!):\n";
    std::cout << "   Don't manage resources manually!\n";
    std::cout << "   Use std::vector, std::string, std::unique_ptr, etc.\n\n";

    std::cout << "   class Widget {\n";
    std::cout << "       std::string name_;        // Self-managing\n";
    std::cout << "       std::vector<int> data_;   // Self-managing\n";
    std::cout << "       // No special members needed!\n";
    std::cout << "   };\n\n";

    std::cout << "  RULE OF THREE (C++98):\n";
    std::cout << "   If you define ONE, define ALL THREE:\n";
    std::cout << "   1. Destructor\n";
    std::cout << "   2. Copy Constructor\n";
```

```
    std::cout << "   3. Copy Assignment Operator\n\n";

    std::cout << " RULE OF FIVE (C++11+):\n";
    std::cout << "   If managing resources, define ALL FIVE:\n";
    std::cout << "   1. Destructor\n";
    std::cout << "   2. Copy Constructor\n";
    std::cout << "   3. Copy Assignment Operator\n";
    std::cout << "   4. Move Constructor (mark noexcept!)\n";
    std::cout << "   5. Move Assignment Operator (mark noexcept!)\n\n";

    std::cout << "   class Resource {\n";
    std::cout << "   public:\n";
    std::cout << "       ~Resource();                            // 1\n";
    std::cout << "       Resource(const Resource&);              // 2\n";
    std::cout << "       Resource& operator=(const Resource&);   // 3\n";
    std::cout << "       Resource(Resource&&) noexcept;          // 4\n";
    std::cout << "       Resource& operator=(Resource&&) noexcept; // 5\n";
    std::cout << "   };\n\n";

    std::cout << " DELETE UNWANTED OPERATIONS:\n";
    std::cout << "   Make class non-copyable but movable:\n\n";
    std::cout << "   class NonCopyable {\n";
    std::cout << "   public:\n";
    std::cout << "       NonCopyable(const NonCopyable&) = delete;\n";
    std::cout << "       NonCopyable& operator=(const NonCopyable&) = delete;\n";
    std::cout << "       NonCopyable(NonCopyable&&) = default;\n";
    std::cout << "       NonCopyable& operator=(NonCopyable&&) = default;\n";
    std::cout << "   };\n\n";

    std::cout << " BEST PRACTICE:\n";
    std::cout << "   1. Prefer Rule of Zero (use standard library types)\n";
    std::cout << "   2. If managing resources, use Rule of Five\n";
    std::cout << "   3. Always mark move operations noexcept\n";
    std::cout << "   4. Use =delete for unwanted operations\n";
    std::cout << "   5. Use =default when compiler-generated is correct\n";
}

} // namespace rule_of_five

// ==================================================================
// SECTION 6: REAL-WORLD EXAMPLES
// ==================================================================

namespace real_world_examples {

void demonstrate_vector() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 6: REAL-WORLD EXAMPLES\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << " 1 std::vector Push Operations:\n\n";

    std::vector<std::string> vec;
```

```
    vec.reserve(3);

    std::string s1 = "String1";
    std::string s2 = "String2";

    std::cout << "   vec.push_back(s1);            // Copy (s1 still valid)\n";
    vec.push_back(s1);
    std::cout << "   s1 after: \"" << s1 << "\"\n\n";

    std::cout << "   vec.push_back(std::move(s2)); // Move (s2 now empty)\n";
    vec.push_back(std::move(s2));
    std::cout << "   s2 after: \"" << s2 << "\" (moved-from)\n\n";

    std::cout << "   vec.emplace_back(\"String3\");  // Construct in-place\n";
    vec.emplace_back("String3");
    std::cout << "\n";
}

void demonstrate_rvo() {
    std::cout << " 2 Return Value Optimization (RVO):\n\n";

    std::cout << "     CORRECT:\n";
    std::cout << "   vector<int> create() {\n";
    std::cout << "       vector<int> result(1000);\n";
    std::cout << "       return result;  // RVO or move (NOT copy!)\n";
    std::cout << "   }\n\n";

    std::cout << "     WRONG:\n";
    std::cout << "   vector<int> create() {\n";
    std::cout << "       vector<int> result(1000);\n";
    std::cout << "       return std::move(result);  // Prevents RVO!\n";
    std::cout << "   }\n\n";
}

void demonstrate_unique_ptr() {
    std::cout << " 3 Unique Ownership Transfer:\n\n";

    std::cout << "   auto ptr1 = make_unique<int>(42);\n";
    auto ptr1 = std::make_unique<int>(42);
    std::cout << "   ptr1 owns: " << *ptr1 << "\n\n";

    std::cout << "   auto ptr2 = std::move(ptr1);  // Transfer ownership\n";
    auto ptr2 = std::move(ptr1);
    std::cout << "   ptr1 is now: " << (ptr1 ? "valid" : "nullptr") << "\n";
    std::cout << "   ptr2 owns: " << *ptr2 << "\n\n";
}

void demonstrate_factory() {
    std::cout << " 4 Factory Functions with Perfect Forwarding:\n\n";

    std::cout << "   See Section 4 for detailed factory examples!\n\n";
}

```

```
void demonstrate_all() {
    demonstrate_vector();
    demonstrate_rvo();
    demonstrate_unique_ptr();
    demonstrate_factory();
}

} // namespace real_world_examples

// ====================================================================
// SECTION 7: PERFORMANCE COMPARISON
// ====================================================================

namespace performance_comparison {

std::vector<int> create_large_vector(size_t size) {
    std::vector<int> result(size);
    for (size_t i = 0; i < size; ++i) {
        result[i] = static_cast<int>(i);
    }
    return result;  // RVO or move
}

void benchmark_copy_vs_move() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 7: PERFORMANCE COMPARISON\n";
    std::cout << std::string(70, '=') << "\n\n";

    const size_t size = 1000000;  // 1 million integers
    const int iterations = 100;

    std::cout << "  Benchmark Configuration:\n";
    std::cout << "    Buffer size: " << size << " integers\n";
    std::cout << "    Iterations: " << iterations << "\n\n";

    // Benchmark COPY
    std::cout << " 1 Testing COPY operations...\n";
    auto start_copy = std::chrono::high_resolution_clock::now();

    std::vector<int> source = create_large_vector(size);
    for (int i = 0; i < iterations; ++i) {
        std::vector<int> dest = source;  // Copy
        (void)dest;  // Prevent optimization
    }

    auto end_copy = std::chrono::high_resolution_clock::now();
    auto copy_time = std::chrono::duration_cast<std::chrono::milliseconds>(
        end_copy - start_copy).count();

    // Benchmark MOVE
    std::cout << " 2 Testing MOVE operations...\n";
    auto start_move = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < iterations; ++i) {
```

```
        std::vector<int> source_temp = create_large_vector(size);
        std::vector<int> dest = std::move(source_temp);  // Move
        (void)dest;  // Prevent optimization
    }

    auto end_move = std::chrono::high_resolution_clock::now();
    auto move_time = std::chrono::duration_cast<std::chrono::milliseconds>(
        end_move - start_move).count();

    // Results
    std::cout << "\ n RESULTS:\n";
    std::cout << "   Copy time: " << copy_time << " ms\n";
    std::cout << "   Move time: " << move_time << " ms\n";
    std::cout << "   Speedup:   " << (double)copy_time / move_time << "x
        faster!\n\n";

    std::cout << "  INTERPRETATION:\n";
    std::cout << "   Copy: O(n) - copies all " << size << " integers\n";
    std::cout << "   Move: O(1) - just transfers pointer ownership\n";
    std::cout << "   For large objects, move is 10x-1000x+ faster!\n";
}

} // namespace performance_comparison

// ==================================================================
// MAIN - Run All Demonstrations
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "      IMPLEMENTING MOVE SEMANTICS AND PERFECT FORWARDING
                 \n";
    std::cout << "             Comprehensive Guide to Modern C++ Features
                  \n";
    std::cout << "                                    \n";

    try {
        // Section 1: Lvalues vs Rvalues
        lvalues_rvalues::demonstrate();

        // Section 2: Move Semantics
        move_semantics::demonstrate();

        // Section 3: Understanding std::move
        understanding_move::demonstrate();

        // Section 4: Perfect Forwarding
        perfect_forwarding::demonstrate();
        perfect_forwarding::demonstrate_factory();

        // Section 5: Rule of Zero/Three/Five
        rule_of_five::demonstrate();

```

```
        // Section 6: Real-World Examples
        real_world_examples::demonstrate_all();

        // Section 7: Performance Comparison
        performance_comparison::benchmark_copy_vs_move();

        // Summary
        std::cout << "\n" << std::string(70, '=') << "\n";
        std::cout << "SUMMARY: KEY TAKEAWAYS\n";
        std::cout << std::string(70, '=') << "\n\n";

        std::cout << "  Move Semantics:\n";
        std::cout << " •   Enables O(1) resource transfer instead of O(n) copy
            \n";
        std::cout << " •   Always mark move operations noexcept\n";
        std::cout << " •   10x-1000x+ performance improvement for large
            objects\n\n";

        std::cout << "  Perfect Forwarding:\n";
        std::cout << " •   Preserves lvalue/rvalue category through templates\
            n";
        std::cout << " •   Use T&& (universal reference) with std::forward<T>\
            n";
        std::cout << " •   Essential for factory functions and wrappers\n\n";

        std::cout << "  Best Practices:\n";
        std::cout << " •   Prefer Rule of Zero (use std types)\n";
        std::cout << " •   If managing resources, implement Rule of Five\n";
        std::cout << " •   Return by value (trust RVO and move)\n";
        std::cout << " •   Use std::move for explicit ownership transfer\n";
        std::cout << " •   Use std::forward in template forwarding\n";
        std::cout << " •   Don't use objects after moving from them\n\n";

        std::cout << "                                    \n";
        std::cout << "                        ALL EXAMPLES COMPLETED!
                                \n";
        std::cout << "                                    \n\n";

    } catch (const std::exception& e) {
        std::cerr << "  Error: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

44 Source Code: MultiThreadedMicroservices.cpp

File: `src/MultiThreadedMicroservices.cpp`
Repository: View on GitHub

```cpp
// MultiThreadedMicroservices.cpp
// Demonstrates multi-threaded microservices architecture with different
// exception handling strategies: REST threads use pthread_exit(), core threads use abort()

#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <vector>
#include <string>
#include <memory>
#include <atomic>
#include <chrono>
#include <sstream>
#include <iomanip>
#include <cstring>
#include <csignal>
#include <functional>

#ifdef __linux__
    #include <pthread.h>
    #include <execinfo.h>
    #include <cxxabi.h>
#elif _WIN32
    #include <windows.h>
    #include <dbghelp.h>
#endif

using namespace std::chrono;
using namespace std::chrono_literals;

// ==========================================================================
// SECTION 1: Stack Trace and Logging Infrastructure
// ==========================================================================

class Logger {
private:
    static std::mutex log_mutex_;

    static std::string get_timestamp() {
        auto now = system_clock::now();
        auto time = system_clock::to_time_t(now);
        auto ms = duration_cast<milliseconds>(now.time_since_epoch()) % 1000;
```

```

        std::ostringstream oss;
        oss << std::put_time(std::localtime(&time), "%H:%M:%S")
            << '.' << std::setfill('0') << std::setw(3) << ms.count();
        return oss.str();
    }

    static std::string get_thread_id() {
        std::ostringstream oss;
        oss << std::this_thread::get_id();
        return oss.str();
    }

public:
    enum Level { INFO, WARNING, ERROR, CRITICAL };

    static void log(Level level, const std::string& message,
                    const std::string& thread_name = "") {
        std::lock_guard<std::mutex> lock(log_mutex_);

        const char* level_str[] = {"INFO", "WARN", "ERROR", "CRIT"};
        const char* color[] = {"\033[32m", "\033[33m", "\033[31m", "\033[35m"
            };
        const char* reset = "\033[0m";

        std::cout << "[" << get_timestamp() << "] "
                  << color[level] << "[" << level_str[level] << "]" << reset
                  << " [TID:" << get_thread_id().substr(0, 6) << "]";

        if (!thread_name.empty()) {
            std::cout << " [" << thread_name << "]";
        }

        std::cout << " " << message << "\n";
    }

    static std::vector<std::string> capture_stack_trace() {
        std::vector<std::string> frames;

#ifdef __linux__
        void* buffer[32];
        int frame_count = backtrace(buffer, 32);
        char** symbols = backtrace_symbols(buffer, frame_count);

        if (symbols) {
            for (int i = 0; i < frame_count && i < 10; ++i) {
                std::string frame = symbols[i];

                // Try to demangle
                const char* begin = strchr(symbols[i], '(');
                const char* end = strchr(symbols[i], '+');

                if (begin && end && begin < end) {
                    begin++;
```

```
                    std::string mangled(begin, end - begin);
                    int status;
                    char* demangled = abi::__cxa_demangle(mangled.c_str(),
                        nullptr, nullptr, &status);
                    if (status == 0 && demangled) {
                        frame = demangled;
                        free(demangled);
                    }
                }

                frames.push_back(frame);
            }
            free(symbols);
        }
#elif _WIN32
        void* buffer[32];
        HANDLE process = GetCurrentProcess();
        SymInitialize(process, NULL, TRUE);

        WORD frame_count = CaptureStackBackTrace(0, 32, buffer, NULL);
        SYMBOL_INFO* symbol = (SYMBOL_INFO*)calloc(sizeof(SYMBOL_INFO) + 256,
            1);

        if (symbol) {
            symbol->MaxNameLen = 255;
            symbol->SizeOfStruct = sizeof(SYMBOL_INFO);

            for (WORD i = 0; i < frame_count && i < 10; ++i) {
                if (SymFromAddr(process, (DWORD64)buffer[i], 0, symbol)) {
                    frames.push_back(symbol->Name);
                } else {
                    std::ostringstream oss;
                    oss << "0x" << std::hex << (uint64_t)buffer[i];
                    frames.push_back(oss.str());
                }
            }
            free(symbol);
        }
        SymCleanup(process);
#else
        frames.push_back("[Stack trace not available]");
#endif

        return frames;
    }

    static void log_exception_with_stack(const std::exception& e,
                                         const std::string& thread_name) {
        log(CRITICAL, std::string("EXCEPTION: ") + e.what(), thread_name);

        auto stack = capture_stack_trace();
        if (!stack.empty()) {
            log(CRITICAL, "Stack trace:", thread_name);
            for (size_t i = 0; i < stack.size(); ++i) {
```

```
                std::ostringstream oss;
                oss << "  #" << i << ": " << stack[i];
                log(CRITICAL, oss.str(), thread_name);
            }
        }
    }
};

std::mutex Logger::log_mutex_;

// ==========================================================================
// SECTION 2: Thread Type Identification and Exception Policies
// ==========================================================================

enum class ThreadType {
    CORE_SERVICE,    // Critical - abort() on exception
    REST_SERVICE,    // Non-critical - pthread_exit() on exception
    MONITORING       // Non-critical - pthread_exit() on exception
};

class ThreadContext {
private:
    ThreadType type_;
    std::string name_;

public:
    ThreadContext(ThreadType type, const std::string& name)
        : type_(type), name_(name) {}

    ThreadType get_type() const { return type_; }
    const std::string& get_name() const { return name_; }

    bool is_core_service() const { return type_ == ThreadType::CORE_SERVICE; }
    bool is_rest_service() const { return type_ == ThreadType::REST_SERVICE; }

    void handle_exception(const std::exception& e) {
        Logger::log_exception_with_stack(e, name_);

        if (is_core_service()) {
            Logger::log(Logger::CRITICAL,
                        "CORE SERVICE FAILURE - Calling abort() to terminate application!",
                        name_);
            std::cerr << "\                    n\n";
            std::cerr << "    CRITICAL: CORE SERVICE CRASHED        \n";
            std::cerr << "    Terminating entire application        \n";
            std::cerr << "                    \n\n";

            std::abort();  // Terminate entire process
        } else {
```

```
            Logger::log(Logger::ERROR,
                        "REST/MONITORING SERVICE FAILURE - Exiting thread only (core services continue)",
                        name_);
            std::cerr << "\                    n\n";
            std::cerr << "    REST/Monitor thread exiting          \n";
            std::cerr << "    Core services still running OK       \n";
            std::cerr << "                    \n\n";

#ifdef __linux__
            pthread_exit(nullptr);  // Exit only this thread
#elif _WIN32
            ExitThread(1);  // Windows equivalent
#else
            std::this_thread::sleep_for(1s);
            throw;  // Fallback
#endif
        }
    }
};

// Thread-local storage for thread context
thread_local std::unique_ptr<ThreadContext> g_thread_context;

// ==========================================================================
// SECTION 3: Simple JSON Parser (for demonstration)
// ==========================================================================

class JsonParseException : public std::runtime_error {
public:
    explicit JsonParseException(const std::string& msg)
        : std::runtime_error("JSON Parse Error: " + msg) {}
};

class SimpleJson {
private:
    std::string data_;

public:
    explicit SimpleJson(const std::string& json_str) : data_(json_str) {
        validate();
    }

    void validate() const {
        // Simple validation
        if (data_.empty()) {
            throw JsonParseException("Empty JSON string");
        }

        // Must start with { and end with }
```

```
        size_t start = data_.find_first_not_of(" \t\n\r");
        size_t end = data_.find_last_not_of(" \t\n\r");

        if (start == std::string::npos || end == std::string::npos) {
            throw JsonParseException("Invalid JSON: whitespace only");
        }

        if (data_[start] != '{' || data_[end] != '}') {
            throw JsonParseException("Invalid JSON: must start with { and end with }");
        }

        // Count braces
        int brace_count = 0;
        for (char c : data_) {
            if (c == '{') brace_count++;
            if (c == '}') brace_count--;
            if (brace_count < 0) {
                throw JsonParseException("Invalid JSON: unmatched closing brace");
            }
        }

        if (brace_count != 0) {
            throw JsonParseException("Invalid JSON: unmatched opening brace");
        }
    }

    std::string get_field(const std::string& field_name) const {
        std::string search = "\"" + field_name + "\"";
        size_t pos = data_.find(search);

        if (pos == std::string::npos) {
            throw JsonParseException("Field not found: " + field_name);
        }

        // Find value after colon
        size_t colon = data_.find(':', pos);
        if (colon == std::string::npos) {
            throw JsonParseException("Malformed field: " + field_name);
        }

        size_t value_start = data_.find_first_not_of(" \t\n\r", colon + 1);
        size_t value_end = data_.find_first_of(",}", value_start);

        if (value_start == std::string::npos || value_end == std::string::npos) {
            throw JsonParseException("Cannot extract value for: " + field_name);
        }

        std::string value = data_.substr(value_start, value_end - value_start);

```

```
        // Remove quotes if present
        if (value.front() == '"' && value.back() == '"') {
            value = value.substr(1, value.length() - 2);
        }

        return value;
    }

    const std::string& raw() const { return data_; }
};

// ==========================================================================
// SECTION 4: Core Services (Critical - abort on exception)
// ==========================================================================

class DatabaseService {
private:
    std::atomic<bool> running_{false};
    std::atomic<int> query_count_{0};
    std::mutex mutex_;
    std::condition_variable cv_;
    std::queue<std::string> query_queue_;

public:
    void start() {
        running_ = true;

        g_thread_context = std::make_unique<ThreadContext>(
            ThreadType::CORE_SERVICE, "DatabaseService");

        Logger::log(Logger::INFO, "Database service started", "DatabaseService");

        try {
            while (running_) {
                std::unique_lock<std::mutex> lock(mutex_);
                cv_.wait_for(lock, 500ms, [this] {
                    return !query_queue_.empty() || !running_;
                });

                if (!running_) break;

                while (!query_queue_.empty()) {
                    std::string query = query_queue_.front();
                    query_queue_.pop();
                    lock.unlock();

                    execute_query(query);

                    lock.lock();
```

```
                }
            }
        }
        catch (const std::exception& e) {
            g_thread_context->handle_exception(e);
        }

        Logger::log(Logger::INFO, "Database service stopped", "DatabaseService");
    }

    void execute_query(const std::string& query) {
        query_count_++;

        // Simulate query execution
        std::this_thread::sleep_for(50ms);

        Logger::log(Logger::INFO,
                    "Executed query #" + std::to_string(query_count_.load()) + ": " + query,
                    "DatabaseService");

        // Simulate critical error in core service (triggered by special query)
        if (query.find("TRIGGER_CORE_FAILURE") != std::string::npos) {
            Logger::log(Logger::ERROR,
                        "   SIMULATING CRITICAL DATABASE CORRUPTION!",
                        "DatabaseService");
            std::this_thread::sleep_for(100ms);
            throw std::runtime_error("CRITICAL: Database corruption detected! Data integrity compromised!");
        }
    }

    void submit_query(const std::string& query) {
        std::lock_guard<std::mutex> lock(mutex_);
        query_queue_.push(query);
        cv_.notify_one();
    }

    void stop() {
        running_ = false;
        cv_.notify_all();
    }

    int get_query_count() const { return query_count_.load(); }
};

class CacheService {
private:
    std::atomic<bool> running_{false};
    std::atomic<int> cache_hits_{0};
    std::atomic<int> cache_misses_{0};

```

```
public:
    void start() {
        running_ = true;

        g_thread_context = std::make_unique<ThreadContext>(
            ThreadType::CORE_SERVICE, "CacheService");

        Logger::log(Logger::INFO, "Cache service started", "CacheService");

        try {
            while (running_) {
                std::this_thread::sleep_for(1s);

                // Simulate cache maintenance
                if (cache_hits_ + cache_misses_ > 0) {
                    int total = cache_hits_ + cache_misses_;
                    float hit_rate = (100.0f * cache_hits_) / total;

                    std::ostringstream oss;
                    oss << "Cache stats: " << cache_hits_.load() << " hits, "
                        << cache_misses_.load() << " misses (hit rate: "
                        << std::fixed << std::setprecision(1) << hit_rate << "
                            %)";

                    Logger::log(Logger::INFO, oss.str(), "CacheService");
                }
            }
        }
        catch (const std::exception& e) {
            g_thread_context->handle_exception(e);
        }

        Logger::log(Logger::INFO, "Cache service stopped", "CacheService");
    }

    bool get(const std::string& key) {
        // Simulate cache lookup
        bool hit = (std::hash<std::string>{}(key) % 3) == 0;

        if (hit) {
            cache_hits_++;
        } else {
            cache_misses_++;
        }

        return hit;
    }

    void stop() {
        running_ = false;
    }
};

//
```

```
    ==============================================================================
// SECTION 5: REST Service (Non-critical - pthread_exit on exception)
//
    ==============================================================================

class RestApiService {
private:
    std::atomic<bool> running_{false};
    std::atomic<int> request_count_{0};
    std::atomic<int> error_count_{0};
    DatabaseService& db_;
    CacheService& cache_;

public:
    RestApiService(DatabaseService& db, CacheService& cache)
        : db_(db), cache_(cache) {}

    void start() {
        running_ = true;

        g_thread_context = std::make_unique<ThreadContext>(
            ThreadType::REST_SERVICE, "RestApiService");

        Logger::log(Logger::INFO, "REST API service started on port 8080 (simulated)",
                    "RestApiService");

        try {
            simulate_rest_requests();
        }
        catch (const std::exception& e) {
            g_thread_context->handle_exception(e);
        }

        Logger::log(Logger::INFO, "REST API service stopped", "RestApiService");
    }

    void simulate_rest_requests() {
        // Simulate various REST requests
        std::vector<std::string> requests = {
            R"({"action": "get_user", "user_id": "123"})",
            R"({"action": "create_order", "product": "laptop", "quantity": 1})",
            R"({"action": "update_profile", "name": "John Doe"})",
            R"({invalid json - missing closing brace)",  // This will cause exception!
            R"({"action": "delete_item", "item_id": "456"})",
            R"()",  // Empty - will cause exception!
        };

        int request_num = 0;
```

```

        while (running_ && request_num < requests.size()) {
            std::this_thread::sleep_for(800ms);

            const std::string& request_body = requests[request_num];
            request_count_++;

            Logger::log(Logger::INFO,
                        "Received REST request #" + std::to_string(
                            request_count_.load()),
                        "RestApiService");

            try {
                handle_request(request_body);
            }
            catch (const JsonParseException& e) {
                error_count_++;

                Logger::log(Logger::ERROR,
                            "Invalid JSON in request #" + std::to_string(
                                request_count_.load()),
                            "RestApiService");

                // This will trigger pthread_exit() through ThreadContext
                throw;
            }

            request_num++;
        }

        // If we get here without exception, run indefinitely
        while (running_) {
            std::this_thread::sleep_for(1s);
        }
    }

    void handle_request(const std::string& json_body) {
        // Parse JSON (may throw JsonParseException)
        SimpleJson json(json_body);

        std::string action = json.get_field("action");

        Logger::log(Logger::INFO, "Processing action: " + action, "
            RestApiService");

        // Check cache
        if (cache_.get(action)) {
            Logger::log(Logger::INFO, "Cache HIT for action: " + action, "
                RestApiService");
        } else {
            Logger::log(Logger::INFO, "Cache MISS for action: " + action, "
                RestApiService");
            // Submit query to database
            db_.submit_query("SELECT * FROM actions WHERE action='" + action +
```

```
                "'");
        }
    }

    void stop() {
        running_ = false;
    }

    int get_request_count() const { return request_count_.load(); }
    int get_error_count() const { return error_count_.load(); }
};

// ==========================================================================
// SECTION 6: Monitoring Service (Non-critical)
// ==========================================================================

class MonitoringService {
private:
    std::atomic<bool> running_{false};
    DatabaseService& db_;
    RestApiService& rest_;

public:
    MonitoringService(DatabaseService& db, RestApiService& rest)
        : db_(db), rest_(rest) {}

    void start() {
        running_ = true;

        g_thread_context = std::make_unique<ThreadContext>(
            ThreadType::MONITORING, "MonitoringService");

        Logger::log(Logger::INFO, "Monitoring service started", "MonitoringService");

        try {
            while (running_) {
                std::this_thread::sleep_for(2s);

                std::ostringstream oss;
                oss << "System Health: DB queries=" << db_.get_query_count()
                    << ", REST requests=" << rest_.get_request_count()
                    << ", REST errors=" << rest_.get_error_count();

                Logger::log(Logger::INFO, oss.str(), "MonitoringService");
            }
        }
        catch (const std::exception& e) {
            g_thread_context->handle_exception(e);
        }
```

```

        Logger::log(Logger::INFO, "Monitoring service stopped", "MonitoringService");
    }

    void stop() {
        running_ = false;
    }
};

// ==========================================================================
// SECTION 7: Microservices Orchestrator
// ==========================================================================

class MicroservicesOrchestrator {
private:
    DatabaseService db_service_;
    CacheService cache_service_;
    RestApiService rest_service_;
    MonitoringService monitoring_service_;

    std::vector<std::thread> threads_;
    bool simulate_core_failure_;

public:
    MicroservicesOrchestrator(bool simulate_core_failure = false)
        : rest_service_(db_service_, cache_service_),
          monitoring_service_(db_service_, rest_service_),
          simulate_core_failure_(simulate_core_failure) {}

    void start() {
        std::cout << "\n";
        std::cout << "                                    \n";
        std::cout << "    Multi-Threaded Microservices Architecture \n";
        std::cout << "                                                                \n";
        std::cout << "    Core Services (abort on exception): \n";
        std::cout << "  •     DatabaseService                                    \n";
        std::cout << "  •     CacheService                                       \n";
        std::cout << "                                                                \n";
        std::cout << "    REST Services (pthread_exit on exception): \n";
        std::cout << "  •     RestApiService                                     \n";
        std::cout << "  •     MonitoringService
```

```
                                                \n";
        std::cout << "                                \n\n";

        Logger::log(Logger::INFO, "Starting microservices orchestrator", "Orchestrator");

        // Start core services first
        threads_.emplace_back(&DatabaseService::start, &db_service_);
        threads_.emplace_back(&CacheService::start, &cache_service_);

        std::this_thread::sleep_for(500ms);

        // Start REST and monitoring services
        threads_.emplace_back(&RestApiService::start, &rest_service_);
        threads_.emplace_back(&MonitoringService::start, &monitoring_service_);

        Logger::log(Logger::INFO, "All services started", "Orchestrator");

        // If simulating core failure, trigger it after a few seconds
        if (simulate_core_failure_) {
            std::this_thread::sleep_for(3s);
            Logger::log(Logger::WARNING,
                        "   Triggering core service failure in 1 second...",
                        "Orchestrator");
            std::this_thread::sleep_for(1s);
            db_service_.submit_query("TRIGGER_CORE_FAILURE - Simulated corruption");
        }
    }

    void wait_and_stop(int seconds) {
        std::this_thread::sleep_for(std::chrono::seconds(seconds));

        Logger::log(Logger::INFO, "Initiating graceful shutdown...", "Orchestrator");

        // Stop services in reverse order
        monitoring_service_.stop();
        rest_service_.stop();
        cache_service_.stop();
        db_service_.stop();

        // Wait for threads to finish
        for (auto& thread : threads_) {
            if (thread.joinable()) {
                thread.join();
            }
        }

        Logger::log(Logger::INFO, "All services stopped", "Orchestrator");
    }
};

```

```
//
    ==========================================================================
// SECTION 8: Demonstrations
//
    ==========================================================================

void demonstrate_rest_service_exception() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Demonstration 1: REST Service Exception Handling ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Scenario:\n";
    std::cout << " •  REST service will receive invalid JSON requests\n";
    std::cout << " •  JsonParseException will be thrown\n";
    std::cout << " •  REST thread will call pthread_exit() and terminate\n";
    std::cout << " •  Core services (Database, Cache) will continue running\n\
        n";

    std::cout << "Press Enter to start demonstration...\n";
    std::cin.get();

    MicroservicesOrchestrator orchestrator(false);  // No core failure
    orchestrator.start();

    // Let it run for 10 seconds
    // REST service will encounter invalid JSON around request #4
    orchestrator.wait_and_stop(10);

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "  REST service terminated (pthread_exit) but core services
        survived!\n";
    std::cout << std::string(70, '=') << "\n\n";
}

void demonstrate_core_service_exception() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Demonstration 2: CORE Service Exception Handling ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Scenario:\n";
    std::cout << " •  Core service (Database) will encounter critical error\n"
        ;
    std::cout << " •  Exception will be thrown in DatabaseService\n";
    std::cout << " •  DatabaseService thread will call abort()\n";
    std::cout << " •     ENTIRE APPLICATION WILL TERMINATE  \n\n";

    std::cout << "   WARNING: This will terminate the process with abort()!\n"
        ;
    std::cout << "Press Enter to start demonstration...\n";
    std::cin.get();

    MicroservicesOrchestrator orchestrator(true);  // Simulate core failure
```

```
    orchestrator.start();

    // Wait for core failure (will never return - abort() will be called)
    orchestrator.wait_and_stop(30);

    // This line will never be reached
    std::cout << "\n This message should NEVER appear (abort() was called)\n";
}

void demonstrate_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Best Practices for Multi-Threaded Microservices ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  THREAD CLASSIFICATION:\n";
    std::cout << "   1. Core/Critical Services:\n";
    std::cout << " •     Essential for application function\n";
    std::cout << " •     Exception → abort() entire process\n";
    std::cout << " •     Examples: Database, Message Queue, State Manager\n\n";

    std::cout << "   2. REST/External Services:\n";
    std::cout << " •     Handle external/untrusted input\n";
    std::cout << " •     Exception → pthread_exit() current thread only\n";
    std::cout << " •     Examples: HTTP servers, gRPC endpoints, WebSocket handlers\n\n";

    std::cout << "   3. Monitoring/Auxiliary Services:\n";
    std::cout << " •     Non-essential functionality\n";
    std::cout << " •     Exception → pthread_exit() current thread only\n";
    std::cout << " •     Examples: Metrics, logging, health checks\n\n";

    std::cout << "  EXCEPTION HANDLING STRATEGY:\n";
    std::cout << "   1. Always log exceptions with stack traces\n";
    std::cout << "   2. Validate all external input (JSON, XML, protobuf)\n";
    std::cout << "   3. Use thread-local context to identify thread type\n";
    std::cout << "   4. Implement graceful degradation for non-critical threads\n";
    std::cout << "   5. Use RAII for resource cleanup before pthread_exit()\n\n";

    std::cout << "  ISOLATION AND RESILIENCE:\n";
    std::cout << "   1. Run REST handlers in separate threads/thread pool\n";
    std::cout << "   2. Implement circuit breakers for failing services\n";
    std::cout << "   3. Use message queues to decouple services\n";
    std::cout << "   4. Monitor thread health and restart if needed\n";
    std::cout << "   5. Test exception paths thoroughly\n\n";

    std::cout << "   CRITICAL WARNINGS:\n";
    std::cout << "   1. pthread_exit() does NOT call destructors for local objects!\n";
    std::cout << "   2. Clean up resources manually before pthread_exit()\n";
    std::cout << "   3. Don't use pthread_exit() from main thread\n";
```

```
    std::cout << "  4. On Windows, use ExitThread() instead of pthread_exit()\n";
    std::cout << "  5. Consider std::terminate() as alternative to abort()\n";
}

// ==========================================================================
// MAIN FUNCTION
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "    Multi-Threaded Microservices with Exception Handling \n";
    std::cout << "                                                         \n";
    std::cout << "    Demonstrates different exception handling strategies: \n";
    std::cout << "  •     REST Service: pthread_exit() (thread-local termination) \n";
    std::cout << "  •     Core Service: abort() (process-wide termination) \n";
    std::cout << "                                    \n";

    std::cout << "\nSelect demonstration:\n";
    std::cout << "  1. REST Service Exception (pthread_exit - graceful)\n";
    std::cout << "  2. CORE Service Exception (abort - terminates process)\n";
    std::cout << "  3. Best Practices Guide (no execution)\n";
    std::cout << "\nEnter choice (1-3): ";

    int choice;
    std::cin >> choice;
    std::cin.ignore();  // Clear newline

    switch (choice) {
        case 1:
            demonstrate_rest_service_exception();
            demonstrate_best_practices();
            break;

        case 2:
            demonstrate_core_service_exception();
            // Will never reach here - abort() terminates process
            break;

        case 3:
            demonstrate_best_practices();
            break;

        default:
```

```
            std::cout << "\nInvalid choice. Running REST demonstration by default.\n";
            demonstrate_rest_service_exception();
            demonstrate_best_practices();
            break;
    }

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << "\nKEY TAKEAWAYS:\n";
    std::cout << "  1. REST threads handle untrusted input → pthread_exit() on error\n";
    std::cout << "  2. Core threads are critical → abort() on error\n";
    std::cout << "  3. Always log exceptions with stack traces\n";
    std::cout << "  4. Use thread-local context for thread identification\n";
    std::cout << "  5. Implement graceful degradation for non-critical services\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

45 Source Code: NVIIdiomTemplateMethod.cpp

File: `src/NVIIdiomTemplateMethod.cpp`
Repository: View on GitHub

```cpp
// NVIIdiomTemplateMethod.cpp
// Demonstrates the Non-Virtual Interface (NVI) Idiom and Template Method Pattern
// Based on Herb Sutter's "Virtuality" guidelines
//
// KEY CONCEPTS:
// 1. Public non-virtual methods define the interface (Template Method)
// 2. Private/protected virtual methods are customization points
// 3. Base class controls the algorithm flow
// 4. Derived classes customize specific steps
//
// HERB SUTTER'S GUIDELINES:
// Guideline #1: Prefer to make interfaces non-virtual (Template Method)
// Guideline #2: Prefer to make virtual functions private
// Guideline #3: Only if derived needs base implementation, make virtual protected
// Guideline #4: Destructor should be public+virtual OR protected+non-virtual
//
// WHEN TO USE NVI:
//   Need invariant algorithm with customizable steps
//   Want to enforce pre/post conditions
//   Need to guarantee certain code always runs
//   Want separation of interface from implementation
//
// WHEN TO AVOID NVI:
//   Simple interfaces with no invariants
//   Performance-critical code (tiny extra overhead)
//   When derived classes need full control
//   Pure abstract interfaces (use public virtual)

#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <chrono>
#include <fstream>
#include <stdexcept>

using namespace std::chrono_literals;

// ============================================================================
// SECTION 1: Basic NVI Idiom - Template Method Pattern
// ============================================================================

namespace basic_nvi {
```

```cpp

class DataProcessor {
public:
    // PUBLIC NON-VIRTUAL: This is the Template Method
    // Defines the algorithm structure that CANNOT be changed by derived classes
    void process() {
        std::cout << "  [DataProcessor::process] Algorithm started\n";

        // Invariant: Always validate before processing
        if (!validate()) {
            std::cout << "    Validation failed, aborting\n";
            return;
        }

        // Call customization points (private virtuals)
        load();
        transform();
        save();

        std::cout << "  [DataProcessor::process] Algorithm completed\n";
    }

    virtual ~DataProcessor() = default;

private:
    // PRIVATE VIRTUAL: Customization points that derived classes override
    // These are the "steps" of the algorithm

    virtual bool validate() {
        std::cout << "    [Base] Default validation\n";
        return true;  // Base default: always valid
    }

    virtual void load() = 0;      // Pure virtual - must override
    virtual void transform() = 0;  // Pure virtual - must override
    virtual void save() = 0;       // Pure virtual - must override
};

class CSVProcessor : public DataProcessor {
private:
    // Override private virtuals to provide CSV-specific behavior
    void load() override {
        std::cout << "    [CSV] Loading CSV file...\n";
    }

    void transform() override {
        std::cout << "    [CSV] Transforming CSV data...\n";
    }

    void save() override {
        std::cout << "    [CSV] Saving CSV results...\n";
    }
};
```

```

class JSONProcessor : public DataProcessor {
private:
    bool validate() override {
        std::cout << "    [JSON] Custom JSON validation\n";
        return true;
    }

    void load() override {
        std::cout << "    [JSON] Loading JSON file...\n";
    }

    void transform() override {
        std::cout << "    [JSON] Transforming JSON data...\n";
    }

    void save() override {
        std::cout << "    [JSON] Saving JSON results...\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1: Basic NVI Idiom - Template Method ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Processing CSV data:\n";
    CSVProcessor csv;
    csv.process();  // Client calls public non-virtual method

    std::cout << "\n2. Processing JSON data:\n";
    JSONProcessor json;
    json.process();

    std::cout << "\ n Benefits:\n";
    std::cout << " •  Base class controls algorithm flow (validation always runs)\n";
    std::cout << " •  Derived classes customize only the steps they need\n";
    std::cout << " •  Cannot accidentally skip validation or change algorithm\n";
    std::cout << " •  Clear separation: public interface vs private implementation\n";
}

} // namespace basic_nvi

// ==========================================================================
// SECTION 2: NVI with Pre/Post Conditions - Real Power
// ==========================================================================

```

```
namespace preconditions_example {

class DatabaseConnection {
    bool connected_ = false;
    int transaction_count_ = 0;

public:
    // PUBLIC NON-VIRTUAL: Template Method with pre/post conditions
    void executeQuery(const std::string& query) {
        std::cout << "  [executeQuery] Starting query execution\n";

        // PRE-CONDITION: Must be connected
        if (!connected_) {
            std::cout << "    Error: Not connected to database!\n";
            return;
        }

        std::cout << "  [Pre] Acquiring lock...\n";
        std::cout << "  [Pre] Starting transaction " << ++transaction_count_
            << "\n";

        // Call customization point
        try {
            doExecuteQuery(query);

            // POST-CONDITION: Commit transaction
            std::cout << "  [Post] Committing transaction\n";
            std::cout << "  [Post] Releasing lock\n";

        } catch (const std::exception& e) {
            // POST-CONDITION: Rollback on error
            std::cout << "  [Post] Rolling back transaction due to error\n";
            std::cout << "  [Post] Releasing lock\n";
            throw;
        }

        std::cout << "  [executeQuery] Query completed successfully\n";
    }

    void connect() {
        std::cout << "  [Connection] Connected to database\n";
        connected_ = true;
    }

    virtual ~DatabaseConnection() = default;

private:
    // PRIVATE VIRTUAL: Derived classes provide query logic
    virtual void doExecuteQuery(const std::string& query) = 0;
};

class MySQLConnection : public DatabaseConnection {
private:
    void doExecuteQuery(const std::string& query) override {
```

```
        std::cout << "    [MySQL] Executing: " << query << "\n";
        std::cout << "    [MySQL] Query executed successfully\n";
    }
};

class PostgreSQLConnection : public DatabaseConnection {
private:
    void doExecuteQuery(const std::string& query) override {
        std::cout << "    [PostgreSQL] Executing: " << query << "\n";
        if (query.find("DROP") != std::string::npos) {
            throw std::runtime_error("DROP statements not allowed!");
        }
        std::cout << "    [PostgreSQL] Query executed successfully\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 2: NVI with Pre/Post Conditions ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. MySQL query execution:\n";
    MySQLConnection mysql;
    mysql.connect();
    mysql.executeQuery("SELECT * FROM users");

    std::cout << "\n2. PostgreSQL with error handling:\n";
    PostgreSQLConnection postgres;
    postgres.connect();
    try {
        postgres.executeQuery("DROP TABLE users");
    } catch (const std::exception& e) {
        std::cout << "    Caught exception: " << e.what() << "\n";
        std::cout << "    Transaction rolled back automatically!\n";
    }

    std::cout << "\n3. Attempting query without connection:\n";
    MySQLConnection mysql2;
    mysql2.executeQuery("SELECT * FROM users");

    std::cout << "\ n NVI ensures:\n";
    std::cout << " •  Pre-conditions always checked (connection, locks)\n";
    std::cout << " •  Post-conditions always executed (commit, rollback,
        cleanup)\n";
    std::cout << " •  Derived classes cannot bypass these guarantees!\n";
}

} // namespace preconditions_example

//
    ==========================================================================
// SECTION 3: Protected Virtual - When Derived Needs Base Implementation
//
```

```
    ==========================================================================

namespace protected_virtual_example {

class Logger {
public:
    // PUBLIC NON-VIRTUAL: Template Method
    void log(const std::string& message) {
        std::cout << "  [log] Preparing log entry\n";
        addTimestamp();
        writeLog(message);
        flush();
    }

    virtual ~Logger() = default;

protected:
    // PROTECTED VIRTUAL: Derived classes may want to call base implementation
    virtual void writeLog(const std::string& message) {
        std::cout << "    [Base Logger] " << message << "\n";
    }

private:
    void addTimestamp() {
        std::cout << "    [Timestamp] 2026-01-02 10:30:45\n";
    }

    void flush() {
        std::cout << "    [Flush] Log buffer flushed\n";
    }
};

class FileLogger : public Logger {
protected:
    void writeLog(const std::string& message) override {
        // Call base implementation first
        Logger::writeLog(message);

        // Add file-specific logging
        std::cout << "    [FileLogger] Writing to file: logs.txt\n";
    }
};

class NetworkLogger : public Logger {
protected:
    void writeLog(const std::string& message) override {
        // Call base implementation
        Logger::writeLog(message);

        // Send over network
        std::cout << "    [NetworkLogger] Sending to log server 192.168.1.100\
            n";
    }
```

```
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 3: Protected Virtual (Guideline #3) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. File logger with base implementation:\n";
    FileLogger file_logger;
    file_logger.log("Application started");

    std::cout << "\n2. Network logger with base implementation:\n";
    NetworkLogger net_logger;
    net_logger.log("User logged in");

    std::cout << "\n Protected virtual when:\n";
    std::cout << " • Derived classes need to call base implementation\n";
    std::cout << " • Want to extend, not replace, base behavior\n";
    std::cout << " • Common pattern: call base, then add specific logic\n";
}

} // namespace protected_virtual_example

// ==========================================================================
// SECTION 4: When NOT to Use NVI - Alternatives
// ==========================================================================

namespace when_not_to_use_nvi {

// ANTI-PATTERN: NVI overkill for simple interfaces
class BadExample_OverengineeredNVI {
public:
    void getValue() {
        doGetValue();  // Unnecessary indirection!
    }

private:
    virtual void doGetValue() = 0;
};

// BETTER: Simple public virtual for simple interfaces
class GoodExample_SimpleInterface {
public:
    virtual int getValue() const = 0;  // Direct and clear
    virtual ~GoodExample_SimpleInterface() = default;
};

class SimpleImpl : public GoodExample_SimpleInterface {
public:
    int getValue() const override { return 42; }
```

```
};

// WHEN TO USE PUBLIC VIRTUAL: Pure abstract interfaces (like Java interfaces)
class IDrawable {
public:
    virtual void draw() = 0;
    virtual void move(int x, int y) = 0;
    virtual ~IDrawable() = default;
};

class Circle : public IDrawable {
public:
    void draw() override {
        std::cout << "    Drawing circle\n";
    }

    void move(int x, int y) override {
        std::cout << "    Moving circle to (" << x << ", " << y << ")\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 4: When NOT to Use NVI ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Simple interface - public virtual is fine:\n";
    SimpleImpl impl;
    std::cout << "    Value: " << impl.getValue() << "\n";

    std::cout << "\n2. Pure abstract interface - use public virtual:\n";
    Circle circle;
    circle.draw();
    circle.move(10, 20);

    std::cout << "\ n DON'T use NVI when:\n";
    std::cout << " •  Simple getter/setter interfaces\n";
    std::cout << " •  No invariants to enforce\n";
    std::cout << " •  No pre/post conditions needed\n";
    std::cout << " •  Pure abstract interfaces (like Java interfaces)\n";
    std::cout << " •  Performance critical (tiny overhead of extra call)\n\n";

    std::cout << "  DO use NVI when:\n";
    std::cout << " •  Need to enforce pre/post conditions\n";
    std::cout << " •  Have invariant algorithm with customizable steps\n";
    std::cout << " •  Want to guarantee certain code always runs\n";
    std::cout << " •  Need clear separation of interface from implementation\n
        ";
}

} // namespace when_not_to_use_nvi

//
    ==============================================================================
```

```cpp
// SECTION 5: Real-World Example - HTTP Request Handler
// ==========================================================================

namespace http_handler_example {

class HTTPRequestHandler {
public:
    // PUBLIC NON-VIRTUAL: Template Method for handling HTTP requests
    void handleRequest(const std::string& method, const std::string& path) {
        std::cout << "  [HTTP] Received " << method << " " << path << "\n";

        // PRE: Always authenticate
        std::cout << "  [Pre] Authenticating request...\n";
        if (!authenticate()) {
            std::cout << "    401 Unauthorized\n";
            return;
        }

        // PRE: Always validate input
        std::cout << "  [Pre] Validating input...\n";
        if (!validateInput(method, path)) {
            std::cout << "    400 Bad Request\n";
            return;
        }

        // PRE: Check rate limiting
        std::cout << "  [Pre] Checking rate limits...\n";
        if (!checkRateLimit()) {
            std::cout << "    429 Too Many Requests\n";
            return;
        }

        // Call customization point
        try {
            std::cout << "  [Processing] Handling request...\n";
            processRequest(method, path);

            // POST: Always log successful requests
            std::cout << "  [Post] Logging successful request\n";
            logAccess(method, path, 200);
            std::cout << "    200 OK\n";

        } catch (const std::exception& e) {
            // POST: Always log errors
            std::cout << "  [Post] Logging error\n";
            logAccess(method, path, 500);
            std::cout << "    500 Internal Server Error: " << e.what() << "\n";
        }
    }

```

```
    virtual ~HTTPRequestHandler() = default;

private:
    // PRIVATE VIRTUAL: Customization points
    virtual bool authenticate() {
        return true;  // Default: no auth
    }

    virtual bool validateInput([[maybe_unused]] const std::string& method,
                               [[maybe_unused]] const std::string& path) {
        return true;  // Default: always valid
    }

    virtual bool checkRateLimit() {
        return true;  // Default: no limits
    }

    virtual void processRequest(const std::string& method,
                                const std::string& path) = 0;

    virtual void logAccess(const std::string& method,
                           const std::string& path,
                           int status_code) {
        std::cout << "    [Log] " << method << " " << path
                  << " - " << status_code << "\n";
    }
};

class UserAPIHandler : public HTTPRequestHandler {
private:
    bool authenticate() override {
        std::cout << "    [Auth] Checking JWT token...\n";
        return true;  // Simplified
    }

    bool validateInput(const std::string& method,
                       const std::string& path) override {
        if (path.find("/api/users") != 0) {
            std::cout << "    [Validation] Invalid path\n";
            return false;
        }
        return true;
    }

    void processRequest(const std::string& method,
                        const std::string& path) override {
        if (method == "GET") {
            std::cout << "    [Handler] Fetching user data from database\n";
            std::cout << "    [Handler] Returning user list\n";
        } else if (method == "POST") {
            std::cout << "    [Handler] Creating new user\n";
            std::cout << "    [Handler] User created successfully\n";
        }
    }
```

```
};

class PublicAPIHandler : public HTTPRequestHandler {
private:
    int request_count_ = 0;

    bool checkRateLimit() override {
        if (++request_count_ > 3) {
            std::cout << "    [RateLimit] Exceeded rate limit!\n";
            return false;
        }
        return true;
    }

    void processRequest([[maybe_unused]] const std::string& method,
                        const std::string& path) override {
        std::cout << "    [Handler] Processing public API request: " << path
            << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 5: Real-World - HTTP Request Handler ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. User API with authentication:\n";
    UserAPIHandler user_api;
    user_api.handleRequest("GET", "/api/users/123");

    std::cout << "\n2. Public API with rate limiting:\n";
    PublicAPIHandler public_api;
    for (int i = 1; i <= 5; ++i) {
        std::cout << "\n  Request " << i << ":\n";
        public_api.handleRequest("GET", "/api/public/data");
        if (i >= 3) {
            break;  // Stop after rate limit
        }
    }

    std::cout << "\ n NVI guarantees:\n";
    std::cout << " •  Authentication always checked before processing\n";
    std::cout << " •  Input validation cannot be bypassed\n";
    std::cout << " •  Rate limiting enforced consistently\n";
    std::cout << " •  Logging always happens (success or error)\n";
    std::cout << " •  Derived classes customize logic, not security/logging!\n
        ";
}

} // namespace http_handler_example

//
    ==========================================================================
```

```
// SECTION 6: Guideline #4 - Destructor Rules
// ==========================================================================

namespace destructor_rules {

// CORRECT: Public virtual destructor for polymorphic base class
class PolymorphicBase {
public:
    virtual ~PolymorphicBase() {
        std::cout << "    [PolymorphicBase] Destructor\n";
    }

    virtual void doSomething() = 0;
};

class DerivedFromPolymorphic : public PolymorphicBase {
public:
    ~DerivedFromPolymorphic() override {
        std::cout << "    [DerivedFromPolymorphic] Destructor\n";
    }

    void doSomething() override {
        std::cout << "    [DerivedFromPolymorphic] Doing something\n";
    }
};

// CORRECT: Protected non-virtual destructor for non-polymorphic base
class NonPolymorphicBase {
protected:
    ~NonPolymorphicBase() {
        std::cout << "    [NonPolymorphicBase] Destructor\n";
    }

public:
    void doWork() {
        std::cout << "    [NonPolymorphicBase] Working\n";
    }
};

class DerivedFromNonPolymorphic : public NonPolymorphicBase {
public:
    ~DerivedFromNonPolymorphic() {
        std::cout << "    [DerivedFromNonPolymorphic] Destructor\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 6: Guideline #4 - Destructor Rules ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Polymorphic base (public virtual destructor):\n";
```

```
    {
        PolymorphicBase* ptr = new DerivedFromPolymorphic();
        ptr->doSomething();
        delete ptr;  //  Calls derived destructor first, then base
    }

    std::cout << "\n2. Non-polymorphic base (protected non-virtual destructor):\n";
    {
        DerivedFromNonPolymorphic obj;
        obj.doWork();
        // Cannot do: NonPolymorphicBase* ptr = new DerivedFromNonPolymorphic();
        // delete ptr;  // Would be undefined behavior!
        // Protected destructor prevents polymorphic deletion
    }

    std::cout << "\n Guideline #4 (Herb Sutter):\n";
    std::cout << " •  Polymorphic base class: public + virtual destructor\n";
    std::cout << " •  Non-polymorphic base class: protected + non-virtual destructor\n";
    std::cout << " •  Protected destructor prevents polymorphic deletion\n";
    std::cout << " •  Virtual destructor allows safe polymorphic deletion\n";
}

} // namespace destructor_rules

// ==========================================================================
// SECTION 7: Summary - Complete Guidelines
// ==========================================================================

void show_summary() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Herb Sutter's Virtuality Guidelines - Complete Summary ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "GUIDELINE #1: Prefer non-virtual interfaces (Template Method)\n";
    std::cout << " •  Public non-virtual methods define the interface\n";
    std::cout << " •  These are the \"Template Methods\" that orchestrate the algorithm\n";
    std::cout << " •  Benefits: Enforce invariants, pre/post conditions, logging, etc.\n\n";

    std::cout << "GUIDELINE #2: Prefer private virtual functions\n";
    std::cout << " •  Private virtual methods are customization points\n";
    std::cout << " •  Derived classes override to provide specific behavior\n";
    std::cout << " •  Cannot be called directly by client code or derived
```

```
        classes\n";
    std::cout << " •  Clearest separation of interface from implementation\n\n
        ";

    std::cout << "GUIDELINE #3: Make virtual protected if derived needs base
        impl\n";
    std::cout << " •  Use protected when derived classes need to call base
        version\n";
    std::cout << " •  Common pattern: call base, then add derived-specific
        logic\n";
    std::cout << " •  Still not part of public interface\n\n";

    std::cout << "GUIDELINE #4: Destructor rules\n";
    std::cout << " •  Polymorphic base: public AND virtual destructor\n";
    std::cout << " •  Non-polymorphic base: protected AND non-virtual
        destructor\n";
    std::cout << " •  Public virtual: allows safe polymorphic deletion\n";
    std::cout << " •  Protected non-virtual: prevents polymorphic deletion\n\n
        ";

    std::cout << std::string(70, '-') << "\n\n";

    std::cout << "WHEN TO USE NVI:\n";
    std::cout << "    Need to enforce pre/post conditions\n";
    std::cout << "    Have invariant algorithm with customizable steps\n";
    std::cout << "    Want to guarantee certain code always runs\n";
    std::cout << "    Need separation of interface from implementation\n";
    std::cout << "    Building frameworks or libraries with hooks\n\n";

    std::cout << "WHEN NOT TO USE NVI:\n";
    std::cout << "    Simple interfaces with no invariants\n";
    std::cout << "    Pure abstract interfaces (use public virtual)\n";
    std::cout << "    No pre/post conditions needed\n";
    std::cout << "    Performance-critical inner loops (tiny overhead)\n";
    std::cout << "    Simple getter/setter methods\n\n";

    std::cout << "COMPARISON:\n";
    std::cout << "  Traditional (public virtual):\n";
    std::cout << " •    Simpler for basic polymorphism\n";
    std::cout << " •    No invariant enforcement\n";
    std::cout << " •    Derived classes have full control\n\n";

    std::cout << "  NVI Idiom (public non-virtual, private virtual):\n";
    std::cout << " •    Enforces invariants and contracts\n";
    std::cout << " •    Base controls algorithm flow\n";
    std::cout << " •    Clear interface/implementation separation\n";
    std::cout << " •    Better encapsulation\n\n";

    std::cout << "REFERENCE:\n";
    std::cout << "  Herb Sutter's \"Virtuality\" article:\n";
    std::cout << "  http://www.gotw.ca/publications/mill18.htm\n";
}

//
```

```
    ==============================================================================

// MAIN FUNCTION
//
    ==============================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "   Non-Virtual Interface (NVI) Idiom & Template Method
        Pattern   \n";
    std::cout << "                Herb Sutter's Virtuality Guidelines
                      \n";
    std::cout << "                                    \n";

    // Section 1: Basic NVI
    basic_nvi::demonstrate();

    // Section 2: Pre/post conditions
    preconditions_example::demonstrate();

    // Section 3: Protected virtual
    protected_virtual_example::demonstrate();

    // Section 4: When NOT to use NVI
    when_not_to_use_nvi::demonstrate();

    // Section 5: Real-world example
    http_handler_example::demonstrate();

    // Section 6: Destructor rules
    destructor_rules::demonstrate();

    // Section 7: Complete summary
    show_summary();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

46 Source Code: NlohmannJsonExample.cpp

File: `src/NlohmannJsonExample.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// NLOHMANN JSON LIBRARY EXAMPLE
// ==================================================================
// This example demonstrates the nlohmann/json library - a modern C++
// JSON library that provides an intuitive API similar to Python/JavaScript.
//
// TOPICS COVERED:
// 1. Basic JSON parsing and serialization
// 2. Working with JSON objects and arrays
// 3. Type conversions and safety
// 4. Custom struct serialization
// 5. JSON Merge Patch (RFC 7386)
// 6. JSON Pointer (RFC 6901)
// 7. CBOR/MessagePack/BSON/UBJSON support
// 8. Performance and embedded systems considerations
//
// WHAT IS NLOHMANN JSON?
// - Header-only C++ JSON library
// - Intuitive API similar to Python dictionaries
// - No external dependencies
// - Supports C++11 and later
// - MIT licensed, widely adopted
//
// WHY USE NLOHMANN JSON?
//   Easy to use: json["key"] = value
//   Header-only: Just #include <nlohmann/json.hpp>
//   STL-like: Integrates seamlessly with C++ containers
//   Type-safe: Strong typing with automatic conversions
//   Standards-compliant: RFC 7159, RFC 6901, RFC 7386
//   Modern C++: Uses C++11/14/17/20 features
//
// INSTALLATION:
// - Ubuntu: sudo apt-get install nlohmann-json3-dev
// - vcpkg: vcpkg install nlohmann-json
// - CMake: find_package(nlohmann_json REQUIRED)
// - Single header: Download json.hpp from GitHub
//
// ==================================================================

#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <map>
#include <set>
#include <nlohmann/json.hpp>

// For convenience
using json = nlohmann::json;
```

```cpp

// ==================================================================
// EXAMPLE 1: BASIC JSON CREATION AND PARSING
// ==================================================================

void example_basic_json() {
    std::cout << "=== Example 1: Basic JSON Creation and Parsing ===\n";

    // Create JSON from initializer list (most readable)
    json j = {
        {"name", "Alice"},
        {"age", 30},
        {"city", "San Francisco"},
        {"active", true}
    };

    std::cout << "Created JSON:\n" << j.dump(2) << "\n\n";

    // Access values (Python-like syntax)
    std::cout << "Name: " << j["name"] << "\n";
    std::cout << "Age: " << j["age"] << "\n";

    // Type-safe access with .get<T>()
    std::string name = j["name"].get<std::string>();
    int age = j["age"].get<int>();
    std::cout << "Type-safe: " << name << " is " << age << " years old\n";

    // Parse from string
    std::string json_str = R"({"device": "sensor_001", "temperature": 23.5})";
    json parsed = json::parse(json_str);
    std::cout << "\nParsed JSON:\n" << parsed.dump(2) << "\n";

    // Check if key exists
    if (parsed.contains("temperature")) {
        std::cout << "Temperature: " << parsed["temperature"] << "°C\n";
    }

    std::cout << "\n Easy syntax like Python/JavaScript\n";
    std::cout << "  Automatic type conversions\n";
    std::cout << "  contains() for safe key checking\n\n";
}

// ==================================================================
// EXAMPLE 2: WORKING WITH ARRAYS
// ==================================================================

void example_arrays() {
    std::cout << "=== Example 2: JSON Arrays ===\n";

    // Create array
    json arr = json::array();
    arr.push_back(10);
    arr.push_back(20);
    arr.push_back(30);
```

```

    std::cout << "Array: " << arr.dump() << "\n";

    // Create from vector
    std::vector<int> vec = {1, 2, 3, 4, 5};
    json j_vec = vec;
    std::cout << "From vector: " << j_vec.dump() << "\n";

    // Convert back to vector
    std::vector<int> vec2 = j_vec.get<std::vector<int>>();
    std::cout << "Back to vector: size = " << vec2.size() << "\n";

    // Array of objects
    json sensors = json::array();
    for (int i = 0; i < 5; ++i) {
        sensors.push_back({
            {"id", "sensor_" + std::to_string(i)},
            {"value", 20.0 + i * 0.5},
            {"online", i % 2 == 0}
        });
    }

    std::cout << "\nSensor array:\n" << sensors.dump(2) << "\n";

    // Iterate over array
    std::cout << "\nOnline sensors:\n";
    for (const auto& sensor : sensors) {
        if (sensor["online"].get<bool>()) {
            std::cout << "  " << sensor["id"] << ": " << sensor["value"] << "\n";
        }
    }

    std::cout << "\n STL container integration\n";
    std::cout << "  Range-based for loops\n";
    std::cout << "  Array manipulation like JavaScript\n\n";
}

// ==================================================================
// EXAMPLE 3: CUSTOM STRUCT SERIALIZATION
// ==================================================================

struct SensorReading {
    std::string device_id;
    double temperature;
    double humidity;
    bool online;
};

// Define JSON serialization (method 1: macro)
NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(SensorReading, device_id, temperature, humidity, online)

// Alternative custom struct with intrusive serialization
```

```
struct GpsCoordinate {
    double latitude;
    double longitude;
    double altitude;

    // Method 2: Member functions (more control)
    NLOHMANN_DEFINE_TYPE_INTRUSIVE(GpsCoordinate, latitude, longitude,
        altitude)
};

void example_custom_types() {
    std::cout << "=== Example 3: Custom Struct Serialization ===\n";

    // Create struct
    SensorReading reading{
        "sensor_001",
        23.5,
        65.3,
        true
    };

    // Automatic conversion to JSON
    json j = reading;
    std::cout << "Struct to JSON:\n" << j.dump(2) << "\n";

    // Automatic conversion from JSON
    json j2 = R"({
        "device_id": "sensor_002",
        "temperature": 25.0,
        "humidity": 70.5,
        "online": false
    })"_json;  // User-defined literal

    SensorReading reading2 = j2.get<SensorReading>();
    std::cout << "\nJSON to struct:\n";
    std::cout << "  Device: " << reading2.device_id << "\n";
    std::cout << "  Temperature: " << reading2.temperature << "°C\n";
    std::cout << "  Online: " << (reading2.online ? "yes" : "no") << "\n";

    // Array of structs
    std::vector<SensorReading> readings = {
        {"sensor_001", 20.0, 60.0, true},
        {"sensor_002", 21.5, 62.0, true},
        {"sensor_003", 19.0, 58.0, false}
    };

    json j_array = readings;
    std::cout << "\nVector of structs to JSON:\n" << j_array.dump(2) << "\n";

    std::cout << "\n Automatic struct  JSON conversion\n";
    std::cout << "  NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE macro\n";
    std::cout << "  Works with STL containers\n\n";
}

```

```
// ==================================================================
// EXAMPLE 4: FILE I/O
// ==================================================================

void example_file_io() {
    std::cout << "=== Example 4: File I/O ===\n";

    // Create configuration JSON
    json config = {
        {"version", "1.0.0"},
        {"server", {
            {"host", "localhost"},
            {"port", 8080},
            {"ssl", true}
        }},
        {"sensors", json::array({
            {{"id", "temp_01"}, {"enabled", true}},
            {{"id", "humid_01"}, {"enabled", true}}
        })}
    };

    // Write to file
    std::ofstream file("config.json");
    file << std::setw(2) << config << std::endl;
    file.close();
    std::cout << "  Written to config.json\n";

    // Read from file
    std::ifstream input("config.json");
    json loaded_config;
    input >> loaded_config;
    input.close();

    std::cout << "  Loaded from config.json:\n";
    std::cout << "   Version: " << loaded_config["version"] << "\n";
    std::cout << "   Server port: " << loaded_config["server"]["port"] << "\n";
    std::cout << "   Sensors: " << loaded_config["sensors"].size() << "\n";

    std::cout << "\n Easy file serialization\n";
    std::cout << "  Pretty printing with setw()\n";
    std::cout << "  Stream operators << and >>\n\n";
}

// ==================================================================
// EXAMPLE 5: JSON POINTER (RFC 6901)
// ==================================================================

void example_json_pointer() {
    std::cout << "=== Example 5: JSON Pointer (RFC 6901) ===\n";

    json data = {
        {"user", {
            {"name", "Alice"},
            {"address", {
```

```
                {"city", "San Francisco"},
                {"zip", "94102"}
            }},
            {"scores", {85, 92, 78}}
        }}
    };

    // Access nested values using JSON Pointer
    std::cout << "Using JSON Pointer:\n";
    std::cout << "  /user/name: " << data["/user/name"_json_pointer] << "\n";
    std::cout << "  /user/address/city: " << data["/user/address/city"
        _json_pointer] << "\n";
    std::cout << "  /user/scores/0: " << data["/user/scores/0"_json_pointer]
        << "\n";

    // Check if pointer exists
    if (data.contains(json::json_pointer("/user/address/country"))) {
        std::cout << "  Country exists\n";
    } else {
        std::cout << "  Country does not exist\n";
    }

    // Modify using pointer
    data["/user/address/country"_json_pointer] = "USA";
    std::cout << "\n Added country: " << data["/user/address/country"
        _json_pointer] << "\n";

    std::cout << "\n Navigate nested structures easily\n";
    std::cout << "  RFC 6901 compliant\n";
    std::cout << "  Safe access with contains()\n\n";
}

// ==================================================================
// EXAMPLE 6: JSON MERGE PATCH (RFC 7386)
// ==================================================================

void example_merge_patch() {
    std::cout << "=== Example 6: JSON Merge Patch (RFC 7386) ===\n";

    json original = {
        {"name", "Alice"},
        {"age", 30},
        {"city", "San Francisco"},
        {"active", true}
    };

    std::cout << "Original:\n" << original.dump(2) << "\n";

    // Merge patch: update existing fields, add new fields
    json patch = {
        {"age", 31},            // Update
        {"city", nullptr},      // Delete
        {"country", "USA"}      // Add
    };
```

```

    original.merge_patch(patch);

    std::cout << "\nAfter merge patch:\n" << original.dump(2) << "\n";

    std::cout << "\n Update multiple fields at once\n";
    std::cout << "  null value deletes field\n";
    std::cout << "  RFC 7386 compliant\n\n";
}

// ==================================================================
// EXAMPLE 7: BINARY FORMATS (CBOR, MessagePack, BSON)
// ==================================================================

void example_binary_formats() {
    std::cout << "=== Example 7: Binary Formats ===\n";

    json data = {
        {"device", "sensor_001"},
        {"temperature", 23.5},
        {"humidity", 65.3},
        {"readings", {20, 21, 22, 23, 24}}
    };

    // JSON string
    std::string json_str = data.dump();
    std::cout << "JSON size: " << json_str.size() << " bytes\n";

    // CBOR (Concise Binary Object Representation)
    std::vector<uint8_t> cbor = json::to_cbor(data);
    std::cout << "CBOR size: " << cbor.size() << " bytes";
    std::cout << " (" << std::fixed << std::setprecision(1)
              << (1.0 - (double)cbor.size() / json_str.size()) * 100 << "% smaller)\n";

    // MessagePack
    std::vector<uint8_t> msgpack = json::to_msgpack(data);
    std::cout << "MessagePack size: " << msgpack.size() << " bytes";
    std::cout << " (" << (1.0 - (double)msgpack.size() / json_str.size()) * 100 << "% smaller)\n";

    // BSON
    std::vector<uint8_t> bson = json::to_bson(data);
    std::cout << "BSON size: " << bson.size() << " bytes";
    std::cout << " (" << (1.0 - (double)bson.size() / json_str.size()) * 100 << "% smaller)\n";

    // UBJSON (Universal Binary JSON)
    std::vector<uint8_t> ubjson = json::to_ubjson(data);
    std::cout << "UBJSON size: " << ubjson.size() << " bytes\n";

    // Deserialize CBOR
    json restored = json::from_cbor(cbor);
    std::cout << "\n Restored from CBOR:\n" << restored.dump(2) << "\n";
```

```

    std::cout << "\n Multiple binary format support\n";
    std::cout << "  30-50% smaller than JSON text\n";
    std::cout << "  Faster parsing than text JSON\n\n";
}

// ==================================================================
// EXAMPLE 8: ERROR HANDLING
// ==================================================================

void example_error_handling() {
    std::cout << "=== Example 8: Error Handling ===\n";

    // Safe parsing with exception handling
    std::string invalid_json = R"({"name": "Alice", "age": })";

    try {
        json j = json::parse(invalid_json);
    } catch (const json::parse_error& e) {
        std::cout << "Parse error caught:\n";
        std::cout << "  Message: " << e.what() << "\n";
        std::cout << "  Exception ID: " << e.id << "\n";
        std::cout << "  Byte position: " << e.byte << "\n";
    }

    // Safe access with value()
    json data = {{"name", "Alice"}, {"age", 30}};

    std::string name = data.value("name", "Unknown");
    std::string country = data.value("country", "Unknown");  // Default if
        missing

    std::cout << "\n Safe value() with default:\n";
    std::cout << "  Name: " << name << "\n";
    std::cout << "  Country: " << country << " (default)\n";

    // Type checking
    if (data["age"].is_number_integer()) {
        int age = data["age"];
        std::cout << "  Age is integer: " << age << "\n";
    }

    std::cout << "\n Exception-based error handling\n";
    std::cout << "  value() method with defaults\n";
    std::cout << "  Type checking with is_xxx()\n\n";
}

// ==================================================================
// EXAMPLE 9: ADVANCED STL CONTAINER CONVERSIONS
// ==================================================================

void example_stl_conversions() {
    std::cout << "=== Example 9: Advanced STL Container Conversions ===\n";

```

```
    // Map conversions
    std::map<std::string, int> scores = {
        {"Alice", 95},
        {"Bob", 87},
        {"Charlie", 92}
    };

    json j_map = scores;
    std::cout << "Map to JSON:\n" << j_map.dump(2) << "\n";

    auto restored_map = j_map.get<std::map<std::string, int>>();
    std::cout << "\n Restored map: " << restored_map.size() << " entries\n";

    // Nested containers
    std::vector<std::map<std::string, std::vector<int>>> complex = {
        {{"data", {1, 2, 3}}, {"scores", {90, 85, 88}}},
        {{"data", {4, 5, 6}}, {"scores", {92, 87, 91}}}
    };

    json j_complex = complex;
    std::cout << "\nNested containers:\n" << j_complex.dump(2) << "\n";

    // Set conversions
    std::set<std::string> tags = {"cpp", "json", "modern", "c++17"};
    json j_set = tags;
    std::cout << "\nSet to JSON array: " << j_set.dump() << "\n";

    // Tuple conversions
    std::tuple<std::string, int, double> data = {"sensor_001", 42, 23.5};
    json j_tuple = data;
    std::cout << "Tuple to JSON array: " << j_tuple.dump() << "\n";

    std::cout << "\n Automatic STL container serialization\n";
    std::cout << "  Works with nested containers\n";
    std::cout << "  Two-way conversion\n\n";
}

// ==================================================================
// EXAMPLE 10: SAX PARSING FOR LARGE FILES
// ==================================================================

void example_sax_parsing() {
    std::cout << "=== Example 10: SAX Parsing for Large Files ===\n";

    // SAX (Simple API for XML) style parsing - event-driven
    // Useful for large JSON files to avoid loading entire document

    struct MySaxHandler {
        bool null() {
            std::cout << "  Event: null\n";
            return true;
        }

        bool boolean(bool val) {
```

```
            std::cout << "  Event: boolean = " << std::boolalpha << val << "\n
                ";
            return true;
        }

        bool number_integer(json::number_integer_t val) {
            std::cout << "  Event: integer = " << val << "\n";
            return true;
        }

        bool number_unsigned(json::number_unsigned_t val) {
            std::cout << "  Event: unsigned = " << val << "\n";
            return true;
        }

        bool number_float(json::number_float_t val, const std::string&) {
            std::cout << "  Event: float = " << val << "\n";
            return true;
        }

        bool binary(json::binary_t& val) {
            std::cout << "  Event: binary (" << val.size() << " bytes)\n";
            return true;
        }

        bool string(std::string& val) {
            std::cout << "  Event: string = \"" << val << "\"\n";
            return true;
        }

        bool start_object(std::size_t) {
            std::cout << "  Event: start_object\n";
            return true;
        }

        bool end_object() {
            std::cout << "  Event: end_object\n";
            return true;
        }

        bool start_array(std::size_t) {
            std::cout << "  Event: start_array\n";
            return true;
        }

        bool end_array() {
            std::cout << "  Event: end_array\n";
            return true;
        }

        bool key(std::string& val) {
            std::cout << "  Event: key = \"" << val << "\"\n";
            return true;
        }
```

```

        bool parse_error(std::size_t, const std::string&, const json::exception& ex) {
            std::cout << "  Parse error: " << ex.what() << "\n";
            return false;
        }
    };

    std::cout << "\nParsing JSON with SAX events:\n";
    std::string json_str = R"({"name":"Alice","age":30,"scores":[95,87,92]})";

    MySaxHandler handler;
    bool result = json::sax_parse(json_str, &handler);

    std::cout << "\n SAX parsing " << (result ? "succeeded" : "failed") << "\n";

    std::cout << "\n Memory-efficient for large files\n";
    std::cout << "  Event-driven processing\n";
    std::cout << "  Can stop parsing early\n";
    std::cout << "  Useful for streaming data\n\n";
}

// ==================================================================
// EXAMPLE 11: JSON SCHEMA VALIDATION
// ==================================================================

void example_json_schema() {
    std::cout << "=== Example 11: JSON Schema-like Validation ===\n";

    // Note: nlohmann::json doesn't have built-in schema validation
    // But we can implement basic validation checks

    json schema = {
        {"type", "object"},
        {"required", {"name", "age", "email"}},
        {"properties", {
            {"name", {{"type", "string"}}},
            {"age", {{"type", "number"}, {"minimum", 0}, {"maximum", 150}}},
            {"email", {{"type", "string"}}}
        }}
    };

    auto validate = [](const json& data, const json& schema) -> bool {
        // Simple validation example
        if (schema["type"] == "object") {
            // Check required fields
            for (const auto& field : schema["required"]) {
                if (!data.contains(field.get<std::string>())) {
                    std::cout << "    Missing required field: " << field << "\n";
                    return false;
                }
            }
```

```

            // Check types
            for (const auto& [key, prop_schema] : schema["properties"].items() ) {
                if (data.contains(key)) {
                    std::string expected_type = prop_schema["type"];
                    const auto& value = data[key];

                    if (expected_type == "string" && !value.is_string()) {
                        std::cout << "    Field '" << key << "' should be string\n";
                        return false;
                    }
                    if (expected_type == "number" && !value.is_number()) {
                        std::cout << "    Field '" << key << "' should be number\n";
                        return false;
                    }

                    // Check number constraints
                    if (value.is_number() && prop_schema.contains("minimum")) {
                        if (value.get<double>() < prop_schema["minimum"].get<double>()) {
                            std::cout << "    Field '" << key << "' below minimum\n";
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    };

    json valid_data = {
        {"name", "Alice"},
        {"age", 30},
        {"email", "alice@example.com"}
    };

    json invalid_data = {
        {"name", "Bob"},
        {"age", "thirty"}  // Wrong type
    };

    std::cout << "\nValidating correct data:\n";
    if (validate(valid_data, schema)) {
        std::cout << "    Validation passed\n";
    }

    std::cout << "\nValidating incorrect data:\n";
    validate(invalid_data, schema);

```

```
    std::cout << "\n For full JSON Schema support, use:\n";
    std::cout << " •   nlohmann/json-schema-validator library\n";
    std::cout << " •   valijson library\n\n";
}

// ==================================================================
// EXAMPLE 12: CUSTOM ALLOCATORS FOR EMBEDDED SYSTEMS
// ==================================================================

void example_custom_allocator() {
    std::cout << "=== Example 12: Custom Allocators for Embedded ===\n";

    // nlohmann::json supports custom allocators via template parameter
    // Useful for embedded systems with custom memory management

    std::cout << "\nDefault allocator usage:\n";
    json j = {{"sensor", "temp_01"}, {"value", 23.5}};
    std::cout << "  Created JSON: " << j.dump() << "\n";

    // For embedded systems, you can use:
    // 1. Compile with -DJSON_NOEXCEPTION (no exceptions)
    // 2. Use custom allocator with basic_json template
    // 3. Disable certain features to reduce code size

    std::cout << "\n Embedded system optimizations:\n";
    std::cout << "   1. Compile flags:\n";
    std::cout << " •      -DJSON_NOEXCEPTION - Disable exceptions\n";
    std::cout << " •      -DJSON_NO_IO - Disable file I/O\n";
    std::cout << " •      -DJSON_DIAGNOSTICS=0 - Smaller binary\n";
    std::cout << "   2. Use binary formats (CBOR/MessagePack)\n";
    std::cout << "   3. Use SAX parsing for large data\n";
    std::cout << "   4. Pre-allocate with reserve()\n";
    std::cout << "   5. Consider alternatives like ArduinoJson\n\n";

    std::cout << "Memory footprint:\n";
    std::cout << " •  Header-only: ~20KB code size\n";
    std::cout << " •  Runtime: depends on JSON size\n";
    std::cout << " •  Stack usage: minimal (recursion depth)\n\n";
}

// ==================================================================
// EXAMPLE 13: PERFORMANCE TIPS
// ==================================================================

void example_performance() {
    std::cout << "=== Example 13: Performance Tips ===\n";

    // Tip 1: Use references to avoid copies
    json large_data = {
        {"sensors", json::array()}
    };

    for (int i = 0; i < 100; ++i) {
        large_data["sensors"].push_back({
```

```
            {"id", i},
            {"value", 20.0 + i * 0.1}
        });
    }

    // BAD: Creates copies
    // for (auto sensor : large_data["sensors"]) { ... }

    // GOOD: Use const reference
    int count = 0;
    for (const auto& sensor : large_data["sensors"]) {
        if (sensor["value"].get<double>() > 25.0) {
            ++count;
        }
    }
    std::cout << "Found " << count << " sensors > 25.0 (using const ref)\n";

    // Tip 2: Reserve array capacity
    json arr = json::array();
    // arr.get_ref<json::array_t&>().reserve(1000);  // Pre-allocate

    // Tip 3: Use binary formats for network/storage
    std::vector<uint8_t> cbor = json::to_cbor(large_data);
    std::cout << "Binary format: " << cbor.size() << " bytes\n";

    // Tip 4: Disable exceptions for embedded systems
    // Compile with -DJSON_NOEXCEPTION

    std::cout << "\nPerformance tips:\n";
    std::cout << "    Use const auto& in loops\n";
    std::cout << "    Reserve array capacity\n";
    std::cout << "    Use binary formats for I/O\n";
    std::cout << "    Compile with -DJSON_NOEXCEPTION for embedded\n\n";
}

// ==================================================================
// EXAMPLE 14: COMPARISON WITH ALTERNATIVES
// ==================================================================

void comparison_with_alternatives() {
    std::cout << "=== Comparison with Alternatives ===\n\n";

    std::cout << "RapidJSON:\n";
    std::cout << "    Faster parsing (2-3x)\n";
    std::cout << "    Lower memory usage\n";
    std::cout << "    More complex API\n";
    std::cout << "    Manual memory management\n\n";

    std::cout << "jsoncpp:\n";
    std::cout << "    Mature and stable\n";
    std::cout << "    Older API design\n";
    std::cout << "    Slower than nlohmann\n";
    std::cout << "    Requires compilation\n\n";

```

```
    std::cout << "simdjson:\n";
    std::cout << "    Extremely fast (SIMD optimized)\n";
    std::cout << "    Read-only parsing\n";
    std::cout << "    No JSON creation\n";
    std::cout << "    Different API paradigm\n\n";

    std::cout << "Protocol Buffers:\n";
    std::cout << "    Smaller binary format\n";
    std::cout << "    Faster parsing\n";
    std::cout << "    Requires schema\n";
    std::cout << "    Code generation needed\n\n";

    std::cout << "When to use nlohmann/json:\n";
    std::cout << "    Need easy-to-use API\n";
    std::cout << "    Header-only library preferred\n";
    std::cout << "    Configuration files\n";
    std::cout << "    REST API communication\n";
    std::cout << "    Rapid prototyping\n";
    std::cout << "    Not for extreme performance (use RapidJSON/simdjson)\n\n";
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "==========================================================\n";
    std::cout << "NLOHMANN JSON LIBRARY EXAMPLES\n";
    std::cout << "==========================================================\n\n";

    example_basic_json();
    example_arrays();
    example_custom_types();
    example_file_io();
    example_json_pointer();
    example_merge_patch();
    example_binary_formats();
    example_error_handling();
    example_stl_conversions();
    example_sax_parsing();
    example_json_schema();
    example_custom_allocator();
    example_performance();
    comparison_with_alternatives();

    std::cout << "==========================================================\n";
    std::cout << "INSTALLATION:\n";
    std::cout << "==========================================================\n\n";

```

```
    std::cout << "Ubuntu/Debian:\n";
    std::cout << "  sudo apt-get install nlohmann-json3-dev\n\n";

    std::cout << "vcpkg:\n";
    std::cout << "  vcpkg install nlohmann-json\n\n";

    std::cout << "CMake:\n";
    std::cout << "  find_package(nlohmann_json REQUIRED)\n";
    std::cout << "  target_link_libraries(YourTarget nlohmann_json::
        nlohmann_json)\n\n";

    std::cout << "Single header:\n";
    std::cout << "  Download: https://github.com/nlohmann/json/releases\n";
    std::cout << "  Just #include \"json.hpp\"\n\n";

    std::cout << "==========================================================\n"
        ;
    std::cout << "RESOURCES:\n";
    std::cout << "==========================================================\n"
        ;
    std::cout << "GitHub: https://github.com/nlohmann/json\n";
    std::cout << "Documentation: https://json.nlohmann.me/\n";
    std::cout << "API Reference: https://json.nlohmann.me/api/basic_json/\n\n"
        ;

    return 0;
}
```

47 Source Code: NoexceptBestPractices.cpp

File: `src/NoexceptBestPractices.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// NOEXCEPT BEST PRACTICES IN MODERN C++
// ==================================================================
// Comprehensive guide to when and when NOT to use noexcept
//
// Topics Covered:
// 1. What is noexcept and why it matters
// 2. When to ALWAYS use noexcept
// 3. When to NEVER use noexcept
// 4. Conditional noexcept
// 5. Performance implications
// 6. std::vector optimization with noexcept
// 7. Exception safety guarantees
// 8. Best practices and guidelines
//
// Build: g++ -std=c++20 -Wall -Wextra -O2 -o NoexceptBestPractices
    NoexceptBestPractices.cpp
// ==================================================================

#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <utility>
#include <type_traits>
#include <chrono>
#include <cmath>  // for std::log2
#include <algorithm>  // for std::copy_n

// ==================================================================
// SECTION 1: WHAT IS NOEXCEPT?
// ==================================================================

namespace what_is_noexcept {

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 1: WHAT IS NOEXCEPT?\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DEFINITION:\n";
    std::cout << "    noexcept is a specifier that tells the compiler:\n";
    std::cout << "    'This function will NEVER throw an exception'\n\n";

    std::cout << "  WHAT HAPPENS IF YOU LIE?\n";
    std::cout << "    If a noexcept function throws:\n";
    std::cout << "    1. std::terminate() is called immediately\n";
    std::cout << "    2. Stack unwinding does NOT happen\n";
    std::cout << "    3. Destructors are NOT called\n";
```

```
    std::cout << "   4. Program terminates (no recovery possible)\n\n";

    std::cout << "  WHY USE NOEXCEPT?\n";
    std::cout << "   1. Performance: Compiler can optimize more aggressively\n";
    std::cout << "   2. std::vector uses move only if noexcept\n";
    std::cout << "   3. Documents that function is exception-safe\n";
    std::cout << "   4. Enables certain optimizations at call sites\n";
}

} // namespace what_is_noexcept

// ==================================================================
// SECTION 2: WHEN TO ALWAYS USE NOEXCEPT
// ==================================================================

namespace always_use_noexcept {

// RULE 1: DESTRUCTORS - Always noexcept (implicit in C++)
class Resource {
private:
    int* data_;

public:
    Resource() : data_(new int(42)) {
        std::cout << "   [Resource] Constructor\n";
    }

    // Destructors are implicitly noexcept
    ~Resource() noexcept {  // Explicit is good for documentation
        delete data_;
        std::cout << "   [Resource] Destructor (noexcept)\n";
    }
};

// RULE 2: MOVE CONSTRUCTORS AND MOVE ASSIGNMENT
// Critical for std::vector optimization!
class MoveableResource {
private:
    int* data_;
    size_t size_;

public:
    explicit MoveableResource(size_t size)
        : data_(new int[size]), size_(size) {
        std::cout << "   [MoveableResource] Constructor\n";
    }

    ~MoveableResource() noexcept {
        delete[] data_;
    }

    //  Move constructor - ALWAYS mark noexcept
    MoveableResource(MoveableResource&& other) noexcept
```

```
        : data_(other.data_), size_(other.size_) {
        other.data_ = nullptr;
        other.size_ = 0;
        std::cout << "   [MoveableResource] Move constructor (noexcept)\n";
    }

    //  Move assignment - ALWAYS mark noexcept
    MoveableResource& operator=(MoveableResource&& other) noexcept {
        if (this != &other) {
            delete[] data_;
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
        }
        std::cout << "   [MoveableResource] Move assignment (noexcept)\n";
        return *this;
    }

    // Copy operations (not noexcept - can throw on allocation)
    MoveableResource(const MoveableResource& other)
        : data_(new int[other.size_]), size_(other.size_) {
        std::copy_n(other.data_, size_, data_);
        std::cout << "   [MoveableResource] Copy constructor (can throw)\n";
    }

    MoveableResource& operator=(const MoveableResource& other) {
        if (this != &other) {
            delete[] data_;
            size_ = other.size_;
            data_ = new int[size_];  // Can throw!
            std::copy_n(other.data_, size_, data_);
        }
        std::cout << "   [MoveableResource] Copy assignment (can throw)\n";
        return *this;
    }
};

// RULE 3: SWAP FUNCTIONS - Always noexcept
class Swappable {
private:
    int value_;

public:
    explicit Swappable(int val) : value_(val) {}

    //  Swap should always be noexcept
    friend void swap(Swappable& a, Swappable& b) noexcept {
        using std::swap;
        swap(a.value_, b.value_);
        std::cout << "   [Swappable] Swapped (noexcept)\n";
    }
};

```

```
// RULE 4: Simple getters and setters (no allocation, no complex logic)
class DataHolder {
private:
    int value_;

public:
    //  Simple getter - noexcept
    int getValue() const noexcept {
        return value_;
    }

    //  Simple setter - noexcept
    void setValue(int val) noexcept {
        value_ = val;
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 2: WHEN TO ALWAYS USE NOEXCEPT\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  ALWAYS USE NOEXCEPT FOR:\n\n";

    std::cout << " 1 DESTRUCTORS (implicit noexcept):\n";
    {
        Resource r;
    }
    std::cout << "\n";

    std::cout << " 2 MOVE OPERATIONS (critical for std::vector):\n";
    MoveableResource mr1(10);
    MoveableResource mr2 = std::move(mr1);
    std::cout << "\n";

    std::cout << " 3 SWAP FUNCTIONS:\n";
    Swappable s1(10), s2(20);
    swap(s1, s2);
    std::cout << "\n";

    std::cout << " 4 SIMPLE GETTERS/SETTERS (no allocation):\n";
    DataHolder dh;
    dh.setValue(42);
    std::cout << "   getValue() = " << dh.getValue() << " (noexcept)\n\n";

    std::cout << "  WHY THESE MUST BE NOEXCEPT:\n";
    std::cout << " •   Destructors: Can't handle exceptions during cleanup\n";
    std::cout << " •   Move ops: std::vector falls back to copy if not
        noexcept\n";
    std::cout << " •   Swap: Used in exception-safe code patterns\n";
    std::cout << " •   Simple ops: No reason to throw, performance benefit\n";
}

} // namespace always_use_noexcept
```

```

// ==================================================================
// SECTION 3: WHEN TO NEVER USE NOEXCEPT
// ==================================================================

namespace never_use_noexcept {

//   NEVER mark noexcept if function can throw!

// DON'T use noexcept: Functions that allocate memory
class Container {
private:
    std::vector<int> data_;

public:
    //   DON'T mark noexcept - can throw std::bad_alloc
    void addElement(int value) {  // No noexcept!
        data_.push_back(value);  // Can throw
        std::cout << "   [Container] Added element (can throw)\n";
    }

    //   DON'T mark noexcept - can throw on allocation
    Container(size_t size) : data_(size) {  // No noexcept!
        std::cout << "   [Container] Constructor (can throw)\n";
    }
};

// DON'T use noexcept: Functions that perform I/O
class FileHandler {
public:
    //   DON'T mark noexcept - I/O can fail
    void writeToFile(const std::string& filename, const std::string& data) {
        std::cout << "   [FileHandler] Writing to file (can throw)\n";
        // File operations can throw
        // if (!success) throw std::runtime_error("Write failed");
    }

    //   DON'T mark noexcept - I/O can fail
    std::string readFromFile(const std::string& filename) {
        std::cout << "   [FileHandler] Reading from file (can throw)\n";
        // File operations can throw
        return "data";
    }
};

// DON'T use noexcept: Functions that validate input
class Validator {
public:
    //   DON'T mark noexcept - validation can throw
    void validateAge(int age) {
        if (age < 0 || age > 150) {
            throw std::invalid_argument("Invalid age");
        }
        std::cout << "   [Validator] Age validated (can throw)\n";
```

```
    }

    //  DON'T mark noexcept - validation can throw
    void checkNonEmpty(const std::string& str) {
        if (str.empty()) {
            throw std::invalid_argument("String cannot be empty");
        }
        std::cout << "   [Validator] String validated (can throw)\n";
    }
};

// DON'T use noexcept: Functions calling other non-noexcept functions
class Processor {
private:
    Container container_;

public:
    Processor() : container_(10) {}

    //  DON'T mark noexcept - calls non-noexcept function
    void process(int value) {  // No noexcept!
        container_.addElement(value);  // Can throw
        std::cout << "   [Processor] Processed (can throw)\n";
    }
};

// DON'T use noexcept: Functions that throw std::runtime_error
class RuntimeErrorExample {
public:
    //  DANGEROUS: Marked noexcept but throws std::runtime_error
    // void dangerousFunction() noexcept {
    //     throw std::runtime_error("This causes std::terminate()!");
    //     // If this runs, program terminates immediately!
    // }

    //  CORRECT: NOT marked noexcept, can throw
    void safeFunction(bool shouldFail) {
        if (shouldFail) {
            throw std::runtime_error("Safe to throw - not noexcept");
        }
        std::cout << "   [RuntimeErrorExample] Executed successfully\n";
    }

    //  WRONG: Claims noexcept but allocates (can throw bad_alloc)
    // std::vector<int> wrongNoexcept(size_t size) noexcept {
    //     return std::vector<int>(size);  // Can throw! Causes terminate!
    // }

    //  CORRECT: Not marked noexcept
    std::vector<int> correctVersion(size_t size) {
        std::cout << "   [RuntimeErrorExample] Creating vector (can throw
            bad_alloc)\n";
        return std::vector<int>(size);
    }
```

```
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 3: WHEN TO NEVER USE NOEXCEPT\n";
    std::cout << " 5 FUNCTIONS THAT THROW std::runtime_error:\n";
    RuntimeErrorExample rte;
    try {
        rte.safeFunction(false);
        std::cout << "   Success case handled\n";
    } catch (const std::runtime_error& e) {
        std::cout << "   Caught: " << e.what() << "\n";
    }

    auto vec = rte.correctVersion(10);
    std::cout << "   Vector created with size: " << vec.size() << "\n\n";

    std::cout << "  CRITICAL: If safeFunction() was marked noexcept:\n";
    std::cout << " •   Throwing std::runtime_error would call std::terminate()
        \n";
    std::cout << " •   No catch block would execute\n";
    std::cout << " •   No stack unwinding, no destructors called\n";
    std::cout << " •   Program crashes immediately\n\n";

    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  NEVER USE NOEXCEPT FOR:\n\n";

    std::cout << " 1 FUNCTIONS THAT ALLOCATE MEMORY:\n";
    Container c(10);
    c.addElement(42);
    std::cout << "   Allocation can throw std::bad_alloc\n\n";

    std::cout << " 2 FUNCTIONS THAT PERFORM I/O:\n";
    FileHandler fh;
    fh.writeToFile("test.txt", "data");
    std::cout << "   I/O operations can fail and throw\n\n";

    std::cout << " 3 FUNCTIONS THAT VALIDATE INPUT:\n";
    Validator v;
    try {
        v.checkNonEmpty("hello");
        std::cout << "   Validation passed\n";
    } catch (...) {
        std::cout << "   Validation can throw\n";
    }
    std::cout << "\n";

    std::cout << " 4 FUNCTIONS CALLING NON-NOEXCEPT FUNCTIONS:\n";
    Processor p;
    p.process(100);
    std::cout << "   Chain of calls inherits throwing behavior\n\n";

    std::cout << "  WHY NEVER MARK THESE NOEXCEPT:\n";
```

```
    std::cout << " •   Lying about noexcept causes std::terminate()\n";
    std::cout << " •   No recovery possible if exception thrown\n";
    std::cout << " •   Better to propagate exception to caller\n";
    std::cout << " •   Allows proper error handling up the stack\n";
}

} // namespace never_use_noexcept

// ==================================================================
// SECTION 4: CONDITIONAL NOEXCEPT
// ==================================================================

namespace conditional_noexcept {

// Use conditional noexcept for template operations
template<typename T>
class Wrapper {
private:
    T value_;

public:
    explicit Wrapper(const T& val) : value_(val) {}

    // Conditional noexcept based on T's move constructor
    Wrapper(Wrapper&& other) noexcept(std::is_nothrow_move_constructible_v<T>)
        : value_(std::move(other.value_)) {
        std::cout << "   [Wrapper] Move constructor (conditionally noexcept)\n";
    }

    // Conditional noexcept based on T's swap
    void swap(Wrapper& other) noexcept(std::is_nothrow_swappable_v<T>) {
        using std::swap;
        swap(value_, other.value_);
        std::cout << "   [Wrapper] Swap (conditionally noexcept)\n";
    }
};

// Example: std::pair uses conditional noexcept
template<typename T1, typename T2>
class MyPair {
private:
    T1 first_;
    T2 second_;

public:
    MyPair(const T1& f, const T2& s) : first_(f), second_(s) {}

    // noexcept only if both T1 and T2 have noexcept move constructors
    MyPair(MyPair&& other) noexcept(
        std::is_nothrow_move_constructible_v<T1> &&
        std::is_nothrow_move_constructible_v<T2>)
        : first_(std::move(other.first_))
        , second_(std::move(other.second_)) {
```

```
        std::cout << "   [MyPair] Move (conditionally noexcept)\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 4: CONDITIONAL NOEXCEPT\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  CONDITIONAL NOEXCEPT:\n";
    std::cout << "   Use noexcept(condition) for templates\n";
    std::cout << "   noexcept status depends on template parameter\n\n";

    std::cout << "  EXAMPLE 1: Wrapper with int (noexcept move):\n";
    Wrapper<int> w1(42);
    Wrapper<int> w2(std::move(w1));
    std::cout << "   int move is noexcept: "
              << std::is_nothrow_move_constructible_v<int> << "\n\n";

    std::cout << "  EXAMPLE 2: Wrapper with std::string (noexcept move):\n";
    Wrapper<std::string> w3("hello");
    Wrapper<std::string> w4(std::move(w3));
    std::cout << "   string move is noexcept: "
              << std::is_nothrow_move_constructible_v<std::string> << "\n\n";

    std::cout << "  EXAMPLE 3: MyPair with noexcept types:\n";
    MyPair<int, int> p1(10, 20);
    MyPair<int, int> p2(std::move(p1));
    std::cout << "\n";

    std::cout << "  WHEN TO USE CONDITIONAL NOEXCEPT:\n";
    std::cout << " •   Template classes wrapping user types\n";
    std::cout << " •   Want to preserve noexcept guarantee when possible\n";
    std::cout << " •   Standard library containers use this extensively\n";
}

} // namespace conditional_noexcept

// ==================================================================
// SECTION 5: STD::VECTOR OPTIMIZATION WITH NOEXCEPT
// ==================================================================

namespace vector_optimization {

// Class WITHOUT noexcept move constructor
class WithoutNoexcept {
private:
    int* data_;

public:
    explicit WithoutNoexcept(int val = 0) : data_(new int(val)) {
        // std::cout << "   [WithoutNoexcept] Constructor\n";
    }

```

```
    ~WithoutNoexcept() {
        delete data_;
    }

    //  Move constructor WITHOUT noexcept
    WithoutNoexcept(WithoutNoexcept&& other)
        : data_(other.data_) {
        other.data_ = nullptr;
        std::cout << "   [WithoutNoexcept] MOVE constructor (not noexcept)\n";
    }

    // Copy constructor
    WithoutNoexcept(const WithoutNoexcept& other)
        : data_(new int(*other.data_)) {
        std::cout << "   [WithoutNoexcept] COPY constructor\n";
    }

    WithoutNoexcept& operator=(WithoutNoexcept&&) = default;
    WithoutNoexcept& operator=(const WithoutNoexcept&) = default;
};

// Class WITH noexcept move constructor
class WithNoexcept {
private:
    int* data_;

public:
    explicit WithNoexcept(int val = 0) : data_(new int(val)) {
        // std::cout << "   [WithNoexcept] Constructor\n";
    }

    ~WithNoexcept() {
        delete data_;
    }

    //  Move constructor WITH noexcept
    WithNoexcept(WithNoexcept&& other) noexcept
        : data_(other.data_) {
        other.data_ = nullptr;
        std::cout << "   [WithNoexcept] MOVE constructor (noexcept)\n";
    }

    // Copy constructor
    WithNoexcept(const WithNoexcept& other)
        : data_(new int(*other.data_)) {
        std::cout << "   [WithNoexcept] COPY constructor\n";
    }

    WithNoexcept& operator=(WithNoexcept&&) noexcept = default;
    WithNoexcept& operator=(const WithNoexcept&) = default;
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
```

```
    std::cout << "SECTION 5: STD::VECTOR OPTIMIZATION WITH NOEXCEPT\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  THE CRITICAL DIFFERENCE:\n";
    std::cout << "    std::vector uses MOVE only if noexcept\n";
    std::cout << "    Otherwise, it uses COPY for exception safety\n\n";

    std::cout << "  WITHOUT noexcept move (vector uses COPY on resize):\n";
    {
        std::vector<WithoutNoexcept> vec;
        vec.reserve(2);
        vec.emplace_back(1);
        vec.emplace_back(2);

        std::cout << "   Resizing vector (triggers reallocation):\n";
        vec.emplace_back(3);  // Triggers resize - uses COPY!
    }
    std::cout << "\n";

    std::cout << "  WITH noexcept move (vector uses MOVE on resize):\n";
    {
        std::vector<WithNoexcept> vec;
        vec.reserve(2);
        vec.emplace_back(1);
        vec.emplace_back(2);

        std::cout << "   Resizing vector (triggers reallocation):\n";
        vec.emplace_back(3);  // Triggers resize - uses MOVE!
    }
    std::cout << "\n";

    std::cout << "  PERFORMANCE IMPACT:\n";
    std::cout << "    Without noexcept: O(n) COPY during resize\n";
    std::cout << "    With noexcept:    O(n) MOVE during resize (much faster!)\
        n";
    std::cout << "    For 1000 objects: 1000x COPY vs 1000x MOVE\n";
    std::cout << "    Speedup can be 10x-100x+ depending on object size!\n";
}

} // namespace vector_optimization

// ==================================================================
// SECTION 6: PERFORMANCE COMPARISON
// ==================================================================

namespace performance_comparison {

class HeavyObject {
private:
    std::vector<int> data_;

public:
    explicit HeavyObject(size_t size = 1000) : data_(size, 42) {}

```

```
    // Copy constructor (expensive)
    HeavyObject(const HeavyObject& other) : data_(other.data_) {}

    // Move constructor (cheap) - WITH noexcept
    HeavyObject(HeavyObject&& other) noexcept : data_(std::move(other.data_))
        {}

    HeavyObject& operator=(const HeavyObject&) = default;
    HeavyObject& operator=(HeavyObject&&) noexcept = default;
};

void benchmark() {
    const size_t num_objects = 10000;

    auto start = std::chrono::high_resolution_clock::now();

    std::vector<HeavyObject> vec;
    for (size_t i = 0; i < num_objects; ++i) {
        vec.emplace_back(1000);
    }

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end
        - start);

    std::cout << "   Created " << num_objects << " objects\n";
    std::cout << "   Total resizes performed: ~"
              << static_cast<int>(std::log2(num_objects)) << "\n";
    std::cout << "   Time: " << duration.count() << " ms\n";
    std::cout << "   Because move is noexcept, vector uses MOVE on resize\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 6: PERFORMANCE COMPARISON\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  BENCHMARK: Vector growth with noexcept move\n";
    benchmark();
    std::cout << "\n";

    std::cout << "  WITHOUT NOEXCEPT:\n";
    std::cout << "   Same test would use COPY instead of MOVE\n";
    std::cout << "   Could be 10x-100x slower!\n";
}

} // namespace performance_comparison

// ==================================================================
// SECTION 7: BEST PRACTICES SUMMARY
// ==================================================================

namespace best_practices {

```

```
void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SECTION 7: BEST PRACTICES SUMMARY\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  ALWAYS MARK NOEXCEPT:\n";
    std::cout << "    1. Destructors (implicit, but explicit is clearer)\n";
    std::cout << "    2. Move constructors\n";
    std::cout << "    3. Move assignment operators\n";
    std::cout << "    4. Swap functions\n";
    std::cout << "    5. Simple getters/setters (no allocation)\n";
    std::cout << "    6. Default constructors (if they don't allocate)\n\n";

    std::cout << "  NEVER MARK NOEXCEPT:\n";
    std::cout << "    1. Functions that allocate memory\n";
    std::cout << "    2. Functions that perform I/O\n";
    std::cout << "    3. Functions that validate/throw on bad input\n";
    std::cout << "    4. Copy constructors/assignment (allocation can fail)\n";
    std::cout << "    5. Functions calling non-noexcept functions\n\n";

    std::cout << "  USE CONDITIONAL NOEXCEPT:\n";
    std::cout << "    1. Template functions wrapping user types\n";
    std::cout << "    2. Operations depending on template parameter traits\n";
    std::cout << "    3. When you want to preserve noexcept when possible\n\n";

    std::cout << "   CRITICAL WARNINGS:\n";
    std::cout << " •   Lying about noexcept = std::terminate() (instant crash)\n";
    std::cout << " •   No recovery possible, no stack unwinding\n";
    std::cout << " •   When in doubt, DON'T use noexcept\n";
    std::cout << " •   Better to allow exception than cause termination\n\n";

    std::cout << "  THE GOLDEN RULE:\n";
    std::cout << "    'Use noexcept ONLY when you're 100% certain\n";
    std::cout << "     the function will NEVER throw under ANY circumstances'\n\n";

    std::cout << "  PERFORMANCE BENEFITS:\n";
    std::cout << " •   std::vector uses move instead of copy (10x-100x faster)\n";
    std::cout << " •   Compiler can optimize more aggressively\n";
    std::cout << " •   No exception handling overhead\n";
    std::cout << " •   Better code generation at call sites\n";
}

} // namespace best_practices

// ==================================================================
// MAIN - Demonstrate All Sections
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
```

```
    std::cout << "                NOEXCEPT BEST PRACTICES IN MODERN C++
                        \n";
    std::cout << "            When to Use and When NOT to Use noexcept
                         \n";
    std::cout << "                                  \n";

    try {
        what_is_noexcept::demonstrate();
        always_use_noexcept::demonstrate();
        never_use_noexcept::demonstrate();
        conditional_noexcept::demonstrate();
        vector_optimization::demonstrate();
        performance_comparison::demonstrate();
        best_practices::demonstrate();

        std::cout << "\                                n\n";
        std::cout << "            ALL NOEXCEPT CONCEPTS DEMONSTRATED!
                              \n";
        std::cout << "                                \n\n";

    } catch (const std::exception& e) {
        std::cerr << "  Error: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

48 Source Code: ObjectSlicingCpp20.cpp

File: `src/ObjectSlicingCpp20.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <memory>
#include <vector>
#include <string>
#include <span>
#include <ranges>
#include <concepts>

// ==================================================================
// PREVENTING OBJECT SLICING WITH C++20 FEATURES
// ==================================================================

// Base class
class Shape {
protected:
    std::string name;

public:
    Shape(const std::string& n) : name(n) {}
    virtual ~Shape() = default;

    virtual void draw() const {
        std::cout << "Drawing Shape: " << name << std::endl;
    }

    virtual double area() const { return 0.0; }

    virtual std::string type() const { return "Shape"; }
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(const std::string& n, double r) : Shape(n), radius(r) {}

    void draw() const override {
        std::cout << "Drawing Circle: " << name << " (radius=" << radius << ")
            " << std::endl;
    }

    double area() const override {
        return 3.14159 * radius * radius;
    }

    std::string type() const override { return "Circle"; }
};

```

```
class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(const std::string& n, double w, double h)
        : Shape(n), width(w), height(h) {}

    void draw() const override {
        std::cout << "Drawing Rectangle: " << name << " (" << width << "x" << height << ")" << std::endl;
    }

    double area() const override {
        return width * height;
    }

    std::string type() const override { return "Rectangle"; }
};

// ==================================================================
// 1. C++20 CONCEPTS - ENFORCE POINTER TYPES AT COMPILE TIME
// ==================================================================

// Concept: Must be a pointer or smart pointer to a type derived from Shape
template<typename T>
concept ShapePointer = requires(T t) {
    { *t };  // Must be dereferenceable
    requires std::derived_from<std::remove_reference_t<decltype(*t)>, Shape>;
};

// Concept: Must be a smart pointer
template<typename T>
concept SmartPointer = requires(T t) {
    typename T::element_type;  // Must have element_type (shared_ptr/unique_ptr trait)
    { t.get() } -> std::same_as<typename T::element_type*>;
    { *t } -> std::same_as<typename T::element_type&>;
};

// Function that ONLY accepts pointers/smart pointers - prevents slicing at compile time
template<ShapePointer T>
void draw_shape(const T& shape_ptr) {
    shape_ptr->draw();
}

// Function that ONLY accepts smart pointers specifically
template<SmartPointer T>
    requires std::derived_from<typename T::element_type, Shape>
void process_shape(const T& shape_ptr) {
    std::cout << "Type: " << shape_ptr->type() << ", Area: " << shape_ptr->area() << std::endl;
}
```

```

void example_concepts_prevent_slicing() {
    std::cout << "\n=== 1. C++20 CONCEPTS - COMPILE-TIME PREVENTION ===" << std::endl;
    std::cout << "Solution: Use concepts to enforce pointer usage at compile time\n" << std::endl;

    auto circle = std::make_shared<Circle>("ConceptCircle", 5.0);
    auto rect = std::make_unique<Rectangle>("ConceptRect", 4.0, 6.0);

    std::cout << " These calls compile (using pointers):" << std::endl;
    draw_shape(circle);           //  Works with shared_ptr
    draw_shape(rect);             //  Works with unique_ptr

    Circle* raw_circle = circle.get();
    draw_shape(raw_circle);       //  Works with raw pointer

    process_shape(circle);        //  Works with smart pointers
    process_shape(rect);          //  Works with smart pointers

    std::cout << "\n Concepts prevent slicing at compile time!" << std::endl;
    std::cout << "   (Uncommenting the line below would cause a compile error)" << std::endl;

    // Circle circle_obj("DirectCircle", 3.0);
    // draw_shape(circle_obj);    //  Compile error! Not a pointer type
}

// ==================================================================
// 2. C++20 RANGES - WORK WITH REFERENCES, NOT COPIES
// ==================================================================

void example_ranges_prevent_slicing() {
    std::cout << "\n=== 2. C++20 RANGES - REFERENCE-BASED OPERATIONS ===" << std::endl;
    std::cout << "Solution: Ranges work with references, preventing copies\n" << std::endl;

    std::vector<std::shared_ptr<Shape>> shapes;
    shapes.push_back(std::make_shared<Circle>("RangeCircle1", 3.0));
    shapes.push_back(std::make_shared<Rectangle>("RangeRect1", 5.0, 4.0));
    shapes.push_back(std::make_shared<Circle>("RangeCircle2", 2.5));

    std::cout << "Using ranges to filter and transform (no slicing):" << std::endl;

    // Filter shapes with area > 20 and draw them
    auto large_shapes = shapes
        | std::views::filter([](const auto& s) { return s->area() > 20.0; })
        | std::views::transform([](const auto& s) { return s.get(); });

    for (const auto* shape : large_shapes) {
        shape->draw();
        std::cout << "  Area: " << shape->area() << std::endl;
```

```
    }

    std::cout << "\n Ranges operate on references - no slicing occurs!" << std::endl;
}

// ==================================================================
// 3. C++20 SPAN - NON-OWNING VIEWS WITHOUT SLICING
// ==================================================================

// Function taking span of Shape pointers
void draw_all_shapes(std::span<const std::shared_ptr<Shape>> shapes) {
    std::cout << "Drawing " << shapes.size() << " shapes:" << std::endl;
    for (const auto& shape : shapes) {
        shape->draw();
    }
}

// Function taking span of raw pointers
void process_shapes(std::span<const Shape* const> shapes) {
    double total_area = 0.0;
    for (const auto* shape : shapes) {
        total_area += shape->area();
    }
    std::cout << "Total area: " << total_area << std::endl;
}

void example_span_prevent_slicing() {
    std::cout << "\n=== 3. C++20 SPAN - NON-OWNING VIEWS ===" << std::endl;
    std::cout << "Solution: std::span provides views without copying\n" << std::endl;

    std::vector<std::shared_ptr<Shape>> shapes;
    shapes.push_back(std::make_shared<Circle>("SpanCircle", 4.0));
    shapes.push_back(std::make_shared<Rectangle>("SpanRect", 3.0, 5.0));

    std::cout << "Passing span of smart pointers:" << std::endl;
    draw_all_shapes(shapes);  //  No slicing - span of pointers

    std::cout << "\nPassing span of raw pointers:" << std::endl;
    std::vector<const Shape*> raw_ptrs;
    for (const auto& s : shapes) {
        raw_ptrs.push_back(s.get());
    }
    process_shapes(raw_ptrs);  //  No slicing - span of raw pointers

    std::cout << "\n std::span is non-owning - operates on existing pointers!" << std::endl;
}

// ==================================================================
// 4. C++20 CONSTRAINTS - DELETE SLICING-PRONE FUNCTIONS
// ==================================================================

```

```cpp
template<typename T>
    requires std::derived_from<T, Shape>
class ShapeContainer {
private:
    std::vector<std::unique_ptr<T>> items;

public:
    //  CORRECT: Add by pointer (transfers ownership)
    void add(std::unique_ptr<T> shape) {
        items.push_back(std::move(shape));
    }

    //  DELETED: Prevent adding by value (would cause slicing)
    void add(const T& shape) = delete;
    void add(T&& shape) = delete;

    //  CORRECT: Access via reference
    const T& get(size_t index) const {
        return *items[index];
    }

    void draw_all() const {
        for (const auto& item : items) {
            item->draw();
        }
    }

    size_t size() const { return items.size(); }
};

void example_delete_slicing_functions() {
    std::cout << "\n=== 4. DELETE SLICING-PRONE FUNCTIONS ===" << std::endl;
    std::cout << "Solution: Use = delete to prevent value-based operations\n"
        << std::endl;

    ShapeContainer<Shape> container;

    std::cout << "  These operations work (using pointers):" << std::endl;
    container.add(std::make_unique<Circle>("DeleteCircle", 5.0));
    container.add(std::make_unique<Rectangle>("DeleteRect", 4.0, 3.0));

    container.draw_all();

    std::cout << "\n Deleted functions prevent slicing at compile time!" <<
        std::endl;
    std::cout << "   (Uncommenting the lines below would cause compile errors)
        " << std::endl;

    // Circle circle("ByValueCircle", 3.0);
    // container.add(circle);          //  Compile error! Function deleted
    // container.add(std::move(circle)); //  Compile error! Function deleted
}

// ==================================================================
```

```
// 5. C++20 DESIGNATED INITIALIZERS WITH SMART POINTERS
// ==================================================================

struct ShapeConfig {
    std::shared_ptr<Shape> primary;
    std::shared_ptr<Shape> secondary;
    std::string label;
};

void example_designated_initializers() {
    std::cout << "\n=== 5. DESIGNATED INITIALIZERS - SAFE INITIALIZATION ===" << std::endl;
    std::cout << "Solution: Initialize with pointers using designated initializers\n" << std::endl;

    //  CORRECT: Designated initializers with smart pointers
    ShapeConfig config {
        .primary = std::make_shared<Circle>("Primary", 6.0),
        .secondary = std::make_shared<Rectangle>("Secondary", 5.0, 4.0),
        .label = "MyShapes"
    };

    std::cout << "Configuration: " << config.label << std::endl;
    config.primary->draw();
    config.secondary->draw();

    std::cout << "\n Designated initializers ensure proper pointer initialization!" << std::endl;
}

// ==================================================================
// 6. C++20 REQUIRES CLAUSE - POLYMORPHIC CONTAINERS ONLY
// ==================================================================

template<typename Container>
    requires requires(Container c) {
        // Require that container holds pointers to Shape-derived types
        { *c.begin() } -> std::convertible_to<std::shared_ptr<Shape>>;
    }
void draw_container(const Container& shapes) {
    std::cout << "Drawing " << shapes.size() << " shapes from container:" << std::endl;
    for (const auto& shape : shapes) {
        shape->draw();
    }
}

void example_requires_clause() {
    std::cout << "\n=== 6. REQUIRES CLAUSE - CONSTRAIN CONTAINER TYPES ===" << std::endl;
    std::cout << "Solution: Require containers to hold pointers only\n" << std::endl;

    std::vector<std::shared_ptr<Shape>> shapes;
```

```
    shapes.push_back(std::make_shared<Circle>("RequiresCircle", 4.5));
    shapes.push_back(std::make_shared<Rectangle>("RequiresRect", 6.0, 2.0));

    std::cout << " This works (container of shared_ptr):" << std::endl;
    draw_container(shapes);

    std::cout << "\n Requires clause ensures polymorphic container usage!" << std::endl;
    std::cout << "   (Uncommenting the lines below would cause a compile error)" << std::endl;

    // std::vector<Circle> value_circles;
    // value_circles.push_back(Circle("ValueCircle", 3.0));
    // draw_container(value_circles);  //  Compile error! Container holds values, not pointers
}

// ==================================================================
// 7. COMPARISON: C++11/14 VS C++20 PREVENTION
// ==================================================================

void example_comparison() {
    std::cout << "\n=== 7. C++11/14 VS C++20 PREVENTION COMPARISON ===" << std::endl;

    std::cout << "\n C++11/14 APPROACH (Runtime/Developer Discipline):" << std::endl;
    std::cout << " •   Manually use smart pointers everywhere" << std::endl;
    std::cout << " •   Remember to not pass by value" << std::endl;
    std::cout << " •   Runtime errors if you forget" << std::endl;
    std::cout << " •   Code reviews needed to catch mistakes" << std::endl;

    std::cout << "\n C++20 APPROACH (Compile-Time Enforcement):" << std::endl;
    std::cout << " •   Concepts enforce pointer usage at compile time" << std::endl;
    std::cout << " •   = delete prevents slicing-prone operations" << std::endl;
    std::cout << " •   Requires clauses constrain templates" << std::endl;
    std::cout << " •   Compiler catches mistakes before runtime!" << std::endl;
    std::cout << " •   std::span provides safe non-owning views" << std::endl;
    std::cout << " •   Ranges work with references by default" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  PREVENTING OBJECT SLICING WITH C++20 FEATURES" << std::
```

```
    endl;
    std::cout << "
        ================================================================" <<
        std::endl;

    example_concepts_prevent_slicing();
    example_ranges_prevent_slicing();
    example_span_prevent_slicing();
    example_delete_slicing_functions();
    example_designated_initializers();
    example_requires_clause();
    example_comparison();

    std::cout << "\n
        ================================================================" <<
        std::endl;
    std::cout << "  C++20 SLICING PREVENTION SUMMARY" << std::endl;
    std::cout << "
        ================================================================" <<
        std::endl;
    std::cout << "\ n NEW C++20 FEATURES TO PREVENT SLICING:" << std::endl;
    std::cout << "\n1.  CONCEPTS:" << std::endl;
    std::cout << " •  Define ShapePointer concept to enforce pointer types"
        << std::endl;
    std::cout << " •  Compile-time error if trying to pass by value" << std::
        endl;
    std::cout << " •  Example: template<ShapePointer T> void draw(const T&
        ptr)" << std::endl;

    std::cout << "\n2.  RANGES:" << std::endl;
    std::cout << " •  Operate on references, not copies" << std::endl;
    std::cout << " •  views::filter and views::transform preserve types" <<
        std::endl;
    std::cout << " •  No accidental slicing during transformations" << std::
        endl;

    std::cout << "\n3.  STD::SPAN:" << std::endl;
    std::cout << " •  Non-owning view of container" << std::endl;
    std::cout << " •  std::span<shared_ptr<Shape>> - no copies" << std::endl;
    std::cout << " •  Safe passing of pointer collections" << std::endl;

    std::cout << "\n4.  = DELETE:" << std::endl;
    std::cout << " •  Explicitly delete value-based operations" << std::endl;
    std::cout << " •  void add(const T& shape) = delete" << std::endl;
    std::cout << " •  Compiler prevents slicing attempts" << std::endl;

    std::cout << "\n5.  REQUIRES CLAUSES:" << std::endl;
    std::cout << " •  Constrain templates to pointer-holding containers" <<
        std::endl;
    std::cout << " •  requires { *c.begin() } -> convertible_to<shared_ptr<
        Shape>>" << std::endl;
    std::cout << " •  Type-safe generic programming" << std::endl;

    std::cout << "\n6.  DESIGNATED INITIALIZERS:" << std::endl;
```

```
    std::cout << " •   Clear initialization syntax" << std::endl;
    std::cout << " •   Config { .shape = make_shared<Circle>(...) }" << std::endl;
    std::cout << " •   Prevents accidental by-value initialization" << std::endl;

    std::cout << "\n KEY ADVANTAGE OF C++20:" << std::endl;
    std::cout << " •   C++11/14: Prevent slicing through discipline (runtime errors)" << std::endl;
    std::cout << " •   C++20: Prevent slicing through type system (compile-time errors)" << std::endl;
    std::cout << " •   Shift from \"remember not to\" → \"impossible to\"" << std::endl;

    std::cout << "\n BEST PRACTICE (C++20):" << std::endl;
    std::cout << "     Use concepts to enforce pointer parameters" << std::endl;
    std::cout << "     Use requires clauses for container constraints" << std::endl;
    std::cout << "     Use = delete for slicing-prone operations" << std::endl;
    std::cout << "     Use std::span for non-owning views" << std::endl;
    std::cout << "     Use ranges for safe transformations" << std::endl;
    std::cout << "     Let the compiler catch slicing at compile time!" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

49 Source Code: ObjectSlicingSmartPtr.cpp

File: `src/ObjectSlicingSmartPtr.cpp`
Repository: View on GitHub

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

// ====================================================================
// OBJECT SLICING AND SMART POINTERS (C++11/14)
// ====================================================================

// Base class
class Shape {
protected:
    std::string name;

public:
    Shape(const std::string& n) : name(n) {
        std::cout << "Shape constructor: " << name << std::endl;
    }

    virtual ~Shape() {
        std::cout << "Shape destructor: " << name << std::endl;
    }

    virtual void draw() const {
        std::cout << "Drawing Shape: " << name << std::endl;
    }

    virtual double area() const {
        return 0.0;
    }

    virtual void info() const {
        std::cout << "Shape: " << name << std::endl;
    }
};

// Derived class: Circle
class Circle : public Shape {
private:
    double radius;

public:
    Circle(const std::string& n, double r) : Shape(n), radius(r) {
        std::cout << "Circle constructor: " << name << " (radius=" << radius
            << ")" << std::endl;
    }

    ~Circle() override {
        std::cout << "Circle destructor: " << name << std::endl;
```

```
    }

    void draw() const override {
        std::cout << "Drawing Circle: " << name << " with radius " << radius
            << std::endl;
    }

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void info() const override {
        std::cout << "Circle: " << name << ", radius=" << radius << ", area="
            << area() << std::endl;
    }
};

// Derived class: Rectangle
class Rectangle : public Shape {
private:
    double width;
    double height;

public:
    Rectangle(const std::string& n, double w, double h)
        : Shape(n), width(w), height(h) {
        std::cout << "Rectangle constructor: " << name
                  << " (width=" << width << ", height=" << height << ")" <<
                      std::endl;
    }

    ~Rectangle() override {
        std::cout << "Rectangle destructor: " << name << std::endl;
    }

    void draw() const override {
        std::cout << "Drawing Rectangle: " << name
                  << " (" << width << "x" << height << ")" << std::endl;
    }

    double area() const override {
        return width * height;
    }

    void info() const override {
        std::cout << "Rectangle: " << name << ", " << width << "x" << height
                  << ", area=" << area() << std::endl;
    }
};

// ==================================================================
// 1. CLASSIC OBJECT SLICING (WITHOUT SMART POINTERS)
// ==================================================================

```

```
void example_classic_slicing() {
    std::cout << "\n=== 1. CLASSIC OBJECT SLICING ===" << std::endl;
    std::cout << "Problem: Assigning derived object to base object by value\n"
        << std::endl;

    Circle circle("MyCircle", 5.0);
    Shape shape = circle;  //  SLICING! Circle parts are lost

    std::cout << "\nCalling draw() on sliced object:" << std::endl;
    shape.draw();  // Calls Shape::draw(), not Circle::draw()!

    std::cout << "\nArea of sliced object: " << shape.area() << std::endl;  //
        Returns 0.0, not circle area!

    std::cout << "\n The Circle-specific data (radius) was sliced off!" <<
        std::endl;
}

// ==================================================================
// 2. OBJECT SLICING WITH SHARED_PTR - WRONG WAY
// ==================================================================

void example_shared_ptr_slicing_wrong() {
    std::cout << "\n=== 2. OBJECT SLICING WITH SHARED_PTR (WRONG) ===" << std
        ::endl;
    std::cout << "Problem: Creating shared_ptr to base from derived object by
        value\n" << std::endl;

    Circle circle("SlicedCircle", 7.0);

    //  WRONG: This creates a shared_ptr<Shape> pointing to a SLICED copy
    std::shared_ptr<Shape> shape_ptr = std::make_shared<Shape>(circle);

    std::cout << "\nCalling draw() through shared_ptr:" << std::endl;
    shape_ptr->draw();  // Calls Shape::draw(), not Circle::draw()!

    std::cout << "\nArea: " << shape_ptr->area() << std::endl;  // Returns
        0.0!

    std::cout << "\n Object was sliced when passed by value to make_shared<
        Shape>!" << std::endl;
}

// ==================================================================
// 3. CONTAINER OF SHARED_PTR - WRONG WAY (SLICING)
// ==================================================================

void example_container_slicing_wrong() {
    std::cout << "\n=== 3. CONTAINER WITH SLICING (WRONG) ===" << std::endl;
    std::cout << "Problem: Storing objects by value in container\n" << std::
        endl;

    std::vector<Shape> shapes;  //  WRONG: Stores Shape objects by value

```

```
    shapes.push_back(Circle("Circle1", 3.0));      // Sliced!
    shapes.push_back(Rectangle("Rect1", 4.0, 5.0)); // Sliced!

    std::cout << "\nDrawing shapes from container:" << std::endl;
    for (const auto& shape : shapes) {
        shape.draw();  // Always calls Shape::draw()!
        std::cout << "  Area: " << shape.area() << std::endl;  // Always returns 0.0!
    }

    std::cout << "\n All derived class data was sliced off!" << std::endl;
}

// ==================================================================
// 4. COPYING SMART POINTERS - POTENTIAL SLICING
// ==================================================================

void process_shape_wrong(std::shared_ptr<Shape> shape) {
    //  If we copy the object instead of using the pointer
    Shape copy = *shape;  // SLICING!
    copy.draw();  // Calls Shape::draw(), not the derived version
}

void example_copying_smart_ptr_wrong() {
    std::cout << "\n=== 4. COPYING OBJECT FROM SMART_PTR (WRONG) ===" << std::endl;
    std::cout << "Problem: Dereferencing smart_ptr and copying by value\n" << std::endl;

    auto circle = std::make_shared<Circle>("PointerCircle", 6.0);

    std::cout << "\nOriginal (via pointer):" << std::endl;
    circle->draw();

    std::cout << "\nCopied value from pointer:" << std::endl;
    process_shape_wrong(circle);  // Slicing happens inside function

    std::cout << "\n Dereferencing and copying caused slicing!" << std::endl;
}

// ==================================================================
// 5. CORRECT WAY: SHARED_PTR TO DERIVED CLASS
// ==================================================================

void example_shared_ptr_correct() {
    std::cout << "\n=== 5. CORRECT: SHARED_PTR TO DERIVED CLASS ===" << std::endl;
    std::cout << "Solution: Create shared_ptr directly to derived class\n" << std::endl;

    //  CORRECT: Create shared_ptr<Circle> directly
    std::shared_ptr<Circle> circle = std::make_shared<Circle>("CorrectCircle", 8.0);

```

```
    //  Upcast to base class pointer (no slicing!)
    std::shared_ptr<Shape> shape_ptr = circle;

    std::cout << "\nCalling draw() through base pointer:" << std::endl;
    shape_ptr->draw();  //  Calls Circle::draw() via polymorphism

    std::cout << "\nArea: " << shape_ptr->area() << std::endl;  //  Returns actual circle area

    std::cout << "\n Polymorphism works correctly - no slicing!" << std::endl;
}

// ==================================================================
// 6. CORRECT WAY: CONTAINER OF SMART POINTERS
// ==================================================================

void example_container_correct() {
    std::cout << "\n=== 6. CORRECT: CONTAINER OF SMART_PTR ===" << std::endl;
    std::cout << "Solution: Store pointers to base class, not objects\n" << std::endl;

    //  CORRECT: Vector of pointers to Shape
    std::vector<std::shared_ptr<Shape>> shapes;

    shapes.push_back(std::make_shared<Circle>("Circle2", 4.0));
    shapes.push_back(std::make_shared<Rectangle>("Rect2", 6.0, 3.0));
    shapes.push_back(std::make_shared<Circle>("Circle3", 2.5));

    std::cout << "\nDrawing shapes from container:" << std::endl;
    for (const auto& shape : shapes) {
        shape->draw();  //  Calls correct derived class method
        std::cout << "  Area: " << shape->area() << std::endl;  //  Returns actual area
    }

    std::cout << "\n Polymorphism works - each object retains its type!" << std::endl;
}

// ==================================================================
// 7. CORRECT WAY: UNIQUE_PTR (C++11/14)
// ==================================================================

void example_unique_ptr_correct() {
    std::cout << "\n=== 7. CORRECT: UNIQUE_PTR FOR OWNERSHIP ===" << std::endl;
    std::cout << "Solution: Use unique_ptr for exclusive ownership\n" << std::endl;

    //  CORRECT: unique_ptr to derived class
    std::unique_ptr<Circle> circle = std::make_unique<Circle>("UniqueCircle", 5.5);

```

```
    //  Move to base class pointer (transfer ownership)
    std::unique_ptr<Shape> shape_ptr = std::move(circle);

    std::cout << "\nCalling methods through unique_ptr:" << std::endl;
    shape_ptr->draw();  //  Calls Circle::draw()
    shape_ptr->info();  //  Calls Circle::info()

    std::cout << "\n unique_ptr maintains polymorphism without slicing!" << std::endl;
}

// ==================================================================
// 8. FACTORY PATTERN WITH SMART POINTERS (CORRECT)
// ==================================================================

std::unique_ptr<Shape> createShape(const std::string& type, const std::string& name) {
    if (type == "circle") {
        return std::make_unique<Circle>(name, 10.0);
    } else if (type == "rectangle") {
        return std::make_unique<Rectangle>(name, 8.0, 6.0);
    }
    return std::make_unique<Shape>(name);
}

void example_factory_pattern() {
    std::cout << "\n=== 8. FACTORY PATTERN (CORRECT) ===" << std::endl;
    std::cout << "Solution: Return base class pointer to derived objects\n" << std::endl;

    auto shape1 = createShape("circle", "FactoryCircle");
    auto shape2 = createShape("rectangle", "FactoryRect");

    std::cout << "\nUsing factory-created objects:" << std::endl;
    shape1->draw();
    shape2->draw();

    std::cout << "\n Factory pattern prevents slicing by returning pointers!" << std::endl;
}

// ==================================================================
// 9. COMPARISON: WRONG VS RIGHT
// ==================================================================

void example_side_by_side_comparison() {
    std::cout << "\n=== 9. SIDE-BY-SIDE COMPARISON ===" << std::endl;

    std::cout << "\n WRONG - Object Slicing:" << std::endl;
    Circle c1("WrongCircle", 5.0);
    Shape s1 = c1;  // Slicing!
    s1.draw();      // Calls Shape::draw()
    std::cout << "   Area: " << s1.area() << " (lost circle data!)" << std::endl;
```

```

    std::cout << "\n CORRECT - Pointer (no slicing):" << std::endl;
    auto c2 = std::make_shared<Circle>("RightCircle", 5.0);
    std::shared_ptr<Shape> s2 = c2;  // No slicing!
    s2->draw();     // Calls Circle::draw()
    std::cout << "   Area: " << s2->area() << " (preserves circle data!)" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  OBJECT SLICING AND SMART POINTERS (C++11/14)" << std::endl;
    std::cout << " ==================================================================" << std::endl;

    // Wrong ways (demonstrating slicing)
    example_classic_slicing();
    example_shared_ptr_slicing_wrong();
    example_container_slicing_wrong();
    example_copying_smart_ptr_wrong();

    // Correct ways (avoiding slicing)
    example_shared_ptr_correct();
    example_container_correct();
    example_unique_ptr_correct();
    example_factory_pattern();

    // Comparison
    example_side_by_side_comparison();

    std::cout << "\n ==================================================================" << std::endl;
    std::cout << "  KEY TAKEAWAYS" << std::endl;
    std::cout << " ==================================================================" << std::endl;
    std::cout << "\n CAUSES OF OBJECT SLICING:" << std::endl;
    std::cout << "   1. Assigning derived object to base object by value" << std::endl;
    std::cout << "   2. Passing derived object by value to function expecting base" << std::endl;
    std::cout << "   3. Storing objects by value in containers of base type" << std::endl;
    std::cout << "   4. Creating smart_ptr<Base>(derived_object) - copies by value" << std::endl;
```

```
    std::cout << "   5. Dereferencing smart pointer and copying: Base copy = *ptr" << std::endl;

    std::cout << "\n HOW TO AVOID SLICING WITH SMART POINTERS:" << std::endl;
    std::cout << "   1. Always create smart_ptr to the actual derived type" << std::endl;
    std::cout << "   2. Use make_shared<DerivedType>() or make_unique<DerivedType>()" << std::endl;
    std::cout << "   3. Store smart_ptr<Base> in containers, not objects" << std::endl;
    std::cout << "   4. Pass smart pointers by reference or const reference" << std::endl;
    std::cout << "   5. Never dereference and copy: always use pointer access (->)" << std::endl;

    std::cout << "\n C++11/14 BEST PRACTICES:" << std::endl;
    std::cout << " •   Use std::unique_ptr for exclusive ownership" << std::endl;
    std::cout << " •   Use std::shared_ptr for shared ownership" << std::endl;
    std::cout << " •   Always use make_shared/make_unique (C++14)" << std::endl;
    std::cout << " •   Prefer polymorphism via pointers, not by value" << std::endl;
    std::cout << " •   Virtual destructors are essential for polymorphic classes" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

50 Source Code: OptionalExamples.cpp

File: `src/OptionalExamples.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <optional>
#include <vector>
#include <map>
#include <algorithm>
#include <cmath>

// ====================================================================
// 1. BASIC OPTIONAL USAGE
// ====================================================================
void example_basic_optional() {
    std::cout << "\n=== 1. BASIC OPTIONAL USAGE ===" << std::endl;

    std::optional<int> maybe_value;  // Empty optional

    std::cout << "Has value: " << (maybe_value.has_value() ? "Yes" : "No") << std::endl;

    maybe_value = 42;  // Assign value
    std::cout << "Has value: " << (maybe_value.has_value() ? "Yes" : "No") << std::endl;
    std::cout << "Value: " << *maybe_value << std::endl;
    std::cout << "Value (using .value()): " << maybe_value.value() << std::endl;
}

// ====================================================================
// 2. OPTIONAL WITH VALUE_OR
// ====================================================================
void example_value_or() {
    std::cout << "\n=== 2. OPTIONAL WITH VALUE_OR ===" << std::endl;

    std::optional<int> empty_opt;
    std::optional<int> filled_opt = 100;

    std::cout << "Empty optional value_or(0): " << empty_opt.value_or(0) << std::endl;
    std::cout << "Filled optional value_or(0): " << filled_opt.value_or(0) << std::endl;

    std::optional<std::string> empty_str;
    std::cout << "Empty string value_or: " << empty_str.value_or("default") << std::endl;
}

// ====================================================================
// 3. OPTIONAL AS FUNCTION RETURN TYPE
// ====================================================================
```

```
std::optional<int> find_value(const std::vector<int>& vec, int target) {
    auto it = std::find(vec.begin(), vec.end(), target);
    if (it != vec.end()) {
        return *it;
    }
    return std::nullopt;  // Return empty optional
}

std::optional<std::string> get_name_by_id(int id) {
    std::map<int, std::string> database = {
        {1, "Alice"},
        {2, "Bob"},
        {3, "Charlie"}
    };

    auto it = database.find(id);
    if (it != database.end()) {
        return it->second;
    }
    return std::nullopt;
}

void example_optional_return() {
    std::cout << "\n=== 3. OPTIONAL AS FUNCTION RETURN TYPE ===" << std::endl;

    std::vector<int> numbers = {10, 20, 30, 40, 50};

    auto result1 = find_value(numbers, 30);
    if (result1) {
        std::cout << "Found: " << *result1 << std::endl;
    } else {
        std::cout << "Not found" << std::endl;
    }

    auto result2 = find_value(numbers, 99);
    if (result2) {
        std::cout << "Found: " << *result2 << std::endl;
    } else {
        std::cout << "Not found" << std::endl;
    }

    auto name1 = get_name_by_id(2);
    std::cout << "ID 2: " << name1.value_or("Unknown") << std::endl;

    auto name2 = get_name_by_id(99);
    std::cout << "ID 99: " << name2.value_or("Unknown") << std::endl;
}

// ==================================================================
// 4. OPTIONAL WITH IF STATEMENT
// ==================================================================
void example_optional_if() {
    std::cout << "\n=== 4. OPTIONAL WITH IF STATEMENT ===" << std::endl;

```

```
    auto maybe_name = get_name_by_id(1);

    // C++17 if with initializer
    if (auto name = get_name_by_id(1); name.has_value()) {
        std::cout << "Found name: " << *name << std::endl;
    } else {
        std::cout << "Name not found" << std::endl;
    }

    if (auto name = get_name_by_id(999); name) {
        std::cout << "Found name: " << *name << std::endl;
    } else {
        std::cout << "Name not found" << std::endl;
    }
}

// ==================================================================
// 5. OPTIONAL WITH EXCEPTIONS
// ==================================================================
void example_optional_exceptions() {
    std::cout << "\n=== 5. OPTIONAL WITH EXCEPTIONS ===" << std::endl;

    std::optional<int> empty_opt;
    std::optional<int> filled_opt = 42;

    try {
        std::cout << "Filled optional value: " << filled_opt.value() << std::
            endl;
        std::cout << "Empty optional value: " << empty_opt.value() << std::
            endl;  // Throws!
    } catch (const std::bad_optional_access& e) {
        std::cout << "Exception caught: " << e.what() << std::endl;
    }
}

// ==================================================================
// 6. OPTIONAL WITH CUSTOM TYPES
// ==================================================================
struct Person {
    std::string name;
    int age;

    Person(const std::string& n, int a) : name(n), age(a) {}

    friend std::ostream& operator<<(std::ostream& os, const Person& p) {
        os << p.name << " (age " << p.age << ")";
        return os;
    }
};

std::optional<Person> find_person(const std::string& name) {
    std::vector<Person> people = {
        {"Alice", 30},
        {"Bob", 25},
```

```
        {"Charlie", 35}
    };

    auto it = std::find_if(people.begin(), people.end(),
        [&name](const Person& p) { return p.name == name; });

    if (it != people.end()) {
        return *it;
    }
    return std::nullopt;
}

void example_optional_custom_type() {
    std::cout << "\n=== 6. OPTIONAL WITH CUSTOM TYPES ===" << std::endl;

    auto person1 = find_person("Alice");
    if (person1) {
        std::cout << "Found: " << *person1 << std::endl;
    }

    auto person2 = find_person("David");
    if (person2) {
        std::cout << "Found: " << *person2 << std::endl;
    } else {
        std::cout << "Person not found" << std::endl;
    }
}

// ==================================================================
// 7. OPTIONAL WITH EMPLACE
// ==================================================================
void example_optional_emplace() {
    std::cout << "\n=== 7. OPTIONAL WITH EMPLACE ===" << std::endl;

    std::optional<Person> opt_person;

    std::cout << "Before emplace: " << (opt_person.has_value() ? "Has value" :
        "Empty") << std::endl;

    opt_person.emplace("John", 28);  // Construct in-place

    std::cout << "After emplace: " << *opt_person << std::endl;

    opt_person.emplace("Jane", 32);  // Replace existing value
    std::cout << "After second emplace: " << *opt_person << std::endl;
}

// ==================================================================
// 8. OPTIONAL WITH RESET
// ==================================================================
void example_optional_reset() {
    std::cout << "\n=== 8. OPTIONAL WITH RESET ===" << std::endl;

    std::optional<int> value = 100;
```

```
    std::cout << "Initial value: " << *value << std::endl;

    value.reset();  // Clear the optional
    std::cout << "After reset: " << (value.has_value() ? "Has value" : "Empty"
        ) << std::endl;

    value = 200;
    std::cout << "After reassignment: " << *value << std::endl;

    value = std::nullopt;  // Another way to clear
    std::cout << "After nullopt: " << (value.has_value() ? "Has value" : "
        Empty") << std::endl;
}

// ==================================================================
// 9. OPTIONAL CHAINING WITH TRANSFORM (C++23-style)
// ==================================================================
std::optional<double> safe_sqrt(double value) {
    if (value >= 0.0) {
        return std::sqrt(value);
    }
    return std::nullopt;
}

std::optional<double> safe_divide(double numerator, double denominator) {
    if (denominator != 0.0) {
        return numerator / denominator;
    }
    return std::nullopt;
}

void example_optional_chaining() {
    std::cout << "\n=== 9. OPTIONAL CHAINING ===" << std::endl;

    auto result1 = safe_divide(100, 4);
    if (result1) {
        auto sqrt_result = safe_sqrt(*result1);
        if (sqrt_result) {
            std::cout << "sqrt(100/4) = " << *sqrt_result << std::endl;
        }
    }

    auto result2 = safe_divide(100, 0);
    std::cout << "Division by zero result: "
              << (result2.has_value() ? "Has value" : "Empty") << std::endl;
}

// ==================================================================
// 10. OPTIONAL WITH COMPARISON
// ==================================================================
void example_optional_comparison() {
    std::cout << "\n=== 10. OPTIONAL WITH COMPARISON ===" << std::endl;

    std::optional<int> opt1 = 10;
```

```
    std::optional<int> opt2 = 20;
    std::optional<int> opt3 = 10;
    std::optional<int> empty;

    std::cout << "opt1 == opt3: " << (opt1 == opt3 ? "true" : "false") << std::endl;
    std::cout << "opt1 < opt2: " << (opt1 < opt2 ? "true" : "false") << std::endl;
    std::cout << "opt1 == 10: " << (opt1 == 10 ? "true" : "false") << std::endl;
    std::cout << "empty == nullopt: " << (empty == std::nullopt ? "true" : "false") << std::endl;
}

// ==================================================================
// 11. OPTIONAL IN CONTAINERS
// ==================================================================
void example_optional_in_containers() {
    std::cout << "\n=== 11. OPTIONAL IN CONTAINERS ===" << std::endl;

    std::vector<std::optional<int>> values = {
        10,
        std::nullopt,
        20,
        std::nullopt,
        30
    };

    std::cout << "Processing optional values:" << std::endl;
    for (size_t i = 0; i < values.size(); ++i) {
        if (values[i]) {
            std::cout << "  Index " << i << ": " << *values[i] << std::endl;
        } else {
            std::cout << "  Index " << i << ": empty" << std::endl;
        }
    }

    // Count non-empty values
    int count = 0;
    for (const auto& opt : values) {
        if (opt) ++count;
    }
    std::cout << "Non-empty values: " << count << std::endl;
}

// ==================================================================
// 12. OPTIONAL WITH MAKE_OPTIONAL
// ==================================================================
std::optional<Person> create_person_if_valid(const std::string& name, int age) {
    if (!name.empty() && age > 0 && age < 150) {
        return Person(name, age);
    }
    return std::nullopt;
```

```
}

void example_make_optional() {
    std::cout << "\n=== 12. OPTIONAL WITH MAKE_OPTIONAL ===" << std::endl;

    auto person1 = create_person_if_valid("Alice", 30);
    if (person1) {
        std::cout << "Valid person: " << *person1 << std::endl;
    }

    auto person2 = create_person_if_valid("", 25);
    if (person2) {
        std::cout << "Valid person: " << *person2 << std::endl;
    } else {
        std::cout << "Invalid person (empty name)" << std::endl;
    }

    auto person3 = create_person_if_valid("Bob", -5);
    if (person3) {
        std::cout << "Valid person: " << *person3 << std::endl;
    } else {
        std::cout << "Invalid person (negative age)" << std::endl;
    }
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    C++17 STD::OPTIONAL EXAMPLES" << std::endl;
    std::cout << "================================================" << std::
        endl;

    example_basic_optional();
    example_value_or();
    example_optional_return();
    example_optional_if();
    example_optional_exceptions();
    example_optional_custom_type();
    example_optional_emplace();
    example_optional_reset();
    example_optional_chaining();
    example_optional_comparison();
    example_optional_in_containers();
    example_make_optional();

    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    ALL EXAMPLES COMPLETED" << std::endl;
    std::cout << "================================================\n" << std::
        endl;

```

```
    return 0;
}
```

51 Source Code: PerfectForwardingAndRequires.cpp

File: `src/PerfectForwardingAndRequires.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// ADVANCED PERFECT FORWARDING, CONCEPTS, AND REQUIRES-EXPRESSIONS
// ==================================================================
// This example covers:
// 1. std::forward and perfect forwarding with concepts
// 2. "requires requires" - nested requires explained
// 3. Requirement clauses vs requirement expressions
// 4. When to use and when to avoid "requires requires"
// 5. Embedded systems considerations
//
// TOPICS:
// - Perfect forwarding with universal references
// - Concepts constraining forwarding functions
// - requires clause (starts constraint)
// - requires expression (tests validity)
// - SFINAE vs Concepts comparison
// - Zero-cost abstractions for embedded systems
// ==================================================================

#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <type_traits>
#include <concepts>
#include <utility>

// ==================================================================
// SECTION 1: PERFECT FORWARDING BASICS WITH std::forward
// ==================================================================

// Without perfect forwarding - INEFFICIENT
template<typename T>
void call_by_value(T arg) {
    std::cout << "  Value parameter (copy made)" << std::endl;
}

// With perfect forwarding - EFFICIENT
template<typename T>
void call_with_forward(T&& arg) {
    std::cout << "  Universal reference with forward" << std::endl;
    // std::forward<T>(arg) preserves value category:
    // - lvalue stays lvalue (no move)
    // - rvalue stays rvalue (enables move)
}

void demonstrate_perfect_forwarding() {
    std::cout << "\n=== 1. PERFECT FORWARDING BASICS ===" << std::endl;

```

```
    std::string lvalue = "Hello";

    std::cout << "\nCalling by value:" << std::endl;
    call_by_value(lvalue);              // Copy
    call_by_value(std::string("World")); // Copy + move

    std::cout << "\nCalling with forward:" << std::endl;
    call_with_forward(lvalue);              // No copy
    call_with_forward(std::string("World")); // No copy

    std::cout << "\n KEY POINT:" << std::endl;
    std::cout << "   T&& in template = universal reference" << std::endl;
    std::cout << "   std::forward<T> preserves value category" << std::endl;
}

// ==================================================================
// SECTION 2: REQUIRES CLAUSE VS REQUIRES EXPRESSION
// ==================================================================

// TERMINOLOGY:
// - requires clause: Introduces a constraint (keyword 'requires')
// - requires expression: Tests if expressions are valid

// Example 1: requires clause WITHOUT requires expression
template<typename T>
requires std::integral<T>  // <- This is a requires CLAUSE
T square(T x) {
    return x * x;
}

// Example 2: requires clause WITH requires expression
template<typename T>
requires requires(T x) {   // <- First 'requires' = clause, second = expression
    x + x;                 // Check if x can be added
    x * x;                 // Check if x can be multiplied
}
T compute(T x) {
    return x * x + x;
}

void demonstrate_requires_syntax() {
    std::cout << "\n=== 2. REQUIRES CLAUSE VS REQUIRES EXPRESSION ===" << std::endl;

    std::cout << "\n TERMINOLOGY:" << std::endl;
    std::cout << "   requires clause:      'requires <constraint>'" << std::endl;
    std::cout << "   requires expression:  'requires(params) { tests; }'" << std::endl;

    std::cout << "\n square(5) = " << square(5) << std::endl;
    std::cout << "  compute(5) = " << compute(5) << std::endl;

```

```
    // square(3.14);   // ERROR: not integral
    // compute("hi");  // ERROR: can't add/multiply strings
}

// ==================================================================
// SECTION 3: THE "REQUIRES REQUIRES" PATTERN
// ==================================================================

// WHAT IS "requires requires"?
//
// Template<typename T>
// requires requires(T t) { ... }
//          ^       ^
//          |       |
//          |       +-- requires EXPRESSION (tests code validity)
//          +---------- requires CLAUSE (introduces constraint)
//
// WHEN TO USE:
// - You need to test if specific operations are valid
// - Standard concepts don't cover your needs
// - You're checking syntax, not just type properties

// Example: Check if type has specific member function
template<typename T>
requires requires(T obj) {           // "requires requires"
    { obj.serialize() } -> std::same_as<std::string>;
}
void save(const T& obj) {
    std::string data = obj.serialize();
    std::cout << "   Saved: " << data << std::endl;
}

// Example: Check if type supports arithmetic and comparison
template<typename T>
requires requires(T a, T b) {        // "requires requires"
    { a + b } -> std::convertible_to<T>;
    { a - b } -> std::convertible_to<T>;
    { a < b } -> std::convertible_to<bool>;
    { a > b } -> std::convertible_to<bool>;
}
T clamp(T value, T min, T max) {
    if (value < min) return min;
    if (value > max) return max;
    return value;
}

// Test classes
class Serializable {
public:
    std::string serialize() const { return "Serializable{data}"; }
};

void demonstrate_requires_requires() {
    std::cout << "\n=== 3. THE 'REQUIRES REQUIRES' PATTERN ===" << std::endl;
```

```

    std::cout << "\n EXPLANATION:" << std::endl;
    std::cout << "   First 'requires':  Starts the constraint clause" << std::endl;
    std::cout << "   Second 'requires': Begins the expression testing code" << std::endl;
    std::cout << "   Inside { ... }:    Operations that must be valid" << std::endl;

    std::cout << "\n Using save() with Serializable:" << std::endl;
    Serializable obj;
    save(obj);

    std::cout << "\n Using clamp() with int:" << std::endl;
    std::cout << "   clamp(150, 0, 100) = " << clamp(150, 0, 100) << std::endl;
    std::cout << "   clamp(-10, 0, 100) = " << clamp(-10, 0, 100) << std::endl;
    std::cout << "   clamp(50, 0, 100) = " << clamp(50, 0, 100) << std::endl;
}

// ==================================================================
// SECTION 4: PERFECT FORWARDING WITH CONCEPTS
// ==================================================================

// Constrain what types can be forwarded
template<typename T>
concept Movable = std::is_move_constructible_v<T>;

template<typename T>
concept Copyable = std::is_copy_constructible_v<T>;

// Factory function with perfect forwarding and concepts
template<typename T, typename... Args>
requires std::constructible_from<T, Args...>  // Concept constrains construction
std::unique_ptr<T> make_unique_constrained(Args&&... args) {
    return std::make_unique<T>(std::forward<Args>(args)...);
}

// Forwarding wrapper with requires requires
template<typename Func, typename... Args>
requires requires(Func f, Args... args) {    // Check if callable
    f(std::forward<Args>(args)...);
}
auto forward_call(Func&& func, Args&&... args) {
    std::cout << "   Forwarding call..." << std::endl;
    return func(std::forward<Args>(args)...);
}

void demonstrate_forwarding_with_concepts() {
    std::cout << "\n=== 4. PERFECT FORWARDING WITH CONCEPTS ===" << std::endl;

    std::cout << "\n make_unique_constrained:" << std::endl;
```

```
    auto ptr1 = make_unique_constrained<std::string>("Hello", 5, 'X');
    std::cout << "   Created: " << *ptr1 << std::endl;

    auto ptr2 = make_unique_constrained<std::vector<int>>(10, 42);
    std::cout << "   Created vector of size: " << ptr2->size() << std::endl;

    std::cout << "\n forward_call with lambda:" << std::endl;
    auto lambda = [](int x, int y) {
        return x + y;
    };
    int result = forward_call(lambda, 10, 20);
    std::cout << "   Result: " << result << std::endl;
}

// ==================================================================
// SECTION 5: WHEN TO USE "REQUIRES REQUIRES"
// ==================================================================

void explain_when_to_use_requires_requires() {
    std::cout << "\n=== 5. WHEN TO USE 'REQUIRES REQUIRES' ===" << std::endl;

    std::cout << "\n USE 'requires requires' WHEN:" << std::endl;
    std::cout << "   1. Testing specific syntax/operations:" << std::endl;
    std::cout << "      requires requires(T t) { t.foo(); t.bar(); }" << std::endl;
    std::cout << "\n   2. No standard concept exists:" << std::endl;
    std::cout << "      requires requires(T t) { t.serialize(); }" << std::endl;
    std::cout << "\n   3. Complex compound expressions:" << std::endl;
    std::cout << "      requires requires(T a, T b) { a + b * 2; }" << std::endl;
    std::cout << "\n   4. Return type checking:" << std::endl;
    std::cout << "      requires requires(T t) { { t.size() } -> std::same_as<size_t>; }" << std::endl;

    std::cout << "\n AVOID 'requires requires' WHEN:" << std::endl;
    std::cout << "   1. Standard concept exists:" << std::endl;
    std::cout << "        requires requires(T t) { t == t; }" << std::endl;
    std::cout << "        requires std::equality_comparable<T>" << std::endl;
    std::cout << "\n   2. Simple type trait works:" << std::endl;
    std::cout << "        requires requires { typename T::value_type; }" << std::endl;
    std::cout << "        requires requires { typename T::value_type; }  # OK if needed" << std::endl;
    std::cout << "\n   3. Makes code harder to read:" << std::endl;
    std::cout << "      Define a named concept instead!" << std::endl;

    std::cout << "\n BEST PRACTICE:" << std::endl;
    std::cout << "   Wrap 'requires requires' in a named concept:" << std::endl;
    std::cout << "   template<typename T>" << std::endl;
    std::cout << "   concept MyConstraint = requires(T t) { /* tests */ };" << std::endl;
}
```

```cpp

// ==================================================================
// SECTION 6: READABILITY - NAMED CONCEPTS VS INLINE REQUIRES
// ==================================================================

// BAD: Inline requires requires - hard to read and reuse
template<typename T>
requires requires(T container) {
    { container.begin() } -> std::input_or_output_iterator;
    { container.end() } -> std::input_or_output_iterator;
    { container.size() } -> std::convertible_to<std::size_t>;
    requires std::is_same_v<typename T::value_type, int>;
}
void process_bad(const T& container) {
    std::cout << "Size: " << container.size() << std::endl;
}

// GOOD: Named concept - readable and reusable
template<typename T>
concept IntContainer = requires(T container) {
    { container.begin() } -> std::input_or_output_iterator;
    { container.end() } -> std::input_or_output_iterator;
    { container.size() } -> std::convertible_to<std::size_t>;
    requires std::is_same_v<typename T::value_type, int>;
};

template<IntContainer T>
void process_good(const T& container) {
    std::cout << "Size: " << container.size() << std::endl;
}

void demonstrate_readability() {
    std::cout << "\n=== 6. READABILITY: NAMED CONCEPTS VS INLINE ===" << std::endl;

    std::cout << "\n BAD - Inline 'requires requires':" << std::endl;
    std::cout << "   template<typename T>" << std::endl;
    std::cout << "   requires requires(T t) { /* 10 lines */ }" << std::endl;
    std::cout << "   void func(T x) { ... }" << std::endl;
    std::cout << "   Problems: Hard to read, can't reuse, error messages unclear" << std::endl;

    std::cout << "\n GOOD - Named concept:" << std::endl;
    std::cout << "   template<typename T>" << std::endl;
    std::cout << "   concept MyConstraint = requires(T t) { /* tests */ };" << std::endl;
    std::cout << "   \n   template<MyConstraint T>" << std::endl;
    std::cout << "   void func(T x) { ... }" << std::endl;
    std::cout << "   Benefits: Readable, reusable, clear error messages" << std::endl;

    std::vector<int> vec = {1, 2, 3, 4, 5};
    process_good(vec);
}
```

```

// ==================================================================
// SECTION 7: EMBEDDED SYSTEMS CONSIDERATIONS
// ==================================================================

void explain_embedded_systems() {
    std::cout << "\n=== 7. EMBEDDED SYSTEMS CONSIDERATIONS ===" << std::endl;

    std::cout << "\n ARE CONCEPTS USEFUL FOR EMBEDDED SYSTEMS?" << std::endl;
    std::cout << "     YES! Here's why:" << std::endl;

    std::cout << "\n1.  ZERO RUNTIME COST:" << std::endl;
    std::cout << " •   Concepts are compile-time only" << std::endl;
    std::cout << " •   No runtime overhead vs unconstrained templates" << std::endl;
    std::cout << " •   Generated code is identical" << std::endl;
    std::cout << " •   Same as SFINAE but cleaner" << std::endl;

    std::cout << "\n2.  REDUCE CODE SIZE:" << std::endl;
    std::cout << " •   Catch errors early = fewer template instantiations" << std::endl;
    std::cout << " •   Better error messages = less debugging code" << std::endl;
    std::cout << " •   Explicit constraints prevent accidental instantiations" << std::endl;

    std::cout << "\n3.  TYPE SAFETY:" << std::endl;
    std::cout << " •   Catch type errors at compile time" << std::endl;
    std::cout << " •   No runtime checks needed" << std::endl;
    std::cout << " •   Perfect for safety-critical systems" << std::endl;

    std::cout << "\n4.  SELF-DOCUMENTING:" << std::endl;
    std::cout << " •   Requirements are explicit in code" << std::endl;
    std::cout << " •   No need for extensive comments" << std::endl;
    std::cout << " •   Easier code review and maintenance" << std::endl;

    std::cout << "\n WHEN TO BE CAUTIOUS:" << std::endl;
    std::cout << "   1. Compiler support: Need C++20" << std::endl;
    std::cout << "   2. Compilation time: Complex concepts slow builds" << std::endl;
    std::cout << "   3. Code size: Templates can increase binary size" << std::endl;
    std::cout << "   4. Debugging: Template errors can be complex" << std::endl;

    std::cout << "\n EMBEDDED BEST PRACTICES:" << std::endl;
    std::cout << "     Use concepts for HAL (Hardware Abstraction Layer)" << std::endl;
    std::cout << "     Constrain register access templates" << std::endl;
    std::cout << "     Type-safe peripheral interfaces" << std::endl;
    std::cout << "     Compile-time buffer size checking" << std::endl;
    std::cout << "     Avoid deep template nesting" << std::endl;
    std::cout << "     Measure binary size impact" << std::endl;
}
```

```

// ==================================================================
// SECTION 8: EMBEDDED EXAMPLE - REGISTER ACCESS
// ==================================================================

// Embedded systems often need type-safe register access
template<typename T>
concept RegisterType = requires {
    requires std::is_integral_v<T>;
    requires sizeof(T) <= 4;  // 32-bit or smaller
    requires std::is_trivially_copyable_v<T>;
};

template<RegisterType T>
class Register {
private:
    volatile T* address_;

public:
    explicit Register(uintptr_t addr) : address_(reinterpret_cast<volatile T
        *>(addr)) {}

    void write(T value) const {
        *address_ = value;
    }

    T read() const {
        return *address_;
    }

    void set_bit(unsigned bit) requires std::unsigned_integral<T> {
        *address_ |= (T{1} << bit);
    }

    void clear_bit(unsigned bit) requires std::unsigned_integral<T> {
        *address_ &= ~(T{1} << bit);
    }
};

void demonstrate_embedded_concepts() {
    std::cout << "\n=== 8. EMBEDDED EXAMPLE - REGISTER ACCESS ===" << std::
        endl;

    std::cout << "\n Type-safe register access with concepts:" << std::endl;

    // Simulate hardware registers
    uint32_t mock_register = 0x00000000;
    Register<uint32_t> gpio_control(reinterpret_cast<uintptr_t>(&mock_register
        ));

    std::cout << "   Initial value: 0x" << std::hex << mock_register << std::
        dec << std::endl;

    gpio_control.write(0xAABBCCDD);
```

```
    std::cout << "   After write:   0x" << std::hex << mock_register << std::dec << std::endl;

    gpio_control.set_bit(4);
    std::cout << "   After set_bit(4): 0x" << std::hex << mock_register << std::dec << std::endl;

    std::cout << "\n Benefits:" << std::endl;
    std::cout << " •   Compile-time type checking" << std::endl;
    std::cout << " •   No runtime overhead" << std::endl;
    std::cout << " •   Can't use wrong size types" << std::endl;
    std::cout << " •   set_bit/clear_bit only for unsigned types" << std::endl;
}

// ==================================================================
// SECTION 9: COMPARISON - SFINAE VS CONCEPTS
// ==================================================================

// Old way: SFINAE (Substitution Failure Is Not An Error)
template<typename T, typename = std::enable_if_t<std::is_integral_v<T>>>
T old_square(T x) {
    return x * x;
}

// New way: Concepts
template<std::integral T>
T new_square(T x) {
    return x * x;
}

void compare_sfinae_vs_concepts() {
    std::cout << "\n=== 9. COMPARISON: SFINAE VS CONCEPTS ===" << std::endl;

    std::cout << "\n OLD WAY (SFINAE):" << std::endl;
    std::cout << "   template<typename T, typename = std::enable_if_t<...>>" << std::endl;
    std::cout << "   T func(T x) { ... }" << std::endl;
    std::cout << "\n   Problems:" << std::endl;
    std::cout << " •   Hard to read and write" << std::endl;
    std::cout << " •   Terrible error messages" << std::endl;
    std::cout << " •   Verbose and error-prone" << std::endl;

    std::cout << "\n NEW WAY (CONCEPTS):" << std::endl;
    std::cout << "   template<std::integral T>" << std::endl;
    std::cout << "   T func(T x) { ... }" << std::endl;
    std::cout << "\n   Benefits:" << std::endl;
    std::cout << " •   Clean and readable" << std::endl;
    std::cout << " •   Clear error messages" << std::endl;
    std::cout << " •   Intent is obvious" << std::endl;
    std::cout << " •   Easier to maintain" << std::endl;

    std::cout << "\n Both work the same:" << std::endl;
    std::cout << "   old_square(5) = " << old_square(5) << std::endl;
```

```
    std::cout << "   new_square(5) = " << new_square(5) << std::endl;
    std::cout << "\n Same runtime cost: ZERO" << std::endl;
}

// ==================================================================
// SECTION 10: COMPLETE EXAMPLE - THREAD-SAFE QUEUE
// ==================================================================

#include <mutex>
#include <queue>
#include <optional>

template<typename T>
concept ThreadSafeElement = requires {
    requires std::is_move_constructible_v<T> || std::is_copy_constructible_v<T
        >;
    requires std::is_destructible_v<T>;
};

template<ThreadSafeElement T>
class ThreadSafeQueue {
private:
    mutable std::mutex mutex_;
    std::queue<T> queue_;

public:
    // Perfect forwarding with concepts
    template<typename U>
    requires std::constructible_from<T, U>
    void push(U&& item) {
        std::lock_guard<std::mutex> lock(mutex_);
        queue_.push(std::forward<U>(item));
    }

    std::optional<T> try_pop() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (queue_.empty()) {
            return std::nullopt;
        }
        T item = std::move(queue_.front());
        queue_.pop();
        return item;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(mutex_);
        return queue_.empty();
    }
};

void demonstrate_complete_example() {
    std::cout << "\n=== 10. COMPLETE EXAMPLE - THREAD-SAFE QUEUE ===" << std::
        endl;

```

```
    ThreadSafeQueue<std::string> queue;

    std::cout << "\n Pushing items with perfect forwarding:" << std::endl;

    std::string lval = "lvalue string";
    queue.push(lval);                       // Copy (lvalue)
    std::cout << "   Pushed lvalue (copied)" << std::endl;

    queue.push(std::string("rvalue"));   // Move (rvalue)
    std::cout << "   Pushed rvalue (moved)" << std::endl;

    queue.push("literal");                  // Construct in place
    std::cout << "   Pushed literal (constructed)" << std::endl;

    std::cout << "\n Popping items:" << std::endl;
    while (auto item = queue.try_pop()) {
        std::cout << "   Popped: " << *item << std::endl;
    }

    std::cout << "\n Combines:" << std::endl;
    std::cout << " •   Concepts (ThreadSafeElement)" << std::endl;
    std::cout << " •   Perfect forwarding (std::forward)" << std::endl;
    std::cout << " •   requires clause (std::constructible_from)" << std::endl;
    std::cout << " •   Modern C++ (std::optional, std::mutex)" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "      ADVANCED PERFECT FORWARDING & CONCEPTS                    \n";
    std::cout << "                                                                  \n";
    std::cout << "   Topics: std::forward, requires requires, embedded systems   \n";
    std::cout << "                                    \n";

    demonstrate_perfect_forwarding();
    demonstrate_requires_syntax();
    demonstrate_requires_requires();
    demonstrate_forwarding_with_concepts();
    explain_when_to_use_requires_requires();
    demonstrate_readability();
    explain_embedded_systems();
    demonstrate_embedded_concepts();
    compare_sfinae_vs_concepts();
    demonstrate_complete_example();

    std::cout << "\n" << std::string(70, '=') << std::endl;
```

```
    std::cout << "SUMMARY:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n KEY TAKEAWAYS:" << std::endl;
    std::cout << "\n1. PERFECT FORWARDING:" << std::endl;
    std::cout << " •   Use T&& (universal reference) in templates" << std::
        endl;
    std::cout << " •   Always use std::forward<T> when forwarding" << std::
        endl;
    std::cout << " •   Preserves lvalue/rvalue value category" << std::endl;

    std::cout << "\n2. 'REQUIRES REQUIRES':" << std::endl;
    std::cout << " •   First: starts constraint clause" << std::endl;
    std::cout << " •   Second: begins expression testing" << std::endl;
    std::cout << " •   Use for testing specific operations" << std::endl;
    std::cout << " •   Prefer named concepts for readability" << std::endl;

    std::cout << "\n3. EMBEDDED SYSTEMS:" << std::endl;
    std::cout << " •   Zero runtime cost (compile-time only)" << std::endl;
    std::cout << " •   Perfect for safety-critical code" << std::endl;
    std::cout << " •   Use for register access, HAL, type safety" << std::endl
        ;
    std::cout << " •   Watch compilation time and binary size" << std::endl;

    std::cout << "\n4. BEST PRACTICES:" << std::endl;
    std::cout << " •   Use standard concepts when available" << std::endl;
    std::cout << " •   Name complex constraints as concepts" << std::endl;
    std::cout << " •   Prefer concepts over SFINAE" << std::endl;
    std::cout << " •   Document requirements clearly" << std::endl;

    std::cout << "\n Modern C++ = Type Safety + Zero Cost\n" << std::endl;

    return 0;
}
```

52 Source Code: PimplIdiom.cpp

File: `src/PimplIdiom.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// PIMPL IDIOM IN MODERN C++ - STILL RELEVANT?
// ==================================================================
// Comprehensive guide covering:
// 1. What is Pimpl (Pointer to Implementation)?
// 2. Traditional benefits and use cases
// 3. Is it still relevant in Modern C++ (C++11/14/17/20/23)?
// 4. Performance implications (especially for real-time systems)
// 5. Cache locality and memory indirection problems
// 6. Alternatives and mitigations
// 7. When to use and when to avoid Pimpl
//
// KEY QUESTION: Why Pimpl may be BAD for real-time systems?
// ANSWER: Pointer indirection breaks cache locality, causes memory jumps,
//         unpredictable latency, and extra heap allocation overhead
//
// Build: g++ -std=c++20 -Wall -Wextra -O2 -o PimplIdiom PimplIdiom.cpp
// ==================================================================

#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <chrono>
#include <iomanip>

// ==================================================================
// SECTION 1: WHAT IS THE PIMPL IDIOM?
// ==================================================================

namespace what_is_pimpl {

// ==============================
// WITHOUT PIMPL - Traditional approach
// ==============================

// Widget.h (header file)
class WidgetNoPimpl {
private:
    // ALL implementation details exposed in header
    std::string name_;
    int id_;
    std::vector<double> data_;

    // Private helper functions also exposed
    void validateData();
    void processInternal();

public:
```

```
    WidgetNoPimpl(const std::string& name, int id);
    ~WidgetNoPimpl();

    void doSomething();
    void process();
};

//   PROBLEMS WITHOUT PIMPL:
// 1. All private members visible in header (no encapsulation)
// 2. Changing private members requires recompiling ALL clients
// 3. Implementation details leak to users
// 4. Longer compile times (includes propagate)
// 5. Binary compatibility issues (ABI breaks easily)

// ==============================
// WITH PIMPL - Pointer to Implementation
// ==============================

// Widget.h (header file)
class WidgetWithPimpl {
private:
    // ONLY pointer to implementation - opaque pointer
    struct Impl;  // Forward declaration
    std::unique_ptr<Impl> pImpl_;

public:
    WidgetWithPimpl(const std::string& name, int id);
    ~WidgetWithPimpl();  // Must be in .cpp (where Impl is complete)

    // Move operations (copy disabled by default with unique_ptr)
    WidgetWithPimpl(WidgetWithPimpl&&) noexcept;
    WidgetWithPimpl& operator=(WidgetWithPimpl&&) noexcept;

    void doSomething();
    void process();
};

// Widget.cpp (implementation file)
struct WidgetWithPimpl::Impl {
    // ALL implementation details hidden here
    std::string name;
    int id;
    std::vector<double> data;

    Impl(const std::string& n, int i) : name(n), id(i) {}

    void validateData() {
        std::cout << "      Validating data for " << name << "\n";
    }

    void processInternal() {
        std::cout << "      Processing " << data.size() << " elements\n";
    }
};
```

```

// Implementation
WidgetWithPimpl::WidgetWithPimpl(const std::string& name, int id)
    : pImpl_(std::make_unique<Impl>(name, id)) {
    std::cout << "    WidgetWithPimpl constructed: " << name << "\n";
}

// Destructor must be in .cpp where Impl is complete type
WidgetWithPimpl::~WidgetWithPimpl() = default;

// Move operations
WidgetWithPimpl::WidgetWithPimpl(WidgetWithPimpl&&) noexcept = default;
WidgetWithPimpl& WidgetWithPimpl::operator=(WidgetWithPimpl&&) noexcept =
    default;

void WidgetWithPimpl::doSomething() {
    pImpl_->validateData();
    std::cout << "    Doing something with " << pImpl_->name << "\n";
}

void WidgetWithPimpl::process() {
    pImpl_->data.push_back(1.0);
    pImpl_->data.push_back(2.0);
    pImpl_->processInternal();
}

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 1: WHAT IS THE PIMPL IDIOM?\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "PIMPL = Pointer to IMPLementation (also called Opaque
        Pointer)\n\n";

    std::cout << "CONCEPT:\n";
    std::cout << " •  Public class holds ONLY a pointer to implementation\n";
    std::cout << " •  Implementation struct defined in .cpp file\n";
    std::cout << " •  All data members and private functions hidden in Impl\n"
        ;
    std::cout << " •  Forwards all operations to pImpl pointer\n\n";

    std::cout << "EXAMPLE:\n";
    WidgetWithPimpl widget("Sensor", 42);
    widget.doSomething();
    widget.process();

    std::cout << "\ n TRADITIONAL BENEFITS:\n";
    std::cout << "   1. COMPILATION FIREWALL\n";
    std::cout << " •      Change private members without recompiling clients\n
        ";
    std::cout << " •      Faster incremental builds\n";
    std::cout << " •      Reduces header dependencies\n\n";

    std::cout << "   2. BINARY COMPATIBILITY (ABI STABILITY)\n";
```

```
    std::cout << " •      Can modify Impl without breaking ABI\n";
    std::cout << " •      Important for shared libraries\n";
    std::cout << " •      No need to relink client code\n\n";

    std::cout << "   3. INFORMATION HIDING\n";
    std::cout << " •      Private members truly hidden\n";
    std::cout << " •      Can use incomplete types in Impl\n";
    std::cout << " •      Better encapsulation\n\n";

    std::cout << "   4. HEADER FILE SIMPLICITY\n";
    std::cout << " •      Minimal includes in header\n";
    std::cout << " •      Cleaner public interface\n";
    std::cout << " •      Easier to read and maintain\n";
}

} // namespace what_is_pimpl

// ==================================================================
// SECTION 2: MODERN C++ IMPROVEMENTS - STILL RELEVANT?
// ==================================================================

namespace modern_cpp_pimpl {

// C++98: Raw pointer (manual memory management)
class WidgetCpp98 {
private:
    struct Impl;
    Impl* pImpl_;  // Raw pointer

public:
    WidgetCpp98();
    ~WidgetCpp98();

    // Rule of Three needed
    WidgetCpp98(const WidgetCpp98&);
    WidgetCpp98& operator=(const WidgetCpp98&);
};

// C++11: unique_ptr (automatic memory management)
class WidgetCpp11 {
private:
    struct Impl;
    std::unique_ptr<Impl> pImpl_;  // Smart pointer

public:
    WidgetCpp11();
    ~WidgetCpp11();  // Still needed for incomplete type

    // Rule of Five: Move-only by default (unique_ptr)
    WidgetCpp11(WidgetCpp11&&) noexcept;
    WidgetCpp11& operator=(WidgetCpp11&&) noexcept;

    // Copy operations require custom implementation if needed
    WidgetCpp11(const WidgetCpp11&) = delete;
```

```
    WidgetCpp11& operator=(const WidgetCpp11&) = delete;
};

// C++11: shared_ptr (for copyable Pimpl)
class WidgetCpp11Shared {
private:
    struct Impl;
    std::shared_ptr<Impl> pImpl_;  // Shared ownership

public:
    WidgetCpp11Shared();
    ~WidgetCpp11Shared() = default;  // No explicit destructor needed!

    // Copyable and movable by default
    WidgetCpp11Shared(const WidgetCpp11Shared&) = default;
    WidgetCpp11Shared& operator=(const WidgetCpp11Shared&) = default;
    WidgetCpp11Shared(WidgetCpp11Shared&&) noexcept = default;
    WidgetCpp11Shared& operator=(WidgetCpp11Shared&&) noexcept = default;
};

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 2: MODERN C++ IMPROVEMENTS - STILL RELEVANT?\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "EVOLUTION OF PIMPL:\n\n";

    std::cout << "C++98: Raw pointer (manual memory management)\n";
    std::cout << "    Manual new/delete\n";
    std::cout << "    Must implement Rule of Three\n";
    std::cout << "    Risk of memory leaks\n";
    std::cout << "    Exception safety issues\n\n";

    std::cout << "C++11: std::unique_ptr\n";
    std::cout << "    Automatic memory management\n";
    std::cout << "    Move-only by default\n";
    std::cout << "    Exception safe\n";
    std::cout << "     Still need destructor in .cpp (incomplete type)\n\n";

    std::cout << "C++11: std::shared_ptr\n";
    std::cout << "    Automatic memory management\n";
    std::cout << "    Copyable by default\n";
    std::cout << "    No destructor needed in .cpp!\n";
    std::cout << "    Reference counting overhead\n";
    std::cout << "    Two heap allocations (control block + Impl)\n\n";

    std::cout << "IS PIMPL STILL RELEVANT IN MODERN C++?\n\n";

    std::cout << "  YES, when you need:\n";
    std::cout << "    1. BINARY STABILITY (shared libraries, plugins)\n";
    std::cout << " •      Qt framework uses Pimpl extensively\n";
    std::cout << " •      Windows API uses opaque handles (HWND, HANDLE)\n";
    std::cout << " •      Linux kernel uses opaque pointers\n\n";

```

```
    std::cout << "   2. COMPILATION FIREWALL (large projects)\n";
    std::cout << " •      Reduce compilation dependencies\n";
    std::cout << " •      Faster incremental builds\n";
    std::cout << " •      Important for multi-million line codebases\n\n";

    std::cout << "   3. INTERFACE STABILITY (APIs)\n";
    std::cout << " •      Public interface never changes\n";
    std::cout << " •      Implementation can evolve freely\n";
    std::cout << " •      Versioning becomes easier\n\n";

    std::cout << " LESS RELEVANT when:\n";
    std::cout << "   1. Using header-only libraries (templates)\n";
    std::cout << "   2. Internal classes (not exposed in API)\n";
    std::cout << "   3. Performance-critical code (real-time systems)\n";
    std::cout << "   4. Small classes with simple members\n";
    std::cout << "   5. Using modules (C++20) - reduces header dependencies\n"
        ;
}

} // namespace modern_cpp_pimpl

// ==================================================================
// SECTION 3: PERFORMANCE PROBLEMS - THE REAL-TIME KILLER
// ==================================================================

namespace performance_problems {

// Simple data structure for testing
struct SensorData {
    int id;
    double value;
    int64_t timestamp;

    SensorData(int i = 0, double v = 0.0)
        : id(i), value(v), timestamp(0) {}
};

// ==============================
// VERSION 1: NO PIMPL - Direct data access
// ==============================

class SensorNoPimpl {
private:
    int id_;
    double value_;
    int64_t timestamp_;

public:
    SensorNoPimpl(int id, double value)
        : id_(id), value_(value), timestamp_(0) {}

    void update(double new_value) {
        value_ = new_value;
        timestamp_++;
```

```
    }

    double getValue() const { return value_; }
    int getId() const { return id_; }
};

// ==============================
// VERSION 2: WITH PIMPL - Pointer indirection
// ==============================

class SensorWithPimpl {
private:
    struct Impl {
        int id;
        double value;
        int64_t timestamp;

        Impl(int i, double v) : id(i), value(v), timestamp(0) {}
    };

    std::unique_ptr<Impl> pImpl_;

public:
    SensorWithPimpl(int id, double value)
        : pImpl_(std::make_unique<Impl>(id, value)) {}

    ~SensorWithPimpl() = default;

    void update(double new_value) {
        pImpl_->value = new_value;
        pImpl_->timestamp++;
    }

    double getValue() const { return pImpl_->value; }
    int getId() const { return pImpl_->id; }
};

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 3: PERFORMANCE PROBLEMS - THE REAL-TIME KILLER\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "PROBLEM 1: POINTER INDIRECTION OVERHEAD\n\n";

    std::cout << "Without Pimpl:\n";
    std::cout << "  sensor.getValue() →  Direct access: sensor.value_\n";
    std::cout << " •  ONE memory access\n";
    std::cout << " •  CPU loads value directly from sensor object\n";
    std::cout << " •  Predictable: O(1) memory access\n\n";

    std::cout << "With Pimpl:\n";
    std::cout << "  sensor.getValue() →  sensor.pImpl_->value\n";
    std::cout << " •  TWO memory accesses:\n";
    std::cout << "    1. Load pImpl_ pointer from sensor object\n";
```

```
    std::cout << "    2. Follow pointer to load value from Impl\n";
    std::cout << " •  Unpredictable: pointer could point anywhere in heap\n\n"
        ;

    std::cout << "PROBLEM 2: CACHE LOCALITY DESTROYED\n\n";

    std::cout << "Without Pimpl (Array of sensors):\n";
    std::cout << "  Memory layout:  [Sensor1][Sensor2][Sensor3][Sensor4]\n";
    std::cout << " •  Contiguous memory (cache-friendly)\n";
    std::cout << " •  CPU prefetcher loads next sensors automatically\n";
    std::cout << " •  All data in same cache line (64 bytes)\n\n";

    std::cout << "With Pimpl (Array of sensors):\n";
    std::cout << "  Memory layout:  [Sensor1→*]Impl1  [Sensor2→*]Impl2\n";
    std::cout << "                  scattered          scattered\n";
    std::cout << "                  in heap            in heap\n";
    std::cout << " •  Impl objects scattered across heap (cache-hostile)\n";
    std::cout << " •  Each access causes cache miss\n";
    std::cout << " •  CPU prefetcher cannot predict pattern\n";
    std::cout << " •  Every pointer dereference = potential cache miss\n\n";

    std::cout << "PROBLEM 3: HEAP ALLOCATION OVERHEAD\n\n";

    std::cout << "Without Pimpl:\n";
    std::cout << "  SensorNoPimpl sensor(1, 3.14);  // Stack allocation\n";
    std::cout << " •  O(1) construction time\n";
    std::cout << " •  No malloc/free overhead\n";
    std::cout << " •  Deterministic\n\n";

    std::cout << "With Pimpl:\n";
    std::cout << "  SensorWithPimpl sensor(1, 3.14);  // Heap allocation!\n";
    std::cout << " •  Calls malloc() for Impl\n";
    std::cout << " •  O(?) allocation time (non-deterministic)\n";
    std::cout << " •  Heap fragmentation over time\n";
    std::cout << " •  Destructor calls free() (non-deterministic)\n\n";

    // Benchmark
    constexpr int ITERATIONS = 1000000;

    std::cout << "BENCHMARK: " << ITERATIONS << " operations\n\n";

    // Test 1: No Pimpl
    {
        SensorNoPimpl sensor(1, 100.0);
        auto start = std::chrono::high_resolution_clock::now();

        for (int i = 0; i < ITERATIONS; ++i) {
            sensor.update(i * 0.1);
            [[maybe_unused]] volatile double val = sensor.getValue();
        }

        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
            end - start);
```

```

        std::cout << "  WITHOUT Pimpl: " << std::setw(8) << duration.count() << "  s\n";
    }

    // Test 2: With Pimpl
    {
        SensorWithPimpl sensor(1, 100.0);
        auto start = std::chrono::high_resolution_clock::now();

        for (int i = 0; i < ITERATIONS; ++i) {
            sensor.update(i * 0.1);
            [[maybe_unused]] volatile double val = sensor.getValue();
        }

        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        std::cout << "  WITH Pimpl:    " << std::setw(8) << duration.count() << "  s\n";
    }

    std::cout << "\n PIMPL OVERHEAD:\n";
    std::cout << " •   50-200% slower (depends on cache behavior)\n";
    std::cout << " •   Non-deterministic timing variations\n";
    std::cout << " •   Extra heap allocation per object\n";
    std::cout << " •   Destroys cache locality\n";
}

} // namespace performance_problems

// ==================================================================
// SECTION 4: WHY PIMPL IS BAD FOR REAL-TIME SYSTEMS
// ==================================================================

namespace realtime_problems {

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 4: WHY PIMPL IS BAD FOR REAL-TIME SYSTEMS\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "REAL-TIME SYSTEM REQUIREMENTS:\n";
    std::cout << "  1. DETERMINISTIC TIMING - Predictable worst-case execution time (WCET)\n";
    std::cout << "  2. BOUNDED LATENCY - Maximum response time guaranteed\n";
    std::cout << "  3. NO HEAP ALLOCATION - At runtime (malloc/free non-deterministic)\n";
    std::cout << "  4. CACHE PREDICTABILITY - Minimize cache misses\n";
    std::cout << "  5. NO MEMORY JUMPS - Sequential memory access for prefetcher\n\n";

    std::cout << "PIMPL VIOLATIONS:\n\n";
```

```

    std::cout << "  VIOLATION 1: NON-DETERMINISTIC HEAP ALLOCATION\n";
    std::cout << "    Problem:\n";
    std::cout << " •     Each Pimpl object requires malloc() for Impl\n";
    std::cout << " •     malloc() time depends on heap state (fragmentation)\n";
    std::cout << " •     WCET becomes unpredictable\n";
    std::cout << "    Impact on Real-Time:\n";
    std::cout << " •     Cannot prove deadline guarantees\n";
    std::cout << " •     Violates MISRA C++ Rule 18-4-1 (no dynamic allocation)\n";
    std::cout << " •     Violates ISO 26262 (automotive safety) guidelines\n";
    std::cout << " •     Unacceptable for hard real-time (ASIL-D)\n\n";

    std::cout << "  VIOLATION 2: POINTER INDIRECTION = CACHE MISSES\n";
    std::cout << "    Problem:\n";
    std::cout << " •     Every operation requires following pImpl_ pointer\n";
    std::cout << " •     Impl object allocated somewhere in heap\n";
    std::cout << " •     High probability of cache miss on every access\n";
    std::cout << "    Impact on Real-Time:\n";
    std::cout << " •     Cache miss = 100+ cycles (vs 1 cycle for cache hit)\n";
    std::cout << " •     Unpredictable latency spikes\n";
    std::cout << " •     Worst-case timing analysis becomes pessimistic\n";
    std::cout << " •     May cause deadline misses in tight loops\n\n";

    std::cout << "  VIOLATION 3: DESTROYED CACHE LOCALITY\n";
    std::cout << "    Problem:\n";
    std::cout << " •     Array of Pimpl objects:\n";
    std::cout << "       [Obj1 ptr→]Impl1 (heap addr 0x1000)\n";
    std::cout << "       [Obj2 ptr→]Impl2 (heap addr 0x5000) ← 16KB away!\n";
    std::cout << "       [Obj3 ptr→]Impl3 (heap addr 0x2000)\n";
    std::cout << " •     Each Impl scattered randomly in heap\n";
    std::cout << " •     CPU prefetcher cannot help\n";
    std::cout << "    Impact on Real-Time:\n";
    std::cout << " •     Sequential access becomes random access\n";
    std::cout << " •     10-100x slower than contiguous data\n";
    std::cout << " •     Unacceptable for sensor processing, control loops\n\n";

    std::cout << "  VIOLATION 4: HEAP FRAGMENTATION OVER TIME\n";
    std::cout << "    Problem:\n";
    std::cout << " •     Repeated allocation/deallocation of Impl objects\n";
    std::cout << " •     Heap becomes fragmented (Swiss cheese memory)\n";
    std::cout << " •     Allocation time increases over time\n";
    std::cout << "    Impact on Real-Time:\n";
    std::cout << " •     System degrades over runtime\n";
    std::cout << " •     May fail after hours/days of operation\n";
    std::cout << " •     Cannot certify for long-running systems\n";
    std::cout << " •     Requires periodic restart (unacceptable)\n\n";

    std::cout << "  VIOLATION 5: DOUBLE DEALLOCATION (shared_ptr Pimpl)\n";
    std::cout << "    Problem:\n";
    std::cout << " •     shared_ptr has control block (heap allocation)\n";
```

```
    std::cout << " •     Plus Impl object (another heap allocation)\n";
    std::cout << " •     TWO heap operations per object\n";
    std::cout << " •     Reference counting overhead on every copy\n";
    std::cout << "   Impact on Real-Time:\n";
    std::cout << " •     Doubles the allocation overhead\n";
    std::cout << " •     Atomic operations for ref counting (slow)\n";
    std::cout << " •     Even worse for real-time systems\n\n";

    std::cout << "REAL-WORLD EXAMPLE:\n\n";
    std::cout << "Scenario: Automotive ECU processing 100 sensor values at 1 kHz\n\n";

    std::cout << "WITHOUT Pimpl:\n";
    std::cout << " •  struct Sensor { int id; double value; };\n";
    std::cout << " •  std::array<Sensor, 100> sensors;  // Contiguous, no heap\n";
    std::cout << " •  for (auto& s : sensors) process(s.value);  // Cache-friendly\n";
    std::cout << " •  Total time: ~500 cycles (all in cache)\n";
    std::cout << " •  Deterministic: Always 500 ± 10 cycles\n";
    std::cout << "   PASSES WCET analysis\n\n";

    std::cout << "WITH Pimpl:\n";
    std::cout << " •  class Sensor { unique_ptr<Impl> pImpl_; };\n";
    std::cout << " •  std::array<Sensor, 100> sensors;  // 100 heap allocations!\n";
    std::cout << " •  for (auto& s : sensors) process(s->value);  // Cache misses\n";
    std::cout << " •  Total time: ~5000-50000 cycles (cache misses)\n";
    std::cout << " •  Non-deterministic: 5000-50000 cycles (10x variation!)\n";
    std::cout << "   FAILS WCET analysis (unpredictable)\n\n";

    std::cout << "STANDARDS VIOLATIONS:\n\n";
    std::cout << " •  MISRA C++ Rule 18-4-1: Dynamic heap allocation shall not be used\n";
    std::cout << " •  AUTOSAR A18-5-1: malloc/calloc/realloc/free shall not be used\n";
    std::cout << " •  ISO 26262 (ASIL-D): Static allocation required\n";
    std::cout << " •  DO-178C Level A: Dynamic allocation discouraged\n";
    std::cout << " •  JSF AV C++ Rule 207: new/delete not used after initialization\n\n";

    std::cout << "VERDICT:  PIMPL IS FORBIDDEN IN REAL-TIME SYSTEMS\n";
}

} // namespace realtime_problems

// ==================================================================
// SECTION 5: ALTERNATIVES AND MITIGATIONS
// ==================================================================

namespace alternatives {

```

```cpp
// ==============================
// ALTERNATIVE 1: NO PIMPL - Expose implementation (if acceptable)
// ==============================

class DirectImplementation {
private:
    int id_;
    double value_;

public:
    DirectImplementation(int id, double value) : id_(id), value_(value) {}

    void process() { value_ *= 2.0; }
    double getValue() const { return value_; }
};

//   Pros: Fast, cache-friendly, deterministic
//   Cons: Breaks ABI on changes, longer compile times

// ==============================
// ALTERNATIVE 2: FAST PIMPL - Pre-allocated storage
// ==============================

template<typename T, size_t Size, size_t Alignment = alignof(std::max_align_t)
    >
class FastPimpl {
private:
    alignas(Alignment) std::byte storage_[Size];

    T* ptr() { return reinterpret_cast<T*>(storage_); }
    const T* ptr() const { return reinterpret_cast<const T*>(storage_); }

public:
    template<typename... Args>
    FastPimpl(Args&&... args) {
        static_assert(sizeof(T) <= Size, "Storage too small");
        static_assert(alignof(T) <= Alignment, "Alignment too small");
        new (storage_) T(std::forward<Args>(args)...);
    }

    ~FastPimpl() { ptr()->~T(); }

    // Delete copy/move (or implement if needed)
    FastPimpl(const FastPimpl&) = delete;
    FastPimpl& operator=(const FastPimpl&) = delete;

    T* operator->() { return ptr(); }
    const T* operator->() const { return ptr(); }

    T& operator*() { return *ptr(); }
    const T& operator*() const { return *ptr(); }
};

class SensorFastPimpl {
```

```
private:
    struct Impl {
        int id;
        double value;
        Impl(int i, double v) : id(i), value(v) {}
        void process() { value *= 2.0; }
    };

    FastPimpl<Impl, 32> pImpl_;  // Storage in object itself!

public:
    SensorFastPimpl(int id, double value) : pImpl_(id, value) {}

    void process() { pImpl_->process(); }
    double getValue() const { return pImpl_->value; }
};

//   Pros: NO heap allocation, cache-friendly, ABI stable (if size doesn't
    change)
//    Cons: Must know maximum Impl size at compile time

// ==============================
// ALTERNATIVE 3: Type Erasure (std::function style)
// ==============================

class TypeErasedSensor {
private:
    struct ConceptBase {
        virtual ~ConceptBase() = default;
        virtual void process() = 0;
        virtual double getValue() const = 0;
        virtual std::unique_ptr<ConceptBase> clone() const = 0;
    };

    template<typename T>
    struct ConceptModel : ConceptBase {
        T data;

        ConceptModel(T d) : data(std::move(d)) {}

        void process() override { data.process(); }
        double getValue() const override { return data.getValue(); }
        std::unique_ptr<ConceptBase> clone() const override {
            return std::make_unique<ConceptModel<T>>(data);
        }
    };

    std::unique_ptr<ConceptBase> pImpl_;

public:
    template<typename T>
    TypeErasedSensor(T sensor)
        : pImpl_(std::make_unique<ConceptModel<T>>(std::move(sensor))) {}

```

```
    void process() { pImpl_->process(); }
    double getValue() const { return pImpl_->getValue(); }
};

//   Pros: Flexible, any type works, no template in interface
//   Cons: Still uses heap, virtual function overhead

// ==============================
// ALTERNATIVE 4: SBO (Small Buffer Optimization) Pimpl
// ==============================

template<size_t BufferSize = 64>
class SBOPimpl {
private:
    alignas(std::max_align_t) std::byte buffer_[BufferSize];
    void* heap_ptr_ = nullptr;

    void* storage() { return heap_ptr_ ? heap_ptr_ : buffer_; }
    const void* storage() const { return heap_ptr_ ? heap_ptr_ : buffer_; }

public:
    template<typename T, typename... Args>
    void emplace(Args&&... args) {
        if (sizeof(T) <= BufferSize && alignof(T) <= alignof(std::max_align_t)
            ) {
            // Small: use buffer (no heap)
            new (buffer_) T(std::forward<Args>(args)...);
            heap_ptr_ = nullptr;
        } else {
            // Large: use heap
            heap_ptr_ = new T(std::forward<Args>(args)...);
        }
    }

    template<typename T>
    T* get() { return static_cast<T*>(storage()); }

    template<typename T>
    const T* get() const { return static_cast<const T*>(storage()); }

    ~SBOPimpl() {
        if (heap_ptr_) {
            // Must know type to delete - requires type erasure
        }
    }
};

//   Pros: Avoids heap for small objects
//    Cons: Complex, requires type erasure for destruction

// ==============================
// ALTERNATIVE 5: Interface Segregation + Direct Storage
// ==============================

```

```cpp
// Pure interface (header file)
class ISensor {
public:
    virtual ~ISensor() = default;
    virtual void process() = 0;
    virtual double getValue() const = 0;
};

// Concrete implementation (cpp file)
class ConcreteSensor : public ISensor {
private:
    int id_;
    double value_;

public:
    ConcreteSensor(int id, double value) : id_(id), value_(value) {}

    void process() override { value_ *= 2.0; }
    double getValue() const override { return value_; }
};

// Factory function (cpp file)
std::unique_ptr<ISensor> createSensor(int id, double value) {
    return std::make_unique<ConcreteSensor>(id, value);
}

//   Pros: Clean interface, implementation hidden
//   Cons: Still uses heap, virtual function overhead

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 5: ALTERNATIVES AND MITIGATIONS\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "ALTERNATIVE 1: NO PIMPL (Direct Implementation)\n";
    std::cout << "    BEST for performance and real-time systems\n";
    std::cout << "    Zero overhead, cache-friendly\n";
    std::cout << "    Breaks ABI on changes\n";
    std::cout << "    Longer compile times\n";
    std::cout << "  USE WHEN: Performance critical, internal classes\n\n";

    DirectImplementation direct(1, 100.0);
    direct.process();
    std::cout << "  Example: Direct value = " << direct.getValue() << "\n\n";

    std::cout << "ALTERNATIVE 2: FAST PIMPL (Pre-allocated storage)\n";
    std::cout << "    NO heap allocation (storage in object)\n";
    std::cout << "    Cache-friendly (contiguous memory)\n";
    std::cout << "    ABI stable (if size doesn't change)\n";
    std::cout << "     Must know maximum Impl size\n";
    std::cout << "  USE WHEN: Need ABI stability + performance\n\n";

    SensorFastPimpl fast(1, 100.0);
    fast.process();
```

```
    std::cout << "  Example: FastPimpl value = " << fast.getValue() << "\n\n";

    std::cout << "ALTERNATIVE 3: TYPE ERASURE\n";
    std::cout << "    Flexible (any type works)\n";
    std::cout << "    Clean generic interface\n";
    std::cout << "    Still uses heap\n";
    std::cout << "    Virtual function overhead\n";
    std::cout << "  USE WHEN: Need runtime polymorphism\n\n";

    std::cout << "ALTERNATIVE 4: SBO (Small Buffer Optimization)\n";
    std::cout << "    Avoids heap for small objects\n";
    std::cout << "    Fallback to heap for large objects\n";
    std::cout << "    Complex implementation\n";
    std::cout << "    Similar to std::string SSO\n";
    std::cout << "  USE WHEN: Mix of small and large objects\n\n";

    std::cout << "ALTERNATIVE 5: INTERFACE + FACTORY\n";
    std::cout << "    Clean separation of interface/implementation\n";
    std::cout << "    Implementation hidden in .cpp\n";
    std::cout << "    Still uses heap\n";
    std::cout << "    Virtual function overhead\n";
    std::cout << "  USE WHEN: Need polymorphism + information hiding\n\n";

    std::cout << "MITIGATION STRATEGIES FOR REAL-TIME:\n\n";

    std::cout << "1. INITIALIZATION PHASE PATTERN\n";
    std::cout << " •   Allocate ALL Pimpl objects during initialization\n";
    std::cout << " •   Use object pools (pre-allocated)\n";
    std::cout << " •   Never allocate/deallocate during runtime\n";
    std::cout << " •   Acceptable: init phase not time-critical\n\n";

    std::cout << "2. CUSTOM POOL ALLOCATOR\n";
    std::cout << " •   Pre-allocate pool of Impl objects\n";
    std::cout << " •   Custom allocator for unique_ptr<Impl>\n";
    std::cout << " •   Bounded, deterministic allocation\n";
    std::cout << " •   Improves cache locality (pool in contiguous memory)\n\n
        ";

    std::cout << "3. PLACEMENT NEW IN FIXED BUFFER\n";
    std::cout << " •   Static buffer for all Impl objects\n";
    std::cout << " •   Placement new to construct in buffer\n";
    std::cout << " •   Zero heap allocation\n";
    std::cout << " •   Requires careful lifetime management\n\n";

    std::cout << "4. C++20 MODULES\n";
    std::cout << " •   Reduce need for Pimpl (compilation firewall)\n";
    std::cout << " •   Faster builds without Pimpl overhead\n";
    std::cout << " •   Still evolving, limited tooling support\n\n";

    std::cout << "RECOMMENDATION FOR REAL-TIME SYSTEMS:\n\n";
    std::cout << "    FIRST CHOICE: No Pimpl (Direct implementation)\n";
    std::cout << " •     Best performance\n";
    std::cout << " •     Deterministic\n";
    std::cout << " •     Accept longer compile times as trade-off\n\n";
```

```

    std::cout << "    SECOND CHOICE: FastPimpl (in-object storage)\n";
    std::cout << " •     Good compromise\n";
    std::cout << " •     No heap, ABI stable\n";
    std::cout << " •     Requires careful sizing\n\n";

    std::cout << "     THIRD CHOICE: Pimpl with pool allocator\n";
    std::cout << " •     Acceptable with pre-allocation\n";
    std::cout << " •     Only during initialization phase\n";
    std::cout << " •     Requires justification for certification\n\n";

    std::cout << "    NEVER: Traditional heap-based Pimpl at runtime\n";
    std::cout << " •     Non-deterministic\n";
    std::cout << " •     Violates real-time standards\n";
    std::cout << " •     Unacceptable for safety-critical systems\n";
}

} // namespace alternatives

// ==================================================================
// SECTION 6: DECISION GUIDE
// ==================================================================

namespace decision_guide {

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 6: WHEN TO USE PIMPL - DECISION GUIDE\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "DECISION TREE:\n\n";

    std::cout << "  Q1: Is this a real-time or safety-critical system?\n";
    std::cout << " \n";
    std::cout << "  YES →  AVOID PIMPL (use direct implementation or FastPimpl)\n";
    std::cout << "         Reason: Non-deterministic, cache-hostile, heap allocation\n";
    std::cout << " \n";
    std::cout << "  NO → Continue to Q2\n\n";

    std::cout << "  Q2: Is this a public API/library interface?\n";
    std::cout << " \n";
    std::cout << "  YES → Continue to Q3\n";
    std::cout << " \n";
    std::cout << "  NO →  DON'T USE PIMPL (internal class - direct impl)\n\n";

    std::cout << "  Q3: Do you need ABI (binary) stability?\n";
    std::cout << " \n";
    std::cout << "  YES → Continue to Q4\n";
    std::cout << " \n";
    std::cout << "  NO →  DON'T USE PIMPL (recompilation acceptable)\n\n";

    std::cout << "  Q4: Is the class performance-critical (hot path)?\n";
```

```
    std::cout << " \n";
    std::cout << "  YES →   RECONSIDER Pimpl (profile first, consider FastPimpl)\n";
    std::cout << " \n";
    std::cout << "  NO → Continue to Q5\n\n";

    std::cout << "  Q5: Is the implementation complex/large?\n";
    std::cout << " \n";
    std::cout << "  YES →  USE PIMPL (good candidate)\n";
    std::cout << " \n";
    std::cout << "  NO →   Probably not worth it (overhead > benefit)\n\n";

    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "COMPARISON TABLE:\n\n";
    std::cout << "                                    \n";
    std::cout << "  CRITERION           NO PIMPL   PIMPL    FAST PIMPL   REAL-TIME OK?  \n";
    std::cout << "                                    \n";
    std::cout << "  Performance         BEST       POOR     GOOD          /   /  \n";
    std::cout << "  Cache Locality      BEST       WORST    BEST          /   /  \n";
    std::cout << "  Heap Allocation     NO         YES      NO            /   /  \n";
    std::cout << "  ABI Stability       NO         YES      PARTIAL      N/A  \n";
    std::cout << "  Compile Time        SLOW       FAST     FAST         N/A  \n";
    std::cout << "  Information Hiding  NO         YES      YES          N/A  \n";
    std::cout << "  Complexity          LOW        MEDIUM   HIGH         N/A  \n";
    std::cout << "  Memory Overhead     0 bytes    8 bytes Impl size    N/A  \n";
    std::cout << "                                    \n\n";

    std::cout << "USE PIMPL WHEN:\n\n";
    std::cout << "    Shipping shared libraries (ABI stability critical)\n";
    std::cout << "      Examples: Qt, wxWidgets, system libraries\n\n";

    std::cout << "    Plugin architectures (binary compatibility)\n";
    std::cout << "      Examples: Audio plugins, game engines\n\n";

    std::cout << "    Large complex implementations (compilation firewall)\n";
    std::cout << "      Examples: GUI frameworks, network libraries\n\n";

    std::cout << "    Platform-specific code (hide platform details)\n";
    std::cout << "      Examples: Cross-platform libraries\n\n";

    std::cout << "AVOID PIMPL WHEN:\n\n";
    std::cout << "    Real-time systems (non-deterministic)\n";
    std::cout << "      Examples: Control systems, DSP, robotics\n\n";

```

```
    std::cout << "    Performance-critical code (cache-hostile)\n";
    std::cout << "      Examples: Game engines, HFT, rendering\n\n";

    std::cout << "    Safety-critical systems (heap allocation)\n";
    std::cout << "      Examples: Automotive (ISO 26262), avionics (DO-178C)\n\n";

    std::cout << "    Small simple classes (overhead > benefit)\n";
    std::cout << "      Examples: POD types, value types\n\n";

    std::cout << "    Internal implementation classes (not exposed)\n";
    std::cout << "      Examples: Private helpers, implementation details\n\n";

    std::cout << "    Header-only libraries (templates)\n";
    std::cout << "      Examples: Template libraries, generic code\n\n";

    std::cout << "    Using C++20 modules (reduces need)\n";
    std::cout << "      Examples: Modern codebases with module support\n\n";

    std::cout << "FAMOUS LIBRARIES USING PIMPL:\n\n";
    std::cout << " •  Qt Framework - Extensive use for ABI stability\n";
    std::cout << " •  Boost.Asio - Some classes use Pimpl\n";
    std::cout << " •  Windows API - HWND, HANDLE (opaque pointers)\n";
    std::cout << " •  OpenSSL - Internal structures hidden\n";
    std::cout << " •  wxWidgets - Cross-platform abstraction\n\n";

    std::cout << "GOLDEN RULES:\n\n";
    std::cout << "  1. Pimpl is a DESIGN PATTERN, not a default choice\n";
    std::cout << "  2. Only use when benefits outweigh performance cost\n";
    std::cout << "  3. Profile before and after - measure the impact\n";
    std::cout << "  4. Consider FastPimpl for real-time constraints\n";
    std::cout << "  5. Never use traditional Pimpl in real-time code\n";
    std::cout << "  6. C++20 modules may reduce need for Pimpl\n";
}

} // namespace decision_guide

// ==================================================================
// MAIN
// ==================================================================

int main() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "PIMPL IDIOM IN MODERN C++ - STILL RELEVANT?\n";
    std::cout << "Performance Impact on Real-Time Systems\n";
    std::cout << std::string(80, '=') << "\n";

    what_is_pimpl::demonstrate();
    modern_cpp_pimpl::demonstrate();
    performance_problems::demonstrate();
    realtime_problems::demonstrate();
    alternatives::demonstrate();
    decision_guide::demonstrate();

```

```
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "FINAL VERDICT\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "IS PIMPL STILL RELEVANT IN MODERN C++?\n\n";

    std::cout << "  YES, for:\n";
    std::cout << " •   Shared libraries requiring ABI stability\n";
    std::cout << " •   Large projects needing compilation firewalls\n";
    std::cout << " •   Public APIs with evolving implementation\n";
    std::cout << " •   Modern C++ makes it easier (unique_ptr, shared_ptr)\n\n";

    std::cout << "  NO (AVOID), for:\n";
    std::cout << " •   Real-time systems - NON-DETERMINISTIC\n";
    std::cout << " •   Safety-critical systems - HEAP ALLOCATION\n";
    std::cout << " •   Performance-critical code - CACHE-HOSTILE\n";
    std::cout << " •   Simple classes - OVERHEAD > BENEFIT\n\n";

    std::cout << "WHY PIMPL IS BAD FOR REAL-TIME:\n\n";
    std::cout << "  1. POINTER INDIRECTION - Extra memory access (cache miss)\n";
    std::cout << "  2. DESTROYED CACHE LOCALITY - Scattered Impl objects in heap\n";
    std::cout << "  3. HEAP ALLOCATION - Non-deterministic malloc/free\n";
    std::cout << "  4. MEMORY JUMPS - CPU prefetcher cannot help\n";
    std::cout << "  5. FRAGMENTATION - System degrades over time\n\n";

    std::cout << "MITIGATION OPTIONS:\n\n";
    std::cout << "    FastPimpl (in-object storage) - Zero heap, cache-friendly\n";
    std::cout << "    Pre-allocated pools - Bounded allocation during init\n";
    std::cout << "    No Pimpl - Accept longer compile times for performance\n\n";

    std::cout << "BOTTOM LINE:\n\n";
    std::cout << "  \"Pimpl is still relevant for APIs and libraries,\n";
    std::cout << "   but should be AVOIDED in real-time and performance-critical code.\n";
    std::cout << "   The pointer indirection and heap allocation make it\n";
    std::cout << "   fundamentally incompatible with deterministic real-time systems.\"\n\n";

    std::cout << std::string(80, '=') << "\n\n";

    return 0;
}
```

53 Source Code: ProtobufExample.cpp

File: `src/ProtobufExample.cpp`
Repository: View on GitHub

```
// ==================================================================
// GOOGLE PROTOCOL BUFFERS (PROTOBUF) EXAMPLE
// ==================================================================
// This example demonstrates Protocol Buffers usage in modern C++.
//
// TOPICS COVERED:
// 1. Creating and populating protobuf messages
// 2. Serialization to binary format
// 3. Deserialization from binary format
// 4. JSON conversion (protobuf v3 feature)
// 5. Nested messages and repeated fields
// 6. Enums and oneof fields
// 7. Performance and memory efficiency
// 8. Best practices for embedded systems
//
// WHAT IS PROTOCOL BUFFERS?
// - Language-neutral, platform-neutral serialization format
// - Smaller, faster, and simpler than XML/JSON
// - Generates code for multiple languages (C++, Python, Java, Go, etc.)
// - Backward/forward compatible with schema evolution
// - Used by Google, Netflix, Uber, and many others
//
// WHY USE PROTOBUF?
//   Efficient: 3-10x smaller than XML, 20-100x faster
//   Type-safe: Strong typing with generated code
//   Versioned: Schema evolution with backward compatibility
//   Cross-language: Same .proto file works for multiple languages
//   Validated: Automatic validation of message structure
//
// BUILD REQUIREMENTS:
// - Google Protocol Buffers library (libprotobuf-dev)
// - protoc compiler to generate C++ code from .proto files
//
// ==================================================================

#include <iostream>
#include <fstream>
#include <memory>
#include <chrono>
#include <iomanip>

// Include generated protobuf headers
// These are generated by protoc from sensor_data.proto
// #include "sensor_data.pb.h"

// Include generated protobuf headers
// These are generated by protoc from sensor_data.proto
#include "sensor_data.pb.h"

```

```
// ==================================================================
// EXAMPLE 1: CREATING AND POPULATING MESSAGES
// ==================================================================

void example_create_sensor_reading() {
    std::cout << "=== Example 1: Creating Sensor Reading ===\n";

    // Create a temperature sensor reading
    sensors::SensorReading reading;

    // Set basic fields
    reading.set_type(sensors::TEMPERATURE);
    reading.set_device_id("sensor_001");

    // Set timestamp (current time)
    auto* timestamp = reading.mutable_timestamp();
    auto now = std::chrono::system_clock::now();
    auto epoch = now.time_since_epoch();
    timestamp->set_seconds(std::chrono::duration_cast<std::chrono::seconds>(
        epoch).count());
    timestamp->set_nanoseconds(0);

    // Set temperature measurement (using oneof field)
    reading.set_temperature_celsius(23.5f);

    // Add metadata
    (*reading.mutable_metadata())["location"] = "office";
    (*reading.mutable_metadata())["calibrated"] = "true";

    std::cout << "Created sensor reading:\n";
    std::cout << "  Device: " << reading.device_id() << "\n";
    std::cout << "  Type: " << reading.type() << "\n";
    std::cout << "  Temperature: " << reading.temperature_celsius() << "°C\n";
    std::cout << "  Metadata:\n";
    for (const auto& [key, value] : reading.metadata()) {
        std::cout << "    " << key << " = " << value << "\n";
    }

    std::cout << "\nKey features demonstrated:\n";
    std::cout << "   Simple field setters (set_xxx)\n";
    std::cout << "   Nested message creation (mutable_xxx)\n";
    std::cout << "   Oneof field (only one measurement type at a time)\n";
    std::cout << "   Map fields for flexible metadata\n\n";
}

// ==================================================================
// EXAMPLE 2: BINARY SERIALIZATION
// ==================================================================

void example_binary_serialization() {
    std::cout << "=== Example 2: Binary Serialization ===\n";

    sensors::SensorReading reading;
    reading.set_type(sensors::TEMPERATURE);
```

```
    reading.set_device_id("sensor_001");
    reading.set_temperature_celsius(23.5f);

    // Method 1: Serialize to string
    std::string serialized;
    if (!reading.SerializeToString(&serialized)) {
        std::cerr << "Failed to serialize!\n";
        return;
    }

    std::cout << "Serialized size: " << serialized.size() << " bytes\n";
    std::cout << "Binary data (hex): ";
    for (size_t i = 0; i < std::min(serialized.size(), size_t(20)); ++i) {
        printf("%02x ", (unsigned char)serialized[i]);
    }
    if (serialized.size() > 20) std::cout << "...";
    std::cout << "\n";

    // Method 2: Serialize to file
    std::ofstream output("sensor_data.bin", std::ios::binary);
    if (!reading.SerializeToOstream(&output)) {
        std::cerr << "Failed to write to file!\n";
        return;
    }
    output.close();
    std::cout << "  Written to sensor_data.bin\n";

    // Method 3: Serialize to byte array (for embedded systems)
    size_t size = reading.ByteSizeLong();
    std::vector<uint8_t> buffer(size);
    if (!reading.SerializeToArray(buffer.data(), size)) {
        std::cerr << "Failed to serialize to array!\n";
        return;
    }
    std::cout << "  Serialized to byte array (" << size << " bytes)\n";

    std::cout << "\nSerialization methods:\n";
    std::cout << "  1. SerializeToString() - for std::string storage\n";
    std::cout << "  2. SerializeToOstream() - for file I/O\n";
    std::cout << "  3. SerializeToArray() - for fixed buffers (embedded)\n";
    std::cout << "  4. ByteSizeLong() - get serialized size before writing\n\n
        ";
}

// ==================================================================
// EXAMPLE 3: BINARY DESERIALIZATION
// ==================================================================

void example_binary_deserialization() {
    std::cout << "=== Example 3: Binary Deserialization ===\n";

    // First, create and serialize a message
    sensors::SensorReading original;
    original.set_type(sensors::HUMIDITY);
```

```
    original.set_device_id("sensor_002");
    original.set_humidity_percent(65.3f);

    std::string serialized_data;
    original.SerializeToString(&serialized_data);
    std::cout << "Original message serialized (" << serialized_data.size() << " bytes)\n";

    // Method 1: Deserialize from string
    sensors::SensorReading reading;
    if (!reading.ParseFromString(serialized_data)) {
        std::cerr << "Failed to parse!\n";
        return;
    }
    std::cout << "  Deserialized from string:\n";
    std::cout << "   Device: " << reading.device_id() << "\n";
    std::cout << "   Humidity: " << reading.humidity_percent() << "%\n";

    // Method 2: Deserialize from file
    std::ifstream input("sensor_data.bin", std::ios::binary);
    sensors::SensorReading reading2;
    if (input.good() && reading2.ParseFromIstream(&input)) {
        std::cout << "  Deserialized from file:\n";
        std::cout << "   Device: " << reading2.device_id() << "\n";
        std::cout << "   Temperature: " << reading2.temperature_celsius() << "°C\n";
    }
    input.close();

    // Method 3: Deserialize from byte array
    std::vector<uint8_t> buffer(serialized_data.begin(), serialized_data.end());
    sensors::SensorReading reading3;
    if (!reading3.ParseFromArray(buffer.data(), buffer.size())) {
        std::cerr << "Failed to parse from array!\n";
        return;
    }
    std::cout << "  Deserialized from byte array: " << reading3.device_id() << "\n";

    // Demonstrate field checking
    if (reading3.has_humidity_percent()) {
        std::cout << "   Has humidity field: " << reading3.humidity_percent() << "%\n";
    }

    std::cout << "\nDeserialization methods:\n";
    std::cout << "   1. ParseFromString() - from std::string\n";
    std::cout << "   2. ParseFromIstream() - from file\n";
    std::cout << "   3. ParseFromArray() - from buffer\n";
    std::cout << "   4. has_xxx() - check if optional field is set\n\n";
}

// ==================================================================
```

```
// EXAMPLE 4: ADVANCED FILE SERIALIZATION
// ==================================================================

void example_file_serialization() {
    std::cout << "=== Example 4: Advanced File Serialization ===\n";

    // Example 4a: Writing multiple messages to a file with length prefixes
    std::cout << "\n[4a] Writing multiple sensor readings to file:\n";

    const char* multi_file = "sensor_readings_multi.bin";
    std::ofstream output(multi_file, std::ios::binary);

    if (!output) {
        std::cerr << "Failed to open file for writing!\n";
        return;
    }

    // Create multiple sensor readings
    std::vector<sensors::SensorReading> readings;

    // Reading 1: Temperature
    sensors::SensorReading temp_reading;
    temp_reading.set_type(sensors::TEMPERATURE);
    temp_reading.set_device_id("sensor_001");
    temp_reading.set_temperature_celsius(22.5f);
    readings.push_back(temp_reading);

    // Reading 2: Humidity
    sensors::SensorReading humidity_reading;
    humidity_reading.set_type(sensors::HUMIDITY);
    humidity_reading.set_device_id("sensor_002");
    humidity_reading.set_humidity_percent(67.8f);
    readings.push_back(humidity_reading);

    // Reading 3: Pressure
    sensors::SensorReading pressure_reading;
    pressure_reading.set_type(sensors::PRESSURE);
    pressure_reading.set_device_id("sensor_003");
    pressure_reading.set_pressure_hpa(1013.25f);
    readings.push_back(pressure_reading);

    // Write each message with length prefix (for proper deserialization)
    for (const auto& reading : readings) {
        // Serialize to string first
        std::string serialized;
        if (!reading.SerializeToString(&serialized)) {
            std::cerr << "Failed to serialize reading!\n";
            continue;
        }

        // Write length prefix (4 bytes)
        uint32_t size = serialized.size();
        output.write(reinterpret_cast<const char*>(&size), sizeof(size));

```

```
        // Write the actual message
        output.write(serialized.data(), serialized.size());

        std::cout << "    Written " << reading.device_id()
                  << " (" << size << " bytes)\n";
    }

    output.close();
    std::cout << "  Successfully wrote " << readings.size()
              << " readings to " << multi_file << "\n";

    // Example 4b: Reading multiple messages from file
    std::cout << "\n[4b] Reading multiple messages from file:\n";

    std::ifstream input(multi_file, std::ios::binary);
    if (!input) {
        std::cerr << "Failed to open file for reading!\n";
        return;
    }

    int count = 0;
    while (input.good()) {
        // Read length prefix
        uint32_t size;
        input.read(reinterpret_cast<char*>(&size), sizeof(size));

        if (input.eof()) break;
        if (!input.good()) {
            std::cerr << "Error reading length prefix!\n";
            break;
        }

        // Read the message data
        std::vector<char> buffer(size);
        input.read(buffer.data(), size);

        if (!input.good() && !input.eof()) {
            std::cerr << "Error reading message data!\n";
            break;
        }

        // Parse the message
        sensors::SensorReading reading;
        if (!reading.ParseFromArray(buffer.data(), size)) {
            std::cerr << "Failed to parse message!\n";
            continue;
        }

        count++;
        std::cout << "  Reading #" << count << ": "
                  << reading.device_id() << " - ";

        // Display based on type
        if (reading.has_temperature_celsius()) {
```

```
            std::cout << reading.temperature_celsius() << "°C";
        } else if (reading.has_humidity_percent()) {
            std::cout << reading.humidity_percent() << "%";
        } else if (reading.has_pressure_hpa()) {
            std::cout << reading.pressure_hpa() << " hPa";
        }
        std::cout << "\n";
    }

    input.close();
    std::cout << "  Successfully read " << count << " readings from file\n";

    // Example 4c: Using delimited I/O (WriteDelimitedToOstream)
    std::cout << "\n[4c] Using Google's delimited I/O utilities:\n";
    std::cout << R"code(
// For writing multiple messages (requires google/protobuf/io/coded_stream.h):
#include <google/protobuf/io/coded_stream.h>
#include <google/protobuf/io/zero_copy_stream_impl.h>

std::ofstream output("data.bin", std::ios::binary);
google::protobuf::io::OstreamOutputStream ostream(&output);
google::protobuf::io::CodedOutputStream coded_output(&ostream);

// Write each message with length prefix
for (const auto& reading : readings) {
    coded_output.WriteVarint32(reading.ByteSizeLong());
    reading.SerializeToCodedStream(&coded_output);
}

// For reading:
std::ifstream input("data.bin", std::ios::binary);
google::protobuf::io::IstreamInputStream istream(&input);
google::protobuf::io::CodedInputStream coded_input(&istream);

uint32_t size;
while (coded_input.ReadVarint32(&size)) {
    auto limit = coded_input.PushLimit(size);
    sensors::SensorReading reading;
    reading.ParseFromCodedStream(&coded_input);
    coded_input.PopLimit(limit);
}
)code" << "\n";

    std::cout << "File I/O Best Practices:\n";
    std::cout << "  1. Always write length prefix for multiple messages\n";
    std::cout << "  2. Use binary mode (std::ios::binary) for files\n";
    std::cout << "  3. Check file operations (input.good(), output.good())\n";
    std::cout << "  4. Use CodedStream for efficient varint encoding\n";
    std::cout << "  5. Handle EOF and errors gracefully\n";
    std::cout << "  6. Close files explicitly or use RAII\n\n";
}

// ==================================================================
// EXAMPLE 5: FILE FORMAT STRATEGIES
```

```
// ==================================================================

void example_file_formats() {
    std::cout << "=== Example 5: File Format Strategies ===\n\n";

    std::cout << "Strategy 1: Single Message per File\n";
    std::cout << "  Use case: Configuration files, snapshots\n";
    std::cout << "    Simple: just SerializeToOstream() / ParseFromIstream()\n";
    std::cout << "    Easy to inspect and replace\n";
    std::cout << "    Many files for multiple messages\n";
    std::cout << R"code(
  sensors::SensorReading reading;
  // ... populate reading ...

  std::ofstream out("config.bin", std::ios::binary);
  reading.SerializeToOstream(&out);
  out.close();

  std::ifstream in("config.bin", std::ios::binary);
  reading.ParseFromIstream(&in);
)code" << "\n\n";

    std::cout << "Strategy 2: Multiple Messages with Length Prefixes\n";
    std::cout << "  Use case: Log files, sensor data streams\n";
    std::cout << "    Efficient for sequential access\n";
    std::cout << "    Single file for all data\n";
    std::cout << "    No random access\n";
    std::cout << "  See Example 4b above for implementation\n\n";

    std::cout << "Strategy 3: Container Message (Repeated Fields)\n";
    std::cout << "  Use case: Batched data, complete datasets\n";
    std::cout << "    Single SerializeToOstream() call\n";
    std::cout << "    Entire dataset in memory\n";
    std::cout << "    Must load everything at once\n";
    std::cout << R"code(
  // Define in .proto:
  // message SensorDataBatch {
  //   repeated SensorReading readings = 1;
  // }

  sensors::SensorDataBatch batch;
  batch.add_readings()->CopyFrom(reading1);
  batch.add_readings()->CopyFrom(reading2);
  batch.add_readings()->CopyFrom(reading3);

  std::ofstream out("batch.bin", std::ios::binary);
  batch.SerializeToOstream(&out);
)code" << "\n\n";

    std::cout << "Strategy 4: Text Format (for debugging)\n";
    std::cout << "  Use case: Debug logs, human inspection\n";
    std::cout << "    Human-readable\n";
    std::cout << "    Easy to debug\n";
```

```
    std::cout << "    Much larger file size\n";
    std::cout << "    Slower parsing\n";
    std::cout << R"code(
  #include <google/protobuf/text_format.h>

  sensors::SensorReading reading;
  // ... populate ...

  std::ofstream out("data.txt");
  std::string text;
  google::protobuf::TextFormat::PrintToString(reading, &text);
  out << text;

  // Read back:
  std::ifstream in("data.txt");
  std::string content((std::istreambuf_iterator<char>(in)),
                      std::istreambuf_iterator<char>());
  google::protobuf::TextFormat::ParseFromString(content, &reading);
)code" << "\n\n";

    std::cout << "Recommendation by use case:\n";
    std::cout << " •  Configuration files → Strategy 1 (single message)\n";
    std::cout << " •  Sensor data logs → Strategy 2 (length prefixes)\n";
    std::cout << " •  Database dumps → Strategy 3 (container message)\n";
    std::cout << " •  Debug output → Strategy 4 (text format)\n";
    std::cout << " •  Embedded logging → Strategy 2 (append to file)\n\n";
}

// ==================================================================
// EXAMPLE 6: JSON CONVERSION
// ==================================================================

void example_json_conversion() {
    std::cout << "=== Example 6: JSON Conversion (Protobuf v3) ===\n";

    std::cout << R"code(
#include <google/protobuf/util/json_util.h>

sensors::SensorReading reading;
reading.set_type(sensors::TEMPERATURE);
reading.set_device_id("sensor_001");
reading.set_temperature_celsius(23.5f);

// Convert to JSON
std::string json_output;
google::protobuf::util::JsonPrintOptions options;
options.add_whitespace = true;  // Pretty print
options.always_print_primitive_fields = true;  // Show all fields

auto status = google::protobuf::util::MessageToJsonString(
    reading, &json_output, options
);

if (status.ok()) {
```

```
    std::cout << "JSON output:\n" << json_output << "\n";
} else {
    std::cerr << "JSON conversion failed: " << status.message() << "\n";
}

// Convert from JSON
std::string json_input = R"({
    "type": "TEMPERATURE",
    "device_id": "sensor_002",
    "temperature_celsius": 25.3
})";

sensors::SensorReading parsed_reading;
status = google::protobuf::util::JsonStringToMessage(json_input, &
    parsed_reading);

if (status.ok()) {
    std::cout << "Parsed from JSON: " << parsed_reading.device_id() << "\n";
}
)code";

    std::cout << "\nJSON features:\n";
    std::cout << "    Human-readable debugging\n";
    std::cout << "    REST API compatibility\n";
    std::cout << "    Configuration files\n";
    std::cout << "    Larger than binary (use binary for production)\n\n";
}

// ==================================================================
// EXAMPLE 7: REPEATED FIELDS AND BATCHING
// ==================================================================

void example_repeated_fields() {
    std::cout << "=== Example 7: Repeated Fields (Batch Processing) ===\n";

    sensors::SensorBatch batch;
    batch.set_batch_id("batch_001");

    // Add multiple sensor readings
    for (int i = 0; i < 10; ++i) {
        auto* reading = batch.add_readings();  // Add new reading
        reading->set_type(sensors::TEMPERATURE);
        reading->set_device_id("sensor_" + std::to_string(i));
        reading->set_temperature_celsius(20.0f + i * 0.5f);
    }

    // Iterate over readings
    std::cout << "Batch contains " << batch.readings_size() << " readings:\n";
    for (const auto& reading : batch.readings()) {
        std::cout << "  " << reading.device_id()
                  << ": " << reading.temperature_celsius() << "°C\n";
    }

    // Access by index
```

```
    if (batch.readings_size() > 0) {
        const auto& first = batch.readings(0);
        std::cout << "\nFirst reading: " << first.device_id() << "\n";
    }

    // Serialize the batch
    std::string batch_data;
    batch.SerializeToString(&batch_data);
    std::cout << "Batch serialized: " << batch_data.size() << " bytes for "
              << batch.readings_size() << " readings\n";
    std::cout << "Average: " << (batch_data.size() / batch.readings_size()) << " bytes/reading\n";

    std::cout << "\nRepeated field methods:\n";
    std::cout << " •  add_xxx() - append new element\n";
    std::cout << " •  xxx_size() - get count\n";
    std::cout << " •  xxx(index) - access by index\n";
    std::cout << " •  clear_xxx() - remove all elements\n";
    std::cout << " •  Range-based for loop support\n\n";
}

// ==================================================================
// EXAMPLE 6: MEMORY MANAGEMENT AND PERFORMANCE
// ==================================================================

void example_performance() {
    std::cout << "=== Example 6: Performance and Memory Efficiency ===\n";

    std::cout << R"(
// Arena allocation for high-performance scenarios
// Reduces memory allocations and improves cache locality
google::protobuf::Arena arena;

auto* reading = google::protobuf::Arena::CreateMessage<sensors::SensorReading>(&arena);
reading->set_device_id("sensor_001");
reading->set_temperature_celsius(23.5f);

// All nested messages also use arena
auto* timestamp = reading->mutable_timestamp();
timestamp->set_seconds(12345);

// Memory is freed when arena goes out of scope
// No need to delete individual messages

// Performance tips:
// 1. Reuse message objects instead of creating new ones
sensors::SensorReading reusable_msg;
for (int i = 0; i < 1000; ++i) {
    reusable_msg.Clear();  // Reset, don't recreate
    reusable_msg.set_device_id("sensor_" + std::to_string(i));
    // ... serialize and send ...
}

```

```
// 2. Use SerializeToArray with pre-allocated buffer
std::vector<uint8_t> buffer(1024);  // Pre-allocate
size_t size = reading->ByteSizeLong();
if (size <= buffer.size()) {
    reading->SerializeToArray(buffer.data(), size);
}

// 3. For embedded systems: use lite runtime
// In .proto file: option optimize_for = LITE_RUNTIME;
// Reduces binary size by 50-70%
)";

    std::cout << "\nPerformance techniques:\n";
    std::cout << "    Arena allocation - reduces fragmentation\n";
    std::cout << "    Message reuse with Clear() - avoids allocations\n";
    std::cout << "    Pre-allocated buffers - zero-copy serialization\n";
    std::cout << "    Lite runtime - smaller binary for embedded systems\n";
    std::cout << "    Lazy field evaluation - on-demand parsing\n\n";
}

// ==================================================================
// EXAMPLE 9: EMBEDDED SYSTEMS CONSIDERATIONS
// ==================================================================

void example_embedded_systems() {
    std::cout << "=== Example 9: Embedded Systems Best Practices ===\n";

    std::cout << R"(
// 1. Use fixed-size buffers to avoid dynamic allocation
constexpr size_t MAX_MESSAGE_SIZE = 256;
uint8_t tx_buffer[MAX_MESSAGE_SIZE];
uint8_t rx_buffer[MAX_MESSAGE_SIZE];

// 2. Serialize directly to UART/SPI buffer
sensors::SensorReading reading;
reading.set_device_id("mcu_001");
reading.set_temperature_celsius(23.5f);

size_t size = reading.ByteSizeLong();
if (size <= MAX_MESSAGE_SIZE) {
    reading.SerializeToArray(tx_buffer, size);
    // uart_transmit(tx_buffer, size);
}

// 3. Parse from received buffer
// size_t received = uart_receive(rx_buffer, MAX_MESSAGE_SIZE);
size_t received = 50;  // Example

sensors::SensorReading received_msg;
if (received_msg.ParseFromArray(rx_buffer, received)) {
    // Process message
    float temp = received_msg.temperature_celsius();
}

```

```
// 4. Use lite runtime for small footprint
// .proto file option:
//   option optimize_for = LITE_RUNTIME;
//
// Removes:
// - Reflection API
// - Text format support
// - JSON conversion
//
// Result: 50-70% smaller binary

// 5. Error handling without exceptions (embedded systems)
#ifdef PROTOBUF_USE_EXCEPTIONS
    try {
        reading.ParseFromArray(rx_buffer, received);
    } catch (const std::exception& e) {
        // Handle error
    }
#else
    // Check return value
    if (!reading.ParseFromArray(rx_buffer, received)) {
        // Handle error without exceptions
    }
#endif
)";

    std::cout << "\nEmbedded systems considerations:\n";
    std::cout << "   Fixed-size buffers - no heap fragmentation\n";
    std::cout << "   Lite runtime - minimal binary size\n";
    std::cout << "   No exceptions - suitable for bare-metal\n";
    std::cout << "   Zero-copy serialization - DMA-friendly\n";
    std::cout << "   Deterministic memory usage\n\n";
}

// ==================================================================
// EXAMPLE 8: SCHEMA EVOLUTION (BACKWARD COMPATIBILITY)
// ==================================================================

void example_schema_evolution() {
    std::cout << "=== Example 8: Schema Evolution ===\n";

    std::cout << R"(
// Protocol Buffers support schema evolution:
// Old code can read new messages, new code can read old messages

// VERSION 1 of .proto:
// message SensorReading {
//   string device_id = 1;
//   float temperature_celsius = 2;
// }

// VERSION 2 adds new fields (backward compatible):
// message SensorReading {
//   string device_id = 1;
```

```
//   float temperature_celsius = 2;
//   string location = 3;        // NEW FIELD
//   int32 battery_percent = 4;  // NEW FIELD
// }

// Rules for compatibility:
//   Can add new fields (old code ignores them)
//   Can remove optional fields (new code uses defaults)
//   Cannot change field numbers (breaks everything)
//   Cannot change field types (incompatible)
//   Can rename fields (only changes code, not wire format)

// Example: Old client reading new message
// sensors::SensorReading msg;  // From version 1 code
// msg.ParseFromString(new_message_bytes);
//
// float temp = msg.temperature_celsius();  // Works!
// // New fields (location, battery) are ignored by old code

// Example: New client reading old message
// sensors::SensorReading msg;  // From version 2 code
// msg.ParseFromString(old_message_bytes);
//
// float temp = msg.temperature_celsius();  // Works!
// std::string loc = msg.location();         // Empty string (default)
// int battery = msg.battery_percent();      // 0 (default)
)";

    std::cout << "\nBackward compatibility rules:\n";
    std::cout << "  DO:\n";
    std::cout << "      Add new optional fields\n";
    std::cout << "      Remove optional fields (deprecated first)\n";
    std::cout << "      Rename fields (code only, not wire format)\n";
    std::cout << "  DON'T:\n";
    std::cout << "      Change field numbers\n";
    std::cout << "      Change field types\n";
    std::cout << "      Make required fields optional (v2 syntax)\n\n";
}

// ==================================================================
// COMPARISON: PROTOBUF VS ALTERNATIVES
// ==================================================================

void comparison_with_alternatives() {
    std::cout << "=== Comparison with Alternatives ===\n\n";

    std::cout << "XML:\n";
    std::cout << "    3-10x larger than protobuf\n";
    std::cout << "    20-100x slower parsing\n";
    std::cout << "    Human-readable\n";
    std::cout << "    Self-describing\n\n";

    std::cout << "JSON:\n";
    std::cout << "    2-5x larger than protobuf\n";
```

```
    std::cout << "    10-50x slower parsing\n";
    std::cout << "    Human-readable\n";
    std::cout << "    JavaScript-friendly\n";
    std::cout << "    No schema enforcement\n\n";

    std::cout << "MessagePack:\n";
    std::cout << "    Similar size to protobuf\n";
    std::cout << "    Fast parsing\n";
    std::cout << "    No schema (schemaless)\n";
    std::cout << "    No code generation\n\n";

    std::cout << "FlatBuffers:\n";
    std::cout << "    Zero-copy deserialization\n";
    std::cout << "    Very fast access\n";
    std::cout << "    Larger than protobuf\n";
    std::cout << "    Mutable messages more complex\n\n";

    std::cout << "Cap'n Proto:\n";
    std::cout << "    Zero-copy like FlatBuffers\n";
    std::cout << "    Very fast\n";
    std::cout << "    Less mature ecosystem\n";
    std::cout << "    Smaller community\n\n";

    std::cout << "When to use Protocol Buffers:\n";
    std::cout << "    Need efficient binary serialization\n";
    std::cout << "    Cross-language communication\n";
    std::cout << "    Schema evolution important\n";
    std::cout << "    Google's ecosystem (gRPC, etc.)\n";
    std::cout << "    Embedded systems with constraints\n\n";
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "==========================================================\n";
    std::cout << "GOOGLE PROTOCOL BUFFERS (PROTOBUF) EXAMPLES\n";
    std::cout << "==========================================================\n\n";

    example_create_sensor_reading();
    example_binary_serialization();
    example_binary_deserialization();
    example_file_serialization();
    example_file_formats();
    example_json_conversion();
    example_repeated_fields();
    example_performance();
    example_embedded_systems();
    example_schema_evolution();
    comparison_with_alternatives();

```

```
    std::cout << "============================================================\n";
    std::cout << "BUILD INSTRUCTIONS:\n";
    std::cout << "============================================================\n\n";

    std::cout << "1. Install Protocol Buffers:\n";
    std::cout << "   Ubuntu/Debian:\n";
    std::cout << "     sudo apt-get install protobuf-compiler libprotobuf-dev\n\n";

    std::cout << "   macOS:\n";
    std::cout << "     brew install protobuf\n\n";

    std::cout << "   Windows:\n";
    std::cout << "     Download from: https://github.com/protocolbuffers/protobuf/releases\n\n";

    std::cout << "2. Generate C++ code from .proto file:\n";
    std::cout << "   protoc --cpp_out=. proto/sensor_data.proto\n";
    std::cout << "   This creates:\n";
    std::cout << "     - sensor_data.pb.h\n";
    std::cout << "     - sensor_data.pb.cc\n\n";

    std::cout << "3. Compile with CMake:\n";
    std::cout << "   find_package(Protobuf REQUIRED)\n";
    std::cout << "   target_link_libraries(YourTarget protobuf::libprotobuf)\n\n";

    std::cout << "4. Or compile manually:\n";
    std::cout << "   g++ -std=c++17 ProtobufExample.cpp sensor_data.pb.cc \\\n";
    std::cout << "       -lprotobuf -o protobuf_example\n\n";

    std::cout << "============================================================\n";
    std::cout << "RESOURCES:\n";
    std::cout << "============================================================\n";
    std::cout << "Official docs: https://protobuf.dev/\n";
    std::cout << "GitHub: https://github.com/protocolbuffers/protobuf\n";
    std::cout << "C++ tutorial: https://protobuf.dev/getting-started/cpptutorial/\n\n";

    return 0;
}
```

54 Source Code: Pybind11Example.cpp

File: `src/Pybind11Example.cpp`
Repository: View on GitHub

```
// ==================================================================
// PYBIND11: C++ AND PYTHON INTEROPERABILITY
// ==================================================================
// This example demonstrates bidirectional calling between C++ and Python
// using pybind11 (modern, header-only binding library)
//
// BUILD INSTRUCTIONS:
// ===================
// 1. Install pybind11:
//    pip install pybind11
//
// 2. Compile as Python extension module:
//    c++ -O3 -Wall -shared -std=c++17 -fPIC \
//        $(python3 -m pybind11 --includes) \
//        Pybind11Example.cpp \
//        -o pybind_example$(python3-config --extension-suffix)
//
// 3. Use from Python:
//    import pybind_example
//    result = pybind_example.add(2, 3)
//
// ALTERNATIVE FRAMEWORKS:
// =======================
// - Boost.Python: Older, requires Boost library
// - SWIG: Generates bindings for multiple languages
// - ctypes: Python standard library, simpler but less type-safe
// - cffi: C Foreign Function Interface for Python
// - nanobind: Modern, lightweight alternative to pybind11
// ====================================================================

#include <pybind11/pybind11.h>
#include <pybind11/stl.h>        // STL containers conversion
#include <pybind11/functional.h> // std::function conversion
#include <pybind11/numpy.h>      // NumPy arrays support
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <cmath>

namespace py = pybind11;

// ==================================================================
// 1. SIMPLE FUNCTIONS - C++ FUNCTIONS CALLED FROM PYTHON
// ==================================================================

// Simple function
int add(int a, int b) {
    return a + b;
```

```
}

// Function with default arguments
double multiply(double a, double b = 2.0) {
    return a * b;
}

// Function with multiple return values (using tuple)
std::tuple<int, int> divide_with_remainder(int dividend, int divisor) {
    int quotient = dividend / divisor;
    int remainder = dividend % divisor;
    return std::make_tuple(quotient, remainder);
}

// Function returning string
std::string greet(const std::string& name) {
    return "Hello, " + name + "!";
}

// ==================================================================
// 2. WORKING WITH STL CONTAINERS
// ==================================================================

// Function accepting and returning std::vector
std::vector<int> square_elements(const std::vector<int>& vec) {
    std::vector<int> result;
    result.reserve(vec.size());
    for (int val : vec) {
        result.push_back(val * val);
    }
    return result;
}

// Function with std::map
std::map<std::string, int> count_words(const std::vector<std::string>& words)
    {
    std::map<std::string, int> counts;
    for (const auto& word : words) {
        counts[word]++;
    }
    return counts;
}

// ==================================================================
// 3. CLASSES - EXPOSING C++ CLASSES TO PYTHON
// ==================================================================

class Vector2D {
private:
    double x, y;

public:
    // Constructors
    Vector2D() : x(0.0), y(0.0) {}
```

```
    Vector2D(double x, double y) : x(x), y(y) {}

    // Getters/Setters
    double getX() const { return x; }
    double getY() const { return y; }
    void setX(double val) { x = val; }
    void setY(double val) { y = val; }

    // Methods
    double length() const {
        return std::sqrt(x * x + y * y);
    }

    Vector2D operator+(const Vector2D& other) const {
        return Vector2D(x + other.x, y + other.y);
    }

    Vector2D operator*(double scalar) const {
        return Vector2D(x * scalar, y * scalar);
    }

    // String representation
    std::string toString() const {
        return "Vector2D(" + std::to_string(x) + ", " + std::to_string(y) + ")";
    }

    // Static method
    static Vector2D zero() {
        return Vector2D(0.0, 0.0);
    }
};

// ==================================================================
// 4. INHERITANCE - BASE AND DERIVED CLASSES
// ==================================================================

class Shape {
protected:
    std::string name;

public:
    Shape(const std::string& name) : name(name) {}
    virtual ~Shape() = default;

    virtual double area() const = 0;
    virtual std::string getName() const { return name; }
};

class Circle : public Shape {
private:
    double radius;

public:
```

```
    Circle(const std::string& name, double r)
        : Shape(name), radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    double getRadius() const { return radius; }
};

class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(const std::string& name, double w, double h)
        : Shape(name), width(w), height(h) {}

    double area() const override {
        return width * height;
    }
};

// ==================================================================
// 5. CALLBACKS - CALLING PYTHON FUNCTIONS FROM C++
// ==================================================================

// Process data with Python callback
std::vector<int> process_with_callback(
    const std::vector<int>& data,
    std::function<int(int)> callback) {

    std::vector<int> result;
    result.reserve(data.size());

    for (int val : data) {
        result.push_back(callback(val));  // Call Python function!
    }

    return result;
}

// Numerical integration using Python callback
double integrate(std::function<double(double)> func,
                 double start, double end, int steps = 1000) {
    double dx = (end - start) / steps;
    double sum = 0.0;

    for (int i = 0; i < steps; ++i) {
        double x = start + i * dx;
        sum += func(x) * dx;  // Call Python function!
    }

    return sum;
```

```
}

// ==================================================================
// 6. SENSOR DATA PROCESSING (EMBEDDED SYSTEMS EXAMPLE)
// ==================================================================

struct SensorReading {
    double timestamp;
    double value;
    std::string sensor_id;

    SensorReading(double t, double v, const std::string& id)
        : timestamp(t), value(v), sensor_id(id) {}
};

class SensorProcessor {
private:
    std::vector<SensorReading> readings;

public:
    void addReading(double timestamp, double value, const std::string& id) {
        readings.emplace_back(timestamp, value, id);
    }

    size_t getReadingCount() const {
        return readings.size();
    }

    std::vector<SensorReading> getReadings() const {
        return readings;
    }

    // Process readings with Python callback
    std::vector<double> processReadings(
        std::function<double(const SensorReading&)> processor) {

        std::vector<double> results;
        results.reserve(readings.size());

        for (const auto& reading : readings) {
            results.push_back(processor(reading));  // Call Python!
        }

        return results;
    }

    // Filter readings with Python predicate
    std::vector<SensorReading> filterReadings(
        std::function<bool(const SensorReading&)> predicate) {

        std::vector<SensorReading> filtered;

        for (const auto& reading : readings) {
            if (predicate(reading)) {  // Call Python!
```

```
                filtered.push_back(reading);
            }
        }

        return filtered;
    }
};

// ==================================================================
// 7. NUMPY ARRAY PROCESSING (HIGH PERFORMANCE)
// ==================================================================

// Process NumPy array in C++ for performance
py::array_t<double> numpy_square(py::array_t<double> input) {
    // Get buffer info
    py::buffer_info buf = input.request();

    if (buf.ndim != 1) {
        throw std::runtime_error("Number of dimensions must be 1");
    }

    // Create output array
    auto result = py::array_t<double>(buf.size);
    py::buffer_info result_buf = result.request();

    // Get pointers
    double* input_ptr = static_cast<double*>(buf.ptr);
    double* result_ptr = static_cast<double*>(result_buf.ptr);

    // Process in C++ (fast!)
    for (size_t i = 0; i < buf.shape[0]; i++) {
        result_ptr[i] = input_ptr[i] * input_ptr[i];
    }

    return result;
}

// Matrix multiplication (2D NumPy arrays)
py::array_t<double> matrix_multiply(
    py::array_t<double> a,
    py::array_t<double> b) {

    py::buffer_info buf_a = a.request();
    py::buffer_info buf_b = b.request();

    if (buf_a.ndim != 2 || buf_b.ndim != 2) {
        throw std::runtime_error("Inputs must be 2D arrays");
    }

    size_t rows_a = buf_a.shape[0];
    size_t cols_a = buf_a.shape[1];
    size_t rows_b = buf_b.shape[0];
    size_t cols_b = buf_b.shape[1];

```

```
    if (cols_a != rows_b) {
        throw std::runtime_error("Matrix dimensions don't match");
    }

    // Create result matrix
    auto result = py::array_t<double>({rows_a, cols_b});
    py::buffer_info buf_result = result.request();

    double* ptr_a = static_cast<double*>(buf_a.ptr);
    double* ptr_b = static_cast<double*>(buf_b.ptr);
    double* ptr_result = static_cast<double*>(buf_result.ptr);

    // Initialize result to zero
    for (size_t i = 0; i < rows_a * cols_b; i++) {
        ptr_result[i] = 0.0;
    }

    // Matrix multiplication
    for (size_t i = 0; i < rows_a; i++) {
        for (size_t j = 0; j < cols_b; j++) {
            for (size_t k = 0; k < cols_a; k++) {
                ptr_result[i * cols_b + j] +=
                    ptr_a[i * cols_a + k] * ptr_b[k * cols_b + j];
            }
        }
    }

    return result;
}

// ==================================================================
// 8. SMART POINTERS - MANAGING OBJECT LIFETIME
// ==================================================================

class Resource {
private:
    std::string name;
    int id;

public:
    Resource(const std::string& n, int i) : name(n), id(i) {
        std::cout << "Resource created: " << name << " (ID: " << id << ")" << std::endl;
    }

    ~Resource() {
        std::cout << "Resource destroyed: " << name << " (ID: " << id << ")" << std::endl;
    }

    std::string getName() const { return name; }
    int getId() const { return id; }
};

```

```
// Factory function returning unique_ptr
std::unique_ptr<Resource> create_resource(const std::string& name, int id) {
    return std::make_unique<Resource>(name, id);
}

// Factory returning shared_ptr
std::shared_ptr<Resource> create_shared_resource(const std::string& name, int
    id) {
    return std::make_shared<Resource>(name, id);
}

// ================================================================
// PYBIND11 MODULE DEFINITION
// ================================================================

PYBIND11_MODULE(pybind_example, m) {
    m.doc() = "pybind11 example module - C++ and Python interoperability";

    // ============================================================
    // 1. SIMPLE FUNCTIONS
    // ============================================================
    m.def("add", &add, "Add two integers",
          py::arg("a"), py::arg("b"));

    m.def("multiply", &multiply, "Multiply two numbers",
          py::arg("a"), py::arg("b") = 2.0);

    m.def("divide_with_remainder", &divide_with_remainder,
          "Divide and return (quotient, remainder)");

    m.def("greet", &greet, "Greet someone");

    // ============================================================
    // 2. STL CONTAINERS
    // ============================================================
    m.def("square_elements", &square_elements,
          "Square all elements in a list");

    m.def("count_words", &count_words,
          "Count occurrences of each word");

    // ============================================================
    // 3. VECTOR2D CLASS
    // ============================================================
    py::class_<Vector2D>(m, "Vector2D")
        .def(py::init<>())
        .def(py::init<double, double>())
        .def_property("x", &Vector2D::getX, &Vector2D::setX)
        .def_property("y", &Vector2D::getY, &Vector2D::setY)
        .def("length", &Vector2D::length)
        .def("__add__", &Vector2D::operator+)
        .def("__mul__", &Vector2D::operator*)
        .def("__repr__", &Vector2D::toString)
        .def_static("zero", &Vector2D::zero);
```

```

    // ================================================================
    // 4. INHERITANCE - SHAPES
    // ================================================================
    py::class_<Shape>(m, "Shape")
        .def("area", &Shape::area)
        .def("get_name", &Shape::getName);

    py::class_<Circle, Shape>(m, "Circle")
        .def(py::init<const std::string&, double>())
        .def("get_radius", &Circle::getRadius);

    py::class_<Rectangle, Shape>(m, "Rectangle")
        .def(py::init<const std::string&, double, double>());

    // ================================================================
    // 5. CALLBACKS
    // ================================================================
    m.def("process_with_callback", &process_with_callback,
          "Process data with Python callback function");

    m.def("integrate", &integrate,
          "Numerical integration using Python callback",
          py::arg("func"), py::arg("start"), py::arg("end"),
          py::arg("steps") = 1000);

    // ================================================================
    // 6. SENSOR DATA PROCESSING
    // ================================================================
    py::class_<SensorReading>(m, "SensorReading")
        .def(py::init<double, double, const std::string&>())
        .def_readonly("timestamp", &SensorReading::timestamp)
        .def_readonly("value", &SensorReading::value)
        .def_readonly("sensor_id", &SensorReading::sensor_id);

    py::class_<SensorProcessor>(m, "SensorProcessor")
        .def(py::init<>())
        .def("add_reading", &SensorProcessor::addReading)
        .def("get_reading_count", &SensorProcessor::getReadingCount)
        .def("get_readings", &SensorProcessor::getReadings)
        .def("process_readings", &SensorProcessor::processReadings)
        .def("filter_readings", &SensorProcessor::filterReadings);

    // ================================================================
    // 7. NUMPY ARRAYS
    // ================================================================
    m.def("numpy_square", &numpy_square,
          "Square all elements in NumPy array (C++ speed!)");

    m.def("matrix_multiply", &matrix_multiply,
          "Matrix multiplication using NumPy arrays");

    // ================================================================
    // 8. SMART POINTERS
```

```
    // ================================================================
    py::class_<Resource, std::shared_ptr<Resource>>(m, "Resource")
        .def("get_name", &Resource::getName)
        .def("get_id", &Resource::getId);

    m.def("create_resource", &create_resource,
          "Create a resource (returns unique_ptr)");

    m.def("create_shared_resource", &create_shared_resource,
          "Create a shared resource (returns shared_ptr)");
}

// ================================================================
// STANDALONE EXAMPLE (FOR DOCUMENTATION)
// ================================================================

/*
PYTHON USAGE EXAMPLES:
======================

# 1. Simple functions
import pybind_example as pe

result = pe.add(5, 3)
print(f"5 + 3 = {result}")  # Output: 5 + 3 = 8

value = pe.multiply(4.5)
print(f"4.5 * 2.0 = {value}")  # Output: 4.5 * 2.0 = 9.0

quotient, remainder = pe.divide_with_remainder(17, 5)
print(f"17 / 5 = {quotient} remainder {remainder}")  # Output: 17 / 5 = 3
    remainder 2

message = pe.greet("World")
print(message)  # Output: Hello, World!

# 2. STL containers
numbers = [1, 2, 3, 4, 5]
squared = pe.square_elements(numbers)
print(f"Squared: {squared}")  # Output: Squared: [1, 4, 9, 16, 25]

words = ["hello", "world", "hello", "python"]
counts = pe.count_words(words)
print(f"Word counts: {counts}")  # Output: Word counts: {'hello': 2, 'world':
    1, 'python': 1}

# 3. Classes
v1 = pe.Vector2D(3.0, 4.0)
v2 = pe.Vector2D(1.0, 2.0)
print(f"v1 = {v1}")  # Output: v1 = Vector2D(3.000000, 4.000000)
print(f"v1.length() = {v1.length()}")  # Output: v1.length() = 5.0

v3 = v1 + v2
print(f"v1 + v2 = {v3}")  # Output: v1 + v2 = Vector2D(4.000000, 6.000000)
```

```python

v4 = v1 * 2.0
print(f"v1 * 2.0 = {v4}")  # Output: v1 * 2.0 = Vector2D(6.000000, 8.000000)

# 4. Inheritance
circle = pe.Circle("MyCircle", 5.0)
rect = pe.Rectangle("MyRect", 4.0, 6.0)
print(f"{circle.get_name()} area: {circle.area()}")  # Output: MyCircle area: 78.53975
print(f"{rect.get_name()} area: {rect.area()}")  # Output: MyRect area: 24.0

# 5. Callbacks - CALLING PYTHON FROM C++!
def my_python_function(x):
    return x * 2 + 1

data = [1, 2, 3, 4, 5]
result = pe.process_with_callback(data, my_python_function)
print(f"Processed: {result}")  # Output: Processed: [3, 5, 7, 9, 11]

# Integrate a Python function using C++
import math
result = pe.integrate(lambda x: x**2, 0, 1, steps=10000)
print(f"Integral of x^2 from 0 to 1: {result}")  # Output: ~0.333333

result = pe.integrate(math.sin, 0, math.pi, steps=10000)
print(f"Integral of sin(x) from 0 to  : {result}")  # Output: ~2.0

# 6. Sensor processing with callbacks
processor = pe.SensorProcessor()
processor.add_reading(0.0, 25.5, "TEMP_01")
processor.add_reading(1.0, 26.2, "TEMP_01")
processor.add_reading(2.0, 24.8, "TEMP_02")

# Process with Python function
def extract_value(reading):
    return reading.value

values = processor.process_readings(extract_value)
print(f"Values: {values}")  # Output: Values: [25.5, 26.2, 24.8]

# Filter with Python predicate
def high_temp(reading):
    return reading.value > 25.0

high_temps = processor.filter_readings(high_temp)
print(f"High temps: {len(high_temps)}")  # Output: High temps: 2

# 7. NumPy arrays (high performance!)
import numpy as np

arr = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
squared = pe.numpy_square(arr)  # Fast C++ processing!
print(f"NumPy squared: {squared}")  # Output: NumPy squared: [1. 4. 9. 16. 25.]
```

```

A = np.array([[1.0, 2.0], [3.0, 4.0]])
B = np.array([[5.0, 6.0], [7.0, 8.0]])
C = pe.matrix_multiply(A, B)
print(f"Matrix multiply:\n{C}")
# Output:
# [[19. 22.]
#  [43. 50.]]

# 8. Smart pointers
resource = pe.create_shared_resource("MyResource", 42)
print(f"Resource: {resource.get_name()}, ID: {resource.get_id()}")
# Output: Resource: MyResource, ID: 42
# When resource goes out of scope, C++ destructor is automatically called

BUILD SCRIPT (build.sh):
========================
#!/bin/bash

# Compile the pybind11 extension
c++ -O3 -Wall -shared -std=c++17 -fPIC \
    $(python3 -m pybind11 --includes) \
    Pybind11Example.cpp \
    -o pybind_example$(python3-config --extension-suffix)

# Run Python tests
python3 test_pybind.py

ALTERNATIVE FRAMEWORKS:
=======================

1. BOOST.PYTHON (older, requires Boost):
   #include <boost/python.hpp>
   BOOST_PYTHON_MODULE(example) {
       boost::python::def("add", add);
   }

2. SWIG (multi-language support):
   // example.i
   %module example
   %{
   #include "example.h"
   %}
   %include "example.h"

   Build: swig -python -c++ example.i

3. CTYPES (Python standard library):
   // Compile: g++ -shared -fPIC example.cpp -o example.so

   # Python:
   import ctypes
   lib = ctypes.CDLL('./example.so')
   lib.add.argtypes = [ctypes.c_int, ctypes.c_int]
```

```
   lib.add.restype = ctypes.c_int
   result = lib.add(5, 3)

4. CFFI (C Foreign Function Interface):
   from cffi import FFI
   ffi = FFI()
   ffi.cdef("int add(int, int);")
   lib = ffi.dlopen('./example.so')
   result = lib.add(5, 3)

5. NANOBIND (modern, lightweight):
   #include <nanobind/nanobind.h>
   NB_MODULE(example, m) {
       m.def("add", &add);
   }

PERFORMANCE COMPARISON:
=======================
For numerical computations:
- Pure Python: 1x (baseline)
- NumPy: 10-100x faster
- C++ via pybind11: 100-1000x faster
- Direct C++ (no Python): 1000x+ faster

WHEN TO USE:
============
 Use pybind11 when:•
   Need performance-critical code in C++•
   Have existing C++ libraries to expose•
   Want type safety and automatic conversions•
   Need to process NumPy arrays efficiently•
   Want modern C++ features (C++11-20)

 Avoid when:•
   Pure Python is fast enough•
   No C++ expertise available•
   Deployment complexity is a concern•
   Only need simple C functions (use ctypes)
*/
```

55 Source Code: ROMability.cpp

File: `src/ROMability.cpp`
Repository: View on GitHub

```cpp
// ROMability.cpp
// Demonstrates ROM-ability in C++ - Placing data in Read-Only Memory (.rodata section)
//
// KEY CONCEPTS:
// 1. const - Runtime or compile-time constant, placed in ROM if possible
// 2. constexpr - Compile-time constant, guaranteed ROM placement
// 3. consteval - C++20, forced compile-time evaluation
// 4. constinit - C++20, compile-time initialization, runtime mutable
// 5. ROM vs RAM - .rodata vs .data/.bss sections
//
// EMBEDDED SYSTEMS CONTEXT:
// ROM (Read-Only Memory) - Flash memory, non-volatile, cheaper
// RAM (Random Access Memory) - SRAM, volatile, expensive on MCUs
// Goal: Maximize ROM usage, minimize RAM usage
//
// C++ ADVANTAGES OVER C:
//   constexpr functions - compute at compile-time
//   constexpr constructors - complex objects in ROM
//   std::array - type-safe ROM arrays
//   consteval - guarantee compile-time evaluation
//   Template metaprogramming - generate ROM data

#include <iostream>
#include <array>
#include <string_view>
#include <cmath>
#include <iomanip>
#include <cstdint>

// ============================================================================
// SECTION 1: const vs constexpr - Understanding the Difference
// ============================================================================

namespace const_vs_constexpr {

// C-style const - MAY be in ROM, but not guaranteed
const int c_style_const = 42;

// C++ constexpr - GUARANTEED compile-time constant, always in ROM
constexpr int cpp_constexpr = 42;

// const with runtime initialization - goes to RAM!
int get_runtime_value() { return 42; }
const int runtime_const = get_runtime_value();  // RAM, not ROM!
```

```

// constexpr forces compile-time evaluation
constexpr int get_compile_time_value() { return 42; }
constexpr int compile_time_const = get_compile_time_value();  // ROM

// Complex compile-time calculation
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}

constexpr int fact_5 = factorial(5);  // Computed at compile-time, stored in ROM

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1: const vs constexpr ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Traditional const:\n";
    std::cout << "   const int c_style_const = 42;\n";
    std::cout << "   Value: " << c_style_const << "\n";
    std::cout << "   Location: Probably ROM, but not guaranteed\n\n";

    std::cout << "2. constexpr (C++11):\n";
    std::cout << "   constexpr int cpp_constexpr = 42;\n";
    std::cout << "   Value: " << cpp_constexpr << "\n";
    std::cout << "   Location: Guaranteed ROM (.rodata section)\n\n";

    std::cout << "3. const with runtime initialization:\n";
    std::cout << "   const int runtime_const = get_runtime_value();\n";
    std::cout << "   Value: " << runtime_const << "\n";
    std::cout << "   Location: RAM (.data section) - not ROM!\n\n";

    std::cout << "4. constexpr with compile-time function:\n";
    std::cout << "   constexpr int fact_5 = factorial(5);\n";
    std::cout << "   Value: " << fact_5 << " (computed at compile-time!)\n";
    std::cout << "   Location: ROM (.rodata section)\n\n";

    std::cout << "  RULE: Use constexpr for guaranteed ROM placement\n";
    std::cout << "  RULE: const doesn't guarantee ROM (can be initialized at runtime)\n";
}

} // namespace const_vs_constexpr

// ============================================================================
// SECTION 2: ROM-able Arrays - Lookup Tables
// ============================================================================

namespace rom_arrays {
```

```

// C-style array - ROM placement
constexpr int sine_table_c[16] = {
    0, 707, 1000, 707, 0, -707, -1000, -707,
    0, 707, 1000, 707, 0, -707, -1000, -707
};

// C++ std::array - ROM placement with type safety
constexpr std::array<int, 16> sine_table_cpp = {
    0, 707, 1000, 707, 0, -707, -1000, -707,
    0, 707, 1000, 707, 0, -707, -1000, -707
};

// Generate lookup table at compile-time
constexpr auto generate_sine_table() {
    std::array<int, 360> table{};
    for (int i = 0; i < 360; ++i) {
        // Note: std::sin is not constexpr before C++26
        // Using approximation for demonstration
        table[i] = static_cast<int>(1000 * std::sin(i * 3.14159 / 180.0));
    }
    return table;
}

// Entire lookup table generated at compile-time, stored in ROM
constexpr auto sine_lookup = generate_sine_table();

// PWM duty cycle table for LED brightness (gamma correction)
constexpr auto generate_gamma_table() {
    std::array<uint8_t, 256> table{};
    for (int i = 0; i < 256; ++i) {
        // Gamma correction: output = input^2.2
        double normalized = i / 255.0;
        double corrected = std::pow(normalized, 2.2);
        table[i] = static_cast<uint8_t>(corrected * 255.0);
    }
    return table;
}

constexpr auto gamma_table = generate_gamma_table();

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 2: ROM-able Arrays - Lookup Tables ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. C-style const array in ROM:\n";
    std::cout << "   constexpr int sine_table_c[16] = {...};\n";
    std::cout << "   Sample values: " << sine_table_c[0] << ", "
              << sine_table_c[2] << ", " << sine_table_c[6] << "\n";
    std::cout << "   Size: " << sizeof(sine_table_c) << " bytes\n\n";

    std::cout << "2. C++ std::array in ROM (type-safe):\n";
    std::cout << "   constexpr std::array<int, 16> sine_table_cpp = {...};\n";
```

```
    std::cout << "   Sample values: " << sine_table_cpp[0] << ", "
              << sine_table_cpp[2] << ", " << sine_table_cpp[6] << "\n";
    std::cout << "   Size: " << sine_table_cpp.size() << " elements\n\n";

    std::cout << "3. Generated lookup table (360 entries, compile-time):\n";
    std::cout << "   Sine values (degrees):\n";
    std::cout << "     0°: " << sine_lookup[0] << "\n";
    std::cout << "    30°: " << sine_lookup[30] << "\n";
    std::cout << "    60°: " << sine_lookup[60] << "\n";
    std::cout << "    90°: " << sine_lookup[90] << "\n";
    std::cout << "   Total size: " << sizeof(sine_lookup) << " bytes in ROM\n\n";

    std::cout << "4. Gamma correction table (256 entries):\n";
    std::cout << "   PWM duty cycles for linear brightness:\n";
    std::cout << "     Input 0: " << static_cast<int>(gamma_table[0]) << "\n";
    std::cout << "    Input 64: " << static_cast<int>(gamma_table[64]) << "\n";
    std::cout << "   Input 128: " << static_cast<int>(gamma_table[128]) << "\n";
    std::cout << "   Input 255: " << static_cast<int>(gamma_table[255]) << "\n";
    std::cout << "   Total size: " << sizeof(gamma_table) << " bytes in ROM\n\n";

    std::cout << "  Embedded benefit: Lookup tables computed at compile-time\n";
    std::cout << "  No runtime computation needed (saves CPU cycles)\n";
    std::cout << "  All stored in cheap ROM, not expensive RAM\n";
}

} // namespace rom_arrays

// ============================================================================
// SECTION 3: constexpr Objects - Complex Data in ROM
// ============================================================================

namespace constexpr_objects {

// Configuration stored entirely in ROM
struct DeviceConfig {
    const char* device_name;
    uint32_t serial_number;
    uint16_t max_voltage_mv;
    uint8_t num_channels;

    constexpr DeviceConfig(const char* name, uint32_t serial,
                           uint16_t voltage, uint8_t channels)
        : device_name(name), serial_number(serial),
          max_voltage_mv(voltage), num_channels(channels) {}
```

```
};

// Entire config in ROM
constexpr DeviceConfig device_config{
    "STM32F407",
    0x12345678,
    3300,  // 3.3V
    16
};

// Sensor calibration data in ROM
struct SensorCalibration {
    float offset;
    float gain;
    int16_t temp_coefficient;

    constexpr SensorCalibration(float o, float g, int16_t tc)
        : offset(o), gain(g), temp_coefficient(tc) {}

    constexpr float calibrate(int16_t raw_value, int16_t temperature) const {
        float temp_correction = temp_coefficient * (temperature - 25) / 1000.0
            f;
        return (raw_value + offset) * gain * (1.0f + temp_correction);
    }
};

constexpr SensorCalibration pressure_sensor{
    -50.0f,   // offset
    0.125f,   // gain
    -25       // temp coefficient (ppm/°C)
};

// Compile-time test of calibration
static_assert(pressure_sensor.calibrate(1000, 25) > 0, "Calibration sanity
    check");

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 3: constexpr Objects in ROM ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Device configuration (ROM):\n";
    std::cout << "   Device: " << device_config.device_name << "\n";
    std::cout << "   Serial: 0x" << std::hex << device_config.serial_number
              << std::dec << "\n";
    std::cout << "   Max Voltage: " << device_config.max_voltage_mv << " mV\n"
        ;
    std::cout << "   Channels: " << static_cast<int>(device_config.
        num_channels) << "\n";
    std::cout << "   Location: ROM (.rodata section)\n\n";

    std::cout << "2. Sensor calibration with compile-time function:\n";
    std::cout << "   Offset: " << pressure_sensor.offset << "\n";
    std::cout << "   Gain: " << pressure_sensor.gain << "\n";
```

```
    std::cout << "   Temp Coeff: " << pressure_sensor.temp_coefficient << " ppm/°C\n\n";

    std::cout << "   Runtime calibration:\n";
    int16_t raw = 1000;
    int16_t temp = 25;
    float calibrated = pressure_sensor.calibrate(raw, temp);
    std::cout << "     Raw value: " << raw << " at " << temp << "°C\n";
    std::cout << "     Calibrated: " << calibrated << " Pa\n";
    std::cout << "   Location: ROM (.rodata section)\n\n";

    std::cout << "  Complex objects can live entirely in ROM\n";
    std::cout << "  constexpr methods allow compile-time testing\n";
    std::cout << "  Perfect for embedded: config, calibration, constants\n";
}

} // namespace constexpr_objects

// ==========================================================================
// SECTION 4: C++20 Features - consteval and constinit
// ==========================================================================

namespace cpp20_features {

// consteval - MUST be evaluated at compile-time
consteval int must_be_compile_time(int n) {
    return n * n;
}

// This works - compile-time constant
constexpr int square_5 = must_be_compile_time(5);

// This would NOT compile:
// int x = 5;
// int result = must_be_compile_time(x);  // ERROR: x is not constexpr

// constinit - Compile-time initialization, but runtime mutable
constinit int initialized_at_compile_time = 42;

// Compare with const/constexpr
const int const_value = 42;              // Can't modify
constexpr int constexpr_value = 42;      // Can't modify, must be compile-time
// constinit allows modification but guarantees compile-time init

// Use case: Global variables that need compile-time init but runtime modification
constinit int error_count = 0;  // Initialized at compile-time to 0

// String literals in ROM
constexpr std::string_view device_name = "STM32F407";
```

```
constexpr std::string_view error_messages[] = {
    "No error",
    "Sensor timeout",
    "Invalid data",
    "Communication error"
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 4: C++20 - consteval and constinit ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. consteval - Forced compile-time evaluation:\n";
    std::cout << "   consteval int must_be_compile_time(int n) { return n * n; }\n";
    std::cout << "   square_5 = " << square_5 << " (computed at compile-time)\n";
    std::cout << "     Guarantees no runtime computation\n";
    std::cout << "     Perfect for embedded: zero runtime cost\n\n";

    std::cout << "2. constinit - Compile-time init, runtime mutable:\n";
    std::cout << "   constinit int error_count = 0;\n";
    std::cout << "   Initial value: " << error_count << "\n";

    // Can modify at runtime (unlike const/constexpr)
    ++error_count;
    std::cout << "   After increment: " << error_count << "\n";
    std::cout << "     Initialized at compile-time (no static initialization order issues)\n";
    std::cout << "     Can be modified at runtime\n\n";

    std::cout << "3. String literals in ROM:\n";
    std::cout << "   constexpr std::string_view device_name = \"STM32F407\";\n";
    std::cout << "   Device: " << device_name << "\n";
    std::cout << "   String data in ROM, no heap allocation\n\n";

    std::cout << "   Error message table in ROM:\n";
    for (size_t i = 0; i < 4; ++i) {
        std::cout << "     Error " << i << ": " << error_messages[i] << "\n";
    }
    std::cout << "   All strings in ROM (.rodata section)\n\n";

    std::cout << "  consteval: Forces compile-time (zero runtime cost)\n";
    std::cout << "  constinit: Safe global init, runtime mutable\n";
    std::cout << "  string_view: String literals without heap\n";
}

} // namespace cpp20_features

// ==========================================================================
// SECTION 5: Practical Embedded Example - Entire System Config in ROM
```

```
// ==========================================================================

namespace embedded_example {

// Pin configuration
struct PinConfig {
    uint8_t port;
    uint8_t pin;
    const char* function;

    constexpr PinConfig(uint8_t p, uint8_t pn, const char* func)
        : port(p), pin(pn), function(func) {}
};

// All pin configs in ROM
constexpr std::array<PinConfig, 8> pin_configs = {{
    {0, 0, "UART_TX"},
    {0, 1, "UART_RX"},
    {0, 5, "SPI_SCK"},
    {0, 6, "SPI_MISO"},
    {0, 7, "SPI_MOSI"},
    {1, 0, "I2C_SCL"},
    {1, 1, "I2C_SDA"},
    {2, 13, "LED"}
}};

// Timer configuration
struct TimerConfig {
    uint32_t frequency_hz;
    uint16_t prescaler;
    uint16_t period;

    constexpr TimerConfig(uint32_t freq)
        : frequency_hz(freq),
          prescaler(calculate_prescaler(freq)),
          period(calculate_period(freq, calculate_prescaler(freq))) {}

private:
    static constexpr uint16_t calculate_prescaler(uint32_t freq) {
        uint32_t divisor = freq * 65536;
        if (divisor == 0) return 1;
        return static_cast<uint16_t>(84000000 / divisor + 1);
    }

    static constexpr uint16_t calculate_period(uint32_t freq, uint16_t
        prescaler) {
        if (freq == 0 || prescaler == 0) return 1000;
        return static_cast<uint16_t>(84000000 / (freq * prescaler));
    }
};

// Timer configs computed at compile-time, stored in ROM
```

```
constexpr TimerConfig pwm_timer{1000};      // 1 kHz PWM
constexpr TimerConfig adc_timer{10000};     // 10 kHz ADC sampling
constexpr TimerConfig led_timer{2};         // 2 Hz LED blink

// Memory map - register addresses
namespace registers {
    constexpr uint32_t GPIO_BASE = 0x40020000;
    constexpr uint32_t UART_BASE = 0x40011000;
    constexpr uint32_t SPI_BASE = 0x40013000;
    constexpr uint32_t I2C_BASE = 0x40005400;
    constexpr uint32_t TIM_BASE = 0x40000000;
}

// Protocol constants
namespace can_protocol {
    constexpr uint32_t CAN_ID_STATUS = 0x100;
    constexpr uint32_t CAN_ID_SENSOR1 = 0x201;
    constexpr uint32_t CAN_ID_SENSOR2 = 0x202;
    constexpr uint32_t CAN_ID_COMMAND = 0x300;

    constexpr std::array<uint32_t, 4> filter_ids = {
        CAN_ID_STATUS, CAN_ID_SENSOR1, CAN_ID_SENSOR2, CAN_ID_COMMAND
    };
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 5: Embedded System - All Config in ROM ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Pin Configuration Table (ROM):\n";
    for (const auto& pin : pin_configs) {
        std::cout << "   Port " << static_cast<int>(pin.port)
                  << ", Pin " << static_cast<int>(pin.pin)
                  << ": " << pin.function << "\n";
    }
    std::cout << "   Size: " << sizeof(pin_configs) << " bytes in ROM\n\n";

    std::cout << "2. Timer Configurations (computed at compile-time):\n";
    std::cout << "   PWM Timer:\n";
    std::cout << "     Frequency: " << pwm_timer.frequency_hz << " Hz\n";
    std::cout << "     Prescaler: " << pwm_timer.prescaler << "\n";
    std::cout << "     Period: " << pwm_timer.period << "\n\n";

    std::cout << "   ADC Timer:\n";
    std::cout << "     Frequency: " << adc_timer.frequency_hz << " Hz\n";
    std::cout << "     Prescaler: " << adc_timer.prescaler << "\n";
    std::cout << "     Period: " << adc_timer.period << "\n\n";

    std::cout << "3. Register Addresses (ROM):\n";
    std::cout << "   GPIO_BASE: 0x" << std::hex << registers::GPIO_BASE << "\n
        ";
    std::cout << "   UART_BASE: 0x" << registers::UART_BASE << "\n";
    std::cout << "   SPI_BASE:  0x" << registers::SPI_BASE << "\n";
```

```
    std::cout << "   I2C_BASE:  0x" << registers::I2C_BASE << std::dec << "\n\n";

    std::cout << "4. CAN Protocol IDs (ROM):\n";
    for (size_t i = 0; i < can_protocol::filter_ids.size(); ++i) {
        std::cout << "   Filter " << i << ": 0x" << std::hex
                  << can_protocol::filter_ids[i] << std::dec << "\n";
    }
    std::cout << "\n EMBEDDED BENEFITS:\n";
    std::cout << " •  Zero RAM usage for configuration\n";
    std::cout << " •  All configs in cheap Flash ROM\n";
    std::cout << " •  Compile-time validation with static_assert\n";
    std::cout << " •  No runtime initialization overhead\n";
    std::cout << " •  Type-safe, no magic numbers\n";
}

} // namespace embedded_example

// ============================================================================
// SECTION 6: Verifying ROM Placement - Compiler Output
// ============================================================================

namespace verifying_placement {

// These should go to ROM (.rodata)
constexpr int rom_value = 42;
constexpr std::array<int, 10> rom_array = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
const char* const rom_string = "This is in ROM";

// These go to RAM (.data or .bss)
int ram_value = 42;
int ram_array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 6: Verifying ROM Placement ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "To verify ROM vs RAM placement, use these tools:\n\n";

    std::cout << "1. Using 'nm' command:\n";
    std::cout << "   $ nm ROMability | grep rom_value\n";
    std::cout << "   Output shows section:\n";
    std::cout << "     'r' or 'R' = .rodata (ROM)\n";
    std::cout << "     'd' or 'D' = .data (RAM)\n";
    std::cout << "     'b' or 'B' = .bss (uninitialized RAM)\n\n";

    std::cout << "2. Using 'objdump' command:\n";
    std::cout << "   $ objdump -h ROMability\n";
    std::cout << "   Shows sections:\n";
```

```
    std::cout << "     .rodata - Read-only data (ROM/Flash)\n";
    std::cout << "     .data   - Initialized data (RAM)\n";
    std::cout << "     .bss    - Uninitialized data (RAM)\n\n";

    std::cout << "3. Using 'size' command:\n";
    std::cout << "   $ size ROMability\n";
    std::cout << "   Shows memory usage:\n";
    std::cout << "     text   - Code + .rodata (ROM/Flash)\n";
    std::cout << "     data   - Initialized data (RAM)\n";
    std::cout << "     bss    - Uninitialized data (RAM)\n\n";

    std::cout << "4. GCC-specific: __attribute__((section))\n";
    std::cout << "   Can explicitly place data in specific section:\n";
    std::cout << "   __attribute__((section(\".rodata\"))) const int x = 42;\n\n";

    std::cout << "5. Embedded: Check linker map file\n";
    std::cout << "   Linker script (*.ld) and map file show exact placement\n";
    std::cout << "   Critical for embedded systems with limited RAM\n\n";

    std::cout << "  Best Practices:\n";
    std::cout << " •  Use constexpr for guaranteed ROM placement\n";
    std::cout << " •  Verify with nm/objdump during development\n";
    std::cout << " •  Check map file for embedded targets\n";
    std::cout << " •  Profile RAM usage on resource-constrained systems\n";
}

} // namespace verifying_placement

// ============================================================================
// SECTION 7: Summary and Best Practices
// ============================================================================

void show_summary() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== ROM-ability in C++ - Complete Summary ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "                                \n";
    std::cout << "   KEYWORD          COMPILE-TIME     LOCATION    MUTABLE     \n";
    std::cout << "                                \n";
    std::cout << "   const            Maybe            ROM/RAM     No          \n";
    std::cout << "   constexpr        Yes              ROM         No          \n";
    std::cout << "   consteval        Must be          ROM         No          \n";
    std::cout << "   constinit        Yes (init)       RAM         Yes         \
```

```
        n";
    std::cout << "                                \n\n";

    std::cout << "WHEN TO USE WHAT:\n\n";

    std::cout << "1. constexpr - Most common for ROM data\n";
    std::cout << " •   Lookup tables (sine, gamma, CRC, etc.)\n";
    std::cout << " •   Configuration structs\n";
    std::cout << " •   Constants and arrays\n";
    std::cout << " •   Compile-time computed values\n\n";

    std::cout << "2. consteval (C++20) - Force compile-time\n";
    std::cout << " •   Functions that MUST run at compile-time\n";
    std::cout << " •   Zero runtime cost guarantee\n";
    std::cout << " •   Configuration validation\n\n";

    std::cout << "3. constinit (C++20) - Global variables\n";
    std::cout << " •   Globals that need compile-time init\n";
    std::cout << " •   Avoids static initialization order fiasco\n";
    std::cout << " •   Runtime mutable (unlike constexpr)\n\n";

    std::cout << "4. const - Use sparingly\n";
    std::cout << " •   May or may not be ROM\n";
    std::cout << " •   Prefer constexpr for ROM guarantee\n\n";

    std::cout << "C++ ADVANTAGES OVER C:\n";
    std::cout << "    constexpr functions - compute at compile-time\n";
    std::cout << "    constexpr constructors - complex objects in ROM\n";
    std::cout << "    std::array - type-safe ROM arrays\n";
    std::cout << "    consteval - guarantee compile-time\n";
    std::cout << "    Templates - generate ROM data\n";
    std::cout << "    static_assert - compile-time validation\n\n";

    std::cout << "EMBEDDED SYSTEMS BENEFITS:\n";
    std::cout << "    Maximize ROM usage (cheap Flash)\n";
    std::cout << "    Minimize RAM usage (expensive SRAM)\n";
    std::cout << "    Zero runtime initialization overhead\n";
    std::cout << "    Compile-time validation\n";
    std::cout << "    No dynamic memory allocation\n";
    std::cout << "    Deterministic memory layout\n\n";

    std::cout << "BEST PRACTICES:\n";
    std::cout << "  1. Use constexpr by default for constants\n";
    std::cout << "  2. Generate lookup tables at compile-time\n";
    std::cout << "  3. Store configuration in ROM\n";
    std::cout << "  4. Use std::array instead of C arrays\n";
    std::cout << "  5. Use string_view for ROM strings\n";
    std::cout << "  6. Verify placement with nm/objdump\n";
    std::cout << "  7. Profile RAM usage on embedded targets\n";
}

//
    ==========================================================================
```

```cpp
// MAIN FUNCTION
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "                ROM-ability in C++ (constexpr & More)                    \n";
    std::cout << "           Placing Data in Read-Only Memory (.rodata)                    \n";
    std::cout << "                                    \n";

    // Section 1: const vs constexpr
    const_vs_constexpr::demonstrate();

    // Section 2: ROM arrays
    rom_arrays::demonstrate();

    // Section 3: constexpr objects
    constexpr_objects::demonstrate();

    // Section 4: C++20 features
    cpp20_features::demonstrate();

    // Section 5: Embedded example
    embedded_example::demonstrate();

    // Section 6: Verification
    verifying_placement::demonstrate();

    // Section 7: Summary
    show_summary();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "KEY TAKEAWAY: Use constexpr for guaranteed ROM placement!\n";
    std::cout << "Perfect for embedded systems: maximize ROM, minimize RAM.\n\n";

    return 0;
}
```

56 Source Code: RangesExamples.cpp

File: `src/RangesExamples.cpp`
Repository: View on GitHub

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <numeric>
#include <functional>

namespace ranges = std::ranges;
namespace views = std::views;

// ====================================================================
// 1. BASIC RANGES AND VIEWS
// ====================================================================
void example_basic_ranges() {
    std::cout << "\n=== 1. BASIC RANGES AND VIEWS ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::cout << "Original: ";
    for (int n : numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Using ranges::for_each
    std::cout << "Using ranges::for_each: ";
    ranges::for_each(numbers, [](int n) {
        std::cout << n << " ";
    });
    std::cout << std::endl;
}

// ====================================================================
// 2. FILTER VIEW
// ====================================================================
void example_filter_view() {
    std::cout << "\n=== 2. FILTER VIEW ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Filter even numbers
    auto even_numbers = numbers | views::filter([](int n) { return n % 2 == 0;
        });

    std::cout << "Even numbers: ";
    for (int n : even_numbers) {
        std::cout << n << " ";
    }
```

```
    std::cout << std::endl;

    // Filter numbers greater than 5
    auto greater_than_5 = numbers | views::filter([](int n) { return n > 5; })
        ;

    std::cout << "Greater than 5: ";
    for (int n : greater_than_5) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 3. TRANSFORM VIEW
// ==================================================================
void example_transform_view() {
    std::cout << "\n=== 3. TRANSFORM VIEW ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Square each number
    auto squared = numbers | views::transform([](int n) { return n * n; });

    std::cout << "Squared: ";
    for (int n : squared) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Convert to strings
    auto strings = numbers | views::transform([](int n) { return std::
        to_string(n); });

    std::cout << "As strings: ";
    for (const auto& s : strings) {
        std::cout << "'" << s << "' ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 4. TAKE AND DROP VIEWS
// ==================================================================
void example_take_drop_views() {
    std::cout << "\n=== 4. TAKE AND DROP VIEWS ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Take first 5 elements
    auto first_five = numbers | views::take(5);
    std::cout << "First 5: ";
    for (int n : first_five) {
        std::cout << n << " ";
```

```
    }
    std::cout << std::endl;

    // Drop first 5 elements
    auto skip_five = numbers | views::drop(5);
    std::cout << "Skip 5: ";
    for (int n : skip_five) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Take while condition is true
    auto take_while_small = numbers | views::take_while([](int n) { return n <
        6; });
    std::cout << "Take while < 6: ";
    for (int n : take_while_small) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 5. COMPOSING VIEWS
// ==================================================================
void example_composing_views() {
    std::cout << "\n=== 5. COMPOSING VIEWS ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Chain multiple operations: filter even, square, take first 3
    auto result = numbers
        | views::filter([](int n) { return n % 2 == 0; })
        | views::transform([](int n) { return n * n; })
        | views::take(3);

    std::cout << "Even, squared, first 3: ";
    for (int n : result) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 6. REVERSE VIEW
// ==================================================================
void example_reverse_view() {
    std::cout << "\n=== 6. REVERSE VIEW ===" << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5};

    auto reversed = numbers | views::reverse;

    std::cout << "Reversed: ";
    for (int n : reversed) {
```

```
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 7. IOTA VIEW (GENERATE SEQUENCE)
// ==================================================================
void example_iota_view() {
    std::cout << "\n=== 7. IOTA VIEW (GENERATE SEQUENCE) ===" << std::endl;

    // Generate numbers from 1 to 10
    auto range1 = views::iota(1, 11);
    std::cout << "Iota 1 to 10: ";
    for (int n : range1) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Infinite range (take first 5)
    auto infinite_range = views::iota(1) | views::take(5);
    std::cout << "Infinite iota, take 5: ";
    for (int n : infinite_range) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 8. SPLIT VIEW
// ==================================================================
void example_split_view() {
    std::cout << "\n=== 8. SPLIT VIEW ===" << std::endl;

    std::string text = "Hello World C++ Ranges";

    auto words = text | views::split(' ');

    std::cout << "Split by space:" << std::endl;
    for (const auto& word : words) {
        std::cout << "  '";
        for (char c : word) {
            std::cout << c;
        }
        std::cout << "'" << std::endl;
    }
}

// ==================================================================
// 9. JOIN VIEW
// ==================================================================
void example_join_view() {
    std::cout << "\n=== 9. JOIN VIEW ===" << std::endl;

```

```
    std::vector<std::vector<int>> nested = {
        {1, 2, 3},
        {4, 5},
        {6, 7, 8, 9}
    };

    auto flattened = nested | views::join;

    std::cout << "Flattened: ";
    for (int n : flattened) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 10. KEYS AND VALUES VIEWS (WITH MAP)
// ==================================================================
void example_keys_values_views() {
    std::cout << "\n=== 10. KEYS AND VALUES VIEWS ===" << std::endl;

    std::vector<std::pair<std::string, int>> data = {
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35}
    };

    auto keys = data | views::keys;
    std::cout << "Keys: ";
    for (const auto& key : keys) {
        std::cout << key << " ";
    }
    std::cout << std::endl;

    auto values = data | views::values;
    std::cout << "Values: ";
    for (int value : values) {
        std::cout << value << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 11. RANGES ALGORITHMS
// ==================================================================
void example_ranges_algorithms() {
    std::cout << "\n=== 11. RANGES ALGORITHMS ===" << std::endl;

    std::vector<int> numbers = {5, 2, 8, 1, 9, 3, 7, 4, 6};

    // Sort using ranges
    ranges::sort(numbers);
    std::cout << "Sorted: ";
    for (int n : numbers) {
```

```
        std::cout << n << " ";
    }
    std::cout << std::endl;

    // Find element
    auto it = ranges::find(numbers, 5);
    if (it != numbers.end()) {
        std::cout << "Found 5 at position: " << std::distance(numbers.begin(),
            it) << std::endl;
    }

    // Count elements greater than 5
    auto count = ranges::count_if(numbers, [](int n) { return n > 5; });
    std::cout << "Elements > 5: " << count << std::endl;
}

// ==================================================================
// 12. COMMON VIEW
// ==================================================================
void example_common_view() {
    std::cout << "\n=== 12. COMMON VIEW ===" << std::endl;

    auto range = views::iota(1, 11) | views::filter([](int n) { return n % 2
        == 0; });

    // Convert to common range (can use with legacy algorithms)
    auto common_range = range | views::common;

    std::cout << "Common view result: ";
    for (int n : common_range) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 13. ENUMERATE VIEW (C++23 preview style)
// ==================================================================
void example_enumerate_style() {
    std::cout << "\n=== 13. ENUMERATE STYLE ===" << std::endl;

    std::vector<std::string> fruits = {"Apple", "Banana", "Cherry", "Date"};

    // Manual enumeration with iota and zip_transform style
    int index = 0;
    for (const auto& fruit : fruits) {
        std::cout << index++ << ": " << fruit << std::endl;
    }
}

// ==================================================================
// 14. COMPLEX PIPELINE
// ==================================================================
void example_complex_pipeline() {
```

```
    std::cout << "\n=== 14. COMPLEX PIPELINE ===" << std::endl;

    // Generate numbers, filter, transform, and take
    auto pipeline = views::iota(1, 101)
        | views::filter([](int n) { return n % 3 == 0; })  // Divisible by 3
        | views::transform([](int n) { return n * n; })     // Square
        | views::take(5);                                   // First 5

    std::cout << "First 5 squares of numbers divisible by 3:" << std::endl;
    for (int n : pipeline) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// 15. CONVERTING VIEWS TO CONTAINERS
// ==================================================================
void example_to_container() {
    std::cout << "\n=== 15. CONVERTING VIEWS TO CONTAINERS ===" << std::endl;

    auto view = views::iota(1, 11)
        | views::filter([](int n) { return n % 2 == 0; })
        | views::transform([](int n) { return n * 2; });

    // Convert to vector
    std::vector<int> result(view.begin(), view.end());

    std::cout << "Converted to vector: ";
    for (int n : result) {
        std::cout << n << " ";
    }
    std::cout << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n================================================" << std::
        endl;
    std::cout << "    C++20 RANGES EXAMPLES" << std::endl;
    std::cout << "================================================" << std::
        endl;

    example_basic_ranges();
    example_filter_view();
    example_transform_view();
    example_take_drop_views();
    example_composing_views();
    example_reverse_view();
    example_iota_view();
    example_split_view();
    example_join_view();
```

```
    example_keys_values_views();
    example_ranges_algorithms();
    example_common_view();
    example_enumerate_style();
    example_complex_pipeline();
    example_to_container();

    std::cout << "\n==============================================" << std::endl;
    std::cout << "    ALL EXAMPLES COMPLETED" << std::endl;
    std::cout << "==============================================\n" << std::endl;

    return 0;
}
```

57 Source Code: RealTimeProgramming.cpp

File: `src/RealTimeProgramming.cpp`
Repository: View on GitHub

```cpp
// RealTimeProgramming.cpp
// Comprehensive guide to Real-Time Programming in Modern C++
//
// REAL-TIME SYSTEM CHARACTERISTICS:
// 1. Deterministic behavior - predictable execution times
// 2. Bounded worst-case execution time (WCET)
// 3. Low and predictable latency
// 4. No dynamic memory allocation in critical paths
// 5. Avoid operations with unbounded execution time
//
// KEY TOPICS COVERED:
// • Big O Notation and Time Complexity
// • STL containers for real-time systems
// • std::bitset vs std::vector<bool>
// • Memory pre-allocation strategies
// • RAII and deterministic resource management
// • Avoiding heap fragmentation
// • Custom allocators and memory pools

#include <iostream>
#include <vector>
#include <array>
#include <list>
#include <bitset>
#include <chrono>
#include <algorithm>
#include <memory>
#include <functional>
#include <string>
#include <string_view>
#include <optional>
#include <variant>
#include <iomanip>
#include <cstdint>

// ==========================================================================
// SECTION 1: Big O Notation - Understanding Time Complexity
// ==========================================================================

namespace big_o_notation {

void demonstrate_O1() {
    std::cout << "\nO(1) - Constant Time:\n";
    std::cout << " •  Execution time does not depend on input size\n";
    std::cout << " •  Always takes the same amount of time\n\n";
```

```

    std::array<int, 1000> data;
    data.fill(42);

    auto start = std::chrono::high_resolution_clock::now();

    // Array/vector element access - O(1)
    [[maybe_unused]] int value = data[500];

    // Hash table lookup - O(1) average case
    // Stack push/pop - O(1)

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);

    std::cout << "  Example: Array element access data[500]\n";
    std::cout << "  Time: " << duration.count() << " ns (constant, regardless of array size)\n";
    std::cout << "    BEST for real-time systems - predictable!\n";
}

void demonstrate_Ologn() {
    std::cout << "\nO(log n) - Logarithmic Time:\n";
    std::cout << " •  Execution time grows logarithmically with input size\n";
    std::cout << " •  Doubling input size adds constant time\n\n";

    std::vector<int> data(1000);
    for (size_t i = 0; i < data.size(); ++i) {
        data[i] = static_cast<int>(i);
    }

    auto start = std::chrono::high_resolution_clock::now();

    // Binary search - O(log n)
    bool found = std::binary_search(data.begin(), data.end(), 500);

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);

    std::cout << "  Example: Binary search in sorted array of 1000 elements\n";
    std::cout << "  Time: " << duration.count() << " ns\n";
    std::cout << "  Found: " << std::boolalpha << found << "\n";
    std::cout << "  Note: For 1M elements, time increases by only ~10x\n";
    std::cout << "    ACCEPTABLE for real-time systems with bounded size\n";
}

void demonstrate_On() {
    std::cout << "\nO(n) - Linear Time:\n";
    std::cout << " •  Execution time grows linearly with input size\n";
    std::cout << " •  Doubling input doubles execution time\n\n";

```

```
    std::vector<int> data(1000);
    for (size_t i = 0; i < data.size(); ++i) {
        data[i] = static_cast<int>(i);
    }

    auto start = std::chrono::high_resolution_clock::now();

    // Linear search - O(n)
    auto it = std::find(data.begin(), data.end(), 500);

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
        start);

    std::cout << "  Example: Linear search through 1000 elements\n";
    std::cout << "  Time: " << duration.count() << " ns\n";
    std::cout << "  Found: " << (it != data.end()) << "\n";
    std::cout << "    USE WITH CAUTION - bound the maximum container size!\n";
}

void demonstrate_Onlogn() {
    std::cout << "\nO(n log n) - Linearithmic Time:\n";
    std::cout << " •  Execution time grows at n * log(n)\n";
    std::cout << " •  Common in efficient sorting algorithms\n\n";

    std::vector<int> data(1000);
    for (size_t i = 0; i < data.size(); ++i) {
        data[i] = 1000 - static_cast<int>(i);  // Reverse order
    }

    auto start = std::chrono::high_resolution_clock::now();

    // Sorting - O(n log n) for std::sort
    std::sort(data.begin(), data.end());

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end
        - start);

    std::cout << "  Example: std::sort on 1000 elements\n";
    std::cout << "  Time: " << duration.count() << "  s\n";
    std::cout << "    AVOID in time-critical paths - do in initialization
        phase\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1: Big O Notation - Time Complexity ===\n";
    std::cout << std::string(70, '=') << "\n";

    demonstrate_O1();
    demonstrate_Ologn();
    demonstrate_On();
    demonstrate_Onlogn();
```

```

    std::cout << "\nREAL-TIME SYSTEM PRIORITY:\n\n";

    std::cout << "  REAL-TIME THREADS (Hard real-time, strict deadlines):\n";
    std::cout << "      O(1)     - REQUIRED (array access, bitset ops, hash lookup)\n";
    std::cout << "      O(log n) - Only if absolutely necessary AND size bounded\n";
    std::cout << "      O(n)     - AVOID in time-critical paths\n\n";

    std::cout << "  NON-REAL-TIME THREADS (Background tasks, logging, diagnostics):\n";
    std::cout << "      O(1)     - Ideal (always preferred)\n";
    std::cout << "      O(log n) - Acceptable (binary search, std::map)\n";
    std::cout << "      O(n)     - Acceptable with bounded size (linear search, iteration)\n";
    std::cout << "      O(n²)    - Avoid unless dataset very small\n\n";

    std::cout << "    GOLDEN RULE: Separate real-time from non-real-time threads!\n";
    std::cout << "    Use O(1) operations in control loops and sensor processing\n";
    std::cout << "    Offload O(n) operations to background threads\n";
}

} // namespace big_o_notation

// ============================================================================
// SECTION 1.5: Real-Time Thread Architecture
// ============================================================================

namespace thread_architecture {

// Example: Separating real-time and non-real-time concerns

class SensorSystem {
private:
    // Real-time data - accessed by RT thread
    std::array<double, 10> latest_readings_;  // O(1) access
    std::bitset<10> sensor_status_;           // O(1) operations
    size_t current_index_ = 0;

    // Non-real-time data - accessed by background thread
    std::vector<double> historical_data_;      // Can grow, processed in background

public:
    SensorSystem() {
        historical_data_.reserve(10000);  // Pre-allocate for background thread
```

```cpp
    }

    // REAL-TIME THREAD: O(1) operations only!
    void process_sensor_reading_RT(size_t sensor_id, double value) noexcept {
        // O(1) - Store latest reading
        latest_readings_[sensor_id] = value;

        // O(1) - Update status flag
        sensor_status_.set(sensor_id);

        // O(1) - No complex processing here!
    }

    // REAL-TIME THREAD: O(1) read access
    [[nodiscard]] double get_latest_reading_RT(size_t sensor_id) const noexcept {
        return latest_readings_[sensor_id];  // O(1)
    }

    // NON-REAL-TIME THREAD: Can use O(n) operations
    void log_historical_data_NonRT() {
        // O(n) - Iterate and log (acceptable in background thread)
        for (size_t i = 0; i < latest_readings_.size(); ++i) {
            if (sensor_status_.test(i)) {
                historical_data_.push_back(latest_readings_[i]);
            }
        }
    }

    // NON-REAL-TIME THREAD: Can use O(n) operations
    [[nodiscard]] double calculate_average_NonRT() const noexcept {
        if (historical_data_.empty()) return 0.0;

        // O(n) - Acceptable in non-RT thread
        double sum = 0.0;
        for (double val : historical_data_) {
            sum += val;
        }
        return sum / historical_data_.size();
    }

    // NON-REAL-TIME THREAD: Can use O(n log n) operations
    [[nodiscard]] double calculate_median_NonRT() {
        if (historical_data_.empty()) return 0.0;

        // O(n log n) - Sorting is expensive, do in background!
        auto data_copy = historical_data_;  // Copy to avoid modifying original
        std::sort(data_copy.begin(), data_copy.end());

        size_t mid = data_copy.size() / 2;
        return data_copy[mid];
    }
};
```

```

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1.5: Real-Time Thread Architecture ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "THREAD SEPARATION PRINCIPLE:\n\n";

    std::cout << "REAL-TIME THREADS:\n";
    std::cout << " • Purpose: Control loops, sensor processing, motor control\n";
    std::cout << " • Requirements: Deterministic, predictable timing\n";
    std::cout << " • Allowed: O(1) operations ONLY\n";
    std::cout << " • Examples:\n";
    std::cout << "    - Reading sensor value from array\n";
    std::cout << "    - Updating control output\n";
    std::cout << "    - Checking/setting status flags (bitset)\n";
    std::cout << "    - Simple arithmetic calculations\n\n";

    std::cout << "NON-REAL-TIME THREADS:\n";
    std::cout << " • Purpose: Logging, diagnostics, data aggregation, UI updates\n";
    std::cout << " • Requirements: Eventually complete, no strict deadlines\n";
    std::cout << " • Allowed: O(1), O(log n), O(n) operations\n";
    std::cout << " • Examples:\n";
    std::cout << "    - Writing logs to disk\n";
    std::cout << "    - Calculating statistics (average, median)\n";
    std::cout << "    - Searching historical data\n";
    std::cout << "    - Generating reports\n\n";

    std::cout << "EXAMPLE: Sensor System with Thread Separation\n\n";

    SensorSystem system;

    // Simulate real-time thread
    std::cout << "  REAL-TIME THREAD (1 kHz control loop):\n";
    auto rt_start = std::chrono::high_resolution_clock::now();

    for (size_t i = 0; i < 10; ++i) {
        system.process_sensor_reading_RT(i % 10, i * 0.5);  // O(1) per reading
    }

    auto rt_end = std::chrono::high_resolution_clock::now();
    auto rt_duration = std::chrono::duration_cast<std::chrono::nanoseconds>(rt_end - rt_start);

    std::cout << "    Processed 10 readings in " << rt_duration.count() << " ns\n";
    std::cout << "    Per-reading time: " << (rt_duration.count() / 10) << " ns (deterministic!)\n\n";

    // Simulate non-real-time thread
```

```
    std::cout << "  NON-REAL-TIME THREAD (background processing):\n";

    system.log_historical_data_NonRT();  // O(n) - acceptable

    auto bg_start = std::chrono::high_resolution_clock::now();
    double avg = system.calculate_average_NonRT();  // O(n)
    auto bg_end = std::chrono::high_resolution_clock::now();
    auto bg_duration = std::chrono::duration_cast<std::chrono::microseconds>(
        bg_end - bg_start);

    std::cout << "    Calculated average: " << avg << "\n";
    std::cout << "    Time: " << bg_duration.count() << " s (non-critical!)\n
        \n";

    std::cout << "    ARCHITECTURE: Real-time thread stays O(1), background
        handles O(n)\n";
    std::cout << "    BENEFIT: Control loop remains deterministic and fast\n";
    std::cout << "    BENEFIT: Complex processing doesn't block time-critical
        operations\n";
}

} // namespace thread_architecture

//
    ==========================================================================
// SECTION 2: std::bitset vs std::vector<bool> - The Better Choice
//
    ==========================================================================

namespace bitset_vs_vector_bool {

// std::vector<bool> is a problematic specialization
// - Uses space optimization (1 bit per element)
// - Does NOT behave like a regular container
// - operator[] returns a proxy object, not a reference
// - Cannot get pointer to element
// - NOT suitable for real-time systems due to proxy complexity

// std::bitset is designed for bit manipulation
// - Fixed size at compile time
// - O(1) access and modification
// - Efficient bit operations (AND, OR, XOR, shift)
// - No dynamic memory allocation
// - PERFECT for real-time systems

template<size_t N>
class RealTimeFlagManager {
private:
    std::bitset<N> flags_;

public:
    // O(1) - Set a flag
```

```
    constexpr void set_flag(size_t index) noexcept {
        if (index < N) {
            flags_.set(index);
        }
    }

    // O(1) - Clear a flag
    constexpr void clear_flag(size_t index) noexcept {
        if (index < N) {
            flags_.reset(index);
        }
    }

    // O(1) - Toggle a flag
    constexpr void toggle_flag(size_t index) noexcept {
        if (index < N) {
            flags_.flip(index);
        }
    }

    // O(1) - Check a flag
    [[nodiscard]] constexpr bool is_set(size_t index) const noexcept {
        return index < N ? flags_.test(index) : false;
    }

    // O(1) - Count set bits (hardware instruction on modern CPUs)
    [[nodiscard]] constexpr size_t count() const noexcept {
        return flags_.count();
    }

    // O(1) - Check if any flags are set
    [[nodiscard]] constexpr bool any() const noexcept {
        return flags_.any();
    }

    // O(1) - Check if all flags are set
    [[nodiscard]] constexpr bool all() const noexcept {
        return flags_.all();
    }

    // O(1) - Check if no flags are set
    [[nodiscard]] constexpr bool none() const noexcept {
        return flags_.none();
    }

    // Bitwise operations - O(1)
    constexpr RealTimeFlagManager operator&(const RealTimeFlagManager& other)
        const noexcept {
        RealTimeFlagManager result;
        result.flags_ = flags_ & other.flags_;
        return result;
    }

    constexpr RealTimeFlagManager operator|(const RealTimeFlagManager& other)
```

```
        const noexcept {
         RealTimeFlagManager result;
         result.flags_ = flags_ | other.flags_;
         return result;
    }
};

// Example: Sensor status flags for embedded system
constexpr size_t NUM_SENSORS = 32;

enum class SensorStatus : size_t {
    TEMPERATURE_OK = 0,
    PRESSURE_OK = 1,
    HUMIDITY_OK = 2,
    VOLTAGE_OK = 3,
    CURRENT_OK = 4,
    GPS_LOCK = 5,
    IMU_CALIBRATED = 6,
    COMMUNICATION_OK = 7,
    // ... up to 32 sensors
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 2: std::bitset vs std::vector<bool> ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "WHY std::vector<bool> IS PROBLEMATIC:\n";
    std::cout << "    Space-optimized specialization (1 bit per element)\n";
    std::cout << "    operator[] returns proxy object, not bool&\n";
    std::cout << "    Cannot take address of elements\n";
    std::cout << "    Non-standard container behavior\n";
    std::cout << "    Dynamic memory allocation\n";
    std::cout << "    Unpredictable performance\n\n";

    std::cout << "WHY std::bitset IS BETTER FOR REAL-TIME:\n";
    std::cout << "    Fixed size at compile time (no heap allocation)\n";
    std::cout << "    O(1) access and modification\n";
    std::cout << "    Efficient bitwise operations (AND, OR, XOR)\n";
    std::cout << "    Hardware-optimized bit counting\n";
    std::cout << "    Standard, predictable behavior\n";
    std::cout << "    Deterministic performance\n\n";

    // Demonstrate real-time flag manager
    RealTimeFlagManager<NUM_SENSORS> sensor_status;

    std::cout << "EXAMPLE: Real-Time Sensor Status Manager\n";
    std::cout << "  Using std::bitset<32> for 32 sensor flags\n\n";

    // Set some flags
    sensor_status.set_flag(static_cast<size_t>(SensorStatus::TEMPERATURE_OK));
    sensor_status.set_flag(static_cast<size_t>(SensorStatus::PRESSURE_OK));
    sensor_status.set_flag(static_cast<size_t>(SensorStatus::GPS_LOCK));

```

```
    std::cout << "  Set flags: TEMPERATURE_OK, PRESSURE_OK, GPS_LOCK\n";
    std::cout << "  Active sensors: " << sensor_status.count() << " / " << NUM_SENSORS << "\n";
    std::cout << "  All sensors OK? " << std::boolalpha << sensor_status.all() << "\n";
    std::cout << "  Any sensor OK? " << sensor_status.any() << "\n";

    // Check specific flag
    bool temp_ok = sensor_status.is_set(static_cast<size_t>(SensorStatus::TEMPERATURE_OK));
    std::cout << "  Temperature sensor OK? " << temp_ok << "\n\n";

    // Timing comparison
    constexpr size_t ITERATIONS = 1000000;

    std::bitset<64> bitset_flags;
    auto start_bitset = std::chrono::high_resolution_clock::now();
    for (size_t i = 0; i < ITERATIONS; ++i) {
        bitset_flags.set(i % 64);
        [[maybe_unused]] bool val = bitset_flags.test(i % 64);
    }
    auto end_bitset = std::chrono::high_resolution_clock::now();
    auto bitset_duration = std::chrono::duration_cast<std::chrono::microseconds>(end_bitset - start_bitset);

    std::vector<bool> vector_bool(64, false);
    auto start_vector = std::chrono::high_resolution_clock::now();
    for (size_t i = 0; i < ITERATIONS; ++i) {
        vector_bool[i % 64] = true;
        [[maybe_unused]] bool val = vector_bool[i % 64];
    }
    auto end_vector = std::chrono::high_resolution_clock::now();
    auto vector_duration = std::chrono::duration_cast<std::chrono::microseconds>(end_vector - start_vector);

    std::cout << "PERFORMANCE COMPARISON (" << ITERATIONS << " operations):\n";
    std::cout << "  std::bitset<64>:     " << std::setw(8) << bitset_duration.count() << "  s\n";
    std::cout << "  std::vector<bool>:   " << std::setw(8) << vector_duration.count() << "  s\n";
    std::cout << "  Speedup:             " << std::fixed << std::setprecision(2)
              << (static_cast<double>(vector_duration.count()) / bitset_duration.count()) << "x\n\n";

    std::cout << "   RECOMMENDATION: Use std::bitset for real-time flag management!\n";
}

} // namespace bitset_vs_vector_bool

// ==========================================================================
```

```
// SECTION 3: STL Containers - Real-Time Best Practices
// ==========================================================================

namespace stl_containers_realtime {

// Container Complexity Reference:
//
// std::vector:
//   - Access:       O(1)
//   - Insert front: O(n)
//   - Insert back:  O(1) amortized, O(n) worst case (reallocation!)
//   - Insert mid:   O(n)
//   - Real-time:    Use reserve() to pre-allocate!
//
// std::array:
//   - Access:       O(1)
//   - Size:         Fixed at compile time
//   - Real-time:    PERFECT - no dynamic allocation, stack-based
//
// std::deque:
//   - Access:       O(1)
//   - Insert front: O(1)
//   - Insert back:  O(1)
//   - Real-time:    Acceptable if size bounded
//
// std::list:
//   - Access:       O(n) - must traverse!
//   - Insert:       O(1) if you have iterator
//   - Real-time:    AVOID - dynamic allocation per element, poor cache
//
// std::map/std::set:
//   - Access:       O(log n)
//   - Insert:       O(log n)
//   - Real-time:    Use with custom allocator or bounded size

// Example 1: Pre-allocated std::vector - The Right Way
class RealTimeDataBuffer {
private:
    static constexpr size_t MAX_SAMPLES = 1000;
    std::vector<double> samples_;

public:
    RealTimeDataBuffer() {
        //  CRITICAL: Pre-allocate during initialization
        // This prevents reallocation during real-time operation
        samples_.reserve(MAX_SAMPLES);

        std::cout << "  Buffer created with capacity: " << samples_.capacity()
            << "\n";
        std::cout << "  No reallocation will occur until this limit\n";
    }
```

```

    // O(1) - Add sample (no reallocation if within capacity)
    bool add_sample(double value) noexcept {
        if (samples_.size() < MAX_SAMPLES) {
            samples_.push_back(value);  // O(1) - deterministic!
            return true;
        }
        return false;  // Buffer full
    }

    // O(1) - Get sample
    [[nodiscard]] std::optional<double> get_sample(size_t index) const noexcept {
        if (index < samples_.size()) {
            return samples_[index];
        }
        return std::nullopt;
    }

    // O(n) - Clear buffer (but no deallocation!)
    void clear() noexcept {
        samples_.clear();
        // Capacity remains unchanged - memory is retained!
    }

    [[nodiscard]] size_t size() const noexcept { return samples_.size(); }
    [[nodiscard]] size_t capacity() const noexcept { return samples_.capacity(); }
};

// Example 2: Fixed-size std::array - Even Better
template<size_t N>
class CircularBuffer {
private:
    std::array<double, N> buffer_;
    size_t head_ = 0;
    size_t count_ = 0;

public:
    // O(1) - Add element (overwrites oldest if full)
    constexpr void push(double value) noexcept {
        buffer_[head_] = value;
        head_ = (head_ + 1) % N;
        if (count_ < N) {
            ++count_;
        }
    }

    // O(1) - Get element (0 = most recent)
    [[nodiscard]] constexpr std::optional<double> get(size_t index) const noexcept {
        if (index >= count_) {
            return std::nullopt;
        }
```

```
        size_t actual_index = (head_ + N - count_ + index) % N;
        return buffer_[actual_index];
    }

    [[nodiscard]] constexpr size_t size() const noexcept { return count_; }
    [[nodiscard]] constexpr bool is_full() const noexcept { return count_ == N; }

    // Statistics - O(n) but bounded by N
    [[nodiscard]] double average() const noexcept {
        if (count_ == 0) return 0.0;
        double sum = 0.0;
        for (size_t i = 0; i < count_; ++i) {
            size_t idx = (head_ + N - count_ + i) % N;
            sum += buffer_[idx];
        }
        return sum / count_;
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 3: STL Containers - Real-Time Best Practices ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "CONTAINER SELECTION FOR REAL-TIME SYSTEMS:\n\n";

    std::cout << "  PREFERRED:\n";
    std::cout << "   1. std::array<T, N>           - O(1) access, no heap, compile-time size\n";
    std::cout << "   2. std::vector<T> + reserve() - O(1) access, pre-allocated\n";
    std::cout << "   3. std::bitset<N>             - O(1) bit ops, no heap\n";
    std::cout << "   4. std::string_view           - O(1) string viewing, no allocation\n\n";

    std::cout << "  USE WITH CAUTION (bound size!):\n";
    std::cout << " •  std::deque<T>                - O(1) both ends, but allocates chunks\n";
    std::cout << " •  std::map/set<T>              - O(log n), allocates per element\n\n";

    std::cout << "  AVOID:\n";
    std::cout << " •  std::list<T>                 - Poor cache locality, alloc per element\n";
    std::cout << " •  std::forward_list<T>         - Poor cache locality\n";
    std::cout << " •  std::vector<bool>            - Use std::bitset instead!\n";
    std::cout << " •  Unbounded growth containers - Always set max size!\n\n";

    // Demonstrate pre-allocated vector
    std::cout << "EXAMPLE 1: Pre-allocated std::vector\n";
    RealTimeDataBuffer buffer;
```

```

    for (int i = 0; i < 100; ++i) {
        buffer.add_sample(i * 0.1);
    }

    std::cout << "  Added 100 samples\n";
    std::cout << "  Current size: " << buffer.size() << "\n";
    std::cout << "  Capacity: " << buffer.capacity() << "\n";
    std::cout << "    No reallocation occurred - deterministic O(1) insertion!\n\n";

    // Demonstrate fixed-size array
    std::cout << "EXAMPLE 2: Fixed-size circular buffer (std::array<T, N>)\n";
    CircularBuffer<10> circ_buffer;

    for (int i = 0; i < 15; ++i) {
        circ_buffer.push(i * 1.5);
    }

    std::cout << "  Pushed 15 values into buffer of size 10\n";
    std::cout << "  Current size: " << circ_buffer.size() << "\n";
    std::cout << "  Average: " << circ_buffer.average() << "\n";
    std::cout << "    No heap allocation - pure stack-based, deterministic!\n\n";

    // Demonstrate reserve() importance
    std::cout << "CRITICAL: Always use reserve() for std::vector in real-time code!\n\n";

    std::vector<int> bad_vector;  // No reserve
    auto start_bad = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; ++i) {
        bad_vector.push_back(i);  // May trigger reallocations!
    }
    auto end_bad = std::chrono::high_resolution_clock::now();
    auto bad_duration = std::chrono::duration_cast<std::chrono::microseconds>(end_bad - start_bad);

    std::vector<int> good_vector;
    good_vector.reserve(1000);  // Pre-allocate
    auto start_good = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; ++i) {
        good_vector.push_back(i);  // No reallocations!
    }
    auto end_good = std::chrono::high_resolution_clock::now();
    auto good_duration = std::chrono::duration_cast<std::chrono::microseconds>(end_good - start_good);

    std::cout << "  Without reserve(): " << bad_duration.count() << " s (unpredictable!)\n";
    std::cout << "  With reserve():    " << good_duration.count() << " s (deterministic!)\n";
    std::cout << "  Speedup:           " << std::fixed << std::setprecision(2)
              << (static_cast<double>(bad_duration.count()) / good_duration.
```

```
                    count()) << "x\n\n";

    std::cout << "    GOLDEN RULE: reserve() prevents non-deterministic reallocations!\n";
}

} // namespace stl_containers_realtime

// ============================================================================
// SECTION 3.5: std::list::splice() - O(1) Element Movement
// ============================================================================

namespace list_splice_realtime {

// std::list::splice() is PERFECT for real-time systems because:
// 1. O(1) complexity for moving elements (constant time!)
// 2. NO MEMORY ALLOCATION - just pointer manipulation
// 3. NO COPYING/MOVING - elements stay in place
// 4. Deterministic and predictable performance
// 5. Iterators remain valid after splice
//
// Use cases in real-time systems:
// - Free lists / object pools
// - Task queue management
// - Event scheduling
// - Moving data between priority queues

struct Task {
    int id;
    std::string name;
    int priority;

    Task(int i, std::string n, int p) : id(i), name(std::move(n)), priority(p) {}
};

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 3.5: std::list::splice() - O(1) Element Movement\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "WHY splice() IS CRITICAL FOR REAL-TIME SYSTEMS:\n";
    std::cout << " •  O(1) constant time - no iteration required\n";
    std::cout << " •  NO memory allocation - just rewires pointers\n";
    std::cout << " •  NO copying/moving - elements stay in original memory\n";
    std::cout << " •  Iterators remain valid - predictable behavior\n";
    std::cout << " •  Deterministic performance - perfect for hard real-time\n\n";

    // ========================================================================
```

```
    // EXAMPLE 1: splice() entire list - O(1)
    // ======================================================================
    std::cout << std::string(80, '-') << "\n";
    std::cout << "EXAMPLE 1: Splice Entire List - O(1) Operation\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::list<Task> active_tasks;
    active_tasks.emplace_back(1, "Process Sensor Data", 10);
    active_tasks.emplace_back(2, "Update Control Loop", 20);
    active_tasks.emplace_back(3, "Send Telemetry", 5);

    std::list<Task> pending_tasks;
    pending_tasks.emplace_back(4, "Log Event", 1);
    pending_tasks.emplace_back(5, "Check Diagnostics", 3);

    std::cout << "BEFORE splice():\n";
    std::cout << "  Active tasks: " << active_tasks.size() << "\n";
    for (const auto& t : active_tasks) {
        std::cout << "    Task " << t.id << ": " << t.name << " (priority: "
            << t.priority << ")\n";
    }
    std::cout << "  Pending tasks: " << pending_tasks.size() << "\n";
    for (const auto& t : pending_tasks) {
        std::cout << "    Task " << t.id << ": " << t.name << " (priority: "
            << t.priority << ")\n";
    }

    // Splice entire pending_tasks list into active_tasks at the end
    // This is O(1) - just rewires a few pointers!
    auto start = std::chrono::high_resolution_clock::now();
    active_tasks.splice(active_tasks.end(), pending_tasks);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
        start);

    std::cout << "\nAFTER splice(active.end(), pending):\n";
    std::cout << "  Active tasks: " << active_tasks.size() << "\n";
    for (const auto& t : active_tasks) {
        std::cout << "    Task " << t.id << ": " << t.name << " (priority: "
            << t.priority << ")\n";
    }
    std::cout << "  Pending tasks: " << pending_tasks.size() << " (empty!)\n";
    std::cout << "   Time taken: " << duration.count() << " ns (O(1) -
        constant time!)\n";
    std::cout << "   NO allocation, NO copying, just pointer rewiring!\n\n";

    // ======================================================================
    // EXAMPLE 2: splice() single element - O(1)
    // ======================================================================
    std::cout << std::string(80, '-') << "\n";
    std::cout << "EXAMPLE 2: Splice Single Element - O(1) Operation\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::list<Task> high_priority;
```

```
    high_priority.emplace_back(10, "Critical Shutdown", 100);
    high_priority.emplace_back(11, "Emergency Stop", 99);

    std::list<Task> normal_priority;
    normal_priority.emplace_back(20, "Routine Check", 10);
    normal_priority.emplace_back(21, "Update Display", 8);
    normal_priority.emplace_back(22, "Log Status", 5);

    std::cout << "SCENARIO: Emergency task detected in normal queue!\n\n";

    std::cout << "BEFORE splice():\n";
    std::cout << "  High priority: " << high_priority.size() << " tasks\n";
    std::cout << "  Normal priority: " << normal_priority.size() << " tasks\n";
    for (const auto& t : normal_priority) {
        std::cout << "    Task " << t.id << ": " << t.name << " (priority: " << t.priority << ")\n";
    }

    // Find the "Update Display" task and move it to high priority
    auto it = std::find_if(normal_priority.begin(), normal_priority.end(),
                           [](const Task& t) { return t.id == 21; });

    if (it != normal_priority.end()) {
        std::cout << "\n  Found Task 21: " << it->name << " - moving to high priority!\n";

        // Splice single element - O(1)!
        start = std::chrono::high_resolution_clock::now();
        high_priority.splice(high_priority.end(), normal_priority, it);
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
    }

    std::cout << "\nAFTER splice(high.end(), normal, iterator):\n";
    std::cout << "  High priority: " << high_priority.size() << " tasks\n";
    for (const auto& t : high_priority) {
        std::cout << "    Task " << t.id << ": " << t.name << " (priority: " << t.priority << ")\n";
    }
    std::cout << "  Normal priority: " << normal_priority.size() << " tasks\n";
    for (const auto& t : normal_priority) {
        std::cout << "    Task " << t.id << ": " << t.name << " (priority: " << t.priority << ")\n";
    }
    std::cout << "   Time taken: " << duration.count() << " ns (O(1) - constant time!)\n";
    std::cout << "   Task moved instantly without copying!\n\n";

    // ======================================================================
    // EXAMPLE 3: splice() range of elements - O(n) where n = range size
    // ======================================================================
```

```
    std::cout << std::string(80, '-') << "\n";
    std::cout << "EXAMPLE 3: Splice Range of Elements - O(n) but NO Allocation
        \n";
    std::cout << std::string(80, '-') << "\n\n";

    std::list<int> source = {10, 20, 30, 40, 50, 60, 70, 80};
    std::list<int> dest = {1, 2, 3};

    std::cout << "BEFORE splice():\n";
    std::cout << "  Source: ";
    for (int v : source) std::cout << v << " ";
    std::cout << "\n  Dest: ";
    for (int v : dest) std::cout << v << " ";
    std::cout << "\n";

    // Move elements 30, 40, 50 from source to dest
    auto range_start = std::find(source.begin(), source.end(), 30);
    auto range_end = std::find(source.begin(), source.end(), 60);  // One past
        last element

    std::cout << "\n  Moving range [30, 40, 50] from source to dest...\n";

    // Splice range - O(n) where n = distance(range_start, range_end)
    // But still NO memory allocation!
    start = std::chrono::high_resolution_clock::now();
    dest.splice(dest.end(), source, range_start, range_end);
    end = std::chrono::high_resolution_clock::now();
    duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
        start);

    std::cout << "\nAFTER splice(dest.end(), source, first, last):\n";
    std::cout << "  Source: ";
    for (int v : source) std::cout << v << " ";
    std::cout << "\n  Dest: ";
    for (int v : dest) std::cout << v << " ";
    std::cout << "\n";
    std::cout << "   Time taken: " << duration.count() << " ns (O(n) but
        predictable!)\n";
    std::cout << "   NO allocation - just pointer updates for range!\n\n";

    // ======================================================================
    // EXAMPLE 4: Real-time pool management with splice()
    // ======================================================================
    std::cout << std::string(80, '-') << "\n";
    std::cout << "EXAMPLE 4: Free List Pool Management - Real-Time Pattern\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "SCENARIO: Object pool using splice() for O(1) allocation/
        deallocation\n\n";

    struct Buffer {
        int id;
        std::array<char, 256> data;
        explicit Buffer(int i) : id(i), data{} {}
```

```
    };

    // Free pool (available buffers)
    std::list<Buffer> free_pool;
    for (int i = 0; i < 5; ++i) {
        free_pool.emplace_back(i);
    }

    // Active pool (in-use buffers)
    std::list<Buffer> active_pool;

    std::cout << "Initial state:\n";
    std::cout << "  Free pool: " << free_pool.size() << " buffers\n";
    std::cout << "  Active pool: " << active_pool.size() << " buffers\n\n";

    // Allocate buffer from free pool - O(1)
    std::cout << "Allocating 3 buffers from free pool...\n";
    for (int i = 0; i < 3; ++i) {
        auto it = free_pool.begin();
        std::cout << "  Allocating buffer " << it->id << "\n";
        active_pool.splice(active_pool.end(), free_pool, it);  // O(1)!
    }

    std::cout << "\nAfter allocation:\n";
    std::cout << "  Free pool: " << free_pool.size() << " buffers\n";
    std::cout << "  Active pool: " << active_pool.size() << " buffers (";
    for (const auto& b : active_pool) std::cout << b.id << " ";
    std::cout << ")\n\n";

    // Deallocate buffer back to free pool - O(1)
    std::cout << "Deallocating buffer 1 back to free pool...\n";
    auto to_free = std::find_if(active_pool.begin(), active_pool.end(),
                                [](const Buffer& b) { return b.id == 1; });
    if (to_free != active_pool.end()) {
        free_pool.splice(free_pool.end(), active_pool, to_free);  // O(1)!
        std::cout << "  Buffer 1 returned to free pool\n";
    }

    std::cout << "\nFinal state:\n";
    std::cout << "  Free pool: " << free_pool.size() << " buffers\n";
    std::cout << "  Active pool: " << active_pool.size() << " buffers (";
    for (const auto& b : active_pool) std::cout << b.id << " ";
    std::cout << ")\n\n";

    std::cout << "   Pool allocation/deallocation in O(1) with splice()!\n";
    std::cout << "   NO dynamic allocation - perfect for real-time systems!\n
        \n";

    // ======================================================================
    // KEY TAKEAWAYS
    // ======================================================================
    std::cout << std::string(80, '=') << "\n";
    std::cout << "KEY TAKEAWAYS: Why splice() is Essential for Real-Time C++\n
        ";
```

```
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "1. PERFORMANCE:\n";
    std::cout << " •   splice() entire list: O(1) - constant time\n";
    std::cout << " •   splice() single element: O(1) - constant time\n";
    std::cout << " •   splice() range: O(n) where n = range size (but predictable!)\n\n";

    std::cout << "2. MEMORY BEHAVIOR:\n";
    std::cout << " •   NO memory allocation - just pointer manipulation\n";
    std::cout << " •   NO copying or moving - elements stay in original memory\n";
    std::cout << " •   Deterministic and bounded - perfect for hard real-time\n\n";

    std::cout << "3. ITERATOR VALIDITY:\n";
    std::cout << " •   All iterators, pointers, and references remain valid\n";
    std::cout << " •   Safe to cache iterators across splice operations\n\n";

    std::cout << "4. REAL-TIME USE CASES:\n";
    std::cout << " •   Free list / object pool management\n";
    std::cout << " •   Task queue reordering (priority changes)\n";
    std::cout << " •   Event scheduling and rescheduling\n";
    std::cout << " •   Moving data between priority queues\n";
    std::cout << " •   Load balancing across worker queues\n\n";

    std::cout << "5. ALTERNATIVES ARE WORSE:\n";
    std::cout << " •   std::copy() + erase(): Requires copying, O(n) erase\n";
    std::cout << " •   std::move() + erase(): Still requires move, O(n) erase\n";
    std::cout << " •   Manual reallocation: Unpredictable timing, allocation overhead\n";
    std::cout << " •   splice() is THE optimal solution for list element movement!\n\n";

    std::cout << "  GOLDEN RULE: Use splice() when you need to move elements\n";
    std::cout << "   between std::list containers in real-time critical code!\n";
}

} // namespace list_splice_realtime

// ==========================================================================
// SECTION 4: Memory Management - RAII and Custom Allocators
// ==========================================================================

namespace memory_management {

```

```cpp
// RAII (Resource Acquisition Is Initialization)
// - Resources acquired in constructor
// - Resources released in destructor
// - Deterministic, exception-safe cleanup
// - No manual memory management needed

// Example: Real-time resource manager using RAII
template<typename T, size_t PoolSize>
class MemoryPool {
private:
    std::array<T, PoolSize> pool_;
    std::array<bool, PoolSize> in_use_;

public:
    MemoryPool() {
        in_use_.fill(false);
        std::cout << "  Memory pool created: " << PoolSize << " objects of size "
                  << sizeof(T) << " bytes\n";
        std::cout << "  Total pool size: " << (PoolSize * sizeof(T)) << " bytes\n";
    }

    // Acquire object from pool - O(n) but bounded by PoolSize
    [[nodiscard]] T* acquire() noexcept {
        for (size_t i = 0; i < PoolSize; ++i) {
            if (!in_use_[i]) {
                in_use_[i] = true;
                return &pool_[i];
            }
        }
        return nullptr;  // Pool exhausted
    }

    // Release object back to pool - O(1)
    void release(T* ptr) noexcept {
        if (!ptr) return;

        // Find the object in pool
        for (size_t i = 0; i < PoolSize; ++i) {
            if (&pool_[i] == ptr) {
                in_use_[i] = false;
                return;
            }
        }
    }

    [[nodiscard]] size_t available() const noexcept {
        return std::count(in_use_.begin(), in_use_.end(), false);
    }
};

// RAII wrapper for pool-allocated object
template<typename T>
```

```
class PoolPtr {
private:
    T* ptr_;
    std::function<void(T*)> deleter_;

public:
    PoolPtr(T* ptr, std::function<void(T*)> deleter)
        : ptr_(ptr), deleter_(std::move(deleter)) {}

    ~PoolPtr() {
        if (ptr_ && deleter_) {
            deleter_(ptr_);
        }
    }

    // Delete copy operations
    PoolPtr(const PoolPtr&) = delete;
    PoolPtr& operator=(const PoolPtr&) = delete;

    // Move operations
    PoolPtr(PoolPtr&& other) noexcept
        : ptr_(other.ptr_), deleter_(std::move(other.deleter_)) {
        other.ptr_ = nullptr;
    }

    PoolPtr& operator=(PoolPtr&& other) noexcept {
        if (this != &other) {
            if (ptr_ && deleter_) {
                deleter_(ptr_);
            }
            ptr_ = other.ptr_;
            deleter_ = std::move(other.deleter_);
            other.ptr_ = nullptr;
        }
        return *this;
    }

    T* get() noexcept { return ptr_; }
    const T* get() const noexcept { return ptr_; }

    T& operator*() noexcept { return *ptr_; }
    const T& operator*() const noexcept { return *ptr_; }

    T* operator->() noexcept { return ptr_; }
    const T* operator->() const noexcept { return ptr_; }

    explicit operator bool() const noexcept { return ptr_ != nullptr; }
};

struct SensorData {
    double temperature;
    double pressure;
    uint64_t timestamp;

```

```
    SensorData() : temperature(0.0), pressure(0.0), timestamp(0) {}
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 4: Memory Management - RAII and Memory Pools ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "REAL-TIME MEMORY MANAGEMENT PRINCIPLES:\n\n";

    std::cout << "  DO:\n";
    std::cout << "   1. Pre-allocate all memory during initialization\n";
    std::cout << "   2. Use memory pools for frequent alloc/dealloc\n";
    std::cout << "   3. Use RAII for deterministic resource cleanup\n";
    std::cout << "   4. Prefer stack allocation over heap\n";
    std::cout << "   5. Use std::unique_ptr for ownership (minimal overhead)\n\n";

    std::cout << "  DON'T:\n";
    std::cout << "   1. Call new/delete in time-critical paths\n";
    std::cout << "   2. Use std::shared_ptr in tight loops (atomic overhead)\n";
    std::cout << "   3. Allow unbounded memory growth\n";
    std::cout << "   4. Rely on garbage collection\n";
    std::cout << "   5. Use dynamic allocation with unpredictable size\n\n";

    std::cout << "EXAMPLE: Memory Pool with RAII\n";

    // Create memory pool (during initialization phase)
    MemoryPool<SensorData, 100> sensor_pool;

    std::cout << "  Available slots: " << sensor_pool.available() << "\n\n";

    // Acquire objects from pool (during real-time operation)
    {
        auto sensor1 = PoolPtr<SensorData>(
            sensor_pool.acquire(),
            [&sensor_pool](SensorData* ptr) { sensor_pool.release(ptr); }
        );

        auto sensor2 = PoolPtr<SensorData>(
            sensor_pool.acquire(),
            [&sensor_pool](SensorData* ptr) { sensor_pool.release(ptr); }
        );

        if (sensor1 && sensor2) {
            sensor1->temperature = 25.5;
            sensor1->pressure = 1013.25;

            std::cout << "  Acquired 2 objects from pool\n";
            std::cout << "  Available slots: " << sensor_pool.available() << "\n";
            std::cout << "  Sensor 1 temp: " << sensor1->temperature << "°C\n"
```

```
                ;
        }

        // Objects automatically released when PoolPtr goes out of scope (RAII!)
        std::cout << "  Exiting scope...\n";
    }

    std::cout << "  Objects released automatically via RAII\n";
    std::cout << "  Available slots: " << sensor_pool.available() << "\n\n";

    std::cout << "    DETERMINISTIC: No heap fragmentation, predictable timing!\n";
    std::cout << "    SAFE: RAII ensures resources are always released!\n";
}

} // namespace memory_management

// ==============================================================================
// SECTION 5: Modern C++ Features for Real-Time Systems
// ==============================================================================

namespace modern_cpp_features {

// C++17 std::string_view - Zero-copy string viewing
void process_command(std::string_view cmd) noexcept {
    // No string copy - just a view into existing data
    if (cmd == "START") {
        std::cout << "    Starting system...\n";
    } else if (cmd == "STOP") {
        std::cout << "    Stopping system...\n";
    }
}

// C++17 std::optional - Express optional values without exceptions
[[nodiscard]] std::optional<double> safe_divide(double a, double b) noexcept {
    if (b == 0.0) {
        return std::nullopt;  // No exception thrown!
    }
    return a / b;
}

// C++17 std::variant - Type-safe union without heap allocation
using SensorValue = std::variant<int, double, std::array<uint8_t, 4>>;

void process_sensor_value(const SensorValue& value) noexcept {
    std::visit([](const auto& val) {
        using T = std::decay_t<decltype(val)>;
        if constexpr (std::is_same_v<T, int>) {
            std::cout << "    Integer sensor: " << val << "\n";
```

```
        } else if constexpr (std::is_same_v<T, double>) {
            std::cout << "    Double sensor: " << val << "\n";
        } else {
            std::cout << "    Array sensor\n";
        }
    }, value);
}

// constexpr functions - Compile-time computation
constexpr uint32_t compute_crc32(std::string_view data) {
    uint32_t crc = 0xFFFFFFFF;
    for (char c : data) {
        crc ^= static_cast<uint32_t>(c);
        for (int i = 0; i < 8; ++i) {
            crc = (crc >> 1) ^ (0xEDB88320 & (-(crc & 1)));
        }
    }
    return ~crc;
}

// Computed at compile time!
constexpr uint32_t header_crc = compute_crc32("HEADER");

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 5: Modern C++ Features for Real-Time ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "MODERN C++ FEATURES SUITABLE FOR REAL-TIME:\n\n";

    // std::string_view
    std::cout << "1. std::string_view (C++17) - Zero-copy string handling:\n";
    std::string command = "START";
    process_command(command);  // No copy, just a view
    process_command("STOP");   // Works with string literals too
    std::cout << "      No allocation, no copy, just a pointer + size\n\n";

    // std::optional
    std::cout << "2. std::optional (C++17) - Safe error handling without
        exceptions:\n";
    auto result1 = safe_divide(10.0, 2.0);
    auto result2 = safe_divide(10.0, 0.0);

    std::cout << "    10.0 / 2.0 = ";
    if (result1) {
        std::cout << *result1 << " (valid)\n";
    }

    std::cout << "    10.0 / 0.0 = ";
    if (result2) {
        std::cout << *result2 << "\n";
    } else {
        std::cout << "error (nullopt)\n";
    }
```

```
    std::cout << "      No exceptions, no heap allocation\n\n";

    // std::variant
    std::cout << "3. std::variant (C++17) - Type-safe union without heap:\n";
    SensorValue val1 = 42;
    SensorValue val2 = 3.14159;
    process_sensor_value(val1);
    process_sensor_value(val2);
    std::cout << "      Type-safe, no heap allocation, no virtual dispatch\n\n";

    // constexpr
    std::cout << "4. constexpr - Compile-time computation:\n";
    std::cout << "    Header CRC32 (computed at compile-time): 0x"
              << std::hex << std::uppercase << header_crc << std::dec << "\n";
    std::cout << "      Zero runtime cost - computed by compiler!\n\n";

    // std::array
    std::cout << "5. std::array - Type-safe stack array:\n";
    std::array<double, 10> samples{1.0, 2.0, 3.0, 4.0, 5.0};
    std::cout << "    Array size: " << samples.size() << "\n";
    std::cout << "      Stack-allocated, compile-time size, bounds checking\n\n";

    std::cout << "FEATURES TO AVOID IN HARD REAL-TIME:\n";
    std::cout << "    Exceptions (use -fno-exceptions)\n";
    std::cout << "    RTTI (use -fno-rtti)\n";
    std::cout << "    Virtual functions in time-critical paths\n";
    std::cout << "    std::shared_ptr (atomic ref counting overhead)\n";
    std::cout << "    Dynamic polymorphism (prefer compile-time polymorphism)\n";
}

} // namespace modern_cpp_features

// ==========================================================================
// SECTION 6: Real-Time System Checklist
// ==========================================================================

void print_realtime_checklist() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 6: Real-Time Programming Checklist ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "INITIALIZATION PHASE (Non-real-time):\n";
    std::cout << "    Pre-allocate all memory (reserve() for vectors)\n";
    std::cout << "    Create memory pools for frequent allocations\n";
    std::cout << "    Load and sort lookup tables\n";
    std::cout << "    Initialize all data structures to max size\n";
    std::cout << "    Warm up caches if applicable\n";
```

```
    std::cout << "    Set thread priorities and CPU affinity\n\n";

    std::cout << "REAL-TIME EXECUTION PHASE:\n";
    std::cout << "    NO dynamic memory allocation (no new/delete/malloc/free)\n";
    std::cout << "    NO operations with unbounded execution time\n";
    std::cout << "    NO file I/O or blocking system calls\n";
    std::cout << "    NO unbounded loops or recursion\n";
    std::cout << "    NO exceptions in critical paths\n";
    std::cout << "    Use O(1) or O(log n) operations only\n";
    std::cout << "    All containers have bounded maximum size\n";
    std::cout << "    Use std::bitset instead of std::vector<bool>\n";
    std::cout << "    Use std::array when size known at compile-time\n";
    std::cout << "    Use RAII for deterministic cleanup\n\n";

    std::cout << "CONTAINER SELECTION:\n";
    std::cout << "    std::array<T, N>           - Fixed size, stack-based\n";
    std::cout << "    std::vector<T> + reserve() - Dynamic, pre-allocated\n";
    std::cout << "    std::bitset<N>             - Flags and bit manipulation\n";
    std::cout << "    std::string_view           - Zero-copy string viewing\n";
    std::cout << "    std::deque<T>              - If size bounded\n";
    std::cout << "    std::list<T>               - Poor cache, heap per element\n";
    std::cout << "    std::vector<bool>          - Use std::bitset instead\n\n";

    std::cout << "ALGORITHM COMPLEXITY TARGETS:\n";
    std::cout << "    O(1)       - Ideal (array access, bitset operations)\n";
    std::cout << "    O(log n)   - Good (binary search, balanced trees)\n";
    std::cout << "    O(n)       - Acceptable if n is bounded and small\n";
    std::cout << "    O(n log n) - Avoid in critical paths (sort)\n";
    std::cout << "    O(n²)      - Never (nested loops, bubble sort)\n\n";

    std::cout << "COMPILER FLAGS FOR REAL-TIME:\n";
    std::cout << "  -fno-exceptions    Disable exception handling\n";
    std::cout << "  -fno-rtti          Disable run-time type information\n";
    std::cout << "  -O3                Maximum optimization\n";
    std::cout << "  -march=native      Target specific CPU architecture\n";
    std::cout << "  -flto              Link-time optimization\n\n";

    std::cout << "PROFILING AND VERIFICATION:\n";
    std::cout << "    Measure worst-case execution time (WCET)\n";
    std::cout << "    Use real-time profiling tools\n";
    std::cout << "    Test under worst-case conditions\n";
    std::cout << "    Verify no unexpected allocations\n";
    std::cout << "    Monitor stack usage\n";
    std::cout << "    Validate timing constraints are met\n";
}

// ==========================================================================
```

```
// Main Function
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "                  REAL-TIME PROGRAMMING IN MODERN C++                    \n";
    std::cout << "                                                                          \n";
    std::cout << "  Comprehensive guide to deterministic, high-performance C++ code     \n";
    std::cout << "  for embedded systems, robotics, and time-critical applications     \n";
    std::cout << "                                    \n";

    try {
        // Section 1: Big O Notation
        big_o_notation::demonstrate();

        // Section 1.5: Thread Architecture
        thread_architecture::demonstrate();

        // Section 2: std::bitset vs std::vector<bool>
        bitset_vs_vector_bool::demonstrate();

        // Section 3: STL Containers
        stl_containers_realtime::demonstrate();

        // Section 3.5: std::list::splice() - O(1) Element Movement
        list_splice_realtime::demonstrate();

        // Section 4: Memory Management
        memory_management::demonstrate();

        // Section 5: Modern C++ Features
        modern_cpp_features::demonstrate();

        // Section 6: Checklist
        print_realtime_checklist();

        std::cout << "\n" << std::string(70, '=') << "\n";
        std::cout << "=== END OF REAL-TIME PROGRAMMING GUIDE ===\n";
        std::cout << std::string(70, '=') << "\n\n";

        std::cout << "KEY TAKEAWAYS:\n";
        std::cout << "  1. Always pre-allocate memory during initialization\n";
        std::cout << "  2. Use std::bitset instead of std::vector<bool>\n";
        std::cout << "  3. Prefer O(1) operations, accept O(log n), bound O(n)\n";
```

```
        std::cout << "  4. Use reserve() for std::vector to prevent reallocations\n";
        std::cout << "  5. std::array is perfect for fixed-size data\n";
        std::cout << "  6. RAII provides deterministic resource management\n";
        std::cout << "  7. Modern C++ features like std::optional and std::variant\n";
        std::cout << "     are real-time friendly (no exceptions, no heap)\n";
        std::cout << "  8. Measure WCET, not average performance!\n\n";

        return 0;

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }
}
```

58 Source Code: ResourceLeaks.cpp

File: `src/ResourceLeaks.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// RESOURCE MANAGEMENT IN MODERN C++
// ==================================================================
// This example demonstrates how Modern C++ solves resource leak problems
// that plagued older C++ code using manual new/delete.
//
// TOPICS COVERED:
// 1. The Old Problem - Manual memory management with new/delete
// 2. Smart Pointers - Automatic memory management
// 3. RAII Pattern - Resource Acquisition Is Initialization
// 4. Other Resources - Files, mutexes, sockets, database connections
// 5. Custom Deleters - Managing non-memory resources
// 6. Exception Safety - Guarantees and resource cleanup
// 7. Modern Best Practices - When raw pointers are still OK
//
// KEY INSIGHT:
// "Modern C++ doesn't use new/delete directly anymore!"
// - Use smart pointers (unique_ptr, shared_ptr, weak_ptr)
// - Use containers (vector, string, map)
// - Use RAII wrappers for all resources
// - Raw pointers for non-owning references only
//
// EMBEDDED SYSTEMS NOTE:
// Even in resource-constrained embedded systems, smart pointers
// have ZERO runtime overhead compared to manual management!
// ==================================================================

#include <iostream>
#include <memory>
#include <vector>
#include <string>
#include <fstream>
#include <mutex>
#include <thread>
#include <chrono>
#include <exception>
#include <functional>
#include <map>

// ==================================================================
// 1. THE OLD PROBLEM - MANUAL MEMORY MANAGEMENT
// ==================================================================

namespace OldCpp {

// Example 1: Classic memory leak
void memory_leak_example() {
    std::cout << "OLD C++: Memory leak example\n";
    std::cout << "----------------------------\n";
```

```

    int* ptr = new int(42);
    std::cout << "Allocated int with value: " << *ptr << "\n";

    // Forgot to delete! Memory leak!
    // delete ptr;  // <-- Missing this line

    std::cout << "Function returns without delete - MEMORY LEAK!\n\n";
}

// Example 2: Exception causes leak
void exception_causes_leak() {
    std::cout << "OLD C++: Exception causes leak\n";
    std::cout << "-------------------------------\n";

    int* data = new int[1000];
    std::cout << "Allocated array of 1000 ints\n";

    try {
        // Some code that might throw
        if (data != nullptr) {
            throw std::runtime_error("Simulated error!");
        }

        delete[] data;  // <-- Never reached if exception thrown!
    } catch (const std::exception& e) {
        std::cout << "Caught exception: " << e.what() << "\n";
        std::cout << "Array was never deleted - MEMORY LEAK!\n\n";
        // Should have delete[] data here, but easy to forget
    }
}

// Example 3: Multiple exit paths
void multiple_exit_paths(bool condition1, bool condition2) {
    std::cout << "OLD C++: Multiple exit paths\n";
    std::cout << "----------------------------\n";

    int* buffer = new int[100];
    std::cout << "Allocated buffer\n";

    if (condition1) {
        std::cout << "Early return #1\n";
        // Forgot to delete!
        return;  // LEAK!
    }

    // ... do some work ...

    if (condition2) {
        std::cout << "Early return #2\n";
        // Forgot to delete!
        return;  // LEAK!
    }

```

```
    delete[] buffer;  // Only reached if both conditions false
    std::cout << "Normal cleanup\n\n";
}

// Example 4: Wrong delete operator
void wrong_delete_operator() {
    std::cout << "OLD C++: Wrong delete operator\n";
    std::cout << "------------------------------\n";

    int* single = new int(42);
    int* array = new int[10];

    // WRONG! Undefined behavior!
    delete[] single;  // Should be: delete single;
    delete array;      // Should be: delete[] array;

    std::cout << "Used wrong delete operator - UNDEFINED BEHAVIOR!\n\n";
}

// Example 5: Double delete
void double_delete_problem() {
    std::cout << "OLD C++: Double delete\n";
    std::cout << "----------------------\n";

    int* ptr = new int(42);

    delete ptr;
    std::cout << "Deleted once\n";

    // WRONG! Undefined behavior!
    delete ptr;  // Double delete - CRASH or corruption!

    std::cout << "Double delete - UNDEFINED BEHAVIOR!\n\n";
}

class ResourceIntensive {
    int* data;
    size_t size;

public:
    ResourceIntensive(size_t n) : size(n) {
        data = new int[n];
        std::cout << "  Allocated " << n << " ints\n";
    }

    // PROBLEM: No destructor! Memory leak!
    // ~ResourceIntensive() { delete[] data; }  // <-- Forgot this!
};

void demonstrate_old_problems() {
    std::cout << "\n";
    std::cout << "============================================================\n"
        ;
    std::cout << "1. THE OLD PROBLEM - MANUAL MEMORY MANAGEMENT\n";
```

```
    std::cout << "==========================================================\n\n";

    memory_leak_example();
    exception_causes_leak();
    multiple_exit_paths(true, false);

    // Note: Can't safely run wrong_delete_operator() and double_delete_problem()
    // as they cause undefined behavior

    std::cout << "Summary of Problems:\n";
    std::cout << "    Forgot to delete\n";
    std::cout << "    Exception before delete\n";
    std::cout << "    Multiple exit paths\n";
    std::cout << "    Wrong delete operator (delete vs delete[])\n";
    std::cout << "    Double delete\n";
    std::cout << "    Forgot destructor in class\n";
    std::cout << "\n";
}

} // namespace OldCpp

// ==================================================================
// 2. MODERN C++ SOLUTION - SMART POINTERS
// ==================================================================

namespace ModernCpp {

// Example 1: unique_ptr - Exclusive ownership
void unique_ptr_example() {
    std::cout << "MODERN C++: unique_ptr (exclusive ownership)\n";
    std::cout << "---------------------------------------------\n";

    // No manual new!
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    std::cout << "Created unique_ptr with value: " << *ptr << "\n";

    // Automatically deleted when scope ends!
    std::cout << "Leaving scope - automatic cleanup!\n\n";

    // NO delete needed!
}

// Example 2: Exception-safe with unique_ptr
void exception_safe_unique_ptr() {
    std::cout << "MODERN C++: Exception-safe with unique_ptr\n";
    std::cout << "------------------------------------------\n";

    auto data = std::make_unique<int[]>(1000);
    std::cout << "Allocated array of 1000 ints\n";

    try {
        throw std::runtime_error("Simulated error!");
```

```

        // Never reached, but doesn't matter!
    } catch (const std::exception& e) {
        std::cout << "Caught exception: " << e.what() << "\n";
        std::cout << "unique_ptr automatically cleaned up - NO LEAK!\n\n";
    }

    // Cleanup happens automatically in catch block
}

// Example 3: Multiple exit paths - no problem!
void multiple_exit_paths_safe(bool condition1, bool condition2) {
    std::cout << "MODERN C++: Multiple exit paths (safe)\n";
    std::cout << "--------------------------------------\n";

    auto buffer = std::make_unique<int[]>(100);
    std::cout << "Allocated buffer\n";

    if (condition1) {
        std::cout << "Early return #1 - buffer auto-deleted!\n\n";
        return;  // NO LEAK! unique_ptr cleans up
    }

    if (condition2) {
        std::cout << "Early return #2 - buffer auto-deleted!\n\n";
        return;  // NO LEAK! unique_ptr cleans up
    }

    std::cout << "Normal return - buffer auto-deleted!\n\n";
    // Automatic cleanup regardless of path!
}

// Example 4: shared_ptr - Shared ownership
void shared_ptr_example() {
    std::cout << "MODERN C++: shared_ptr (shared ownership)\n";
    std::cout << "-----------------------------------------\n";

    std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
    std::cout << "Created shared_ptr, ref count: " << ptr1.use_count() << "\n"
        ;

    {
        std::shared_ptr<int> ptr2 = ptr1;  // Share ownership
        std::cout << "Shared ownership, ref count: " << ptr1.use_count() << "\
            n";

        std::shared_ptr<int> ptr3 = ptr1;  // Another share
        std::cout << "More sharing, ref count: " << ptr1.use_count() << "\n";

    } // ptr2 and ptr3 destroyed

    std::cout << "After scope, ref count: " << ptr1.use_count() << "\n";
    std::cout << "Memory deleted when last shared_ptr destroyed!\n\n";
}
```

```

// Example 5: weak_ptr - Non-owning observer (doesn't keep object alive)
void weak_ptr_example() {
    std::cout << "MODERN C++: weak_ptr (non-owning observer)\n";
    std::cout << "-------------------------------------------\n";

    std::weak_ptr<int> weak;

    {
        std::shared_ptr<int> shared = std::make_shared<int>(42);
        weak = shared;  // weak_ptr observes, but doesn't own

        std::cout << "shared_ptr alive, ref count: " << shared.use_count() << "\n";
        std::cout << "weak_ptr expired? " << (weak.expired() ? "YES" : "NO") << "\n";

        // ALWAYS check if weak_ptr is still valid before using!
        if (auto locked = weak.lock()) {  // lock() returns shared_ptr
            std::cout << "Locked weak_ptr, value: " << *locked << "\n";
            std::cout << "Ref count during lock: " << locked.use_count() << "\n";
        } else {
            std::cout << "Object was deleted!\n";
        }

    } // shared destroyed here

    std::cout << "\nAfter shared_ptr destroyed:\n";
    std::cout << "weak_ptr expired? " << (weak.expired() ? "YES" : "NO") << "\n";

    if (auto locked = weak.lock()) {
        std::cout << "Locked successfully, value: " << *locked << "\n";
    } else {
        std::cout << "Cannot lock - object was deleted!\n";
    }
    std::cout << "\n";
}

// Example 6: weak_ptr - Breaking circular references
class Node {
public:
    std::string name;
    std::shared_ptr<Node> next;    // Strong reference (owns next)
    std::weak_ptr<Node> prev;      // Weak reference (doesn't own prev)

    Node(const std::string& n) : name(n) {
        std::cout << "  Node '" << name << "' created\n";
    }

    ~Node() {
        std::cout << "  Node '" << name << "' destroyed\n";
    }
```

```
};

void circular_reference_example() {
    std::cout << "MODERN C++: weak_ptr breaks circular references\n";
    std::cout << "-----------------------------------------------\n";

    auto node1 = std::make_shared<Node>("First");
    auto node2 = std::make_shared<Node>("Second");
    auto node3 = std::make_shared<Node>("Third");

    // Build doubly-linked list
    node1->next = node2;
    node2->prev = node1;  // weak_ptr - doesn't create cycle!

    node2->next = node3;
    node3->prev = node2;  // weak_ptr - doesn't create cycle!

    std::cout << "\nNavigate forward (using shared_ptr):\n";
    auto current = node1;
    while (current) {
        std::cout << "  At node: " << current->name << "\n";
        current = current->next;
    }

    std::cout << "\nNavigate backward (using weak_ptr):\n";
    current = node3;
    while (current) {
        std::cout << "  At node: " << current->name << "\n";

        // Must lock() weak_ptr to use it!
        if (auto prev = current->prev.lock()) {
            current = prev;
        } else {
            break;  // No previous node
        }
    }

    std::cout << "\nLeaving scope - all nodes will be destroyed:\n";
}

// Example 7: weak_ptr - Cache implementation
class ExpensiveResource {
    std::string data;
public:
    ExpensiveResource(const std::string& d) : data(d) {
        std::cout << "  Expensive resource created: " << data << "\n";
    }
    ~ExpensiveResource() {
        std::cout << "  Expensive resource destroyed: " << data << "\n";
    }
    const std::string& get_data() const { return data; }
};

class ResourceCache {
```

```
    std::map<std::string, std::weak_ptr<ExpensiveResource>> cache;

public:
    std::shared_ptr<ExpensiveResource> get_resource(const std::string& key) {
        // Try to get from cache
        auto it = cache.find(key);
        if (it != cache.end()) {
            // Check if cached resource is still alive
            if (auto resource = it->second.lock()) {
                std::cout << "  Cache HIT: " << key << "\n";
                return resource;
            } else {
                std::cout << "  Cache EXPIRED: " << key << "\n";
                cache.erase(it);  // Clean up expired entry
            }
        }

        // Create new resource
        std::cout << "  Cache MISS: Creating " << key << "\n";
        auto resource = std::make_shared<ExpensiveResource>(key);
        cache[key] = resource;  // Store weak_ptr in cache
        return resource;
    }
};

void cache_example() {
    std::cout << "MODERN C++: weak_ptr for caching\n";
    std::cout << "--------------------------------\n";

    ResourceCache cache;

    {
        auto res1 = cache.get_resource("data1");
        auto res2 = cache.get_resource("data1");  // Should be cache hit

        std::cout << "Both pointing to same resource: "
                  << (res1 == res2 ? "YES" : "NO") << "\n";
    }

    std::cout << "\nResources destroyed, trying to access again:\n";
    auto res3 = cache.get_resource("data1");  // Should be cache miss

    std::cout << "\n";
}

// Example 8: Modern class with RAII
class ResourceIntensiveModern {
    std::unique_ptr<int[]> data;
    size_t size;

public:
    ResourceIntensiveModern(size_t n) : data(std::make_unique<int[]>(n)), size
        (n) {
        std::cout << "  Allocated " << n << " ints (RAII)\n";
```

```
    }

    // No destructor needed! unique_ptr handles cleanup!
    // Compiler-generated destructor is perfect!

    // Safe to copy if we want (Rule of Zero)
    // Or delete copy if we want move-only:
    ResourceIntensiveModern(const ResourceIntensiveModern&) = delete;
    ResourceIntensiveModern& operator=(const ResourceIntensiveModern&) = delete;

    // Move operations provided automatically!
    ResourceIntensiveModern(ResourceIntensiveModern&&) = default;
    ResourceIntensiveModern& operator=(ResourceIntensiveModern&&) = default;

    int& operator[](size_t i) { return data[i]; }
};

void demonstrate_modern_solutions() {
    std::cout << "\n";
    std::cout << "============================================================\n";
    std::cout << "2. MODERN C++ SOLUTION - SMART POINTERS\n";
    std::cout << "============================================================\n\n";

    unique_ptr_example();
    exception_safe_unique_ptr();
    multiple_exit_paths_safe(true, false);
    shared_ptr_example();
    weak_ptr_example();
    circular_reference_example();
    cache_example();

    std::cout << "Modern class with RAII:\n";
    {
        ResourceIntensiveModern obj(100);
        // Automatic cleanup when obj goes out of scope!
    }
    std::cout << "  Automatically cleaned up!\n\n";

    std::cout << "Summary of Solutions:\n";
    std::cout << "    No manual new/delete\n";
    std::cout << "    Exception-safe automatically\n";
    std::cout << "    Multiple exit paths safe\n";
    std::cout << "    Can't use wrong delete operator\n";
    std::cout << "    Can't double-delete\n";
    std::cout << "    Compiler-generated destructor works\n";
    std::cout << "    weak_ptr breaks circular references\n";
    std::cout << "    weak_ptr for cache/observer patterns\n";
    std::cout << "\n";
}

} // namespace ModernCpp
```

```

// ==================================================================
// 3. RAII PATTERN - BEYOND MEMORY
// ==================================================================

namespace RAII {

// File handle RAII
class FileRAII {
    std::ofstream file;
    std::string filename;

public:
    FileRAII(const std::string& name) : filename(name) {
        file.open(filename);
        if (!file) {
            throw std::runtime_error("Failed to open file");
        }
        std::cout << "  File opened: " << filename << "\n";
    }

    ~FileRAII() {
        if (file.is_open()) {
            file.close();
            std::cout << "  File closed: " << filename << "\n";
        }
    }

    // Delete copy, allow move
    FileRAII(const FileRAII&) = delete;
    FileRAII& operator=(const FileRAII&) = delete;
    FileRAII(FileRAII&&) = default;
    FileRAII& operator=(FileRAII&&) = default;

    void write(const std::string& data) {
        file << data;
    }
};

// Mutex lock RAII (std::lock_guard does this!)
class MutexLockRAII {
    std::mutex& mtx;

public:
    explicit MutexLockRAII(std::mutex& m) : mtx(m) {
        mtx.lock();
        std::cout << "  Mutex locked\n";
    }

    ~MutexLockRAII() {
        mtx.unlock();
        std::cout << "  Mutex unlocked\n";
    }

```

```
    // Non-copyable, non-movable
    MutexLockRAII(const MutexLockRAII&) = delete;
    MutexLockRAII& operator=(const MutexLockRAII&) = delete;
};

// Timer RAII (measures scope duration)
class ScopeTimer {
    std::string name;
    std::chrono::high_resolution_clock::time_point start;

public:
    explicit ScopeTimer(const std::string& n)
        : name(n), start(std::chrono::high_resolution_clock::now()) {
        std::cout << "  [" << name << "] Started\n";
    }

    ~ScopeTimer() {
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
            end - start);
        std::cout << "  [" << name << "] Duration: " << duration.count() << "
            s\n";
    }
};

// Generic RAII wrapper
template<typename ResourceType, typename AcquireFunc, typename ReleaseFunc>
class GenericRAII {
    ResourceType resource;
    ReleaseFunc release;
    bool active;

public:
    GenericRAII(AcquireFunc acquire, ReleaseFunc rel)
        : resource(acquire()), release(rel), active(true) {}

    ~GenericRAII() {
        if (active) {
            release(resource);
        }
    }

    ResourceType& get() { return resource; }

    // Delete copy
    GenericRAII(const GenericRAII&) = delete;
    GenericRAII& operator=(const GenericRAII&) = delete;

    // Allow move
    GenericRAII(GenericRAII&& other) noexcept
        : resource(std::move(other.resource)),
          release(std::move(other.release)),
          active(other.active) {
        other.active = false;
```

```
    }
};

void demonstrate_raii() {
    std::cout << "\n";
    std::cout << "============================================================\n";
    std::cout << "3. RAII PATTERN - BEYOND MEMORY\n";
    std::cout << "============================================================\n\n";

    std::cout << "File RAII example:\n";
    {
        FileRAII file("test.txt");
        file.write("Hello, RAII!\n");
        // File automatically closed when scope ends
    }
    std::cout << "\n";

    std::cout << "Mutex RAII example:\n";
    std::mutex mtx;
    {
        MutexLockRAII lock(mtx);
        // Critical section
        std::cout << "  In critical section\n";
        // Mutex automatically unlocked when scope ends
    }
    std::cout << "\n";

    std::cout << "Scope timer example:\n";
    {
        ScopeTimer timer("MyOperation");
        // Simulate work
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        // Timer automatically reports duration
    }
    std::cout << "\n";

    std::cout << "RAII Resources:\n";
    std::cout << "    Memory (unique_ptr, shared_ptr)\n";
    std::cout << "    Files (fstream, custom wrappers)\n";
    std::cout << "    Locks (lock_guard, unique_lock)\n";
    std::cout << "    Timers (scope profiling)\n";
    std::cout << "    Network sockets\n";
    std::cout << "    Database connections\n";
    std::cout << "    OpenGL contexts\n";
    std::cout << "    Hardware registers (embedded)\n";
    std::cout << "\n";
}

} // namespace RAII

// ================================================================
// 4. CUSTOM DELETERS
```

```
// ==================================================================

namespace CustomDeleters {

// Example: C-style FILE*
void file_deleter(FILE* fp) {
    if (fp) {
        std::cout << "  Custom deleter: closing FILE*\n";
        fclose(fp);
    }
}

void custom_deleter_file_example() {
    std::cout << "Custom deleter with FILE*:\n";

    std::unique_ptr<FILE, decltype(&file_deleter)> file(
        fopen("test.txt", "w"),
        file_deleter
    );

    if (file) {
        fprintf(file.get(), "Hello from C-style FILE!\n");
    }

    // FILE* automatically closed by custom deleter!
}

// Example: Hardware register (embedded systems)
struct HardwareRegister {
    volatile uint32_t* address;
    uint32_t original_value;

    HardwareRegister(volatile uint32_t* addr)
        : address(addr), original_value(*addr) {
        std::cout << "  Saving register value: 0x" << std::hex <<
            original_value << std::dec << "\n";
    }

    ~HardwareRegister() {
        *address = original_value;
        std::cout << "  Restored register value: 0x" << std::hex <<
            original_value << std::dec << "\n";
    }
};

void embedded_register_example() {
    std::cout << "\nEmbedded hardware register RAII:\n";

    uint32_t simulated_register = 0x12345678;

    {
        HardwareRegister reg(&simulated_register);

        // Modify register
```

```
        simulated_register = 0xABCDEF00;
        std::cout << "  Modified register: 0x" << std::hex << simulated_register << std::dec << "\n";

        // Automatically restored on scope exit!
    }

    std::cout << "  Register after scope: 0x" << std::hex << simulated_register << std::dec << "\n";
}

void demonstrate_custom_deleters() {
    std::cout << "\n";
    std::cout << "============================================================\n";
    std::cout << "4. CUSTOM DELETERS\n";
    std::cout << "============================================================\n\n";

    custom_deleter_file_example();
    embedded_register_example();

    std::cout << "\nCustom deleters for:\n";
    std::cout << "    C API resources (FILE*, malloc, etc.)\n";
    std::cout << "    Operating system handles\n";
    std::cout << "    Hardware registers (embedded)\n";
    std::cout << "    Third-party library resources\n";
    std::cout << "\n";
}

} // namespace CustomDeleters

// ================================================================
// 5. WHEN RAW POINTERS ARE OK IN MODERN C++
// ================================================================

namespace RawPointersOK {

class Node {
public:
    int value;
    std::unique_ptr<Node> next;  // Owns the next node
    Node* prev;                   // Non-owning pointer to previous

    Node(int v) : value(v), prev(nullptr) {}
};

void raw_pointer_examples() {
    std::cout << "\n";
    std::cout << "============================================================\n";
    std::cout << "5. WHEN RAW POINTERS ARE OK IN MODERN C++\n";
    std::cout << "============================================================\n\n";
```

```

    std::cout << "Raw pointers are OK for:\n\n";

    std::cout << "1. NON-OWNING REFERENCES:\n";
    std::cout << "   auto owner = std::make_unique<int>(42);\n";
    std::cout << "   int* observer = owner.get();  // OK! Not responsible for deletion\n\n";

    std::cout << "2. FUNCTION PARAMETERS (non-owning):\n";
    std::cout << "   void process(const Widget* widget);  // OK! Doesn't own\n\n";

    std::cout << "3. OPTIONAL PARAMETERS:\n";
    std::cout << "   void render(Texture* texture = nullptr);  // OK! May be null\n\n";

    std::cout << "4. POINTERS TO STACK OBJECTS:\n";
    std::cout << "   int value = 42;\n";
    std::cout << "   int* ptr = &value;  // OK! No ownership\n\n";

    std::cout << "5. BACK POINTERS IN DATA STRUCTURES:\n";
    std::cout << "   struct Node {\n";
    std::cout << "       std::unique_ptr<Node> next;  // Owns next\n";
    std::cout << "       Node* prev;                  // Non-owning back pointer\n";
    std::cout << "   };\n\n";

    std::cout << "6. POINTERS TO GLOBAL/STATIC OBJECTS:\n";
    std::cout << "   static Logger logger;\n";
    std::cout << "   Logger* getLogger() { return &logger; }  // OK!\n\n";

    std::cout << "NEVER use raw pointers for:\n";
    std::cout << "    Ownership (use unique_ptr or shared_ptr)\n";
    std::cout << "    Arrays (use std::vector or std::array)\n";
    std::cout << "    Manual new/delete\n";
    std::cout << "\n";
}

} // namespace RawPointersOK

// ================================================================
// 6. CONTAINERS INSTEAD OF MANUAL ARRAYS
// ================================================================

namespace Containers {

void demonstrate_containers() {
    std::cout << "\n";
    std::cout << "============================================================\n";
    std::cout << "6. CONTAINERS INSTEAD OF MANUAL ARRAYS\n";
    std::cout << "============================================================\n\n";

```

```
    std::cout << "OLD C++: Manual array management\n";
    std::cout << "  int* arr = new int[100];  // Manual allocation\n";
    std::cout << "  // ... use array ...\n";
    std::cout << "  delete[] arr;  // Manual cleanup (easy to forget!)\n\n";

    std::cout << "MODERN C++: Use std::vector\n";
    std::vector<int> vec(100);
    std::cout << "  std::vector<int> vec(100);  // Automatic!\n";
    std::cout << "  // Automatic cleanup, resizing, bounds checking\n";
    std::cout << "  Size: " << vec.size() << " elements\n\n";

    std::cout << "Other containers (all automatic!):\n";
    std::cout << "    std::vector - Dynamic array\n";
    std::cout << "    std::array - Fixed-size array\n";
    std::cout << "    std::string - String (char array)\n";
    std::cout << "    std::map - Key-value pairs\n";
    std::cout << "    std::unordered_map - Hash table\n";
    std::cout << "    std::set - Unique elements\n";
    std::cout << "    std::list - Doubly-linked list\n";
    std::cout << "    std::deque - Double-ended queue\n";
    std::cout << "\n";
}

} // namespace Containers

// ================================================================
// 7. PERFORMANCE - ZERO OVERHEAD
// ================================================================

namespace Performance {

void demonstrate_zero_overhead() {
    std::cout << "\n";
    std::cout << "==========================================================\n";
    std::cout << "7. PERFORMANCE - ZERO OVERHEAD\n";
    std::cout << "==========================================================\n\n";

    std::cout << "Smart pointers have ZERO runtime overhead!\n\n";

    std::cout << "unique_ptr:\n";
    std::cout << " •  Same size as raw pointer\n";
    std::cout << " •  No runtime cost\n";
    std::cout << " •  Optimized to same assembly as manual delete\n";
    std::cout << " •  sizeof(unique_ptr<int>) = " << sizeof(std::unique_ptr<int>) << " bytes\n";
    std::cout << " •  sizeof(int*) = " << sizeof(int*) << " bytes\n\n";

    std::cout << "shared_ptr:\n";
    std::cout << " •  Small overhead for reference counting\n";
    std::cout << " •  sizeof(shared_ptr<int>) = " << sizeof(std::shared_ptr<int>) << " bytes\n";
    std::cout << " •  Only use when you need shared ownership\n\n";
```

```

    std::cout << "Embedded systems:\n";
    std::cout << "    unique_ptr: Perfect! Zero overhead\n";
    std::cout << "    std::vector: Excellent! Deterministic\n";
    std::cout << "    shared_ptr: Use sparingly (ref counting cost)\n";
    std::cout << "    RAII wrappers: Zero cost abstraction\n\n";

    std::cout << "Compiler optimizations:\n";
    std::cout << " •  unique_ptr: Completely optimized away\n";
    std::cout << " •  Move semantics: No copies, just pointer transfer\n";
    std::cout << " •  RVO/NRVO: Return value optimization\n";
    std::cout << " •  Inline expansion: Zero function call overhead\n";
    std::cout << "\n";
}

} // namespace Performance

// ==================================================================
// MAIN DEMONSTRATION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "
        ==================================================================\n"
        ;
    std::cout << "RESOURCE MANAGEMENT IN MODERN C++\n";
    std::cout << "
        ==================================================================\n"
        ;
    std::cout << "\"Modern C++ doesn't use new/delete directly anymore!\"\n";
    std::cout << "
        ==================================================================\n"
        ;

    OldCpp::demonstrate_old_problems();
    ModernCpp::demonstrate_modern_solutions();
    RAII::demonstrate_raii();
    CustomDeleters::demonstrate_custom_deleters();
    RawPointersOK::raw_pointer_examples();
    Containers::demonstrate_containers();
    Performance::demonstrate_zero_overhead();

    std::cout << "\n";
    std::cout << "
        ==================================================================\n"
        ;
    std::cout << "SUMMARY - MODERN C++ RESOURCE MANAGEMENT\n";
    std::cout << "
        ==================================================================\n"
        ;
    std::cout << "\n";
    std::cout << "THE GOLDEN RULES:\n";
    std::cout << "-----------------\n";
```

```
    std::cout << "1. NEVER use 'new' or 'delete' directly\n";
    std::cout << " →   Use std::make_unique<T>() instead\n";
    std::cout << "\n";
    std::cout << "2. NEVER use 'new[]' or 'delete[]'\n";
    std::cout << " →   Use std::vector<T> or std::array<T, N> instead\n";
    std::cout << "\n";
    std::cout << "3. Prefer unique_ptr over shared_ptr\n";
    std::cout << " →   Exclusive ownership is clearer and faster\n";
    std::cout << "\n";
    std::cout << "4. Use RAII for ALL resources\n";
    std::cout << " →   Memory, files, locks, sockets, everything!\n";
    std::cout << "\n";
    std::cout << "5. Raw pointers are OK for non-owning references\n";
    std::cout << " →   Never responsible for deletion\n";
    std::cout << "\n";
    std::cout << "6. Follow the Rule of Zero\n";
    std::cout << " →   Let compiler generate special members\n";
    std::cout << " →   Use standard library types as members\n";
    std::cout << "\n";
    std::cout << "BENEFITS:\n";
    std::cout << "---------\n";
    std::cout << "  No memory leaks (automatic cleanup)\n";
    std::cout << "  Exception-safe (automatic cleanup on unwind)\n";
    std::cout << "  No double-delete bugs\n";
    std::cout << "  No use-after-free bugs\n";
    std::cout << "  Clear ownership semantics\n";
    std::cout << "  Zero runtime overhead (unique_ptr)\n";
    std::cout << "  Easier to reason about code\n";
    std::cout << "  Suitable for embedded systems\n";
    std::cout << "\n";
    std::cout << "WHEN YOU STILL SEE new/delete:\n";
    std::cout << "------------------------------\n";
    std::cout << "• Legacy code (pre-C++11)\n";
    std::cout << "• Educational purposes (showing the old way)\n";
    std::cout << "• Custom memory allocators (advanced)\n";
    std::cout << "• Framework internals (Qt, COM, etc.)\n";
    std::cout << "\n";
    std::cout << "IN MODERN C++ (C++11 and later):\n";
    std::cout << "--------------------------------\n";
    std::cout << "new/delete are considered OBSOLETE for application code!\n";
    std::cout << "\n";
    std::cout << "MIGRATION PATH:\n";
    std::cout << "---------------\n";
    std::cout << "T* ptr = new T; →           auto ptr = std::make_unique<T>();\n";
    std::cout << "T* arr = new T[N]; →        std::vector<T> arr(N);\n";
    std::cout << "delete ptr; →               (automatic)\n";
    std::cout << "delete[] arr; →             (automatic)\n";
    std::cout << "T* shared = new T; →        auto shared = std::make_shared<T>();\n";
    std::cout << "\n";
    std::cout << "==================================================================\n";
```

```
    std::cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!\n";
    std::cout << "
        ==================================================================\n\
        n";

    return 0;
}

/*
EXPECTED OUTPUT HIGHLIGHTS:
===========================

=========================================================
1. THE OLD PROBLEM - MANUAL MEMORY MANAGEMENT
=========================================================

OLD C++: Memory leak example
----------------------------
Allocated int with value: 42
Function returns without delete - MEMORY LEAK!

OLD C++: Exception causes leak
------------------------------
Allocated array of 1000 ints
Caught exception: Simulated error!
Array was never deleted - MEMORY LEAK!

Summary of Problems:
   Forgot to delete
   Exception before delete
   Multiple exit paths
   Wrong delete operator (delete vs delete[])
   Double delete
   Forgot destructor in class

=========================================================
2. MODERN C++ SOLUTION - SMART POINTERS
=========================================================

MODERN C++: unique_ptr (exclusive ownership)
--------------------------------------------
Created unique_ptr with value: 42
Leaving scope - automatic cleanup!

MODERN C++: Exception-safe with unique_ptr
------------------------------------------
Allocated array of 1000 ints
Caught exception: Simulated error!
unique_ptr automatically cleaned up - NO LEAK!

Summary of Solutions:
   No manual new/delete
   Exception-safe automatically
   Multiple exit paths safe
```

```
   Can't use wrong delete operator
   Can't double-delete
   Compiler-generated destructor works

============================================================
3. RAII PATTERN - BEYOND MEMORY
============================================================

File RAII example:
  File opened: test.txt
  File closed: test.txt

Mutex RAII example:
  Mutex locked
  In critical section
  Mutex unlocked

RAII Resources:
   Memory (unique_ptr, shared_ptr)
   Files (fstream, custom wrappers)
   Locks (lock_guard, unique_lock)
   Timers (scope profiling)
   Network sockets
   Database connections
   Hardware registers (embedded)

============================================================
7. PERFORMANCE - ZERO OVERHEAD
============================================================

Smart pointers have ZERO runtime overhead!

unique_ptr:•
   Same size as raw pointer•
   No runtime cost•
   Optimized to same assembly as manual delete•
   sizeof(unique_ptr<int>) = 8 bytes•
   sizeof(int*) = 8 bytes

Embedded systems:
   unique_ptr: Perfect! Zero overhead
   std::vector: Excellent! Deterministic
   shared_ptr: Use sparingly (ref counting cost)
   RAII wrappers: Zero cost abstraction

====================================================================
SUMMARY - MODERN C++ RESOURCE MANAGEMENT
====================================================================

THE GOLDEN RULES:
-----------------
1. NEVER use 'new' or 'delete' directly→
    Use std::make_unique<T>() instead

```

```
2. NEVER use 'new[]' or 'delete[]'→
    Use std::vector<T> or std::array<T, N> instead

3. Prefer unique_ptr over shared_ptr→
    Exclusive ownership is clearer and faster

4. Use RAII for ALL resources→
    Memory, files, locks, sockets, everything!

5. Raw pointers are OK for non-owning references→
    Never responsible for deletion

BENEFITS:
---------
 No memory leaks (automatic cleanup)
 Exception-safe (automatic cleanup on unwind)
 No double-delete bugs
 No use-after-free bugs
 Clear ownership semantics
 Zero runtime overhead (unique_ptr)
 Suitable for embedded systems

IN MODERN C++ (C++11 and later):
--------------------------------
new/delete are considered OBSOLETE for application code!

*/
```

59 Source Code: RestApiExample.cpp

File: `src/RestApiExample.cpp`
Repository: View on GitHub

```
// ==================================================================
// REST API CALLS IN C++ EXAMPLE
// ==================================================================
// This example demonstrates making REST API calls in modern C++ using:
// - libcurl: For HTTP/HTTPS requests
// - nlohmann/json: For JSON parsing and serialization
//
// TOPICS COVERED:
// 1. GET requests with query parameters
// 2. POST requests with JSON payloads
// 3. PUT and DELETE requests
// 4. Custom headers and authentication
// 5. Error handling for network operations
// 6. RAII wrapper for curl resources
// 7. Response parsing with JSON
//
// WHAT IS libcurl?
// - A free and easy-to-use client-side URL transfer library
// - Supports HTTP, HTTPS, FTP, and many other protocols
// - Thread-safe when used correctly
// - Cross-platform (Windows, Linux, macOS)
//
// WHY USE libcurl?
//   Battle-tested: Used in billions of devices
//   Feature-rich: Supports all HTTP methods and features
//   Cross-platform: Works everywhere
//   Well-documented: Extensive documentation and examples
//   Active development: Regular updates and security fixes
//
// INSTALLATION:
// - Ubuntu: sudo apt-get install libcurl4-openssl-dev
// - Windows (vcpkg): vcpkg install curl
// - macOS: brew install curl
// - CMake: find_package(CURL REQUIRED)
//
// ALTERNATIVES TO CONSIDER:
// - cpp-httplib: Header-only, simpler but less features
// - Boost.Beast: Part of Boost, good for async operations
// - cpprestsdk: Microsoft's REST SDK, good for async
// - Qt Network: If already using Qt framework
//
// ==================================================================

#include <iostream>
#include <string>
#include <memory>
#include <stdexcept>
#include <vector>
#include <map>
```

```cpp
#include <curl/curl.h>
#include <nlohmann/json.hpp>

using json = nlohmann::json;

// ==================================================================
// SECTION 1: RAII Wrapper for CURL
// ==================================================================
// Modern C++ principle: Use RAII to manage resources automatically
// This ensures curl_easy_cleanup is always called, even on exceptions

class CurlHandle {
private:
    CURL* handle_;

public:
    CurlHandle() : handle_(curl_easy_init()) {
        if (!handle_) {
            throw std::runtime_error("Failed to initialize CURL");
        }
    }

    ~CurlHandle() {
        if (handle_) {
            curl_easy_cleanup(handle_);
        }
    }

    // Delete copy constructor and assignment (non-copyable)
    CurlHandle(const CurlHandle&) = delete;
    CurlHandle& operator=(const CurlHandle&) = delete;

    // Allow move semantics
    CurlHandle(CurlHandle&& other) noexcept : handle_(other.handle_) {
        other.handle_ = nullptr;
    }

    CurlHandle& operator=(CurlHandle&& other) noexcept {
        if (this != &other) {
            if (handle_) {
                curl_easy_cleanup(handle_);
            }
            handle_ = other.handle_;
            other.handle_ = nullptr;
        }
        return *this;
    }

    CURL* get() const { return handle_; }
    operator CURL*() const { return handle_; }
};

// ==================================================================
// SECTION 2: RAII Wrapper for curl_slist (headers)
```

```
// ==================================================================

class CurlHeaders {
private:
    struct curl_slist* headers_;

public:
    CurlHeaders() : headers_(nullptr) {}

    ~CurlHeaders() {
        if (headers_) {
            curl_slist_free_all(headers_);
        }
    }

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

    void append(const std::string& header) {
        headers_ = curl_slist_append(headers_, header.c_str());
    }

    struct curl_slist* get() const { return headers_; }
};

// ==================================================================
// SECTION 3: HTTP Response Structure
// ==================================================================

struct HttpResponse {
    long status_code;
    std::string body;
    std::map<std::string, std::string> headers;

    bool is_success() const {
        return status_code >= 200 && status_code < 300;
    }

    json to_json() const {
        return json::parse(body);
    }
};

// ==================================================================
// SECTION 4: Callback for Writing Response Data
// ==================================================================
// libcurl uses C-style callbacks. We use a static function that
// calls into our C++ code.

static size_t WriteCallback(void* contents, size_t size, size_t nmemb, void*
    userp) {
    size_t total_size = size * nmemb;
    auto* str = static_cast<std::string*>(userp);
    str->append(static_cast<char*>(contents), total_size);
```

```
    return total_size;
}

// ==================================================================
// SECTION 5: REST API Client Class
// ==================================================================

class RestClient {
private:
    std::string base_url_;
    std::map<std::string, std::string> default_headers_;

public:
    explicit RestClient(const std::string& base_url = "")
        : base_url_(base_url) {
        // Initialize libcurl globally (once per program)
        static bool initialized = false;
        if (!initialized) {
            curl_global_init(CURL_GLOBAL_DEFAULT);
            initialized = true;
        }
    }

    ~RestClient() {
        // Note: curl_global_cleanup() should be called at program exit
        // We don't call it here because multiple RestClient instances may exist
    }

    void set_default_header(const std::string& key, const std::string& value) {
        default_headers_[key] = value;
    }

    void set_bearer_token(const std::string& token) {
        default_headers_["Authorization"] = "Bearer " + token;
    }

    // GET request
    HttpResponse get(const std::string& endpoint,
                     const std::map<std::string, std::string>& params = {}) {
        std::string url = build_url(endpoint, params);
        return perform_request(url, "GET", "", {});
    }

    // POST request with JSON body
    HttpResponse post(const std::string& endpoint,
                      const json& body,
                      const std::map<std::string, std::string>& extra_headers = {}) {
        std::string url = build_url(endpoint);
        auto headers = extra_headers;
        headers["Content-Type"] = "application/json";
        return perform_request(url, "POST", body.dump(), headers);
```

```
    }

    // PUT request with JSON body
    HttpResponse put(const std::string& endpoint,
                     const json& body,
                     const std::map<std::string, std::string>& extra_headers = {}) {
        std::string url = build_url(endpoint);
        auto headers = extra_headers;
        headers["Content-Type"] = "application/json";
        return perform_request(url, "PUT", body.dump(), headers);
    }

    // DELETE request
    HttpResponse del(const std::string& endpoint) {
        std::string url = build_url(endpoint);
        return perform_request(url, "DELETE", "", {});
    }

private:
    std::string build_url(const std::string& endpoint,
                          const std::map<std::string, std::string>& params = {}) {
        std::string url = base_url_ + endpoint;

        if (!params.empty()) {
            url += "?";
            bool first = true;
            for (const auto& [key, value] : params) {
                if (!first) url += "&";
                url += key + "=" + curl_easy_escape_string(value);
                first = false;
            }
        }

        return url;
    }

    std::string curl_easy_escape_string(const std::string& str) {
        CurlHandle curl;
        char* escaped = curl_easy_escape(curl, str.c_str(), str.length());
        std::string result(escaped);
        curl_free(escaped);
        return result;
    }

    HttpResponse perform_request(const std::string& url,
                                 const std::string& method,
                                 const std::string& body,
                                 const std::map<std::string, std::string>& extra_headers) {
        CurlHandle curl;
        HttpResponse response;
        std::string response_body;
```

```

        // Set URL
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());

        // Set HTTP method
        if (method == "POST") {
            curl_easy_setopt(curl, CURLOPT_POST, 1L);
        } else if (method == "PUT") {
            curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "PUT");
        } else if (method == "DELETE") {
            curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "DELETE");
        } else if (method == "GET") {
            curl_easy_setopt(curl, CURLOPT_HTTPGET, 1L);
        }

        // Set request body if provided
        if (!body.empty()) {
            curl_easy_setopt(curl, CURLOPT_POSTFIELDS, body.c_str());
            curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, body.length());
        }

        // Set headers
        CurlHeaders headers;

        // Add default headers
        for (const auto& [key, value] : default_headers_) {
            headers.append(key + ": " + value);
        }

        // Add extra headers (can override defaults)
        for (const auto& [key, value] : extra_headers) {
            headers.append(key + ": " + value);
        }

        if (headers.get()) {
            curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers.get());
        }

        // Set write callback
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response_body);

        // Follow redirects
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);

        // Set timeout
        curl_easy_setopt(curl, CURLOPT_TIMEOUT, 30L);

        // Perform request
        CURLcode res = curl_easy_perform(curl);

        if (res != CURLE_OK) {
            throw std::runtime_error(std::string("CURL error: ") +
                                     curl_easy_strerror(res));
```

```
        }

        // Get response code
        curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response.status_code)
            ;
        response.body = response_body;

        return response;
    }
};

// ==================================================================
// SECTION 6: Example Usage with Public APIs
// ==================================================================

void example_json_placeholder_api() {
    std::cout << "\n=== Example 1: JSONPlaceholder API (GET) ===\n";

    try {
        RestClient client("https://jsonplaceholder.typicode.com");

        // GET request: Fetch a post
        auto response = client.get("/posts/1");

        std::cout << "Status Code: " << response.status_code << "\n";

        if (response.is_success()) {
            auto data = response.to_json();
            std::cout << "Post Title: " << data["title"] << "\n";
            std::cout << "Post Body: " << data["body"] << "\n";
            std::cout << "User ID: " << data["userId"] << "\n";
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}

void example_create_post() {
    std::cout << "\n=== Example 2: Create a Post (POST) ===\n";

    try {
        RestClient client("https://jsonplaceholder.typicode.com");

        // Create JSON payload
        json post_data = {
            {"title", "My C++ REST API Example"},
            {"body", "This post was created using C++ and libcurl!"},
            {"userId", 1}
        };

        // POST request
        auto response = client.post("/posts", post_data);

```

```
        std::cout << "Status Code: " << response.status_code << "\n";

        if (response.is_success()) {
            auto data = response.to_json();
            std::cout << "Created Post ID: " << data["id"] << "\n";
            std::cout << "Response:\n" << data.dump(2) << "\n";
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}

void example_update_post() {
    std::cout << "\n=== Example 3: Update a Post (PUT) ===\n";

    try {
        RestClient client("https://jsonplaceholder.typicode.com");

        // Updated data
        json updated_data = {
            {"id", 1},
            {"title", "Updated Title from C++"},
            {"body", "This post was updated using modern C++!"},
            {"userId", 1}
        };

        // PUT request
        auto response = client.put("/posts/1", updated_data);

        std::cout << "Status Code: " << response.status_code << "\n";

        if (response.is_success()) {
            auto data = response.to_json();
            std::cout << "Updated Post:\n" << data.dump(2) << "\n";
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}

void example_delete_post() {
    std::cout << "\n=== Example 4: Delete a Post (DELETE) ===\n";

    try {
        RestClient client("https://jsonplaceholder.typicode.com");

        // DELETE request
        auto response = client.del("/posts/1");

        std::cout << "Status Code: " << response.status_code << "\n";

        if (response.is_success()) {
```

```
            std::cout << "Post deleted successfully!\n";
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}

void example_with_query_parameters() {
    std::cout << "\n=== Example 5: GET with Query Parameters ===\n";

    try {
        RestClient client("https://jsonplaceholder.typicode.com");

        // GET with query parameters
        std::map<std::string, std::string> params = {
            {"userId", "1"}
        };

        auto response = client.get("/posts", params);

        std::cout << "Status Code: " << response.status_code << "\n";

        if (response.is_success()) {
            auto data = response.to_json();
            std::cout << "Found " << data.size() << " posts by user 1\n";

            // Display first 3 posts
            for (size_t i = 0; i < std::min(data.size(), size_t(3)); ++i) {
                std::cout << "\nPost " << (i + 1) << ":\n";
                std::cout << "  Title: " << data[i]["title"] << "\n";
            }
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}

void example_github_api() {
    std::cout << "\n=== Example 6: GitHub API (Public Data) ===\n";

    try {
        RestClient client("https://api.github.com");

        // GitHub requires User-Agent header
        client.set_default_header("User-Agent", "ModernCppExamples/1.0");

        // GET user information
        auto response = client.get("/users/torvalds");

        std::cout << "Status Code: " << response.status_code << "\n";

        if (response.is_success()) {
```

```
            auto data = response.to_json();
            std::cout << "GitHub User Info:\n";
            std::cout << "  Name: " << data.value("name", "N/A") << "\n";
            std::cout << "  Login: " << data["login"] << "\n";
            std::cout << "  Public Repos: " << data["public_repos"] << "\n";
            std::cout << "  Followers: " << data["followers"] << "\n";
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}

void example_error_handling() {
    std::cout << "\n=== Example 7: Error Handling ===\n";

    try {
        RestClient client("https://jsonplaceholder.typicode.com");

        // Try to access a non-existent resource (404)
        auto response = client.get("/posts/999999");

        std::cout << "Status Code: " << response.status_code << "\n";

        if (!response.is_success()) {
            std::cout << "Request failed!\n";
            std::cout << "Response body: " << response.body << "\n";
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}

// ==================================================================
// SECTION 7: Best Practices Summary
// ==================================================================

void print_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "REST API BEST PRACTICES IN C++:\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Resource Management (RAII):\n";
    std::cout << "    Use RAII wrappers for CURL handles\n";
    std::cout << "    Ensure cleanup happens automatically\n";
    std::cout << "    Exception-safe resource handling\n\n";

    std::cout << "2. Error Handling:\n";
    std::cout << "    Check CURLcode return values\n";
    std::cout << "    Check HTTP status codes\n";
    std::cout << "    Use exceptions for network errors\n";
    std::cout << "    Parse JSON safely with try-catch\n\n";

```

```
    std::cout << "3. Security:\n";
    std::cout << "     Always use HTTPS for sensitive data\n";
    std::cout << "     Validate SSL certificates (enabled by default)\n";
    std::cout << "     Never hardcode API keys (use environment variables)\n";
    std::cout << "     Use Bearer tokens for authentication\n\n";

    std::cout << "4. Performance:\n";
    std::cout << "     Reuse RestClient instances when possible\n";
    std::cout << "     Set appropriate timeouts\n";
    std::cout << "     Consider connection pooling for many requests\n";
    std::cout << "     Use HTTP/2 when supported\n\n";

    std::cout << "5. Modern C++ Features:\n";
    std::cout << "     Use smart pointers and RAII\n";
    std::cout << "     Leverage move semantics\n";
    std::cout << "     Use structured bindings for map iteration\n";
    std::cout << "     std::optional for nullable values\n\n";

    std::cout << "6. Testing:\n";
    std::cout << "     Mock HTTP responses for unit tests\n";
    std::cout << "     Test error conditions\n";
    std::cout << "     Use test APIs like JSONPlaceholder\n\n";
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "           REST API CALLS IN C++ - Complete Examples
                    \n";
    std::cout << "
                                                                    \n";
    std::cout << "   Demonstrates HTTP methods: GET, POST, PUT, DELETE
                     \n";
    std::cout << "   Using: libcurl + nlohmann::json
                                    \n";
    std::cout << "                                    \n";

    // Run all examples
    example_json_placeholder_api();
    example_create_post();
    example_update_post();
    example_delete_post();
    example_with_query_parameters();
    example_github_api();
    example_error_handling();

    // Print best practices
    print_best_practices();

    std::cout << "\ n All examples completed successfully!\n";
```

```
    std::cout << "\nNOTE: These examples use public test APIs:\n";
    std::cout << "  - JSONPlaceholder (https://jsonplaceholder.typicode.com)\n";
    std::cout << "  - GitHub API (https://api.github.com)\n";
    std::cout << "\nNo API key required. Responses are simulated/cached.\n\n";

    return 0;
}
```

60 Source Code: RuleOf3_5_0.cpp

File: `src/RuleOf3_5_0.cpp`
Repository: View on GitHub

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

// ==================================================================
// C++ SPECIAL MEMBER FUNCTIONS: RULE OF 3, 5, AND 0
// ==================================================================

// ==================================================================
// 1. RULE OF ZERO
// ==================================================================
// If you can avoid defining any special member functions (destructor,
// copy/move constructors, copy/move assignment), do so. Let the compiler
// generate them automatically. Use RAII wrappers like std::unique_ptr,
// std::shared_ptr, std::vector, std::string, etc.

class RuleOfZeroExample {
private:
    std::string name;
    std::vector<int> data;
    std::unique_ptr<int> ptr;

public:
    RuleOfZeroExample(const std::string& n, int value)
        : name(n), data{1, 2, 3, 4, 5}, ptr(std::make_unique<int>(value)) {
        std::cout << "RuleOfZero: Constructor for " << name << std::endl;
    }

    // No destructor needed - std::string, std::vector, std::unique_ptr
    // handle cleanup automatically

    // No copy constructor needed - compiler generates correct one
    // (note: std::unique_ptr makes this non-copyable by default)

    // No copy assignment needed

    // No move constructor needed - compiler generates efficient one

    // No move assignment needed

    void display() const {
        std::cout << "  Name: " << name << ", Value: " << *ptr
                  << ", Data size: " << data.size() << std::endl;
    }
};

void example_rule_of_zero() {
```

```
    std::cout << "\n=== RULE OF ZERO ===" << std::endl;
    std::cout << "Use RAII wrappers. No special member functions needed!\n" <<
        std::endl;

    RuleOfZeroExample obj1("Object1", 42);
    obj1.display();

    // Move semantics work automatically
    RuleOfZeroExample obj2 = std::move(obj1);
    obj2.display();

    std::cout << "\n Compiler-generated special members handle everything
        correctly" << std::endl;
}

// ==================================================================
// 2. RULE OF THREE (Pre-C++11)
// ==================================================================
// If you define any of the following, you should define all three:
// 1. Destructor
// 2. Copy constructor
// 3. Copy assignment operator

class RuleOfThreeExample {
private:
    int* data;
    size_t size;
    std::string name;

public:
    // Constructor
    RuleOfThreeExample(const std::string& n, size_t s)
        : data(new int[s]), size(s), name(n) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = static_cast<int>(i);
        }
        std::cout << "RuleOfThree: Constructor for " << name
                  << " (size=" << size << ")" << std::endl;
    }

    // 1. Destructor
    ~RuleOfThreeExample() {
        std::cout << "RuleOfThree: Destructor for " << name << std::endl;
        delete[] data;
    }

    // 2. Copy constructor
    RuleOfThreeExample(const RuleOfThreeExample& other)
        : data(new int[other.size]), size(other.size), name(other.name + "
            _copy") {
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
        std::cout << "RuleOfThree: Copy constructor for " << name << std::endl
```

```
            ;
    }

    // 3. Copy assignment operator
    RuleOfThreeExample& operator=(const RuleOfThreeExample& other) {
        std::cout << "RuleOfThree: Copy assignment for " << name << std::endl;

        if (this != &other) {
            // Free existing resource
            delete[] data;

            // Allocate new resource and copy
            size = other.size;
            data = new int[size];
            for (size_t i = 0; i < size; ++i) {
                data[i] = other.data[i];
            }
            name = other.name + "_assigned";
        }
        return *this;
    }

    void display() const {
        std::cout << "  " << name << " data: [";
        for (size_t i = 0; i < std::min(size, size_t(5)); ++i) {
            std::cout << data[i] << (i < std::min(size, size_t(5)) - 1 ? ", "
                : "");
        }
        std::cout << (size > 5 ? "..." : "") << "]" << std::endl;
    }
};

void example_rule_of_three() {
    std::cout << "\n=== RULE OF THREE ===" << std::endl;
    std::cout << "Define: Destructor, Copy Constructor, Copy Assignment\n" <<
        std::endl;

    RuleOfThreeExample obj1("Original", 10);
    obj1.display();

    // Copy constructor
    RuleOfThreeExample obj2 = obj1;
    obj2.display();

    // Copy assignment
    RuleOfThreeExample obj3("Another", 5);
    obj3 = obj1;
    obj3.display();

    std::cout << "\n All three special members ensure proper resource
        management" << std::endl;
}

// ==================================================================
```

```cpp
// 3. RULE OF FIVE (C++11 and later)
// ================================================================
// If you define any of the following, you should define all five:
// 1. Destructor
// 2. Copy constructor
// 3. Copy assignment operator
// 4. Move constructor
// 5. Move assignment operator

class RuleOfFiveExample {
private:
    int* data;
    size_t size;
    std::string name;

public:
    // Constructor
    RuleOfFiveExample(const std::string& n, size_t s)
        : data(new int[s]), size(s), name(n) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = static_cast<int>(i * 10);
        }
        std::cout << "RuleOfFive: Constructor for " << name
                  << " (size=" << size << ")" << std::endl;
    }

    // 1. Destructor
    ~RuleOfFiveExample() {
        std::cout << "RuleOfFive: Destructor for " << name << std::endl;
        delete[] data;
    }

    // 2. Copy constructor
    RuleOfFiveExample(const RuleOfFiveExample& other)
        : data(new int[other.size]), size(other.size), name(other.name + "
            _copy") {
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
        std::cout << "RuleOfFive: Copy constructor for " << name << std::endl;
    }

    // 3. Copy assignment operator
    RuleOfFiveExample& operator=(const RuleOfFiveExample& other) {
        std::cout << "RuleOfFive: Copy assignment for " << name << std::endl;

        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            for (size_t i = 0; i < size; ++i) {
                data[i] = other.data[i];
            }
            name = other.name + "_copy_assigned";
```

```
        }
        return *this;
    }

    // 4. Move constructor
    RuleOfFiveExample(RuleOfFiveExample&& other) noexcept
        : data(other.data), size(other.size), name(std::move(other.name) + "
            _moved") {
        other.data = nullptr;
        other.size = 0;
        std::cout << "RuleOfFive: Move constructor for " << name << std::endl;
    }

    // 5. Move assignment operator
    RuleOfFiveExample& operator=(RuleOfFiveExample&& other) noexcept {
        std::cout << "RuleOfFive: Move assignment for " << name << std::endl;

        if (this != &other) {
            delete[] data;

            data = other.data;
            size = other.size;
            name = std::move(other.name) + "_move_assigned";

            other.data = nullptr;
            other.size = 0;
        }
        return *this;
    }

    void display() const {
        if (data && size > 0) {
            std::cout << "  " << name << " data: [";
            for (size_t i = 0; i < std::min(size, size_t(5)); ++i) {
                std::cout << data[i] << (i < std::min(size, size_t(5)) - 1 ? "
                    , " : "");
            }
            std::cout << (size > 5 ? "..." : "") << "]" << std::endl;
        } else {
            std::cout << "  " << name << " (moved-from, empty)" << std::endl;
        }
    }
};

void example_rule_of_five() {
    std::cout << "\n=== RULE OF FIVE ===" << std::endl;
    std::cout << "Define: Destructor, Copy Constructor, Copy Assignment, "
              << "Move Constructor, Move Assignment\n" << std::endl;

    RuleOfFiveExample obj1("Original", 8);
    obj1.display();

    // Copy constructor
    RuleOfFiveExample obj2 = obj1;
```

```
    obj2.display();

    // Move constructor
    RuleOfFiveExample obj3 = std::move(obj1);
    obj3.display();
    std::cout << "After move:" << std::endl;
    obj1.display();  // obj1 is in valid but moved-from state

    // Copy assignment
    RuleOfFiveExample obj4("Target1", 3);
    obj4 = obj2;
    obj4.display();

    // Move assignment
    RuleOfFiveExample obj5("Target2", 3);
    obj5 = std::move(obj2);
    obj5.display();
    std::cout << "After move:" << std::endl;
    obj2.display();  // obj2 is in valid but moved-from state

    std::cout << "\n All five special members handle both copy and move
        semantics" << std::endl;
}

// ==================================================================
// 4. MODERN C++17/20 RELEVANCE
// ==================================================================

#include <optional>
#include <variant>

class ModernCpp17Example {
private:
    std::string name;
    std::vector<int> data;
    std::optional<std::string> description;  // C++17
    std::variant<int, double, std::string> value;  // C++17

public:
    ModernCpp17Example(const std::string& n)
        : name(n), data{1, 2, 3}, description("A modern class"), value(42) {
        std::cout << "Modern C++17 class: " << name << std::endl;
    }

    // NO special member functions needed!
    // Compiler generates everything correctly, including:
    // - Destructor (cleans up all RAII members)
    // - Copy constructor (deep copies std::string, std::vector, std::optional
        , std::variant)
    // - Copy assignment (properly assigns all members)
    // - Move constructor (efficient moves for all members)
    // - Move assignment (efficient move assigns)

    void display() const {
```

```
        std::cout << "  Name: " << name;
        if (description) {
            std::cout << ", Desc: " << *description;
        }
        std::cout << ", Variant holds: ";
        std::visit([](auto&& arg) { std::cout << arg; }, value);
        std::cout << std::endl;
    }
};

void example_modern_cpp_relevance() {
    std::cout << "\n=== MODERN C++17/20 RELEVANCE ===" << std::endl;

    std::cout << "\n RULE OF ZERO IS MORE RELEVANT THAN EVER!" << std::endl;
    std::cout << "\nC++17/20 additions that strengthen Rule of Zero:" << std::endl;
    std::cout << " •  std::optional<T> - Optional values (no more raw pointers!)" << std::endl;
    std::cout << " •  std::variant<Ts...> - Type-safe unions" << std::endl;
    std::cout << " •  std::any - Type-safe void*" << std::endl;
    std::cout << " •  std::string_view - Non-owning string references" << std::endl;
    std::cout << " •  std::span<T> (C++20) - Non-owning array views" << std::endl;
    std::cout << " •  std::make_shared array support (C++20)" << std::endl;

    ModernCpp17Example obj1("Original");
    obj1.display();

    std::cout << "\nCopying (compiler-generated):" << std::endl;
    ModernCpp17Example obj2 = obj1;
    obj2.display();

    std::cout << "\nMoving (compiler-generated):" << std::endl;
    ModernCpp17Example obj3 = std::move(obj1);
    obj3.display();

    std::cout << "\n Zero manual resource management, all type-safe!" << std::endl;
}

// ==================================================================
// 5. COMPARISON: WHEN TO USE WHICH RULE
// ==================================================================

void example_comparison() {
    std::cout << "\n=== WHEN TO USE WHICH RULE (2026 PERSPECTIVE) ===" << std::endl;

    std::cout << "\n RULE OF ZERO (STRONGLY PREFERRED - 95%+ of cases):" << std::endl;
    std::cout << "    Default choice for ALL new C++17/20 code" << std::endl;
    std::cout << "    Use std::unique_ptr, std::shared_ptr, std::vector, std::string" << std::endl;
```

```
    std::cout << "    Use std::optional, std::variant, std::any (C++17)" << std::endl;
    std::cout << "    Use std::span for views (C++20)" << std::endl;
    std::cout << "    Let compiler generate ALL special members" << std::endl;
    std::cout << "    Zero bugs from manual memory management" << std::endl;
    std::cout << "    MOST RELEVANT IN MODERN C++!" << std::endl;

    std::cout << "\n RULE OF THREE (LARGELY OBSOLETE):" << std::endl;
    std::cout << "    Pre-C++11 guideline - avoid in new code" << std::endl;
    std::cout << "    Maintain in legacy codebases only" << std::endl;
    std::cout << "    Refactor to Rule of Zero when possible" << std::endl;
    std::cout << "    Don't use for new C++17/20 code" << std::endl;

    std::cout << "\n RULE OF FIVE (RARELY NEEDED - <5% of cases):" << std::endl;
    std::cout << "    Only when Rule of Zero is impossible" << std::endl;
    std::cout << "    Custom allocators or exotic resources" << std::endl;
    std::cout << "    Performance-critical kernel/driver code" << std::endl;
    std::cout << "    Interop with C libraries or hardware" << std::endl;
    std::cout << "    Consider if you really need it first" << std::endl;

    std::cout << "\n MODERN C++17/20 BEST PRACTICES:" << std::endl;
    std::cout << "  1. ALWAYS start with Rule of Zero" << std::endl;
    std::cout << "  2. Use std::optional/variant instead of raw pointers" << std::endl;
    std::cout << "  3. Use std::unique_ptr for ownership" << std::endl;
    std::cout << "  4. Use std::span for non-owning views (C++20)" << std::endl;
    std::cout << "  5. Only implement special members if profiling proves necessary" << std::endl;
    std::cout << "  6. Use = default and = delete for explicit intent" << std::endl;
    std::cout << "  7. Mark move operations noexcept (enables optimizations)" << std::endl;

    std::cout << "\n REALITY CHECK (2026):" << std::endl;
    std::cout << " •  95%+ of classes should use Rule of Zero" << std::endl;
    std::cout << " •  <5% need Rule of Five (custom resource managers)" << std::endl;
    std::cout << " •  Rule of Three is legacy knowledge only" << std::endl;
    std::cout << " •  Modern C++ makes manual memory management unnecessary" << std::endl;
}

// ==================================================================
// 5. EXPLICIT CONTROL: DEFAULT AND DELETE
// ==================================================================

class DefaultAndDeleteExample {
private:
    std::unique_ptr<int> ptr;

public:
    DefaultAndDeleteExample() : ptr(std::make_unique<int>(42)) {}
```

```

    // Explicitly defaulted destructor
    ~DefaultAndDeleteExample() = default;

    // Deleted copy operations (non-copyable)
    DefaultAndDeleteExample(const DefaultAndDeleteExample&) = delete;
    DefaultAndDeleteExample& operator=(const DefaultAndDeleteExample&) = delete;

    // Explicitly defaulted move operations
    DefaultAndDeleteExample(DefaultAndDeleteExample&&) = default;
    DefaultAndDeleteExample& operator=(DefaultAndDeleteExample&&) = default;

    int getValue() const { return *ptr; }
};

void example_default_and_delete() {
    std::cout << "\n=== DEFAULT AND DELETE ===" << std::endl;
    std::cout << "Explicitly control which special members are available\n" << std::endl;

    DefaultAndDeleteExample obj1;
    std::cout << "Created obj1, value: " << obj1.getValue() << std::endl;

    // DefaultAndDeleteExample obj2 = obj1;  // ERROR: Copy deleted

    DefaultAndDeleteExample obj3 = std::move(obj1);  // OK: Move defaulted
    std::cout << "Moved to obj3, value: " << obj3.getValue() << std::endl;

    std::cout << "\n = delete prevents copying, = default enables moving" << std::endl;
}

// ==================================================================
// 6. EXPLICIT KEYWORD - PREVENTING IMPLICIT CONVERSIONS
// ==================================================================
// The explicit keyword prevents implicit type conversions that could
// lead to unexpected behavior or resource management issues

// BAD: Without explicit - allows implicit conversions
class ImplicitBuffer {
private:
    std::unique_ptr<int[]> data;
    size_t size;

public:
    // Constructor without explicit - DANGEROUS!
    ImplicitBuffer(size_t s) : data(std::make_unique<int[]>(s)), size(s) {
        std::cout << "ImplicitBuffer: Allocated " << size << " ints" << std::endl;
    }

    size_t getSize() const { return size; }
};
```

```

// GOOD: With explicit - prevents implicit conversions
class ExplicitBuffer {
private:
    std::unique_ptr<int[]> data;
    size_t size;

public:
    // Constructor with explicit - SAFE!
    explicit ExplicitBuffer(size_t s) : data(std::make_unique<int[]>(s)), size(s) {
        std::cout << "ExplicitBuffer: Allocated " << size << " ints" << std::endl;
    }

    size_t getSize() const { return size; }
};

// Example with conversion operators
class SmartInt {
private:
    int value;

public:
    explicit SmartInt(int v) : value(v) {}

    // C++11: explicit conversion operator
    explicit operator int() const { return value; }

    // Implicit conversion would be dangerous for bool
    explicit operator bool() const { return value != 0; }
};

void processBuffer(const ExplicitBuffer& buf) {
    std::cout << "  Processing buffer of size: " << buf.getSize() << std::endl;
}

void processImplicitBuffer(const ImplicitBuffer& buf) {
    std::cout << "  Processing implicit buffer of size: " << buf.getSize() << std::endl;
}

void example_explicit_keyword() {
    std::cout << "\n=== EXPLICIT KEYWORD ===" << std::endl;
    std::cout << "Prevents dangerous implicit conversions\n" << std::endl;

    // Without explicit - COMPILES BUT DANGEROUS!
    std::cout << "Without explicit keyword:" << std::endl;
    ImplicitBuffer buf1(10);  // OK: Direct initialization

    // DANGEROUS: Implicit conversion from int to ImplicitBuffer!
    processImplicitBuffer(100);  // Creates temporary ImplicitBuffer(100) - wasteful!
```

```

    // Can even do this nonsense:
    ImplicitBuffer buf2 = 50;  // Implicit conversion - looks like int assignment!
    std::cout << "  Created buffer with = 50 (confusing!)" << std::endl;

    std::cout << "\nWith explicit keyword:" << std::endl;
    ExplicitBuffer buf3(10);  // OK: Direct initialization

    // processBuffer(100);  // ERROR: Cannot implicitly convert int to ExplicitBuffer
    // ExplicitBuffer buf4 = 50;  // ERROR: Cannot use copy initialization

    std::cout << "    Must use direct initialization: ExplicitBuffer(100)" << std::endl;
    processBuffer(ExplicitBuffer(100));  // OK: Explicit construction

    // Explicit conversion operators (C++11)
    std::cout << "\nExplicit conversion operators:" << std::endl;
    SmartInt smart(42);

    // int x = smart;  // ERROR: Cannot implicitly convert to int
    int x = static_cast<int>(smart);  // OK: Explicit cast required
    std::cout << "  Explicit cast to int: " << x << std::endl;

    // if (smart) { }  // OK: Contextual conversion to bool allowed
    std::cout << "    Contextual bool conversion works in if/while statements" << std::endl;

    std::cout << "\n MODERN C++ BEST PRACTICES:" << std::endl;
    std::cout << "    ALWAYS use explicit for single-parameter constructors" << std::endl;
    std::cout << "    Use explicit for conversion operators (except bool)" << std::endl;
    std::cout << "    Prevents accidental temporary object creation" << std::endl;
    std::cout << "    Prevents resource allocation surprises" << std::endl;
    std::cout << "    Makes code intent crystal clear" << std::endl;

    std::cout << "\n  WHEN IMPLICIT IS OK:" << std::endl;
    std::cout << " •  String literals to std::string: std::string s = \"hello\";" << std::endl;
    std::cout << " •  Initializer lists: std::vector<int> v = {1, 2, 3};" << std::endl;
    std::cout << " •  Copy/move constructors (never explicit)" << std::endl;
}

// ================================================================
// 7. POLICY-BASED DESIGN - PARAMETERIZED RESOURCE MANAGEMENT
// ================================================================
// Modern C++ technique: Use template parameters to customize behavior
// without runtime overhead or virtual functions

// Policy 1: Deletion Strategies
```

```
struct ArrayDelete {
    template<typename T>
    void operator()(T* ptr) const {
        delete[] ptr;
        std::cout << "    ArrayDelete: delete[] called" << std::endl;
    }
};

struct SingleDelete {
    template<typename T>
    void operator()(T* ptr) const {
        delete ptr;
        std::cout << "    SingleDelete: delete called" << std::endl;
    }
};

struct NoOpDelete {
    template<typename T>
    void operator()(T*) const {
        std::cout << "    NoOpDelete: no deletion (externally managed)" << std
            ::endl;
    }
};

// Policy 2: Copy Strategies
struct DeepCopy {
    template<typename T>
    static T* copy(const T* src, size_t size) {
        std::cout << "    DeepCopy: allocating and copying " << size << "
            elements" << std::endl;
        T* dest = new T[size];
        std::copy(src, src + size, dest);
        return dest;
    }
};

struct ShallowCopy {
    template<typename T>
    static T* copy(const T* src, size_t) {
        std::cout << "    ShallowCopy: returning same pointer (reference
            counting)" << std::endl;
        return const_cast<T*>(src);  // Warning: For demo only!
    }
};

// Policy 3: Thread Safety
struct NoThreadSafety {
    void lock() const { /* no-op */ }
    void unlock() const { /* no-op */ }
};

struct BasicThreadSafety {
    void lock() const {
        std::cout << "    BasicThreadSafety: acquiring lock" << std::endl;
```

```
    }
    void unlock() const {
        std::cout << "    BasicThreadSafety: releasing lock" << std::endl;
    }
};

// Policy-based Resource Manager
template<
    typename T,
    typename DeletePolicy = ArrayDelete,
    typename CopyPolicy = DeepCopy,
    typename ThreadPolicy = NoThreadSafety
>
class PolicyBasedBuffer {
private:
    T* data_;
    size_t size_;
    DeletePolicy deleter_;
    ThreadPolicy thread_policy_;

public:
    // Constructor
    PolicyBasedBuffer(size_t s) : data_(new T[s]), size_(s) {
        std::cout << "  PolicyBasedBuffer: Constructor (size=" << size_ << ")"
            << std::endl;
    }

    // Destructor
    ~PolicyBasedBuffer() {
        std::cout << "  PolicyBasedBuffer: Destructor" << std::endl;
        thread_policy_.lock();
        deleter_(data_);
        thread_policy_.unlock();
    }

    // Copy constructor (uses CopyPolicy)
    PolicyBasedBuffer(const PolicyBasedBuffer& other)
        : data_(CopyPolicy::copy(other.data_, other.size_)),
          size_(other.size_) {
        std::cout << "  PolicyBasedBuffer: Copy constructor" << std::endl;
    }

    // Copy assignment
    PolicyBasedBuffer& operator=(const PolicyBasedBuffer& other) {
        if (this != &other) {
            std::cout << "  PolicyBasedBuffer: Copy assignment" << std::endl;
            thread_policy_.lock();
            deleter_(data_);
            data_ = CopyPolicy::copy(other.data_, other.size_);
            size_ = other.size_;
            thread_policy_.unlock();
        }
        return *this;
    }
```

```

    // Move constructor
    PolicyBasedBuffer(PolicyBasedBuffer&& other) noexcept
        : data_(other.data_), size_(other.size_) {
        other.data_ = nullptr;
        other.size_ = 0;
        std::cout << "  PolicyBasedBuffer: Move constructor" << std::endl;
    }

    // Move assignment
    PolicyBasedBuffer& operator=(PolicyBasedBuffer&& other) noexcept {
        if (this != &other) {
            std::cout << "  PolicyBasedBuffer: Move assignment" << std::endl;
            thread_policy_.lock();
            deleter_(data_);
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
            thread_policy_.unlock();
        }
        return *this;
    }

    size_t size() const { return size_; }
    T* data() { return data_; }
};

// Type aliases for common configurations
template<typename T>
using ArrayBuffer = PolicyBasedBuffer<T, ArrayDelete, DeepCopy, NoThreadSafety
    >;

template<typename T>
using ThreadSafeBuffer = PolicyBasedBuffer<T, ArrayDelete, DeepCopy,
    BasicThreadSafety>;

template<typename T>
using SingleObjectBuffer = PolicyBasedBuffer<T, SingleDelete, DeepCopy,
    NoThreadSafety>;

void example_policy_based_design() {
    std::cout << "\n=== POLICY-BASED DESIGN ===" << std::endl;
    std::cout << "Parameterize behavior with template policies\n" << std::endl
        ;

    std::cout << "1. Standard array buffer (ArrayDelete + DeepCopy):" << std::
        endl;
    {
        ArrayBuffer<int> buf1(10);
        std::cout << "   Copying..." << std::endl;
        ArrayBuffer<int> buf2 = buf1;
        std::cout << "   Moving..." << std::endl;
        ArrayBuffer<int> buf3 = std::move(buf1);
```

```
        std::cout << "   Cleanup:" << std::endl;
    }

    std::cout << "\n2. Thread-safe buffer (with BasicThreadSafety):" << std::endl;
    {
        ThreadSafeBuffer<int> buf(5);
        std::cout << "   Cleanup with thread safety:" << std::endl;
    }

    std::cout << "\n3. Custom policy combination (NoOpDelete for externally managed):" << std::endl;
    {
        PolicyBasedBuffer<int, NoOpDelete, DeepCopy, NoThreadSafety> buf(3);
        std::cout << "   Cleanup (no actual deletion):" << std::endl;
    }

    std::cout << "\n ADVANTAGES OF POLICY-BASED DESIGN:" << std::endl;
    std::cout << "   Zero runtime overhead (compile-time selection)" << std::endl;
    std::cout << "   No virtual functions or vtables needed" << std::endl;
    std::cout << "   Highly composable and reusable" << std::endl;
    std::cout << "   Type-safe customization" << std::endl;
    std::cout << "   Optimized for each configuration" << std::endl;

    std::cout << "\n REAL-WORLD USES:" << std::endl;
    std::cout << " • std::unique_ptr<T, Deleter> - Custom deleters" << std::endl;
    std::cout << " • STL containers with Allocator - Custom allocators" << std::endl;
    std::cout << " • Thread-safety policies in concurrent code" << std::endl;
    std::cout << " • Logging/tracing policies" << std::endl;
    std::cout << " • Different storage strategies (stack vs heap)" << std::endl;

    std::cout << "\n MODERN C++ ALTERNATIVE:" << std::endl;
    std::cout << "  Use std::unique_ptr<T, Deleter> for most cases:" << std::endl;
    std::cout << "    auto ptr = std::unique_ptr<int[], ArrayDelete>(new int[10]);" << std::endl;
}

// ==================================================================
// 8. COPY-AND-SWAP IDIOM
// ==================================================================
// An elegant way to implement assignment operators

class CopyAndSwapExample {
private:
    int* data;
    size_t size;
    std::string name;

public:
```

```
    CopyAndSwapExample(const std::string& n, size_t s)
        : data(new int[s]), size(s), name(n) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = static_cast<int>(i);
        }
    }

    ~CopyAndSwapExample() {
        delete[] data;
    }

    // Copy constructor
    CopyAndSwapExample(const CopyAndSwapExample& other)
        : data(new int[other.size]), size(other.size), name(other.name) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
    }

    // Move constructor
    CopyAndSwapExample(CopyAndSwapExample&& other) noexcept
        : data(other.data), size(other.size), name(std::move(other.name)) {
        other.data = nullptr;
        other.size = 0;
    }

    // Swap function
    friend void swap(CopyAndSwapExample& first, CopyAndSwapExample& second)
        noexcept {
        using std::swap;
        swap(first.data, second.data);
        swap(first.size, second.size);
        swap(first.name, second.name);
    }

    // Unified assignment operator (handles both copy and move)
    CopyAndSwapExample& operator=(CopyAndSwapExample other) {
        swap(*this, other);
        return *this;
    }

    void display() const {
        if (data) {
            std::cout << "  " << name << " [size=" << size << "]" << std::endl
                ;
        } else {
            std::cout << "  " << name << " (empty)" << std::endl;
        }
    }
};

void example_copy_and_swap() {
    std::cout << "\n=== COPY-AND-SWAP IDIOM ===" << std::endl;
    std::cout << "Elegant and exception-safe assignment operator\n" << std::
```

```
        endl;

    CopyAndSwapExample obj1("Object1", 10);
    CopyAndSwapExample obj2("Object2", 5);

    obj1.display();
    obj2.display();

    std::cout << "\nAssigning obj1 = obj2 (copy):" << std::endl;
    obj1 = obj2;
    obj1.display();

    std::cout << "\nAssigning obj1 = std::move(obj2) (move):" << std::endl;
    obj1 = std::move(obj2);
    obj1.display();
    obj2.display();

    std::cout << "\ n Single assignment operator handles both copy and move!"
        << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n
        ==================================================================" <<
        std::endl;
    std::cout << "  C++ SPECIAL MEMBER FUNCTIONS: RULE OF 3, 5, AND 0" << std
        ::endl;
    std::cout << "
        ==================================================================" <<
        std::endl;

    example_rule_of_zero();
    example_rule_of_three();
    example_rule_of_five();
    example_modern_cpp_relevance();
    example_comparison();
    example_default_and_delete();
    example_explicit_keyword();
    example_policy_based_design();
    example_copy_and_swap();

    std::cout << "\n
        ==================================================================" <<
        std::endl;
    std::cout << "  SUMMARY" << std::endl;
    std::cout << " == RULE OF ZERO: Use in 95%+ of modern C++17/20 code!" <<
        std::endl;
    std::cout << "   - std::unique_ptr, std::optional, std::variant, std::
        vector, etc." << std::endl;
    std::cout << "   - Zero manual memory management = zero bugs" << std::endl
```

```
        ;

    std::cout << "\n RULE OF THREE: Legacy guideline (pre-C++11)" << std::endl;
    std::cout << "   - Maintain in old codebases only" << std::endl;
    std::cout << "   - Refactor to Rule of Zero when possible" << std::endl;

    std::cout << "\n  RULE OF FIVE: Rarely needed (<5% of cases)" << std::endl;
    std::cout << "   - Only for custom resource managers" << std::endl;
    std::cout << "   - Performance-critical scenarios" << std::endl;

    std::cout << "\n USE = DEFAULT AND = DELETE FOR EXPLICIT INTENT" << std::endl;
    std::cout << " Copy-and-swap idiom when you need Rule of Five" << std::endl;
    std::cout << "\n ==============================================================\n" << std::endl;

    return 0;
}
```

61 Source Code: RuntimePolymorphism.cpp

File: `src/RuntimePolymorphism.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <chrono>

// ====================================================================
// VIRTUAL FUNCTIONS AND PURE VIRTUAL FUNCTIONS IN MODERN C++
// ====================================================================
// Demonstrates:
// 1. Virtual functions vs pure virtual functions
// 2. Abstract classes and interfaces
// 3. Runtime polymorphism
// 4. Virtual destructors (critical!)
// 5. override and final keywords (C++11)
// 6. When to use vs alternatives (CRTP, concepts)
// 7. Performance considerations
// 8. Modern C++ best practices
// ====================================================================

// ====================================================================
// 1. PURE VIRTUAL FUNCTIONS - ABSTRACT INTERFACE
// ====================================================================
// Pure virtual function: = 0
// Cannot instantiate class with pure virtual functions
// Derived classes MUST implement all pure virtual functions

class IShape {
public:
    // Pure virtual function - no implementation
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual void draw() const = 0;
    virtual std::string name() const = 0;

    // Virtual destructor is CRITICAL for polymorphic classes
    virtual ~IShape() = default;

    // Can have non-virtual functions too
    void info() const {
        std::cout << name() << ": area=" << area()
                  << ", perimeter=" << perimeter() << std::endl;
    }
};

class Circle : public IShape {
private:
    double radius;

```

```
public:
    explicit Circle(double r) : radius(r) {}

    // Must implement ALL pure virtual functions
    double area() const override {
        return 3.14159 * radius * radius;
    }

    double perimeter() const override {
        return 2 * 3.14159 * radius;
    }

    void draw() const override {
        std::cout << "Drawing Circle (radius=" << radius << ")" << std::endl;
    }

    std::string name() const override {
        return "Circle";
    }
};

class Rectangle : public IShape {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const override {
        return width * height;
    }

    double perimeter() const override {
        return 2 * (width + height);
    }

    void draw() const override {
        std::cout << "Drawing Rectangle (" << width << "x" << height << ")" <<
            std::endl;
    }

    std::string name() const override {
        return "Rectangle";
    }
};

void example_pure_virtual() {
    std::cout << "\n=== 1. PURE VIRTUAL FUNCTIONS (ABSTRACT INTERFACE) ===" <<
        std::endl;

    // IShape shape;  //  ERROR: Cannot instantiate abstract class

    std::vector<std::unique_ptr<IShape>> shapes;
    shapes.push_back(std::make_unique<Circle>(5.0));
```

```
    shapes.push_back(std::make_unique<Rectangle>(4.0, 6.0));
    shapes.push_back(std::make_unique<Circle>(3.0));

    std::cout << "\nProcessing shapes polymorphically:" << std::endl;
    for (const auto& shape : shapes) {
        shape->draw();
        shape->info();
        std::cout << std::endl;
    }

    std::cout << "  PURE VIRTUAL (= 0):" << std::endl;
    std::cout << " •   Forces derived classes to implement" << std::endl;
    std::cout << " •   Creates abstract class (cannot instantiate)" << std::
        endl;
    std::cout << " •   Defines interface contract" << std::endl;
    std::cout << " •   Use for: Interfaces, abstract base classes" << std::
        endl;
}

// ==================================================================
// 2. VIRTUAL FUNCTIONS WITH DEFAULT IMPLEMENTATION
// ==================================================================
// Virtual function with implementation - can be overridden but not required

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

    // Virtual with default implementation
    virtual void makeSound() const {
        std::cout << "Some generic animal sound" << std::endl;
    }

    // Virtual with default implementation
    virtual void eat() const {
        std::cout << "Animal is eating" << std::endl;
    }

    // Non-virtual - cannot override
    void breathe() const {
        std::cout << "Animal is breathing" << std::endl;
    }
};

class Dog : public Animal {
public:
    // Override with custom implementation
    void makeSound() const override {
        std::cout << "Woof! Woof!" << std::endl;
    }

    // Override eat
    void eat() const override {
        std::cout << "Dog is eating dog food" << std::endl;
```

```
    }

    // breathe() cannot be overridden (not virtual)
};

class Cat : public Animal {
public:
    // Only override makeSound
    void makeSound() const override {
        std::cout << "Meow!" << std::endl;
    }

    // eat() uses default implementation from Animal
};

void example_virtual_with_default() {
    std::cout << "\n=== 2. VIRTUAL FUNCTIONS WITH DEFAULT IMPLEMENTATION ==="
        << std::endl;

    std::vector<std::unique_ptr<Animal>> animals;
    animals.push_back(std::make_unique<Dog>());
    animals.push_back(std::make_unique<Cat>());

    std::cout << "\nDog:" << std::endl;
    animals[0]->makeSound();  // Custom implementation
    animals[0]->eat();        // Custom implementation
    animals[0]->breathe();    // Non-virtual (same for all)

    std::cout << "\nCat:" << std::endl;
    animals[1]->makeSound();  // Custom implementation
    animals[1]->eat();        // Uses default from Animal
    animals[1]->breathe();    // Non-virtual (same for all)

    std::cout << "\n VIRTUAL WITH DEFAULT:" << std::endl;
    std::cout << " •   Provides base implementation" << std::endl;
    std::cout << " •   Derived classes can override if needed" << std::endl;
    std::cout << " •   Optional customization" << std::endl;
    std::cout << " •   Use for: Template Method pattern" << std::endl;
}

// ================================================================
// 3. VIRTUAL DESTRUCTOR - CRITICAL FOR POLYMORPHISM!
// ================================================================

class Base {
public:
    Base() {
        std::cout << "Base constructor" << std::endl;
    }

    //  BAD: Non-virtual destructor in polymorphic class!
    ~Base() {
        std::cout << "Base destructor" << std::endl;
    }
```

```
};

class DerivedBad : public Base {
private:
    int* data;

public:
    DerivedBad() {
        data = new int[100];
        std::cout << "DerivedBad constructor (allocated 100 ints)" << std::
            endl;
    }

    ~DerivedBad() {
        delete[] data;
        std::cout << "DerivedBad destructor (freed memory)" << std::endl;
    }
};

class BaseGood {
public:
    BaseGood() {
        std::cout << "BaseGood constructor" << std::endl;
    }

    //  GOOD: Virtual destructor!
    virtual ~BaseGood() {
        std::cout << "BaseGood destructor" << std::endl;
    }
};

class DerivedGood : public BaseGood {
private:
    int* data;

public:
    DerivedGood() {
        data = new int[100];
        std::cout << "DerivedGood constructor (allocated 100 ints)" << std::
            endl;
    }

    ~DerivedGood() override {
        delete[] data;
        std::cout << "DerivedGood destructor (freed memory)" << std::endl;
    }
};

void example_virtual_destructor() {
    std::cout << "\n=== 3. VIRTUAL DESTRUCTOR (CRITICAL!) ===" << std::endl;

    std::cout << "\n BAD: Non-virtual destructor:" << std::endl;
    {
        Base* ptr = new DerivedBad();
```

```
        delete ptr;  //  MEMORY LEAK! Only Base destructor called
    }

    std::cout << "\n GOOD: Virtual destructor:" << std::endl;
    {
        BaseGood* ptr = new DerivedGood();
        delete ptr;  //  Both destructors called correctly
    }

    std::cout << "\n RULE: Always make destructor virtual if class has virtual functions!" << std::endl;
    std::cout << " •  Otherwise: Memory leaks, undefined behavior" << std::endl;
    std::cout << " •  Use: virtual ~ClassName() = default;" << std::endl;
}

// ==================================================================
// 4. OVERRIDE AND FINAL KEYWORDS (C++11)
// ==================================================================

class BaseKeywords {
public:
    virtual void foo() const {
        std::cout << "BaseKeywords::foo()" << std::endl;
    }

    virtual void bar() const {
        std::cout << "BaseKeywords::bar()" << std::endl;
    }

    virtual ~BaseKeywords() = default;
};

class DerivedKeywords : public BaseKeywords {
public:
    // override: Ensures we're actually overriding a base function
    void foo() const override {
        std::cout << "DerivedKeywords::foo()" << std::endl;
    }

    //  This would cause compile error with override:
    // void fooo() const override { }  // Typo! No such function in base

    // final: Cannot be overridden further
    void bar() const final {
        std::cout << "DerivedKeywords::bar() - cannot override further" << std::endl;
    }
};

// final: Cannot be inherited from
class FinalClass final {
public:
    void method() {
```

```
        std::cout << "FinalClass::method()" << std::endl;
    }
};

//  This would cause compile error:
// class CannotDeriveFromFinal : public FinalClass { };

void example_override_final() {
    std::cout << "\n=== 4. OVERRIDE AND FINAL KEYWORDS (C++11) ===" << std::endl;

    std::unique_ptr<BaseKeywords> ptr = std::make_unique<DerivedKeywords>();
    ptr->foo();
    ptr->bar();

    std::cout << "\n OVERRIDE KEYWORD:" << std::endl;
    std::cout << " •   Catches typos at compile time" << std::endl;
    std::cout << " •   Documents intent clearly" << std::endl;
    std::cout << " •   Prevents accidental shadowing" << std::endl;
    std::cout << " •   ALWAYS use it when overriding!" << std::endl;

    std::cout << "\n FINAL KEYWORD:" << std::endl;
    std::cout << " •   Prevents further overriding (function)" << std::endl;
    std::cout << " •   Prevents inheritance (class)" << std::endl;
    std::cout << " •   Enables compiler optimizations" << std::endl;
    std::cout << " •   Use for: Performance, design intent" << std::endl;
}

// ==================================================================
// 5. WHEN TO USE VIRTUAL FUNCTIONS IN MODERN C++
// ==================================================================

void example_when_to_use() {
    std::cout << "\n=== 5. WHEN TO USE VIRTUAL FUNCTIONS ===" << std::endl;

    std::cout << "\n USE VIRTUAL FUNCTIONS WHEN:" << std::endl;
    std::cout << "   1. Runtime polymorphism needed" << std::endl;
    std::cout << "      - Heterogeneous containers (vector<Base*>)" << std::endl;
    std::cout << "      - Plugin architectures" << std::endl;
    std::cout << "      - Dynamic type selection" << std::endl;

    std::cout << "\n   2. Interface definition" << std::endl;
    std::cout << "      - Abstract base classes" << std::endl;
    std::cout << "      - Pure virtual functions (= 0)" << std::endl;
    std::cout << "      - Contract for derived classes" << std::endl;

    std::cout << "\n   3. Framework design" << std::endl;
    std::cout << "      - Template Method pattern" << std::endl;
    std::cout << "      - Strategy pattern" << std::endl;
    std::cout << "      - Factory pattern" << std::endl;

    std::cout << "\n   4. Open/Closed principle" << std::endl;
    std::cout << "      - Extend behavior without modifying base" << std::endl
```

```
        ;
    std::cout << "      - Add new types without changing existing code" << std
        ::endl;
}

// ==================================================================
// 6. ALTERNATIVES TO VIRTUAL FUNCTIONS (MODERN C++)
// ==================================================================

// CRTP (Curiously Recurring Template Pattern) - Static Polymorphism
template<typename Derived>
class SensorBase {
public:
    float read() {
        // Static cast to derived - no runtime overhead!
        return static_cast<Derived*>(this)->read_impl();
    }
};

class TemperatureSensor : public SensorBase<TemperatureSensor> {
public:
    float read_impl() {
        return 25.5f;  // Temperature reading
    }
};

class PressureSensor : public SensorBase<PressureSensor> {
public:
    float read_impl() {
        return 1013.25f;  // Pressure reading
    }
};

template<typename Sensor>
void process_sensor(Sensor& sensor) {
    // Compile-time polymorphism - fully inlineable!
    std::cout << "Sensor reading: " << sensor.read() << std::endl;
}

void example_alternatives() {
    std::cout << "\n=== 6. ALTERNATIVES TO VIRTUAL FUNCTIONS ===" << std::endl
        ;

    std::cout << "\n1. CRTP (STATIC POLYMORPHISM):" << std::endl;
    TemperatureSensor temp;
    PressureSensor pressure;

    process_sensor(temp);
    process_sensor(pressure);

    std::cout << "\n     Advantages:" << std::endl;
    std::cout << " •      Zero runtime overhead (no vtable)" << std::endl;
    std::cout << " •      Fully inlineable" << std::endl;
    std::cout << " •      Faster than virtual functions" << std::endl;
```

```
    std::cout << " •      Good for embedded systems" << std::endl;

    std::cout << "\n     Disadvantages:" << std::endl;
    std::cout << " •      No heterogeneous containers" << std::endl;
    std::cout << " •      Compile-time binding only" << std::endl;
    std::cout << " •      More complex syntax" << std::endl;

    std::cout << "\n2. CONCEPTS (C++20):" << std::endl;
    std::cout << " •   Compile-time constraints" << std::endl;
    std::cout << " •   Duck typing with type safety" << std::endl;
    std::cout << " •   No inheritance needed" << std::endl;
    std::cout << " •   Example: template<Drawable T> void draw(T& obj)" << std
        ::endl;

    std::cout << "\n3. std::variant (C++17):" << std::endl;
    std::cout << " •   Type-safe union" << std::endl;
    std::cout << " •   std::visit for polymorphic behavior" << std::endl;
    std::cout << " •   Value semantics (no pointers)" << std::endl;
    std::cout << " •   Good for small, closed set of types" << std::endl;

    std::cout << "\n4. std::function (C++11):" << std::endl;
    std::cout << " •   Type-erased callable" << std::endl;
    std::cout << " •   Works with lambdas, functors" << std::endl;
    std::cout << " •   Runtime overhead (like virtual)" << std::endl;
    std::cout << " •   Good for callbacks, strategies" << std::endl;
}

// ==================================================================
// 7. PERFORMANCE COMPARISON
// ==================================================================

class VirtualBase {
public:
    virtual int compute(int x) { return x * 2; }
    virtual ~VirtualBase() = default;
};

class VirtualDerived : public VirtualBase {
public:
    int compute(int x) override { return x * 3; }
};

void example_performance() {
    std::cout << "\n=== 7. PERFORMANCE CONSIDERATIONS ===" << std::endl;

    constexpr int iterations = 10'000'000;

    // Virtual function call
    VirtualBase* vptr = new VirtualDerived();
    auto start_virtual = std::chrono::high_resolution_clock::now();
    volatile int result_virtual = 0;
    for (int i = 0; i < iterations; ++i) {
        result_virtual += vptr->compute(i);
    }
```

```
    auto end_virtual = std::chrono::high_resolution_clock::now();
    auto duration_virtual = std::chrono::duration_cast<std::chrono::milliseconds>(
        end_virtual - start_virtual).count();

    // Direct function call (non-virtual)
    VirtualDerived direct;
    auto start_direct = std::chrono::high_resolution_clock::now();
    volatile int result_direct = 0;
    for (int i = 0; i < iterations; ++i) {
        result_direct += direct.compute(i);
    }
    auto end_direct = std::chrono::high_resolution_clock::now();
    auto duration_direct = std::chrono::duration_cast<std::chrono::milliseconds>(
        end_direct - start_direct).count();

    delete vptr;

    std::cout << "\nPerformance (10 million calls):" << std::endl;
    std::cout << "   Virtual function:  " << duration_virtual << " ms" << std::endl;
    std::cout << "   Direct call:       " << duration_direct << " ms" << std::endl;
    std::cout << "   Overhead:          ~"
              << (duration_virtual - duration_direct) << " ms" << std::endl;

    std::cout << "\n OVERHEAD SOURCES:" << std::endl;
    std::cout << "   1. Vtable lookup (8 bytes per object)" << std::endl;
    std::cout << "   2. Indirect function call (cache miss)" << std::endl;
    std::cout << "   3. Cannot be inlined by compiler" << std::endl;
    std::cout << "   4. Branch prediction harder" << std::endl;

    std::cout << "\n TRADE-OFF:" << std::endl;
    std::cout << " •   Virtual functions: Small overhead, big flexibility" << std::endl;
    std::cout << " •   Usually worth it for abstraction benefits" << std::endl;
    std::cout << " •   Only optimize if profiling shows bottleneck" << std::endl;
}

// ==================================================================
// 8. MODERN C++ BEST PRACTICES
// ==================================================================

void example_best_practices() {
    std::cout << "\n=== 8. MODERN C++ BEST PRACTICES ===" << std::endl;

    std::cout << "\n DO:" << std::endl;
    std::cout << "   1. ALWAYS use 'override' keyword" << std::endl;
    std::cout << "        void foo() override { }" << std::endl;
    std::cout << "        void foo() { }  // Easy to make mistakes" << std::endl;
```

```

    std::cout << "\n   2. ALWAYS make destructor virtual in polymorphic classes" << std::endl;
    std::cout << "        virtual ~Base() = default;" << std::endl;
    std::cout << "        ~Base() { }  // Memory leaks!" << std::endl;

    std::cout << "\n   3. Use 'final' to prevent further overriding" << std::endl;
    std::cout << "        void foo() final { }  // Design intent + optimization" << std::endl;

    std::cout << "\n   4. Use pure virtual (= 0) for interfaces" << std::endl;
    std::cout << "        virtual void draw() const = 0;  // Must implement" << std::endl;

    std::cout << "\n   5. Prefer smart pointers for polymorphic objects" << std::endl;
    std::cout << "        std::unique_ptr<Base> ptr = std::make_unique<Derived>();" << std::endl;
    std::cout << "        Base* ptr = new Derived();  // Manual delete needed" << std::endl;

    std::cout << "\n DON'T:" << std::endl;
    std::cout << "   1. Don't use virtual for non-polymorphic classes" << std::endl;
    std::cout << "      - Unnecessary overhead" << std::endl;

    std::cout << "\n   2. Don't forget virtual destructor" << std::endl;
    std::cout << "      - Causes memory leaks and undefined behavior" << std::endl;

    std::cout << "\n   3. Don't call virtual functions in constructor/destructor" << std::endl;
    std::cout << "      - Derived class not fully constructed yet" << std::endl;
    std::cout << "      - Will call base version, not derived!" << std::endl;

    std::cout << "\n   4. Don't use virtual if CRTP/concepts work" << std::endl;
    std::cout << "      - Static polymorphism is faster" << std::endl;
    std::cout << "      - Good for performance-critical code" << std::endl;
}

// ==================================================================
// 9. ARE VIRTUAL FUNCTIONS STILL RELEVANT?
// ==================================================================

void example_still_relevant() {
    std::cout << "\n=== 9. ARE VIRTUAL FUNCTIONS STILL RELEVANT? ===" << std::endl;

    std::cout << "\n YES! Virtual functions are STILL HIGHLY RELEVANT:" << std::endl;

```

```
    std::cout << "\n1. RUNTIME POLYMORPHISM:" << std::endl;
    std::cout << " •   Containers of heterogeneous objects" << std::endl;
    std::cout << " •   Plugin systems and dynamic loading" << std::endl;
    std::cout << " •   GUI frameworks (Qt, wxWidgets)" << std::endl;
    std::cout << " •   Game engines (entity systems)" << std::endl;

    std::cout << "\n2. DESIGN PATTERNS:" << std::endl;
    std::cout << " •   Strategy, Observer, Command patterns" << std::endl;
    std::cout << " •   Factory method, Template method" << std::endl;
    std::cout << " •   Visitor pattern (double dispatch)" << std::endl;

    std::cout << "\n3. API DESIGN:" << std::endl;
    std::cout << " •   Stable binary interfaces (ABIs)" << std::endl;
    std::cout << " •   Dynamic libraries (.dll, .so)" << std::endl;
    std::cout << " •   Cross-module boundaries" << std::endl;

    std::cout << "\n4. SIMPLICITY:" << std::endl;
    std::cout << " •   Easier to understand than CRTP" << std::endl;
    std::cout << " •   More maintainable for most teams" << std::endl;
    std::cout << " •   Better error messages" << std::endl;

    std::cout << "\ n MODERN ALTERNATIVES (C++11-20):" << std::endl;
    std::cout << " •   CRTP - Compile-time polymorphism" << std::endl;
    std::cout << " •   Concepts (C++20) - Duck typing with constraints" << std
        ::endl;
    std::cout << " •   std::variant + std::visit (C++17)" << std::endl;
    std::cout << " •   std::function - Type-erased callables" << std::endl;

    std::cout << "\ n WHEN TO CHOOSE:" << std::endl;
    std::cout << "   Virtual Functions:" << std::endl;
    std::cout << "       Need runtime polymorphism" << std::endl;
    std::cout << "       Heterogeneous containers" << std::endl;
    std::cout << "       Plugin architectures" << std::endl;
    std::cout << "       Simplicity and maintainability" << std::endl;

    std::cout << "\n   CRTP/Concepts:" << std::endl;
    std::cout << "       Performance critical code" << std::endl;
    std::cout << "       Compile-time polymorphism sufficient" << std::endl;
    std::cout << "       Embedded systems (no vtable)" << std::endl;
    std::cout << "       Header-only libraries" << std::endl;

    std::cout << "\ n BOTTOM LINE:" << std::endl;
    std::cout << " •   Virtual functions: Core C++ feature, still essential"
        << std::endl;
    std::cout << " •   Modern C++ adds alternatives, not replacements" << std
        ::endl;
    std::cout << " •   Choose based on requirements, not trends" << std::endl;
    std::cout << " •   Most codebases use BOTH virtual and static polymorphism
        " << std::endl;
}

// ==================================================================
// MAIN
// ==================================================================
```

```

int main() {
    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  VIRTUAL FUNCTIONS AND PURE VIRTUAL FUNCTIONS IN MODERN C++" << std::endl;
    std::cout << " ================================================================" << std::endl;

    example_pure_virtual();
    example_virtual_with_default();
    example_virtual_destructor();
    example_override_final();
    example_when_to_use();
    example_alternatives();
    example_performance();
    example_best_practices();
    example_still_relevant();

    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  SUMMARY: VIRTUAL FUNCTIONS IN MODERN C++" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\n KEY CONCEPTS:" << std::endl;
    std::cout << "   1. Pure virtual (= 0) - Abstract interface, must implement" << std::endl;
    std::cout << "   2. Virtual - Can override, optional" << std::endl;
    std::cout << "   3. Virtual destructor - ALWAYS for polymorphic classes" << std::endl;
    std::cout << "   4. override keyword - ALWAYS use when overriding" << std::endl;
    std::cout << "   5. final keyword - Prevent further overriding" << std::endl;

    std::cout << "\n BEST PRACTICES:" << std::endl;
    std::cout << " •   Use 'override' on all virtual function overrides" << std::endl;
    std::cout << " •   Always virtual destructor for polymorphic classes" << std::endl;
    std::cout << " •   Prefer std::unique_ptr/shared_ptr over raw pointers" << std::endl;
    std::cout << " •   Consider CRTP/concepts for performance-critical code" << std::endl;
    std::cout << " •   Don't call virtual functions in constructors/destructors" << std::endl;

    std::cout << "\n STILL RELEVANT IN MODERN C++:" << std::endl;
    std::cout << " •   YES! Essential for runtime polymorphism" << std::endl;
```

```
    std::cout << " •   Plugin systems, frameworks, design patterns" << std::endl;
    std::cout << " •   Modern C++ adds alternatives, not replacements" << std::endl;
    std::cout << " •   Choose based on requirements, not trends" << std::endl;

    std::cout << "\n==================================================================\n" << std::endl;

    return 0;
}
```

62 Source Code: SOLIDPrinciples.cpp

File: `src/SOLIDPrinciples.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// SOLID PRINCIPLES IN C++
// ==================================================================
// A comprehensive demonstration of the five SOLID design principles
// for object-oriented programming in Modern C++
//
// S - Single Responsibility Principle
// O - Open-Closed Principle
// L - Liskov Substitution Principle
// I - Interface Segregation Principle
// D - Dependency Inversion Principle
//
// Build: g++ -std=c++20 -Wall -Wextra -O2 -o SOLIDPrinciples SOLIDPrinciples.cpp
// ==================================================================

#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <fstream>
#include <sstream>
#include <cmath>
#include <stdexcept>

// ==================================================================
// PRINCIPLE 1: SINGLE RESPONSIBILITY PRINCIPLE (SRP)
// ==================================================================
// A class should have only ONE reason to change
// Each class should do ONE thing and do it well

namespace single_responsibility {

//  BAD: Class with multiple responsibilities
class BadEmployee {
private:
    std::string name_;
    double salary_;

public:
    BadEmployee(const std::string& name, double salary)
        : name_(name), salary_(salary) {}

    // Responsibility 1: Employee data management
    std::string getName() const { return name_; }
    double getSalary() const { return salary_; }

    // Responsibility 2: Salary calculation (WRONG!)
    double calculateTax() const {
```

```
        return salary_ * 0.25;
    }

    // Responsibility 3: Database operations (WRONG!)
    void saveToDatabase() {
        std::cout << "   [BAD] Saving " << name_ << " to database...\n";
        // Database logic here...
    }

    // Responsibility 4: Reporting (WRONG!)
    void generatePayslip() {
        std::cout << "   [BAD] Generating payslip for " << name_ << "\n";
        // Report generation logic...
    }
};

//  GOOD: Separate classes for separate responsibilities
class Employee {
private:
    std::string name_;
    double salary_;

public:
    Employee(const std::string& name, double salary)
        : name_(name), salary_(salary) {}

    std::string getName() const { return name_; }
    double getSalary() const { return salary_; }
};

// Responsibility 2: Tax calculation
class TaxCalculator {
public:
    double calculateTax(const Employee& emp) const {
        return emp.getSalary() * 0.25;
    }
};

// Responsibility 3: Database operations
class EmployeeRepository {
public:
    void save(const Employee& emp) {
        std::cout << "   [GOOD] Saving " << emp.getName() << " to database...\n";
        // Database logic here...
    }
};

// Responsibility 4: Reporting
class PayslipGenerator {
public:
    void generate(const Employee& emp) {
        std::cout << "   [GOOD] Generating payslip for " << emp.getName() << "\n";
```

```
        // Report generation logic...
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "PRINCIPLE 1: SINGLE RESPONSIBILITY PRINCIPLE (SRP)\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DEFINITION:\n";
    std::cout << "    A class should have only ONE reason to change.\n";
    std::cout << "    Each class should do ONE thing and do it well.\n\n";

    std::cout << "  BAD Example (Multiple Responsibilities):\n";
    BadEmployee badEmp("John Doe", 50000);
    badEmp.saveToDatabase();
    badEmp.generatePayslip();
    std::cout << "    Problem: Employee class handles data, tax, DB, and
        reports!\n\n";

    std::cout << "  GOOD Example (Single Responsibility):\n";
    Employee goodEmp("Jane Smith", 60000);
    TaxCalculator taxCalc;
    EmployeeRepository repo;
    PayslipGenerator payslip;

    repo.save(goodEmp);
    payslip.generate(goodEmp);
    std::cout << "    Tax: $" << taxCalc.calculateTax(goodEmp) << "\n\n";

    std::cout << "  BENEFITS:\n";
    std::cout << " •   Easier to understand and maintain\n";
    std::cout << " •   Changes to tax logic don't affect database code\n";
    std::cout << " •   Classes are more reusable and testable\n";
    std::cout << " •   Better separation of concerns\n";
}

} // namespace single_responsibility

// ==================================================================
// PRINCIPLE 2: OPEN-CLOSED PRINCIPLE (OCP)
// ==================================================================
// Open for extension, closed for modification
// You should be able to add new functionality without changing existing code

namespace open_closed {

//   BAD: Need to modify existing code for each new shape
class BadShapeCalculator {
public:
    double calculateArea(const std::string& shapeType, double dimension) {
        if (shapeType == "circle") {
            return 3.14159 * dimension * dimension;
        } else if (shapeType == "square") {
```

```
            return dimension * dimension;
        }
        // Need to modify this function for each new shape! (BAD)
        return 0.0;
    }
};

//  GOOD: Use polymorphism - open for extension, closed for modification
class Shape {
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;
    virtual std::string name() const = 0;
};

class Circle : public Shape {
private:
    double radius_;

public:
    explicit Circle(double radius) : radius_(radius) {}

    double area() const override {
        return 3.14159 * radius_ * radius_;
    }

    std::string name() const override { return "Circle"; }
};

class Square : public Shape {
private:
    double side_;

public:
    explicit Square(double side) : side_(side) {}

    double area() const override {
        return side_ * side_;
    }

    std::string name() const override { return "Square"; }
};

class Triangle : public Shape {
private:
    double base_;
    double height_;

public:
    Triangle(double base, double height) : base_(base), height_(height) {}

    double area() const override {
        return 0.5 * base_ * height_;
    }
```

```

    std::string name() const override { return "Triangle"; }
};

// This class is CLOSED for modification, but OPEN for extension
class AreaCalculator {
public:
    double totalArea(const std::vector<std::unique_ptr<Shape>>& shapes) const {
        double total = 0.0;
        for (const auto& shape : shapes) {
            total += shape->area();
        }
        return total;
    }

    void printAreas(const std::vector<std::unique_ptr<Shape>>& shapes) const {
        for (const auto& shape : shapes) {
            std::cout << "   " << shape->name() << " area: "
                      << shape->area() << "\n";
        }
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "PRINCIPLE 2: OPEN-CLOSED PRINCIPLE (OCP)\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DEFINITION:\n";
    std::cout << "    Software entities should be OPEN for extension,\n";
    std::cout << "    but CLOSED for modification.\n\n";

    std::cout << "  BAD Example:\n";
    BadShapeCalculator badCalc;
    std::cout << "    Circle area: " << badCalc.calculateArea("circle", 5.0) << "\n";
    std::cout << "    Problem: Must modify calculateArea() for each new shape!\n\n";

    std::cout << "  GOOD Example (Using Polymorphism):\n";
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(5.0));
    shapes.push_back(std::make_unique<Square>(4.0));
    shapes.push_back(std::make_unique<Triangle>(3.0, 6.0));

    AreaCalculator calc;
    calc.printAreas(shapes);
    std::cout << "    Total area: " << calc.totalArea(shapes) << "\n\n";

    std::cout << "  BENEFITS:\n";
    std::cout << " •   Add new shapes without modifying AreaCalculator\n";
    std::cout << " •   Existing code remains stable and tested\n";
    std::cout << " •   Reduces risk of breaking existing functionality\n";
```

```
    std::cout << " •   Promotes use of interfaces and polymorphism\n";
}

} // namespace open_closed

// ==================================================================
// PRINCIPLE 3: LISKOV SUBSTITUTION PRINCIPLE (LSP)
// ==================================================================
// Objects of a superclass should be replaceable with objects of a subclass
// without breaking the application

namespace liskov_substitution {

//   BAD: Violates LSP - Square changes Rectangle's behavior
class BadRectangle {
protected:
    int width_;
    int height_;

public:
    virtual ~BadRectangle() = default;  // Virtual destructor for polymorphic
        class
    virtual void setWidth(int w) { width_ = w; }
    virtual void setHeight(int h) { height_ = h; }

    int getWidth() const { return width_; }
    int getHeight() const { return height_; }

    virtual int area() const { return width_ * height_; }
};

class BadSquare : public BadRectangle {
public:
    // Violates LSP: Changes the behavior!
    void setWidth(int w) override {
        width_ = w;
        height_ = w;  // Square forces equal sides
    }

    void setHeight(int h) override {
        width_ = h;
        height_ = h;  // Square forces equal sides
    }
};

//   GOOD: Use composition instead of inheritance for Square
class Shape {
public:
    virtual ~Shape() = default;
    virtual int area() const = 0;
    virtual std::string type() const = 0;
};

class Rectangle : public Shape {
```

```
private:
    int width_;
    int height_;

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

    void setWidth(int w) { width_ = w; }
    void setHeight(int h) { height_ = h; }

    int getWidth() const { return width_; }
    int getHeight() const { return height_; }

    int area() const override { return width_ * height_; }
    std::string type() const override { return "Rectangle"; }
};

class Square : public Shape {
private:
    int side_;

public:
    explicit Square(int side) : side_(side) {}

    void setSide(int s) { side_ = s; }
    int getSide() const { return side_; }

    int area() const override { return side_ * side_; }
    std::string type() const override { return "Square"; }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "PRINCIPLE 3: LISKOV SUBSTITUTION PRINCIPLE (LSP)\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DEFINITION:\n";
    std::cout << "    Objects of a superclass should be replaceable with objects\n";
    std::cout << "    of a subclass without breaking the application.\n\n";

    std::cout << "  BAD Example (LSP Violation):\n";
    BadRectangle* rect = new BadSquare();
    rect->setWidth(5);
    rect->setHeight(4);
    std::cout << "    Expected area: 20 (5x4)\n";
    std::cout << "    Actual area: " << rect->area() << " (4x4)\n";
    std::cout << "    Problem: Square changed Rectangle's behavior!\n";
    delete rect;
    std::cout << "\n";

    std::cout << "  GOOD Example (LSP Compliant):\n";
    Rectangle goodRect(5, 4);
    std::cout << "    Rectangle (5x4) area: " << goodRect.area() << "\n";
```

```

    Square goodSquare(4);
    std::cout << "   Square (4x4) area: " << goodSquare.area() << "\n\n";

    std::cout << "  BENEFITS:\n";
    std::cout << " •   Substitutability is preserved\n";
    std::cout << " •   No unexpected behavior when using derived classes\n";
    std::cout << " •   Polymorphism works correctly\n";
    std::cout << " •   \"IS-A\" relationship is properly maintained\n";
}

} // namespace liskov_substitution

// ==================================================================
// PRINCIPLE 4: INTERFACE SEGREGATION PRINCIPLE (ISP)
// ==================================================================
// Clients should not be forced to depend on interfaces they don't use
// Many specific interfaces are better than one general-purpose interface

namespace interface_segregation {

//   BAD: Fat interface forces implementations to provide unused methods
class BadWorker {
public:
    virtual ~BadWorker() = default;
    virtual void work() = 0;
    virtual void eat() = 0;
    virtual void sleep() = 0;
};

class BadHumanWorker : public BadWorker {
public:
    void work() override {
        std::cout << "   [Human] Working...\n";
    }

    void eat() override {
        std::cout << "   [Human] Eating lunch...\n";
    }

    void sleep() override {
        std::cout << "   [Human] Sleeping...\n";
    }
};

class BadRobotWorker : public BadWorker {
public:
    void work() override {
        std::cout << "   [Robot] Working 24/7...\n";
    }

    // Robots don't eat or sleep! Forced to implement unused methods
    void eat() override {
        // Empty or throw exception (both are bad)
```

```
        std::cout << "   [Robot] ERROR: Robots don't eat!\n";
    }

    void sleep() override {
        // Empty or throw exception (both are bad)
        std::cout << "   [Robot] ERROR: Robots don't sleep!\n";
    }
};

//  GOOD: Segregated interfaces - clients use only what they need
class Workable {
public:
    virtual ~Workable() = default;
    virtual void work() = 0;
};

class Eatable {
public:
    virtual ~Eatable() = default;
    virtual void eat() = 0;
};

class Sleepable {
public:
    virtual ~Sleepable() = default;
    virtual void sleep() = 0;
};

// Human implements all interfaces
class HumanWorker : public Workable, public Eatable, public Sleepable {
public:
    void work() override {
        std::cout << "   [Human] Working...\n";
    }

    void eat() override {
        std::cout << "   [Human] Eating lunch...\n";
    }

    void sleep() override {
        std::cout << "   [Human] Sleeping...\n";
    }
};

// Robot only implements what it needs
class RobotWorker : public Workable {
public:
    void work() override {
        std::cout << "   [Robot] Working 24/7...\n";
    }
    // No eat() or sleep() - not forced to implement them!
};

void demonstrate() {
```

```
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "PRINCIPLE 4: INTERFACE SEGREGATION PRINCIPLE (ISP)\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DEFINITION:\n";
    std::cout << "    Clients should not be forced to depend on interfaces\n";
    std::cout << "    they don't use. Prefer many specific interfaces over\n";
    std::cout << "    one general-purpose interface.\n\n";

    std::cout << "  BAD Example (Fat Interface):\n";
    BadRobotWorker badRobot;
    badRobot.work();
    badRobot.eat();   // Forced to implement, but doesn't make sense!
    badRobot.sleep(); // Forced to implement, but doesn't make sense!
    std::cout << "\n";

    std::cout << "  GOOD Example (Segregated Interfaces):\n";
    HumanWorker human;
    human.work();
    human.eat();
    human.sleep();
    std::cout << "\n";

    RobotWorker robot;
    robot.work();
    std::cout << "    [Robot] No eat() or sleep() methods - clean interface!\n\n";

    std::cout << "  BENEFITS:\n";
    std::cout << " •   Classes implement only what they need\n";
    std::cout << " •   No dummy implementations or exceptions\n";
    std::cout << " •   Better decoupling and flexibility\n";
    std::cout << " •   Easier to understand and maintain\n";
}

} // namespace interface_segregation

// ==================================================================
// PRINCIPLE 5: DEPENDENCY INVERSION PRINCIPLE (DIP)
// ==================================================================
// High-level modules should not depend on low-level modules
// Both should depend on abstractions (interfaces)
// Abstractions should not depend on details, details should depend on abstractions

namespace dependency_inversion {

//  BAD: High-level class depends on low-level concrete classes
class BadMySQLDatabase {
public:
    void connect() {
        std::cout << "    [BAD] Connecting to MySQL...\n";
    }

```

```
    void saveData(const std::string& data) {
        std::cout << "   [BAD] Saving to MySQL: " << data << "\n";
    }
};

class BadUserService {
private:
    BadMySQLDatabase database_;  // Tightly coupled to MySQL!

public:
    void saveUser(const std::string& user) {
        database_.connect();
        database_.saveData(user);
        // If we want to switch to PostgreSQL, we must modify this class!
    }
};

//  GOOD: Depend on abstraction, not concrete implementation
class IDatabase {
public:
    virtual ~IDatabase() = default;
    virtual void connect() = 0;
    virtual void saveData(const std::string& data) = 0;
};

// Low-level modules implement the interface
class MySQLDatabase : public IDatabase {
public:
    void connect() override {
        std::cout << "   [GOOD] Connecting to MySQL...\n";
    }

    void saveData(const std::string& data) override {
        std::cout << "   [GOOD] Saving to MySQL: " << data << "\n";
    }
};

class PostgreSQLDatabase : public IDatabase {
public:
    void connect() override {
        std::cout << "   [GOOD] Connecting to PostgreSQL...\n";
    }

    void saveData(const std::string& data) override {
        std::cout << "   [GOOD] Saving to PostgreSQL: " << data << "\n";
    }
};

class MongoDBDatabase : public IDatabase {
public:
    void connect() override {
        std::cout << "   [GOOD] Connecting to MongoDB...\n";
    }

```

```
    void saveData(const std::string& data) override {
        std::cout << "   [GOOD] Saving to MongoDB: " << data << "\n";
    }
};

// High-level module depends on abstraction (IDatabase), not concrete class
class UserService {
private:
    IDatabase& database_;  // Depends on interface, not implementation!

public:
    explicit UserService(IDatabase& db) : database_(db) {}

    void saveUser(const std::string& user) {
        database_.connect();
        database_.saveData(user);
        // Can switch database without modifying this class!
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "PRINCIPLE 5: DEPENDENCY INVERSION PRINCIPLE (DIP)\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  DEFINITION:\n";
    std::cout << "   1. High-level modules should not depend on low-level
       modules.\n";
    std::cout << "      Both should depend on abstractions.\n";
    std::cout << "   2. Abstractions should not depend on details.\n";
    std::cout << "      Details should depend on abstractions.\n\n";

    std::cout << "  BAD Example (Tight Coupling):\n";
    BadUserService badService;
    badService.saveUser("John Doe");
    std::cout << "   Problem: UserService is tightly coupled to MySQL!\n\n";

    std::cout << "  GOOD Example (Dependency Injection):\n";

    MySQLDatabase mysql;
    UserService service1(mysql);
    service1.saveUser("Jane Smith");
    std::cout << "\n";

    PostgreSQLDatabase postgres;
    UserService service2(postgres);
    service2.saveUser("Bob Johnson");
    std::cout << "\n";

    MongoDBDatabase mongo;
    UserService service3(mongo);
    service3.saveUser("Alice Williams");
    std::cout << "\n";

```

```
    std::cout << "  BENEFITS:\n";
    std::cout << " •   Loose coupling between modules\n";
    std::cout << " •   Easy to switch implementations\n";
    std::cout << " •   Better testability (can inject mocks)\n";
    std::cout << " •   Follows \"Program to an interface, not an implementation\"\n";
}

} // namespace dependency_inversion

// ==================================================================
// MAIN - Demonstrate All SOLID Principles
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "                         SOLID PRINCIPLES IN C++                          \n";
    std::cout << "                Object-Oriented Design Best Practices                    \n";
    std::cout << "                                    \n";

    try {
        // S - Single Responsibility Principle
        single_responsibility::demonstrate();

        // O - Open-Closed Principle
        open_closed::demonstrate();

        // L - Liskov Substitution Principle
        liskov_substitution::demonstrate();

        // I - Interface Segregation Principle
        interface_segregation::demonstrate();

        // D - Dependency Inversion Principle
        dependency_inversion::demonstrate();

        // Summary
        std::cout << "\n" << std::string(70, '=') << "\n";
        std::cout << "SUMMARY: SOLID PRINCIPLES\n";
        std::cout << std::string(70, '=') << "\n\n";

        std::cout << "S - Single Responsibility Principle:\n";
        std::cout << "    One class, one job, one reason to change\n\n";

        std::cout << "O - Open-Closed Principle:\n";
        std::cout << "    Open for extension, closed for modification\n\n";

        std::cout << "L - Liskov Substitution Principle:\n";
        std::cout << "    Derived classes must be substitutable for base classes\n\n";

```

```
        std::cout << "I - Interface Segregation Principle:\n";
        std::cout << "    Many specific interfaces > one general interface\n\n";

        std::cout << "D - Dependency Inversion Principle:\n";
        std::cout << "    Depend on abstractions, not concrete implementations\n\n";

        std::cout << "  APPLYING SOLID:\n";
        std::cout << " •   Leads to more maintainable code\n";
        std::cout << " •   Reduces coupling between components\n";
        std::cout << " •   Makes code more testable\n";
        std::cout << " •   Improves code reusability\n";
        std::cout << " •   Easier to understand and modify\n\n";

        std::cout << "                                    \n";
        std::cout << "                ALL SOLID PRINCIPLES DEMONSTRATED!\n";
        std::cout << "                                    \n\n";

    } catch (const std::exception& e) {
        std::cerr << "  Error: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

63 Source Code: STLContainersNoHeap.cpp

File: `src/STLContainersNoHeap.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <array>
#include <vector>
#include <memory>
#include <cstdint>
#include <cstring>
#include <algorithm>
#include <span>

// ==================================================================
// CAN STL CONTAINERS WORK ON SYSTEMS WITHOUT HEAP?
// ==================================================================
// Comprehensive answer:
// 1. Most STL containers (vector, map, list) require heap by default
// 2. std::array works WITHOUT heap (C++11)
// 3. std::span works WITHOUT heap (C++20)
// 4. Custom allocators can use stack/static memory
// 5. C++17 PMR (Polymorphic Memory Resources) for flexible allocation
//
// This example demonstrates ALL approaches for heap-less STL usage
// ==================================================================

// ==================================================================
// 1. STACK-BASED CONTAINERS - std::array (NO HEAP!)
// ==================================================================

void example_std_array() {
    std::cout << "\n=== 1. std::array - STACK-BASED, NO HEAP ===" << std::endl;

    //  std::array: Fixed size, stack allocated, zero heap usage
    std::array<int, 10> sensor_data = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    std::cout << "\nSensor readings:" << std::endl;
    for (size_t i = 0; i < sensor_data.size(); ++i) {
        std::cout << "  Sensor " << i << ": " << sensor_data[i] << std::endl;
    }

    //  Full STL algorithm support
    std::cout << "\nSorted readings:" << std::endl;
    std::sort(sensor_data.begin(), sensor_data.end());
    for (const auto& reading : sensor_data) {
        std::cout << "  " << reading;
    }
    std::cout << std::endl;

    //  constexpr support (C++17)
    constexpr std::array<int, 5> compile_time = {1, 2, 3, 4, 5};
```

```
    [[maybe_unused]] constexpr int sum = compile_time[0] + compile_time[1]; // Compile-time computation

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << " •   Zero heap allocation" << std::endl;
    std::cout << " •   Stack-based (fast access)" << std::endl;
    std::cout << " •   Fixed size known at compile time" << std::endl;
    std::cout << " •   Same interface as std::vector" << std::endl;
    std::cout << " •   constexpr support (C++17)" << std::endl;
    std::cout << " •   Perfect for embedded systems" << std::endl;

    std::cout << "\n LIMITATIONS:" << std::endl;
    std::cout << " •   Fixed size (cannot grow/shrink)" << std::endl;
    std::cout << " •   Size must be known at compile time" << std::endl;
}

// ==================================================================
// 2. NON-OWNING VIEWS - std::span (C++20, NO HEAP!)
// ==================================================================

void process_data(std::span<int> data) {
    std::cout << "  Processing " << data.size() << " elements: ";
    for (auto val : data) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
}

void example_std_span() {
    std::cout << "\n=== 2. std::span - NON-OWNING VIEW, NO HEAP ===" << std::endl;

    // Stack-based data
    std::array<int, 5> arr = {1, 2, 3, 4, 5};
    int c_array[] = {10, 20, 30, 40};

    //  std::span: Non-owning view, zero allocation
    std::span<int> span1(arr);
    std::span<int> span2(c_array, 4);

    std::cout << "\nProcessing array:" << std::endl;
    process_data(span1);

    std::cout << "\nProcessing C array:" << std::endl;
    process_data(span2);

    // Sub-spans (slicing)
    auto first_three = span1.first(3);
    std::cout << "\nFirst 3 elements:" << std::endl;
    process_data(first_three);

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << " •   Zero allocation (view only)" << std::endl;
    std::cout << " •   Works with std::array, C arrays, std::vector" << std::
```

```
        endl;
    std::cout << " •   Safe bounds checking (can enable)" << std::endl;
    std::cout << " •   Sub-span support (slicing)" << std::endl;
    std::cout << " •   Modern replacement for (T*, size_t)" << std::endl;

    std::cout << "\n CAUTION:" << std::endl;
    std::cout << " •   Non-owning (must ensure lifetime)" << std::endl;
    std::cout << " •   Can dangle if underlying data destroyed" << std::endl;
}

// ================================================================
// 3. CUSTOM STACK ALLOCATOR FOR std::vector
// ================================================================

template<typename T, size_t N>
class StackAllocator {
private:
    alignas(T) uint8_t storage[N * sizeof(T)];
    size_t used = 0;

public:
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;

    StackAllocator() = default;

    template<typename U>
    StackAllocator(const StackAllocator<U, N>&) noexcept {}

    template<typename U>
    struct rebind {
        using other = StackAllocator<U, N>;
    };

    T* allocate(size_t n) {
        if (used + n > N) {
            throw std::bad_alloc();  // Or return nullptr in embedded
        }

        T* result = reinterpret_cast<T*>(storage + used * sizeof(T));
        used += n;
        return result;
    }

    void deallocate(T* p, size_t n) noexcept {
        // Simple stack allocator: only deallocate if it's the last allocation
        T* last = reinterpret_cast<T*>(storage + (used - n) * sizeof(T));
        if (p == last) {
            used -= n;
        }
        // Otherwise, memory is "leaked" until allocator is destroyed
        // For better deallocation, use a more sophisticated allocator
    }
```

```

    size_t capacity() const { return N; }
    size_t available() const { return N - used; }
};

template<typename T, size_t N, typename U, size_t M>
bool operator==(const StackAllocator<T, N>&, const StackAllocator<U, M>&) noexcept {
    return false;  // Different allocators are never equal
}

template<typename T, size_t N, typename U, size_t M>
bool operator!=(const StackAllocator<T, N>&, const StackAllocator<U, M>&) noexcept {
    return true;
}

void example_custom_stack_allocator() {
    std::cout << "\n=== 3. CUSTOM STACK ALLOCATOR FOR std::vector ===" << std::endl;

    //  std::vector with stack allocator - NO HEAP!
    using StackVec = std::vector<int, StackAllocator<int, 20>>;
    StackVec vec;

    std::cout << "\nAdding elements (stack allocation):" << std::endl;
    vec.reserve(10);  // Pre-reserve to avoid reallocation
    for (int i = 0; i < 10; ++i) {
        vec.push_back(i * 10);
        std::cout << "  Added " << i * 10
                  << ", size=" << vec.size() << std::endl;
    }

    std::cout << "\nVector contents:" << std::endl;
    for (const auto& val : vec) {
        std::cout << "  " << val;
    }
    std::cout << std::endl;

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << " •  Uses std::vector interface" << std::endl;
    std::cout << " •  Stack-based (no heap)" << std::endl;
    std::cout << " •  Can still push_back(), resize(), etc." << std::endl;
    std::cout << " •  Custom allocator handles memory" << std::endl;

    std::cout << "\n LIMITATIONS:" << std::endl;
    std::cout << " •  Fixed maximum capacity (20 in this case)" << std::endl;
    std::cout << " •  Simple deallocation strategy" << std::endl;
    std::cout << " •  Throws exception when out of memory" << std::endl;
}

// ==================================================================
// 4. STATIC MEMORY POOL ALLOCATOR
// ==================================================================
```

```

template<typename T, size_t PoolSize>
class PoolAllocator {
private:
    struct Block {
        alignas(T) uint8_t data[sizeof(T)];
    };

    // Static storage - shared across all instances
    inline static Block storage[PoolSize];
    inline static bool used[PoolSize] = {};
    inline static size_t allocated_count = 0;

public:
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;

    PoolAllocator() noexcept = default;

    template<typename U>
    PoolAllocator(const PoolAllocator<U, PoolSize>&) noexcept {}

    template<typename U>
    struct rebind {
        using other = PoolAllocator<U, PoolSize>;
    };

    T* allocate(size_t n) {
        if (n > PoolSize || allocated_count + n > PoolSize) {
            throw std::bad_alloc();
        }

        // Find contiguous free blocks
        for (size_t i = 0; i <= PoolSize - n; ++i) {
            bool found = true;
            for (size_t j = 0; j < n; ++j) {
                if (used[i + j]) {
                    found = false;
                    break;
                }
            }

            if (found) {
                for (size_t j = 0; j < n; ++j) {
                    used[i + j] = true;
                }
                allocated_count += n;
                return reinterpret_cast<T*>(&storage[i]);
            }
        }

        throw std::bad_alloc();
    }
```

```

    void deallocate(T* p, size_t n) noexcept {
        if (!p) return;

        auto base = reinterpret_cast<Block*>(storage);
        auto block = reinterpret_cast<Block*>(p);
        size_t index = block - base;

        if (index < PoolSize) {
            for (size_t i = 0; i < n && (index + i) < PoolSize; ++i) {
                used[index + i] = false;
            }
            allocated_count -= n;
        }
    }

    static size_t pool_size() { return PoolSize; }
};

template<typename T, size_t N, typename U, size_t M>
bool operator==(const PoolAllocator<T, N>&, const PoolAllocator<U, M>&) noexcept {
    return N == M;
}

template<typename T, size_t N, typename U, size_t M>
bool operator!=(const PoolAllocator<T, N>&, const PoolAllocator<U, M>&) noexcept {
    return N != M;
}

void example_pool_allocator() {
    std::cout << "\n=== 4. STATIC MEMORY POOL ALLOCATOR ===" << std::endl;

    struct Sensor {
        int id;
        float value;
        Sensor(int i, float v) : id(i), value(v) {
            std::cout << "    Sensor(" << id << ", " << value << ")" << std::endl;
        }
        ~Sensor() {
            std::cout << "    ~Sensor(" << id << ")" << std::endl;
        }
    };

    using SensorVec = std::vector<Sensor, PoolAllocator<Sensor, 10>>;

    std::cout << "\nAllocating sensors from static pool:" << std::endl;
    {
        SensorVec sensors;
        sensors.push_back(Sensor(1, 25.5f));
        sensors.push_back(Sensor(2, 30.2f));
        sensors.push_back(Sensor(3, 28.7f));
```

```

        std::cout << "\nSensors in vector: " << sensors.size() << std::endl;
    }
    std::cout << "\nVector destroyed, memory returned to pool" << std::endl;

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << " •   Static memory (no heap)" << std::endl;
    std::cout << " •   Fast O(1) allocation/deallocation" << std::endl;
    std::cout << " •   No fragmentation" << std::endl;
    std::cout << " •   Works with std::vector, std::list, etc." << std::endl;
    std::cout << " •   Memory reused across instances" << std::endl;

    std::cout << "\n LIMITATIONS:" << std::endl;
    std::cout << " •   Fixed pool size" << std::endl;
    std::cout << " •   Static storage (lifetime of program)" << std::endl;
    std::cout << " •   Simple implementation (one object per allocation)" <<
        std::endl;
}

// ==================================================================
// 5. FIXED CAPACITY VECTOR (NO ALLOCATOR NEEDED)
// ==================================================================

template<typename T, size_t Capacity>
class FixedVector {
private:
    alignas(T) uint8_t storage[Capacity * sizeof(T)];
    size_t count = 0;

public:
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;

    FixedVector() = default;

    ~FixedVector() {
        clear();
    }

    // Copy constructor
    FixedVector(const FixedVector& other) : count(0) {
        for (size_t i = 0; i < other.count; ++i) {
            push_back(other[i]);
        }
    }

    // Move constructor
    FixedVector(FixedVector&& other) noexcept : count(0) {
        for (size_t i = 0; i < other.count; ++i) {
            push_back(std::move(other[i]));
        }
        other.clear();
    }
```

```

    void push_back(const T& value) {
        if (count >= Capacity) {
            // In embedded: return error code or assert
            throw std::length_error("FixedVector capacity exceeded");
        }
        new (storage + count * sizeof(T)) T(value);
        ++count;
    }

    void push_back(T&& value) {
        if (count >= Capacity) {
            throw std::length_error("FixedVector capacity exceeded");
        }
        new (storage + count * sizeof(T)) T(std::move(value));
        ++count;
    }

    template<typename... Args>
    void emplace_back(Args&&... args) {
        if (count >= Capacity) {
            throw std::length_error("FixedVector capacity exceeded");
        }
        new (storage + count * sizeof(T)) T(std::forward<Args>(args)...);
        ++count;
    }

    void pop_back() {
        if (count > 0) {
            --count;
            reinterpret_cast<T*>(storage + count * sizeof(T))->~T();
        }
    }

    void clear() {
        while (count > 0) {
            pop_back();
        }
    }

    T& operator[](size_t index) {
        return *reinterpret_cast<T*>(storage + index * sizeof(T));
    }

    const T& operator[](size_t index) const {
        return *reinterpret_cast<const T*>(storage + index * sizeof(T));
    }

    T& front() { return (*this)[0]; }
    const T& front() const { return (*this)[0]; }
    T& back() { return (*this)[count - 1]; }
    const T& back() const { return (*this)[count - 1]; }

    size_t size() const { return count; }
```

```
    size_t capacity() const { return Capacity; }
    bool empty() const { return count == 0; }
    bool full() const { return count == Capacity; }

    iterator begin() { return reinterpret_cast<T*>(storage); }
    iterator end() { return reinterpret_cast<T*>(storage + count * sizeof(T)); }
    const_iterator begin() const { return reinterpret_cast<const T*>(storage); }
    const_iterator end() const { return reinterpret_cast<const T*>(storage + count * sizeof(T)); }
};

void example_fixed_vector() {
    std::cout << "\n=== 5. FIXED CAPACITY VECTOR (NO ALLOCATOR) ===" << std::endl;

    FixedVector<int, 10> vec;

    std::cout << "\nAdding elements:" << std::endl;
    for (int i = 0; i < 8; ++i) {
        vec.push_back(i * 5);
        std::cout << "  Pushed " << i * 5
                  << ", size=" << vec.size()
                  << "/" << vec.capacity() << std::endl;
    }

    std::cout << "\nVector contents:" << std::endl;
    for (const auto& val : vec) {
        std::cout << "  " << val;
    }
    std::cout << std::endl;

    std::cout << "\n BENEFITS:" << std::endl;
    std::cout << " •   Complete control over memory" << std::endl;
    std::cout << " •   No allocator overhead" << std::endl;
    std::cout << " •   Vector-like interface" << std::endl;
    std::cout << " •   Stack or static storage" << std::endl;
    std::cout << " •   Perfect for embedded systems" << std::endl;
    std::cout << " •   Compile-time capacity checking" << std::endl;
}

// ==================================================================
// 6. SUMMARY TABLE
// ==================================================================

void example_summary() {
    std::cout << "\n=== 6. SUMMARY: STL ON HEAP-LESS SYSTEMS ===" << std::endl;

    std::cout << "\                                n" << std::endl;
    std::cout << "  APPROACH                HEAP?   GROWS?   COMPLEXITY   USE CASE   " << std::endl;
    std::cout << "                                 " << std::endl;
```

```
    std::cout << "  std::array                NO       NO       SIMPLE     Fixed
            " << std::endl;
    std::cout << "  std::span (C++20)         NO       NO       SIMPLE     View
             " << std::endl;
    std::cout << "  Stack Allocator           NO      YES       MEDIUM
       Temporary  " << std::endl;
    std::cout << "  Pool Allocator            NO      YES       MEDIUM
       Reusable   " << std::endl;
    std::cout << "  FixedVector               NO      YES       LOW
       Embedded   " << std::endl;
    std::cout << "  std::vector (default)   YES      YES       SIMPLE
       General    " << std::endl;
    std::cout << "                                  " << std::endl;

    std::cout << "\n ANSWER: Can STL containers work without heap?" << std::
       endl;
    std::cout << "\n YES! Multiple approaches:" << std::endl;
    std::cout << "   1. Use std::array - Fixed size, stack-based" << std::endl
       ;
    std::cout << "   2. Use std::span - Non-owning view (C++20)" << std::endl;
    std::cout << "   3. Custom stack allocator - std::vector on stack" << std
       ::endl;
    std::cout << "   4. Static pool allocator - Pre-allocated memory" << std::
       endl;
    std::cout << "   5. Custom fixed vector - Complete control" << std::endl;

    std::cout << "\n NO (by default):" << std::endl;
    std::cout << " •   std::vector - Uses heap by default" << std::endl;
    std::cout << " •   std::map - Uses heap by default" << std::endl;
    std::cout << " •   std::list - Uses heap by default" << std::endl;
    std::cout << " •   std::string - Uses heap (except SSO)" << std::endl;
    std::cout << " •   All use std::allocator which calls new/delete" << std::
       endl;

    std::cout << "\n BEST PRACTICES FOR EMBEDDED SYSTEMS:" << std::endl;
    std::cout << "\n  1. PREFER std::array:" << std::endl;
    std::cout << "       Simple, safe, zero overhead" << std::endl;
    std::cout << "       Compile-time size checking" << std::endl;
    std::cout << "       Full STL algorithm support" << std::endl;

    std::cout << "\n  2. USE std::span for function parameters:" << std::endl;
    std::cout << "       Works with array, vector, C arrays" << std::endl;
    std::cout << "       Zero allocation" << std::endl;
    std::cout << "       Modern replacement for (T*, size_t)" << std::endl;

    std::cout << "\n  3. CUSTOM ALLOCATORS when needed:" << std::endl;
    std::cout << "       std::vector with stack/pool allocator" << std::endl;
    std::cout << "       Keep std::vector interface" << std::endl;
    std::cout << "        More complex, test thoroughly" << std::endl;

    std::cout << "\n  4. CUSTOM CONTAINERS for critical paths:" << std::endl;
    std::cout << "       FixedVector, CircularBuffer, etc." << std::endl;
    std::cout << "       Complete control over behavior" << std::endl;
    std::cout << "       Optimized for specific use case" << std::endl;
```

```
}

// ==================================================================
// 7. REAL-WORLD EXAMPLE: SENSOR DATA COLLECTION
// ==================================================================

void example_real_world() {
    std::cout << "\n=== 7. REAL-WORLD: SENSOR DATA COLLECTION ===" << std::
        endl;

    std::cout << "\n SCENARIO:" << std::endl;
    std::cout << " •   ARM Cortex-M4 @ 80MHz" << std::endl;
    std::cout << " •   64KB RAM (no heap)" << std::endl;
    std::cout << " •   Collect 100 sensor readings" << std::endl;
    std::cout << " •   Process with STL algorithms" << std::endl;

    //  Solution 1: std::array (simplest)
    std::cout << "\n SOLUTION 1: std::array" << std::endl;
    std::array<float, 100> readings1 = {};
    for (size_t i = 0; i < 100; ++i) {
        readings1[i] = 20.0f + (i % 10) * 0.5f;  // Simulated readings
    }

    auto min_max1 = std::minmax_element(readings1.begin(), readings1.end());
    std::cout << "   Min: " << *min_max1.first
              << ", Max: " << *min_max1.second << std::endl;

    //  Solution 2: FixedVector (dynamic-like)
    std::cout << "\n SOLUTION 2: FixedVector" << std::endl;
    FixedVector<float, 100> readings2;
    for (int i = 0; i < 100; ++i) {
        readings2.push_back(20.0f + (i % 10) * 0.5f);
    }

    auto min_max2 = std::minmax_element(readings2.begin(), readings2.end());
    std::cout << "   Min: " << *min_max2.first
              << ", Max: " << *min_max2.second << std::endl;

    std::cout << "\n BOTH WORK WITHOUT HEAP!" << std::endl;
    std::cout << " •   Zero dynamic allocation" << std::endl;
    std::cout << " •   Full STL algorithm support" << std::endl;
    std::cout << " •   Predictable memory usage" << std::endl;
    std::cout << " •   Real-time safe" << std::endl;
}

// ==================================================================
// MAIN
// ==================================================================

int main() {
    std::cout << "\n
        ==================================================================" <<
        std::endl;
    std::cout << "  CAN STL CONTAINERS WORK ON SYSTEMS WITHOUT HEAP?" << std::
```

```
        endl;
    std::cout << "
        ================================================================" <<
        std::endl;

    example_std_array();
    example_std_span();
    example_custom_stack_allocator();
    example_pool_allocator();
    example_fixed_vector();
    example_summary();
    example_real_world();

    std::cout << "\n
        ================================================================" <<
        std::endl;
    std::cout << "  FINAL ANSWER" << std::endl;
    std::cout << "
        ================================================================" <<
        std::endl;

    std::cout << "\n YES! STL containers CAN work without heap:" << std::endl
        ;

    std::cout << "\n1. BUILT-IN HEAP-LESS CONTAINERS:" << std::endl;
    std::cout << " •   std::array (C++11) - Fixed size, stack-based" << std::
        endl;
    std::cout << " •   std::span (C++20) - Non-owning view" << std::endl;
    std::cout << " •   std::string_view (C++17) - String view" << std::endl;

    std::cout << "\n2. CUSTOM ALLOCATORS:" << std::endl;
    std::cout << " •   Stack allocator - Use stack memory" << std::endl;
    std::cout << " •   Pool allocator - Pre-allocated pools" << std::endl;
    std::cout << " •   PMR allocators - C++17 polymorphic" << std::endl;
    std::cout << " •   Works with: vector, map, list, etc." << std::endl;

    std::cout << "\n3. CUSTOM CONTAINERS:" << std::endl;
    std::cout << " •   FixedVector - Vector with capacity limit" << std::endl;
    std::cout << " •   CircularBuffer - Ring buffer" << std::endl;
    std::cout << " •   StaticString - Fixed-size string" << std::endl;
    std::cout << " •   Full control over allocation" << std::endl;

    std::cout << "\n DEFAULT STL CONTAINERS USE HEAP:" << std::endl;
    std::cout << " •   std::vector - std::allocator uses new/delete" << std::
        endl;
    std::cout << " •   std::map - std::allocator uses new/delete" << std::endl
        ;
    std::cout << " •   std::string - std::allocator (except SSO)" << std::endl
        ;
    std::cout << " •   Must use custom allocator or alternatives" << std::endl
        ;

    std::cout << "\n RECOMMENDATION FOR EMBEDDED/REAL-TIME:" << std::endl;
    std::cout << "   1st choice: std::array (simple, safe)" << std::endl;
```

```
    std::cout << "   2nd choice: std::span for parameters" << std::endl;
    std::cout << "   3rd choice: Custom allocators for std::vector" << std::endl;
    std::cout << "   4th choice: Custom containers (FixedVector)" << std::endl;
    std::cout << "     Avoid: Default std::vector, std::map, std::string" << std::endl;

    std::cout << "\n==================================================================\n" << std::endl;

    return 0;
}
```

64 Source Code: SafetyCriticalSTLContainers.cpp

File: `src/SafetyCriticalSTLContainers.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// STL CONTAINERS FOR SAFETY-CRITICAL SYSTEMS
// ==================================================================
// Comprehensive guide for using STL containers in:
// - ISO 26262 (Automotive functional safety)
// - DO-178C (Avionics software)
// - IEC 61508 (Industrial safety)
// - Medical device software (IEC 62304)
//
// Based on:
// - MISRA C++:2008 and MISRA C++:2023
// - AUTOSAR C++14 Coding Guidelines
// - High Integrity C++ (HICPP)
// - JSF AV C++ Coding Standards
//
// Key concerns in safety-critical systems:
// 1. Deterministic behavior (no unpredictable timing)
// 2. Bounded memory usage (no dynamic allocation at runtime)
// 3. No hidden control flow (exceptions, virtual functions)
// 4. Verifiable and testable code
// 5. No undefined behavior
//
// Build: g++ -std=c++20 -Wall -Wextra -Wpedantic -O2 -o
    SafetyCriticalSTLContainers SafetyCriticalSTLContainers.cpp
// ==================================================================

#include <iostream>
#include <array>
#include <vector>
#include <list>
#include <forward_list>
#include <set>
#include <map>
#include <unordered_set>
#include <unordered_map>
#include <memory>
#include <cstdint>
#include <string>
#include <algorithm>

// ==================================================================
// SECTION 1: CONTAINER CLASSIFICATION FOR SAFETY-CRITICAL SYSTEMS
// ==================================================================

namespace container_classification {

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 1: STL CONTAINERS - SAFETY-CRITICAL CLASSIFICATION\n
```

```
        ";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "                                        \n";
    std::cout << "  CONTAINER            HEAP?   NODE-BASED?   SAFE?   NOTES                    \n";
    std::cout << "                                        \n";
    std::cout << "  std::array            NO         NO              BEST for safety     \n";
    std::cout << "  std::vector           YES        NO              OK with allocator   \n";
    std::cout << "  std::list             YES        YES             AVOID per-node      \n";
    std::cout << "  std::forward_list     YES        YES             AVOID per-node      \n";
    std::cout << "  std::deque            YES        NO              AVOID complex       \n";
    std::cout << "  std::set              YES        YES             AVOID per-node      \n";
    std::cout << "  std::map              YES        YES             AVOID per-node      \n";
    std::cout << "  std::multiset         YES        YES             AVOID per-node      \n";
    std::cout << "  std::multimap         YES        YES             AVOID per-node      \n";
    std::cout << "  std::unordered_set    YES        YES             AVOID per-bucket    \n";
    std::cout << "  std::unordered_map    YES        YES             AVOID per-bucket    \n";
    std::cout << "  std::unordered_*      YES        YES             AVOID per-bucket    \n";
    std::cout << "                                        \n\n";

    std::cout << "KEY ISSUES WITH HEAP ALLOCATION IN SAFETY-CRITICAL SYSTEMS:\n\n";

    std::cout << "1. NON-DETERMINISTIC TIMING:\n";
    std::cout << " •   malloc/new can take variable time depending on:\n";
    std::cout << "     - Current heap fragmentation\n";
    std::cout << "     - Size of allocation request\n";
    std::cout << "     - Operating system state\n";
    std::cout << " •   Worst-case execution time (WCET) becomes unpredictable\n";
    std::cout << " •   Unacceptable for hard real-time systems\n\n";

    std::cout << "2. MEMORY FRAGMENTATION:\n";
    std::cout << " •   Repeated allocations/deallocations cause fragmentation\n";
    std::cout << " •   Can lead to allocation failures even with available memory\n";
    std::cout << " •   Node-based containers (list, map, set) are worst offenders\n";
    std::cout << " •   Each element requires separate heap allocation\n\n";

```

```
    std::cout << "3. ALLOCATION FAILURES:\n";
    std::cout << " •   new can throw std::bad_alloc (exception handling required)\n";
    std::cout << " •   malloc can return nullptr (error handling required)\n";
    std::cout << " •   Difficult to recover gracefully in critical systems\n";
    std::cout << " •   May violate MISRA/AUTOSAR guidelines on exceptions\n\n";

    std::cout << "4. NODE-BASED CONTAINERS (list, map, set, unordered_*):\n";
    std::cout << " •   EACH ELEMENT requires separate heap allocation\n";
    std::cout << " •   1000 elements = 1000 malloc/free calls\n";
    std::cout << " •   Massive fragmentation over time\n";
    std::cout << " •   Iterator/pointer invalidation issues\n";
    std::cout << " •   Hidden per-node overhead (8-24 bytes per element)\n\n";

    std::cout << "5. STANDARDS GUIDANCE:\n";
    std::cout << " •   MISRA C++ Rule 18-4-1: Dynamic heap allocation shall not be used\n";
    std::cout << " •   AUTOSAR A18-5-1: Functions malloc, calloc, realloc, free not used\n";
    std::cout << " •   ISO 26262: Recommends static memory allocation for ASIL-D\n";
    std::cout << " •   DO-178C: Dynamic allocation discouraged for Level A software\n";
}

} // namespace container_classification

// ==================================================================
// SECTION 2: CONTAINERS TO AVOID IN SAFETY-CRITICAL SYSTEMS
// ==================================================================

namespace containers_to_avoid {

void demonstrate_list_issues() {
    std::cout << "\n" << std::string(80, '-') << "\n";
    std::cout << "WHY AVOID std::list, std::forward_list?\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "PROBLEM: Each element requires separate heap allocation\n\n";

    // Demonstrate list overhead
    std::list<int32_t> my_list;

    std::cout << "Adding 5 elements to std::list:\n";
    for (int32_t i = 0; i < 5; ++i) {
        my_list.push_back(i * 10);
        std::cout << "  push_back(" << i * 10 << ") - HEAP ALLOCATION #" << (i + 1) << "\n";
    }

    std::cout << "\n ISSUES:\n";
    std::cout << " •   5 separate malloc() calls - non-deterministic timing\n"
```

```
        ;
    std::cout << " •   Each node has pointer overhead (16-24 bytes per element)\n";
    std::cout << " •   Fragmentation increases with each allocation\n";
    std::cout << " •   Cannot use custom allocator to pre-allocate (node-by-node)\n";
    std::cout << " •   Cache-unfriendly (nodes scattered in memory)\n\n";

    std::cout << "  ALTERNATIVE: std::array or std::vector with reserve()\n";
}

void demonstrate_map_set_issues() {
    std::cout << "\n" << std::string(80, '-') << "\n";
    std::cout << "WHY AVOID std::map, std::set, std::multimap, std::multiset?\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "PROBLEM: Red-black tree with per-node heap allocation\n\n";

    std::map<int32_t, std::string> my_map;

    std::cout << "Adding 3 entries to std::map:\n";
    my_map[1] = "Critical";
    std::cout << "  map[1] = \"Critical\" - HEAP ALLOCATION for node\n";
    my_map[2] = "Warning";
    std::cout << "  map[2] = \"Warning\" - HEAP ALLOCATION for node\n";
    my_map[3] = "Info";
    std::cout << "  map[3] = \"Info\" - HEAP ALLOCATION for node\n";

    std::cout << "\n ISSUES:\n";
    std::cout << " •   Each insertion allocates a tree node (40-64 bytes overhead!)\n";
    std::cout << " •   Rebalancing operations (rotations) at runtime\n";
    std::cout << " •   Non-constant insertion time: O(log n) with allocation\n";
    std::cout << " •   Cannot pre-allocate all nodes\n";
    std::cout << " •   Iterators invalidated on modification\n\n";

    std::cout << "  ALTERNATIVES:\n";
    std::cout << " •   std::array of key-value pairs + binary search\n";
    std::cout << " •   std::vector of pairs + sort + binary search\n";
    std::cout << " •   Fixed-size hash table (compile-time allocation)\n";
}

void demonstrate_unordered_issues() {
    std::cout << "\n" << std::string(80, '-') << "\n";
    std::cout << "WHY AVOID std::unordered_map, std::unordered_set, std::unordered_multimap?\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "PROBLEM: Hash table with bucket allocation + chaining\n\n";

    std::unordered_map<int32_t, std::string> my_unordered_map;

```

```
    std::cout << "Adding 3 entries to std::unordered_map:\n";
    my_unordered_map[100] = "Sensor1";
    std::cout << "  map[100] = \"Sensor1\" - HEAP: buckets + node\n";
    my_unordered_map[200] = "Sensor2";
    std::cout << "  map[200] = \"Sensor2\" - HEAP: buckets + node\n";
    my_unordered_map[300] = "Sensor3";
    std::cout << "  map[300] = \"Sensor3\" - HEAP: buckets + node\n";

    std::cout << "\n ISSUES:\n";
    std::cout << " •   Initial bucket array allocation\n";
    std::cout << " •   Each element allocated separately (chaining)\n";
    std::cout << " •   REHASHING at runtime (load factor > threshold)\n";
    std::cout << "     - Allocates NEW larger bucket array\n";
    std::cout << "     - Rehashes ALL elements (moves to new buckets)\n";
    std::cout << "     - Deallocates old bucket array\n";
    std::cout << "     - MASSIVE non-determinism!\n";
    std::cout << " •   Cannot predict WCET (worst-case execution time)\n";
    std::cout << " •   Hash collisions cause unpredictable performance\n\n";

    std::cout << "  ALTERNATIVES:\n";
    std::cout << " •   Fixed-size hash table (no rehashing)\n";
    std::cout << " •   Perfect hashing (compile-time)\n";
    std::cout << " •   Sorted std::array + binary search\n";
}

void demonstrate_deque_issues() {
    std::cout << "\n" << std::string(80, '-') << "\n";
    std::cout << "WHY AVOID std::deque?\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "PROBLEM: Complex internal structure with multiple allocations\n\n";

    std::cout << "std::deque internal structure:\n";
    std::cout << " •  Array of pointers to fixed-size chunks\n";
    std::cout << " •  Each chunk allocated separately\n";
    std::cout << " •  Grows by allocating new chunks + updating pointer array\n";
    std::cout << " •  Pointer array itself may need reallocation\n\n";

    std::cout << "  ISSUES:\n";
    std::cout << " •   Multiple heap allocations (chunks + pointer array)\n";
    std::cout << " •   Complex iterator invalidation rules\n";
    std::cout << " •   Non-contiguous memory (cache-unfriendly)\n";
    std::cout << " •   Unpredictable memory usage pattern\n";
    std::cout << " •   Difficult to analyze for WCET\n\n";

    std::cout << "  ALTERNATIVES:\n";
    std::cout << " •   std::array (fixed size)\n";
    std::cout << " •   std::vector with reserve() (growable)\n";
    std::cout << " •   Circular buffer (custom implementation)\n";
}

void demonstrate() {
```

```
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 2: CONTAINERS TO AVOID IN SAFETY-CRITICAL SYSTEMS\n"
        ;
    std::cout << std::string(80, '=') << "\n";

    demonstrate_list_issues();
    demonstrate_map_set_issues();
    demonstrate_unordered_issues();
    demonstrate_deque_issues();
}

} // namespace containers_to_avoid

// ==================================================================
// SECTION 3: SAFE ALTERNATIVES - USING CUSTOM ALLOCATORS
// ==================================================================

namespace safe_alternatives {

// Pre-allocated memory pool allocator for safety-critical systems
template<typename T, size_t PoolSize>
class SafetyPoolAllocator {
private:
    struct alignas(T) Block {
        uint8_t data[sizeof(T)];
    };

    // Static storage - allocated at compile time
    inline static Block storage[PoolSize];
    inline static bool used[PoolSize] = {};
    inline static size_t allocated_count = 0;

public:
    using value_type = T;
    using size_type = std::size_t;

    SafetyPoolAllocator() noexcept = default;

    template<typename U>
    SafetyPoolAllocator(const SafetyPoolAllocator<U, PoolSize>&) noexcept {}

    template<typename U>
    struct rebind {
        using other = SafetyPoolAllocator<U, PoolSize>;
    };

    // Allocate from pool - bounded, deterministic
    [[nodiscard]] T* allocate(size_t n) {
        if (n > PoolSize || allocated_count + n > PoolSize) {
            // In safety-critical systems, handle gracefully without exception
            return nullptr;  // Or use error code
        }

        // Find contiguous free blocks
```

```
        for (size_t i = 0; i <= PoolSize - n; ++i) {
            bool found = true;
            for (size_t j = 0; j < n; ++j) {
                if (used[i + j]) {
                    found = false;
                    break;
                }
            }

            if (found) {
                for (size_t j = 0; j < n; ++j) {
                    used[i + j] = true;
                }
                allocated_count += n;
                return reinterpret_cast<T*>(&storage[i]);
            }
        }

        return nullptr;  // Pool exhausted
    }

    void deallocate(T* p, size_t n) noexcept {
        if (!p) return;

        // Find and mark blocks as free
        for (size_t i = 0; i < PoolSize; ++i) {
            if (reinterpret_cast<T*>(&storage[i]) == p) {
                for (size_t j = 0; j < n && (i + j) < PoolSize; ++j) {
                    used[i + j] = false;
                }
                allocated_count -= n;
                return;
            }
        }
    }

    static size_t capacity() { return PoolSize; }
    static size_t available() { return PoolSize - allocated_count; }
};

template<typename T, size_t N, typename U, size_t M>
bool operator==(const SafetyPoolAllocator<T, N>&, const SafetyPoolAllocator<U, M>&) noexcept {
    return N == M;
}

template<typename T, size_t N, typename U, size_t M>
bool operator!=(const SafetyPoolAllocator<T, N>&, const SafetyPoolAllocator<U, M>&) noexcept {
    return N != M;
}

void demonstrate_vector_with_allocator() {
    std::cout << "\n" << std::string(80, '-') << "\n";
```

```
    std::cout << "SAFE: std::vector WITH CUSTOM ALLOCATOR\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "  Solution: Pre-allocated pool at initialization\n\n";

    using SafeVector = std::vector<int32_t, SafetyPoolAllocator<int32_t,
        100>>;

    std::cout << "Creating std::vector with SafetyPoolAllocator<int32_t,
        100>:\n";
    std::cout << " •  Pool allocated at compile-time (static storage)\n";
    std::cout << " •  Max 100 elements (bounded memory)\n";
    std::cout << " •  No heap allocation during runtime\n";
    std::cout << " •  Deterministic allocation from pool\n\n";

    SafeVector safe_vec;
    safe_vec.reserve(10);  // Reserve from pool

    std::cout << "Adding 10 elements:\n";
    for (int32_t i = 0; i < 10; ++i) {
        safe_vec.push_back(i * 10);
        std::cout << "  push_back(" << i * 10 << ") - from POOL, not heap\n";
    }

    std::cout << "\ n BENEFITS:\n";
    std::cout << " •   NO heap allocation (malloc/new)\n";
    std::cout << " •   Deterministic timing (pool allocation is O(n) bounded)\
        n";
    std::cout << " •   No fragmentation\n";
    std::cout << " •   Bounded memory usage (max 100 elements)\n";
    std::cout << " •   Keeps std::vector interface\n";
    std::cout << " •   Compatible with algorithms (sort, find, etc.)\n\n";

    std::cout << "   LIMITATIONS:\n";
    std::cout << " •   Fixed maximum capacity (100 in this example)\n";
    std::cout << " •   Must handle allocation failures gracefully\n";
    std::cout << " •   Pool exhaustion returns nullptr (not exception)\n";
    std::cout << " •   Requires careful sizing during design phase\n";
}

void demonstrate_array_alternative() {
    std::cout << "\n" << std::string(80, '-') << "\n";
    std::cout << "SAFEST: std::array - NO ALLOCATOR NEEDED\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "  BEST SOLUTION for fixed-size data: std::array\n\n";

    std::array<int32_t, 10> safe_array = {0, 10, 20, 30, 40, 50, 60, 70, 80,
        90};

    std::cout << "std::array<int32_t, 10> properties:\n";
    std::cout << " •  Zero heap allocation (stack or static storage)\n";
    std::cout << " •  Size known at compile time\n";
    std::cout << " •  Bounds checking with .at()\n";
```

```
    std::cout << " •  Compatible with STL algorithms\n";
    std::cout << " •  constexpr support (C++17)\n";
    std::cout << " •  No overhead vs C array\n\n";

    std::cout << "Array contents:\n  ";
    for (const auto& val : safe_array) {
        std::cout << val << " ";
    }
    std::cout << "\n\n";

    std::cout << "  PERFECT FOR SAFETY-CRITICAL:\n";
    std::cout << " •   ISO 26262 ASIL-D compliant\n";
    std::cout << " •   MISRA C++ compliant (prefer over C arrays)\n";
    std::cout << " •   AUTOSAR C++14 recommended\n";
    std::cout << " •   DO-178C Level A acceptable\n";
    std::cout << " •   Deterministic behavior (O(1) access)\n";
    std::cout << " •   Predictable memory footprint\n";
    std::cout << " •   No hidden control flow\n";
    std::cout << " •   Verifiable and testable\n";
}

void demonstrate_initialization_phase_pattern() {
    std::cout << "\n" << std::string(80, '-') << "\n";
    std::cout << "PATTERN: Initialization Phase vs Runtime Phase\n";
    std::cout << std::string(80, '-') << "\n\n";

    std::cout << "GOLDEN RULE for safety-critical systems:\n";
    std::cout << "  'Allocate during INITIALIZATION, operate during RUNTIME'\n\n";

    std::cout << "                                  \n";
    std::cout << "  PHASE             ALLOWED                  FORBIDDEN\n";
    std::cout << "                                  \n";
    std::cout << "  INITIALIZATION  •   Dynamic allocation  •    Nothing\n";
    std::cout << "  (startup)  •        reserve()\n";
    std::cout << "   •                  Pool setup\n";
    std::cout << "   •                  Resource acquisition\n";
    std::cout << "                                  \n";
    std::cout << "  RUNTIME  •          Read/write data  •       new/malloc\n";
    std::cout << "  (operational)  •    Fixed operations  •      delete/free\n";
    std::cout << "   •                  Pool allocation  •       realloc\n";
    std::cout << "   •                  Bounded algorithms  •    resize/grow\n";
    std::cout << "   •                                           Exceptions\n";
    std::cout << "                                  \n\n";
```

```

    std::cout << "EXAMPLE: Two-phase lifecycle\n\n";

    std::cout << "// ===== INITIALIZATION PHASE (startup, non-critical) =====\n";
    std::cout << "void initialize() {\n";
    std::cout << "    static std::vector<SensorData> sensor_buffer;\n";
    std::cout << "    sensor_buffer.reserve(1000);  // ONE-TIME allocation\n";
    std::cout << "    // Pre-allocate ALL buffers, pools, resources\n";
    std::cout << "}\n\n";

    std::cout << "// ===== RUNTIME PHASE (operational, time-critical) =====\n";
    std::cout << "void process_sensor_data(SensorData data) {\n";
    std::cout << "    // NO allocation here! Only use pre-allocated memory\n";
    std::cout << "    sensor_buffer.push_back(data);  // No realloc (reserved)\n";
    std::cout << "    if (sensor_buffer.size() > 1000) {\n";
    std::cout << "        sensor_buffer.erase(sensor_buffer.begin());  // Shift\n";
    std::cout << "    }\n";
    std::cout << "}\n\n";

    std::cout << "  KEY PRINCIPLES:\n";
    std::cout << "   1. Allocate ONCE during initialization\n";
    std::cout << "   2. Use reserve() to prevent reallocation\n";
    std::cout << "   3. Never exceed reserved capacity at runtime\n";
    std::cout << "   4. Use static or global storage for long-lived data\n";
    std::cout << "   5. Monitor memory usage during testing\n";
    std::cout << "   6. Assert/verify no runtime allocation in production\n";
}

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 3: SAFE ALTERNATIVES WITH CUSTOM ALLOCATORS\n";
    std::cout << std::string(80, '=') << "\n";

    demonstrate_vector_with_allocator();
    demonstrate_array_alternative();
    demonstrate_initialization_phase_pattern();
}

} // namespace safe_alternatives

// ==================================================================
// SECTION 4: STANDARDS AND GUIDELINES SUMMARY
// ==================================================================

namespace standards_summary {

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 4: STANDARDS AND GUIDELINES SUMMARY\n";
    std::cout << std::string(80, '=') << "\n\n";
```

```
    std::cout << "                                        \n";
    std::cout << "  STANDARD / GUIDELINE   KEY REQUIREMENTS                    \n";
    std::cout << "                                        \n";
    std::cout << "  MISRA C++:2008          Rule 18-4-1: No dynamic heap allocation        \n";
    std::cout << "                          Rule 5-2-4: C-style casts forbidden                    \n";
    std::cout << "                          Rule 27-0-1: <cstdio> functions discouraged    \n";
    std::cout << "                                        \n";
    std::cout << "  MISRA C++:2023          Allows std::array, std::span                  \n";
    std::cout << "                          Allows std::vector with reserve()                   \n";
    std::cout << "                          Custom allocators acceptable                  \n";
    std::cout << "                                        \n";
    std::cout << "  AUTOSAR C++14           A18-5-1: malloc/calloc/realloc/free forbidden  \n";
    std::cout << "                          A18-5-2: new/delete in non-throwing form only  \n";
    std::cout << "                          A18-1-1: Prefer std::array over C arrays       \n";
    std::cout << "                                        \n";
    std::cout << "  ISO 26262 (ASIL-D)      Recommends static allocation                  \n";
    std::cout << "                          Dynamic allocation requires justification      \n";
    std::cout << "                          WCET analysis mandatory                                \n";
    std::cout << "                                        \n";
    std::cout << "  DO-178C Level A         Dynamic memory discouraged                            \n";
    std::cout << "  (Avionics)              Must prove bounded memory usage                   \n";
    std::cout << "                          Full MC/DC coverage required                  \n";
    std::cout << "                                        \n";
    std::cout << "  IEC 61508 (SIL 3/4)     Deterministic behavior required                   \n";
    std::cout << "                          Static analysis mandatory                                \n";
    std::cout << "                          Runtime monitoring recommended                        \n";
    std::cout << "                                        \n";
    std::cout << "  JSF AV C++              Rule 206: malloc/free not used                        \n";
    std::cout << "  (F-35 Fighter)          Rule 207: new/delete not used after init       \n";
    std::cout << "                          Rule 208: Prefer automatic storage                    \n";
    std::cout << "                                        \n\n";
```

```

    std::cout << "CONTAINER RECOMMENDATIONS BY STANDARD:\n\n";

    std::cout << "  UNIVERSALLY ACCEPTED (all standards):\n";
    std::cout << " •   std::array - Fixed size, no allocation\n";
    std::cout << " •   std::span (C++20) - Non-owning view\n";
    std::cout << " •   C arrays with size (if bounds-checked)\n\n";

    std::cout << "   CONDITIONALLY ACCEPTED (with restrictions):\n";
    std::cout << " •   std::vector - ONLY with:\n";
    std::cout << "     - reserve() called during initialization\n";
    std::cout << "     - Custom allocator (pool-based)\n";
    std::cout << "     - Never exceeds reserved capacity at runtime\n";
    std::cout << "     - Documented maximum size\n\n";

    std::cout << "  UNIVERSALLY FORBIDDEN (all standards):\n";
    std::cout << " •   std::list - Per-node heap allocation\n";
    std::cout << " •   std::forward_list - Per-node heap allocation\n";
    std::cout << " •   std::map / std::set - Per-node heap allocation\n";
    std::cout << " •   std::multimap / std::multiset - Per-node heap allocation\n";
    std::cout << " •   std::unordered_map / std::unordered_set - Rehashing\n";
    std::cout << " •   std::unordered_multimap / std::unordered_multiset - Rehashing\n";
    std::cout << " •   std::deque - Complex internal structure\n";
    std::cout << " •   std::string - Dynamic (unless with custom allocator)\n\n";

    std::cout << "RATIONALE FOR PROHIBITIONS:\n\n";

    std::cout << "1. NODE-BASED CONTAINERS (list, map, set):\n";
    std::cout << "   Problem: Each element → separate allocation\n";
    std::cout << "   Impact: 1000 elements = 1000 heap operations\n";
    std::cout << "   Result: Massive fragmentation + non-deterministic timing\n\n";

    std::cout << "2. HASH CONTAINERS (unordered_map, unordered_set):\n";
    std::cout << "   Problem: Rehashing at unpredictable times\n";
    std::cout << "   Impact: Can reallocate entire bucket array at runtime\n";
    std::cout << "   Result: Unpredictable latency spikes (WCET violation)\n\n";

    std::cout << "3. DEQUE:\n";
    std::cout << "   Problem: Complex multi-level allocation\n";
    std::cout << "   Impact: Chunks + pointer array allocations\n";
    std::cout << "   Result: Difficult to analyze and verify\n\n";
}

} // namespace standards_summary

// ==================================================================
// SECTION 5: PRACTICAL DECISION GUIDE
// ==================================================================

```

```
namespace decision_guide {

void demonstrate() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "SECTION 5: PRACTICAL DECISION GUIDE\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "DECISION TREE: Which Container Should I Use?\n\n";

    std::cout << "  Q: Is the size KNOWN at compile time?\n";
    std::cout << " \n";
    std::cout << "  YES → Use std::array<T, N>\n";
    std::cout << "            BEST choice for safety-critical\n";
    std::cout << "  •         Zero overhead\n";
    std::cout << "  •         No allocation\n";
    std::cout << "  •         Compile-time size checking\n";
    std::cout << " \n";
    std::cout << "  NO → Continue to Q2\n\n";

    std::cout << "  Q: Can I determine MAXIMUM size at design time?\n";
    std::cout << " \n";
    std::cout << "  YES → Use std::vector with reserve(MAX_SIZE)\n";
    std::cout << "            ACCEPTABLE with these conditions:\n";
    std::cout << "          1. Call reserve() during initialization\n";
    std::cout << "          2. Never exceed reserved capacity at runtime\n";
    std::cout << "          3. Monitor usage during testing\n";
    std::cout << "          4. Document maximum size\n";
    std::cout << "          5. Consider custom allocator for extra safety\n";
    std::cout << " \n";
    std::cout << "  NO (truly unbounded) → DESIGN PROBLEM!\n";
    std::cout << "            Safety-critical systems MUST have bounded resources\n";
    std::cout << "  •         Revisit requirements\n";
    std::cout << "  •         Determine worst-case maximum\n";
    std::cout << "  •         Add resource limits\n";
    std::cout << "  •         Consider circular buffer pattern\n";
    std::cout << " \n";
    std::cout << "  ALTERNATIVE → Fixed-capacity container (custom)\n\n";

    std::cout << "  Q: Do I need to insert/remove in the MIDDLE frequently?\n";
    std::cout << " \n";
    std::cout << "  YES (was thinking std::list) → RECONSIDER!\n";
    std::cout << "  •         std::list forbidden in safety-critical systems\n";
    std::cout << "  •         Alternatives:\n";
    std::cout << "            1. std::vector + erase/insert (usually fast enough)\n";
    std::cout << "            2. Circular buffer (if queue-like)\n";
    std::cout << "            3. Fixed-size priority queue\n";
    std::cout << "            4. Index-based linked list in std::array\n";
    std::cout << " \n";
    std::cout << "  NO → std::vector or std::array is fine\n\n";

```

```
    std::cout << "  Q: Do I need key-value mapping (was thinking std::map)?\n";
    std::cout << " \n";
    std::cout << "  YES → ALTERNATIVES (no std::map!):\n";
    std::cout << "          1. std::array of std::pair<Key, Value> + binary_search\n";
    std::cout << " •            Sort during initialization\n";
    std::cout << " •            Use binary_search at runtime (O(log n))\n";
    std::cout << " •            No allocation after init\n";
    std::cout << " \n";
    std::cout << "          2. Fixed-size hash table (compile-time)\n";
    std::cout << " •            Perfect hashing (no collisions)\n";
    std::cout << " •            O(1) lookup\n";
    std::cout << " •            Requires design-time hash analysis\n";
    std::cout << " \n";
    std::cout << "          3. std::vector<pair> + sort + binary_search\n";
    std::cout << " •            Sorted during initialization\n";
    std::cout << " •            Read-only at runtime\n";
    std::cout << " •            Good for configuration tables\n";
    std::cout << " \n";
    std::cout << "  NO → Continue\n\n";

    std::cout << "SUMMARY TABLE:\n\n";
    std::cout << "                                        \n";
    std::cout << " NEED                     SAFE SOLUTION             AVOID                 \n";
    std::cout << "                                        \n";
    std::cout << " Fixed-size sequence      std::array                C array (no bounds)   \n";
    std::cout << " Bounded sequence         std::vector + reserve()   std::vector (default) \n";
    std::cout << " FIFO queue               Circular buffer           std::queue/std::list  \n";
    std::cout << " LIFO stack               std::array + index        std::stack (default)  \n";
    std::cout << " Priority queue           std::array + heap alg     std::priority_queue   \n";
    std::cout << " Key-value map            sorted array + search     std::map              \n";
    std::cout << " Hash table               Fixed hash table          std::unordered_map    \n";
    std::cout << " Sorted set               sorted array + search     std::set              \n";
    std::cout << " Linked list              Index-based in array      std::list             \n";
    std::cout << "                                        \n\n";

    std::cout << "GOLDEN RULES:\n\n";
    std::cout << "1. ALWAYS prefer std::array when size is known\n";
    std::cout << "2. NEVER use node-based containers (list, map, set)\n";
    std::cout << "3. NEVER use std::unordered_* (rehashing issue)\n";
    std::cout << "4. IF using std::vector:\n";
    std::cout << " •   Call reserve() during initialization\n";
    std::cout << " •   Never exceed reserved capacity at runtime\n";
```

```
    std::cout << " •   Consider custom allocator for extra safety\n";
    std::cout << "5. DOCUMENT maximum sizes for all containers\n";
    std::cout << "6. VERIFY no runtime allocation (static analysis)\n";
    std::cout << "7. TEST with memory allocation monitoring enabled\n";
}

} // namespace decision_guide

// ==================================================================
// MAIN
// ==================================================================

int main() {
    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "STL CONTAINERS FOR SAFETY-CRITICAL SYSTEMS\n";
    std::cout << "ISO 26262 | DO-178C | MISRA C++ | AUTOSAR C++14\n";
    std::cout << std::string(80, '=') << "\n";

    container_classification::demonstrate();
    containers_to_avoid::demonstrate();
    safe_alternatives::demonstrate();
    standards_summary::demonstrate();
    decision_guide::demonstrate();

    std::cout << "\n" << std::string(80, '=') << "\n";
    std::cout << "FINAL ANSWER TO: Which STL containers to avoid?\n";
    std::cout << std::string(80, '=') << "\n\n";

    std::cout << "  ABSOLUTELY FORBIDDEN in safety-critical systems:\n\n";
    std::cout << "   1. std::list - Per-element heap allocation\n";
    std::cout << "   2. std::forward_list - Per-element heap allocation\n";
    std::cout << "   3. std::map - Per-node allocation (red-black tree)\n";
    std::cout << "   4. std::multimap - Per-node allocation\n";
    std::cout << "   5. std::set - Per-node allocation\n";
    std::cout << "   6. std::multiset - Per-node allocation\n";
    std::cout << "   7. std::unordered_map - Rehashing + per-bucket allocation\n";
    std::cout << "   8. std::unordered_multimap - Rehashing + per-bucket allocation\n";
    std::cout << "   9. std::unordered_set - Rehashing + per-bucket allocation\n";
    std::cout << "   10. std::unordered_multiset - Rehashing + per-bucket allocation\n";
    std::cout << "   11. std::deque - Complex multi-level allocation\n";
    std::cout << "   12. std::string - Dynamic (use with custom allocator only)\n\n";

    std::cout << "  YES, you CAN use containers with CUSTOM ALLOCATORS:\n\n";
    std::cout << " •   std::vector<T, CustomAllocator>\n";
    std::cout << " •   Pre-allocated pool-based allocator\n";
    std::cout << " •   Static memory pool (no heap)\n";
    std::cout << " •   Bounded capacity enforced\n";
    std::cout << " •   Allocation during initialization phase only\n\n";

```

```
    std::cout << "  SAFEST CHOICE (no allocator needed):\n\n";
    std::cout << " •   std::array<T, N> - Stack or static storage\n";
    std::cout << " •   std::span<T> (C++20) - Non-owning view\n";
    std::cout << " •   C++17 std::string_view - String view\n\n";

    std::cout << "  REMEMBER:\n";
    std::cout << "   'If heap is the issue, custom allocators can help,\n";
    std::cout << "    but std::array is ALWAYS the safest choice when size is
        known!'\n\n";

    std::cout << std::string(80, '=') << "\n\n";

    return 0;
}
```

65 Source Code: SearchAnagramsDictionary.cpp

File: `src/SearchAnagramsDictionary.cpp`
Repository: View on GitHub

```
// Example program
#include <iostream>
#include <string>

#include <algorithm>
#include <functional>
#include <iterator>

#include <vector>
#include <set>
#include <map>

using Pair_t = std::multimap< std::string, std::string> ;
using MMAPIterator = std::multimap<std::string, std::string>::iterator ;

std::set<std::string> mDictionary = { "act",  "ant",  "art",  "bat",  "bet",  "boss",  "cat",  "cap",  "cop",  "dear", "dog",
    "dip", "ear", "end", "eel", "fad", "fat", "fog", "gap", "god", "hat", "hit", "hot", "ink", "irk", "jot", "jab",
    "lap", "lip", "lot", "man", "nan", "nat", "net", "pat", "pet", "tap", "tar", "ten", "rat", "woo", "yoo", "zoo"};

int main()
{
    Pair_t Pairs = {};
    std::vector<std::string> emptyVectors = {};

    //std::cout << "All words in the mDictionary are :" << std::endl;
    for(const auto& it : mDictionary) {

       std::string sortedWord = it;
       std::sort(sortedWord.begin(), sortedWord.end());

       Pairs.emplace(std::make_pair(sortedWord, it));

       //std::cout << "for word " << it << ", sorted word = " << sortedWord << std::endl;
    }

    for(auto& obj : Pairs) {

        std::cout << "unsorted word = " << obj.second << ", sorted word : " << obj.first << std::endl;
    }


    for(auto& obj : Pairs) {

        std::pair <MMAPIterator, MMAPIterator> equalPairs = Pairs.equal_range(
```

```
            obj.first);

        if(equalPairs.second != equalPairs.first)
        {
            std::cout << "for the word = " << obj.second << ", its anagrams are : ";

            for (MMAPIterator it=equalPairs.first; it!=equalPairs.second; ++it) {

                if(obj.second != it->second) {
                    std::cout << it->second << ", ";
                }

                //std::cout << it->first << " : " << it->second << ", ";
            }

            std::cout << std::endl;
        } else {
            std::cout << "for the word = " << obj.second << " there are no anagrams!";
        }
    }
}
```

66 Source Code: SinglyLinkedList.cpp

File: `src/SinglyLinkedList.cpp`
Repository: View on GitHub

```
#include <iostream>
#include <string>
#include <string.h>
 /*
      pHead
       |
       |
    +---+------+     +---+-------+       +----+------+
    | 1  | next ----->|  2  | next-----> |  3 | NULL |
    +---+------+     +---+-------+       +----+------+
*/
extern "C" {
struct info;
typedef struct info{
    char name[20];
    int age;
    info *next;
} personal_info_t;

info* pHead = NULL; //by default first node will be point to nothing

extern "C" void deleteNode(info* p); //Forward declaration

extern "C" void deleteNodeByName(info** pH, const char* name) {
    info *prevNode, *currNode = *pH;

    while(currNode->next != NULL) {

        if(strcmp(currNode->name, name) == 0) {
            std::cout << "found the match, will delete node named " <<
currNode->name << std::endl;
            deleteNode(prevNode);
            return;
        }
        prevNode = currNode;
        currNode = currNode->next;
    }

    if(currNode->next == NULL) {
       if(strcmp(currNode->name, name) == 0) {
            std::cout << currNode->name << " is the last node...." << std::
                endl;
            memset((void*) currNode, 0, sizeof(info));
            free((info *) currNode);
            prevNode->next = NULL;
        }
        else {
            std::cout << "cannot find the matching name " << name << std::endl
                ;
```

```
        }
    }
}

extern "C" void deleteNode(info* p) {
    info* pNode = p->next;
    std::cout << p->name << " is followed by " << pNode->name << std::endl;
    p->next = pNode->next;
    memset((void*) pNode, 0, sizeof(info));
    free((info *) pNode);
    pNode = p->next;
    std::cout << p->name << " is followed by " << pNode->name << std::endl;
}
extern "C" void createNode(info** pH, const char* name, const int age) {
    info* pNewNode = (info*) malloc(sizeof(info));
    if(pNewNode != NULL) {
        strcpy(pNewNode->name, name);
        pNewNode->age = age;
        pNewNode->next= NULL;
    }
    else {
        return;
    }

    if(*pH == NULL) {
        pNewNode->next= NULL;
        *pH = pNewNode;
        std::cout << (*pH)->name << " is Head, pointing to memory address 0x"
            << std::hex << std::to_string((size_t)pH) << std::endl;
    }
    else {
        info* pNode = *pH;
        while(pNode->next != NULL) { pNode = pNode->next; }
        pNode->next= pNewNode;
    }
}
extern "C" const char* getName(const info* i) { return i->name; }
extern "C" const int getAge(const info* i) { return i->age;}

};

void print(info* pH) {
    std::cout << getName(pH) << " is aged " << std::to_string(getAge(pH)) << "
        followed by:" << std::endl;

    info* pNode = pH;
    while(pNode->next != NULL){
        pNode = pNode->next;
        std::cout << getName(pNode) << " aged " << getAge(pNode) << std::endl;
    }
}

int main() {
    createNode(&pHead, "AungBu", 20);
```

```
    createNode(&pHead, "Sar Oo", 30);
    createNode(&pHead, "AhBang", 25);
    createNode(&pHead, "AhLain", 27);
    createNode(&pHead, "Ahmedi", 28);

    print(pHead);

    deleteNodeByName(&pHead, "Sar Oo");
    deleteNodeByName(&pHead, "AhLain");

    print(pHead);

    return 0;

}
```

67 Source Code: StopTokenExample.cpp

File: `src/StopTokenExample.cpp`
Repository: View on GitHub

```cpp
// StopTokenExample.cpp
// Demonstrates std::stop_token (C++20) for graceful thread cancellation
// Shows the difference between old-style atomic flags and modern stop_token
//
// KEY CONCEPTS:
// 1. std::jthread - Automatically joins, supports stop_token
// 2. std::stop_token - Cooperative cancellation signal
// 3. std::stop_source - Controls stop requests
// 4. std::stop_callback - Actions on stop request
//
// USE CASES:
//   REST/Monitor threads (non-critical)
//   Worker threads in thread pools
//   Background tasks that need graceful shutdown
//   Any interruptible operation
//
// ADVANTAGES OVER atomic<bool>:
//   Standard, type-safe
//   Callback support (cleanup on stop)
//   Works with condition_variable_any
//   No manual join() needed with jthread
//   Cannot be accidentally set to true

#include <iostream>
#include <thread>
#include <stop_token>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <vector>
#include <atomic>
#include <chrono>
#include <string>
#include <sstream>
#include <iomanip>
#include <functional>

using namespace std::chrono_literals;

// ============================================================================
// SECTION 1: Old Style vs New Style - Comparison
// ============================================================================

namespace old_vs_new {

```

```cpp
// OLD STYLE: Using atomic<bool> for cancellation
class OldStyleThread {
private:
    std::atomic<bool> stop_flag_{false};
    std::thread thread_;
    std::mutex mutex_;
    std::condition_variable cv_;

public:
    void start() {
        thread_ = std::thread([this] {
            std::cout << "  [Old Style] Thread started\n";

            while (!stop_flag_.load()) {
                std::unique_lock<std::mutex> lock(mutex_);

                // Problem: Can't easily interrupt cv.wait()
                // Must use wait_for() with timeout
                if (cv_.wait_for(lock, 100ms, [this] { return stop_flag_.load
                    (); })) {
                    break;
                }

                std::cout << "  [Old Style] Working...\n";
            }

            std::cout << "  [Old Style] Thread stopped\n";
        });
    }

    void stop() {
        stop_flag_ = true;
        cv_.notify_all();
        if (thread_.joinable()) {
            thread_.join();  // Manual join required
        }
    }
};

// NEW STYLE: Using std::jthread with stop_token
class NewStyleThread {
private:
    std::jthread thread_;  // Automatically joins on destruction
    std::mutex mutex_;
    std::condition_variable_any cv_;  // Note: _any variant for stop_token

public:
    void start() {
        thread_ = std::jthread([this](std::stop_token stoken) {
            std::cout << "  [New Style] Thread started\n";

            while (!stoken.stop_requested()) {
                std::unique_lock<std::mutex> lock(mutex_);

```

```
                // Can be interrupted immediately!
                if (cv_.wait(lock, stoken, [] { return false; })) {
                    break;  // Stop requested
                }

                std::cout << "  [New Style] Working...\n";
            }

            std::cout << "  [New Style] Thread stopped\n";
        });
    }

    void stop() {
        thread_.request_stop();  // Signal stop
        // No manual join() needed - jthread does it automatically!
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 1: Old Style vs New Style ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "1. Old style with atomic<bool>:\n";
    {
        OldStyleThread old_thread;
        old_thread.start();
        std::this_thread::sleep_for(250ms);
        std::cout << "  [Main] Requesting stop...\n";
        old_thread.stop();
    }

    std::cout << "\n2. New style with stop_token:\n";
    {
        NewStyleThread new_thread;
        new_thread.start();
        std::this_thread::sleep_for(250ms);
        std::cout << "  [Main] Requesting stop...\n";
        new_thread.stop();
    }

    std::cout << "\n ADVANTAGES of stop_token:\n";
    std::cout << " • Type-safe (can't accidentally set to wrong value)\n";
    std::cout << " • Works with condition_variable_any for immediate interruption\n";
    std::cout << " • jthread automatically joins\n";
    std::cout << " • Standard interface for cancellation\n";
}

} // namespace old_vs_new

// ==========================================================================
```

```cpp
// SECTION 2: REST Service with stop_token (Non-Critical Thread)
// ==========================================================================

namespace rest_service_example {

class RESTService {
private:
    std::jthread thread_;
    std::mutex mutex_;
    std::condition_variable_any cv_;
    std::queue<std::string> request_queue_;
    int request_count_ = 0;

public:
    void start() {
        thread_ = std::jthread([this](std::stop_token stoken) {
            std::cout << "  [REST] Service started (non-critical thread)\n";

            // Register callback for cleanup
            std::stop_callback callback(stoken, [this] {
                std::cout << "  [REST] Stop requested, cleaning up...\n";
                std::lock_guard<std::mutex> lock(mutex_);
                std::cout << "  [REST] Processed " << request_count_
                          << " requests before stop\n";
            });

            try {
                while (!stoken.stop_requested()) {
                    std::unique_lock<std::mutex> lock(mutex_);

                    // Wait for requests or stop signal
                    bool stop = cv_.wait(lock, stoken, [this] {
                        return !request_queue_.empty();
                    });

                    if (stop) {
                        std::cout << "  [REST] Stop signal received during wait\n";
                        break;
                    }

                    // Process request
                    if (!request_queue_.empty()) {
                        std::string request = request_queue_.front();
                        request_queue_.pop();
                        lock.unlock();

                        handle_request(request, stoken);

                        lock.lock();
                    }
                }
```

```
            }
            catch (const std::exception& e) {
                std::cerr << "  [REST] Exception: " << e.what() << "\n";
                std::cerr << "  [REST] Non-critical thread exiting (core services OK)\n";
            }

            std::cout << "  [REST] Service stopped gracefully\n";
        });
    }

    void handle_request(const std::string& request, std::stop_token stoken) {
        std::cout << "  [REST] Handling request: " << request << "\n";

        // Simulate processing with interruptible sleep
        for (int i = 0; i < 5; ++i) {
            if (stoken.stop_requested()) {
                std::cout << "  [REST] Request handling interrupted\n";
                return;
            }

            std::this_thread::sleep_for(100ms);
        }

        ++request_count_;
        std::cout << "  [REST] Request completed (total: " << request_count_ << ")\n";

        // Simulate occasional error in REST service
        if (request.find("CAUSE_ERROR") != std::string::npos) {
            throw std::runtime_error("REST service error (non-critical)");
        }
    }

    void submit_request(const std::string& request) {
        std::lock_guard<std::mutex> lock(mutex_);
        request_queue_.push(request);
        cv_.notify_one();
    }

    void stop() {
        std::cout << "  [Main] Requesting REST service stop...\n";
        thread_.request_stop();
        // jthread automatically joins
    }

    bool is_running() const {
        return thread_.joinable() && !thread_.get_stop_token().stop_requested();
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
```

```
    std::cout << "=== SECTION 2: REST Service with stop_token ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    RESTService rest;
    rest.start();

    std::cout << "Submitting requests...\n";
    rest.submit_request("GET /api/users");
    rest.submit_request("POST /api/data");

    std::this_thread::sleep_for(300ms);

    std::cout << "\nSimulating graceful shutdown...\n";
    rest.stop();

    std::cout << "\ n REST service stopped gracefully\n";
    std::cout << "  Core services (database, cache) would continue running\n";
}

} // namespace rest_service_example

// ==========================================================================
// SECTION 3: Monitoring Service with stop_token
// ==========================================================================

namespace monitor_service_example {

class MonitorService {
private:
    std::jthread thread_;
    std::atomic<int> health_checks_{0};

public:
    void start() {
        thread_ = std::jthread([this](std::stop_token stoken) {
            std::cout << "  [Monitor] Service started (non-critical)\n";

            // Cleanup callback
            std::stop_callback cleanup(stoken, [this] {
                std::cout << "  [Monitor] Performed " << health_checks_.load()
                          << " health checks\n";
            });

            while (!stoken.stop_requested()) {
                // Perform health check
                ++health_checks_;
                std::cout << "  [Monitor] Health check #" << health_checks_.load()
                          << " - System OK\n";

```

```
                // Interruptible sleep
                auto wake_time = std::chrono::steady_clock::now() + 1s;
                while (std::chrono::steady_clock::now() < wake_time) {
                    if (stoken.stop_requested()) {
                        std::cout << "  [Monitor] Stop requested during sleep\n";
                        return;
                    }
                    std::this_thread::sleep_for(100ms);
                }
            }

            std::cout << "  [Monitor] Service stopped\n";
        });
    }

    void stop() {
        thread_.request_stop();
    }

    int get_health_checks() const {
        return health_checks_.load();
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 3: Monitoring Service with stop_token ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    MonitorService monitor;
    monitor.start();

    std::cout << "Monitoring running...\n";
    std::this_thread::sleep_for(2500ms);

    std::cout << "\nStopping monitor service...\n";
    monitor.stop();

    std::cout << "\n Total health checks: " << monitor.get_health_checks() << "\n";
}

} // namespace monitor_service_example

// ==========================================================================
// SECTION 4: Thread Pool with stop_token
// ==========================================================================

namespace thread_pool_example {
```

```

class ThreadPool {
private:
    std::vector<std::jthread> workers_;
    std::queue<std::function<void()>> tasks_;
    std::mutex mutex_;
    std::condition_variable_any cv_;

public:
    explicit ThreadPool(size_t num_threads) {
        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this, i](std::stop_token stoken) {
                std::cout << "  [Worker " << i << "] Started\n";

                while (!stoken.stop_requested()) {
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(mutex_);

                        // Wait for task or stop signal
                        bool stop = cv_.wait(lock, stoken, [this] {
                            return !tasks_.empty();
                        });

                        if (stop) {
                            std::cout << "  [Worker " << i << "] Stop requested\n";
                            break;
                        }

                        if (!tasks_.empty()) {
                            task = std::move(tasks_.front());
                            tasks_.pop();
                        }
                    }

                    if (task) {
                        std::cout << "  [Worker " << i << "] Executing task\n";
                        task();
                    }
                }

                std::cout << "  [Worker " << i << "] Stopped\n";
            });
        }
    }

    template<typename F>
    void enqueue(F&& task) {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            tasks_.emplace(std::forward<F>(task));
```

```
        }
        cv_.notify_one();
    }

    void stop() {
        std::cout << "  [ThreadPool] Requesting stop for all workers...\n";
        for (auto& worker : workers_) {
            worker.request_stop();
        }
        cv_.notify_all();
        // jthread automatically joins all threads
    }

    ~ThreadPool() {
        stop();
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 4: Thread Pool with stop_token ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    ThreadPool pool(3);

    std::cout << "Enqueueing tasks...\n";
    for (int i = 1; i <= 5; ++i) {
        pool.enqueue([i] {
            std::cout << "    Task " << i << " processing...\n";
            std::this_thread::sleep_for(200ms);
            std::cout << "    Task " << i << " completed\n";
        });
    }

    std::this_thread::sleep_for(1s);

    std::cout << "\nStopping thread pool...\n";
    pool.stop();

    std::cout << "\ n All workers stopped gracefully\n";
}

} // namespace thread_pool_example

// ==========================================================================
// SECTION 5: stop_callback - Cleanup on Stop
// ==========================================================================

namespace stop_callback_example {

```

```cpp
class ServiceWithCleanup {
private:
    std::jthread thread_;

public:
    void start() {
        thread_ = std::jthread([](std::stop_token stoken) {
            std::cout << "  [Service] Started\n";

            // Register multiple callbacks
            std::stop_callback callback1(stoken, [] {
                std::cout << "  [Cleanup 1] Closing network connections...\n";
            });

            std::stop_callback callback2(stoken, [] {
                std::cout << "  [Cleanup 2] Flushing buffers...\n";
            });

            std::stop_callback callback3(stoken, [] {
                std::cout << "  [Cleanup 3] Saving state...\n";
            });

            // Simulate work
            int work_count = 0;
            while (!stoken.stop_requested()) {
                ++work_count;
                std::cout << "  [Service] Working... (" << work_count << ")\n";
                std::this_thread::sleep_for(500ms);
            }

            std::cout << "  [Service] Stopped after " << work_count << " iterations\n";
            // Callbacks execute here (in reverse order of registration)
        });
    }

    void stop() {
        thread_.request_stop();
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 5: stop_callback for Cleanup ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    ServiceWithCleanup service;
    service.start();

    std::this_thread::sleep_for(1500ms);

    std::cout << "\nRequesting stop...\n";
    service.stop();
```

```

    std::cout << "\n Cleanup callbacks executed automatically\n";
}

} // namespace stop_callback_example

// ==========================================================================
// SECTION 6: Complete Microservices Example
// ==========================================================================

namespace microservices_example {

// Core service (critical - no stop_token, uses exception for abort)
class DatabaseService {
private:
    std::thread thread_;  // Regular thread, not jthread
    std::atomic<bool> running_{false};

public:
    void start() {
        running_ = true;
        thread_ = std::thread([this] {
            std::cout << "  [Database] CORE service started (critical)\n";

            try {
                while (running_) {
                    std::cout << "  [Database] Processing transactions...\n";
                    std::this_thread::sleep_for(500ms);

                    // Simulate critical error (would call abort())
                    // For demo, we just log it
                }
            }
            catch (const std::exception& e) {
                std::cerr << "  [Database] CRITICAL ERROR: " << e.what() << "\n";
                std::cerr << "  [Database] Would call abort() in real system\n";
            }

            std::cout << "  [Database] Core service stopped\n";
        });
    }

    void stop() {
        running_ = false;
        if (thread_.joinable()) {
            thread_.join();
        }
    }
```

```
};

// Non-core service (non-critical - uses stop_token)
class MetricsService {
private:
    std::jthread thread_;
    std::atomic<int> metrics_collected_{0};

public:
    void start() {
        thread_ = std::jthread([this](std::stop_token stoken) {
            std::cout << "  [Metrics] NON-CORE service started (stop_token)\n";

            std::stop_callback cleanup(stoken, [this] {
                std::cout << "  [Metrics] Collected " << metrics_collected_.load()
                          << " metrics before stop\n";
            });

            while (!stoken.stop_requested()) {
                ++metrics_collected_;
                std::cout << "  [Metrics] Collecting metric #"
                          << metrics_collected_.load() << "\n";

                // Interruptible sleep
                auto wake_time = std::chrono::steady_clock::now() + 800ms;
                while (std::chrono::steady_clock::now() < wake_time) {
                    if (stoken.stop_requested()) {
                        return;
                    }
                    std::this_thread::sleep_for(100ms);
                }
            }
        });
    }

    void stop() {
        std::cout << "  [Metrics] Requesting graceful stop...\n";
        thread_.request_stop();
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== SECTION 6: Complete Microservices Example ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    DatabaseService database;
    MetricsService metrics;

    std::cout << "Starting services...\n";
    database.start();
    metrics.start();
```

```

    std::this_thread::sleep_for(2s);

    std::cout << "\                    n\n";
    std::cout << "   Stopping NON-CORE service             \n";
    std::cout << "   (Metrics - uses stop_token)           \n";
    std::cout << "                    \n\n";

    metrics.stop();

    std::cout << "\ n Metrics service stopped gracefully\n";
    std::cout << "  Database service (core) still running\n";

    std::this_thread::sleep_for(1s);

    std::cout << "\nStopping all services...\n";
    database.stop();

    std::cout << "\ n All services stopped\n";
}

} // namespace microservices_example

// ==========================================================================
// SECTION 7: Summary and Best Practices
// ==========================================================================

void show_summary() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== Summary: stop_token for Non-Core Threads ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "                                \n";
    std::cout << "   THREAD TYPE          STOP MECHANISM                    \n";
    std::cout << "                                \n";
    std::cout << "   Core (Critical)      abort() on exception                    \n";
    std::cout << "                        std::thread + atomic<bool>                    \n";
    std::cout << "                        Ensures entire process stops                    \n";
    std::cout << "                                \n";
    std::cout << "   Non-Core             std::jthread + stop_token                    \n";
    std::cout << "   (REST, Monitor)      Graceful shutdown                    \n";
    std::cout << "                        Core services continue running                    \n";
    std::cout << "                                \n\n";
```

```

    std::cout << "WHY stop_token FOR NON-CORE THREADS:\n";
    std::cout << "    Type-safe cancellation signal\n";
    std::cout << "    Standard C++20 interface\n";
    std::cout << "    Works with condition_variable_any for immediate interruption\n";
    std::cout << "    std::jthread automatically joins (RAII)\n";
    std::cout << "    stop_callback for automatic cleanup\n";
    std::cout << "    Cannot accidentally set wrong value (unlike bool)\n\n";

    std::cout << "COMPARISON: atomic<bool> vs stop_token\n\n";

    std::cout << "  atomic<bool> approach:\n";
    std::cout << "    while (!stop_flag.load()) {\n";
    std::cout << "      cv.wait_for(lock, 100ms, ...);  // Needs timeout\n";
    std::cout << "      // Check flag regularly\n";
    std::cout << "    }\n";
    std::cout << "    // Manual join() required\n\n";

    std::cout << "  stop_token approach:\n";
    std::cout << "    std::jthread thread([](std::stop_token stoken) {\n";
    std::cout << "      while (!stoken.stop_requested()) {\n";
    std::cout << "        cv.wait(lock, stoken, ...);  // Immediate interrupt!\n";
    std::cout << "      }\n";
    std::cout << "    });\n";
    std::cout << "    thread.request_stop();  // Automatic join on destruction\n\n";

    std::cout << "BEST PRACTICES:\n";
    std::cout << "  1. Use stop_token for all non-critical threads\n";
    std::cout << "  2. Use std::jthread (not std::thread) for automatic joining\n";
    std::cout << "  3. Use condition_variable_any (not condition_variable)\n";
    std::cout << "  4. Check stop_requested() in loops\n";
    std::cout << "  5. Use stop_callback for cleanup logic\n";
    std::cout << "  6. Core services: use abort() for critical failures\n";
    std::cout << "  7. Non-core services: graceful shutdown with stop_token\n\n";

    std::cout << "MICROSERVICES PATTERN:\n";
    std::cout << " •  Database, Cache, Auth: CORE (abort on failure)\n";
    std::cout << " •  REST, Metrics, Logs: NON-CORE (stop_token for graceful stop)\n";
    std::cout << " •  Core failure → Process terminates (data integrity)\n";
    std::cout << " •  Non-core failure → Thread exits (core continues)\n";
}

// ============================================================================
// MAIN FUNCTION
// ============================================================================
```

```

int main() {
    std::cout << "\n";
    std::cout << "                                \n";
    std::cout << "          std::stop_token for Non-Core Thread Cancellation
              \n";
    std::cout << "                     C++20 Cooperative Cancellation
                     \n";
    std::cout << "                                \n";

    // Section 1: Old vs New
    old_vs_new::demonstrate();

    // Section 2: REST Service
    rest_service_example::demonstrate();

    // Section 3: Monitoring Service
    monitor_service_example::demonstrate();

    // Section 4: Thread Pool
    thread_pool_example::demonstrate();

    // Section 5: stop_callback
    stop_callback_example::demonstrate();

    // Section 6: Complete Example
    microservices_example::demonstrate();

    // Section 7: Summary
    show_summary();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All demonstrations completed!\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "KEY TAKEAWAY:\n";
    std::cout << "Use std::stop_token with std::jthread for graceful
        cancellation\n";
    std::cout << "of non-critical threads (REST, monitoring, metrics).\n";
    std::cout << "Core threads should still use abort() for critical failures
        .\n\n";

    return 0;
}
```

68 Source Code: StructuredBindings.cpp

File: `src/StructuredBindings.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <tuple>
#include <array>
#include <set>
#include <utility>
#include <algorithm>

// ====================================================================
// 1. BASIC STRUCTURED BINDINGS WITH PAIR
// ====================================================================
void example_basic_structured_bindings() {
    std::cout << "\n=== 1. BASIC STRUCTURED BINDINGS WITH PAIR ===" << std::endl;

    std::pair<int, std::string> person{25, "Alice"};

    // Old way (C++11/14)
    int age_old = person.first;
    std::string name_old = person.second;
    std::cout << "Old way: " << name_old << " is " << age_old << " years old" << std::endl;

    // New way with structured bindings (C++17)
    auto [age, name] = person;
    std::cout << "Structured binding: " << name << " is " << age << " years old" << std::endl;
}

// ====================================================================
// 2. STRUCTURED BINDINGS WITH TUPLE
// ====================================================================
void example_tuple_bindings() {
    std::cout << "\n=== 2. STRUCTURED BINDINGS WITH TUPLE ===" << std::endl;

    std::tuple<int, double, std::string, bool> data{42, 3.14, "Hello", true};

    // Structured binding with tuple
    auto [num, pi, text, flag] = data;

    std::cout << "Number: " << num << std::endl;
    std::cout << "Pi: " << pi << std::endl;
    std::cout << "Text: " << text << std::endl;
    std::cout << "Flag: " << std::boolalpha << flag << std::endl;
}

// ====================================================================
```

```
// 3. STRUCTURED BINDINGS WITH ARRAYS
// ==================================================================
void example_array_bindings() {
    std::cout << "\n=== 3. STRUCTURED BINDINGS WITH ARRAYS ===" << std::endl;

    // C-array
    int c_array[3] = {10, 20, 30};
    auto [a, b, c] = c_array;
    std::cout << "C-array elements: " << a << ", " << b << ", " << c << std::endl;

    // std::array
    std::array<std::string, 3> std_array = {"Red", "Green", "Blue"};
    auto [color1, color2, color3] = std_array;
    std::cout << "Colors: " << color1 << ", " << color2 << ", " << color3 << std::endl;
}

// ==================================================================
// 4. STRUCTURED BINDINGS WITH STRUCTS
// ==================================================================
struct Point {
    int x;
    int y;
    int z;
};

struct Employee {
    std::string name;
    int id;
    double salary;
};

void example_struct_bindings() {
    std::cout << "\n=== 4. STRUCTURED BINDINGS WITH STRUCTS ===" << std::endl;

    Point p{10, 20, 30};
    auto [x, y, z] = p;
    std::cout << "Point coordinates: x=" << x << ", y=" << y << ", z=" << z << std::endl;

    Employee emp{"John Doe", 12345, 75000.50};
    auto [emp_name, emp_id, emp_salary] = emp;
    std::cout << "Employee: " << emp_name << " (ID: " << emp_id
              << ", Salary: $" << emp_salary << ")" << std::endl;
}

// ==================================================================
// 5. STRUCTURED BINDINGS WITH MAP ITERATION
// ==================================================================
void example_map_iteration() {
    std::cout << "\n=== 5. STRUCTURED BINDINGS WITH MAP ITERATION ===" << std::endl;

```

```
    std::map<std::string, int> age_map = {
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35}
    };

    // Old way
    std::cout << "Old way:" << std::endl;
    for (const auto& pair : age_map) {
        std::cout << "  " << pair.first << " is " << pair.second << " years old" << std::endl;
    }

    // New way with structured bindings
    std::cout << "With structured bindings:" << std::endl;
    for (const auto& [name, age] : age_map) {
        std::cout << "  " << name << " is " << age << " years old" << std::endl;
    }
}

// ==================================================================
// 6. STRUCTURED BINDINGS WITH REFERENCES
// ==================================================================
void example_reference_bindings() {
    std::cout << "\n=== 6. STRUCTURED BINDINGS WITH REFERENCES ===" << std::endl;

    std::pair<int, std::string> data{100, "Original"};

    // Non-reference binding (copy)
    auto [val1, str1] = data;
    val1 = 200;
    str1 = "Modified Copy";
    std::cout << "Original after copy modification: "
              << data.first << ", " << data.second << std::endl;

    // Reference binding
    auto& [val2, str2] = data;
    val2 = 300;
    str2 = "Modified Reference";
    std::cout << "Original after reference modification: "
              << data.first << ", " << data.second << std::endl;
}

// ==================================================================
// 7. STRUCTURED BINDINGS WITH CONST
// ==================================================================
void example_const_bindings() {
    std::cout << "\n=== 7. STRUCTURED BINDINGS WITH CONST ===" << std::endl;

    const std::pair<int, std::string> data{42, "Constant"};

    // Const structured binding
```

```
    const auto& [value, text] = data;
    std::cout << "Const binding: " << value << ", " << text << std::endl;

    // value = 100;  // ERROR: cannot modify const
}

// ==================================================================
// 8. STRUCTURED BINDINGS WITH FUNCTION RETURNS
// ==================================================================
std::tuple<int, double, std::string> get_data() {
    return {42, 3.14159, "Function Return"};
}

std::pair<bool, std::string> validate_input(int value) {
    if (value > 0) {
        return {true, "Valid"};
    } else {
        return {false, "Invalid: must be positive"};
    }
}

void example_function_return_bindings() {
    std::cout << "\n=== 8. STRUCTURED BINDINGS WITH FUNCTION RETURNS ===" << std::endl;

    auto [num, pi, message] = get_data();
    std::cout << "From function: " << num << ", " << pi << ", " << message << std::endl;

    auto [is_valid, error_msg] = validate_input(10);
    std::cout << "Validation: " << std::boolalpha << is_valid << " - " << error_msg << std::endl;

    auto [is_valid2, error_msg2] = validate_input(-5);
    std::cout << "Validation: " << std::boolalpha << is_valid2 << " - " << error_msg2 << std::endl;
}

// ==================================================================
// 9. STRUCTURED BINDINGS WITH NESTED STRUCTURES
// ==================================================================
struct Address {
    std::string street;
    std::string city;
    int zipcode;
};

struct Person {
    std::string name;
    int age;
    Address address;
};

void example_nested_bindings() {
```

```
    std::cout << "\n=== 9. STRUCTURED BINDINGS WITH NESTED STRUCTURES ===" << std::endl;

    Person person{"John Smith", 30, {"123 Main St", "Springfield", 12345}};

    auto [name, age, address] = person;
    auto [street, city, zipcode] = address;

    std::cout << "Person: " << name << ", Age: " << age << std::endl;
    std::cout << "Address: " << street << ", " << city << " " << zipcode << std::endl;
}

// ==================================================================
// 10. STRUCTURED BINDINGS WITH STL ALGORITHMS
// ==================================================================
void example_algorithms_with_bindings() {
    std::cout << "\n=== 10. STRUCTURED BINDINGS WITH STL ALGORITHMS ===" << std::endl;

    std::vector<std::pair<std::string, int>> scores = {
        {"Alice", 95},
        {"Bob", 87},
        {"Charlie", 92},
        {"David", 88}
    };

    // Find student with highest score
    auto max_it = std::max_element(scores.begin(), scores.end(),
        [](const auto& a, const auto& b) {
            auto [name1, score1] = a;
            auto [name2, score2] = b;
            return score1 < score2;
        });

    if (max_it != scores.end()) {
        auto [top_student, top_score] = *max_it;
        std::cout << "Top student: " << top_student << " with score " << top_score << std::endl;
    }

    // Print all with structured bindings
    std::cout << "All scores:" << std::endl;
    std::for_each(scores.begin(), scores.end(),
        [](const auto& entry) {
            auto [name, score] = entry;
            std::cout << "  " << name << ": " << score << std::endl;
        });
}

// ==================================================================
// 11. STRUCTURED BINDINGS WITH INSERT OPERATIONS
// ==================================================================
void example_insert_bindings() {
```

```
    std::cout << "\n=== 11. STRUCTURED BINDINGS WITH INSERT OPERATIONS ===" << std::endl;

    std::map<int, std::string> id_map;

    // Insert returns pair<iterator, bool>
    auto [it1, inserted1] = id_map.insert({1, "First"});
    std::cout << "Insert 1: " << (inserted1 ? "Success" : "Failed") << std::endl;

    auto [it2, inserted2] = id_map.insert({1, "Duplicate"});
    std::cout << "Insert duplicate 1: " << (inserted2 ? "Success" : "Failed") << std::endl;

    auto [it3, inserted3] = id_map.insert({2, "Second"});
    std::cout << "Insert 2: " << (inserted3 ? "Success" : "Failed") << std::endl;

    std::cout << "Map contents:" << std::endl;
    for (const auto& [id, value] : id_map) {
        std::cout << "  " << id << ": " << value << std::endl;
    }
}

// ==================================================================
// 12. STRUCTURED BINDINGS WITH MULTIPLE RETURNS
// ==================================================================
struct Stats {
    double mean;
    double median;
    double stddev;
};

Stats calculate_stats(const std::vector<int>& data) {
    double sum = 0;
    for (int val : data) sum += val;
    double mean = sum / data.size();

    return {mean, static_cast<double>(data[data.size()/2]), 10.5}; // simplified
}

void example_multiple_return_values() {
    std::cout << "\n=== 12. STRUCTURED BINDINGS WITH MULTIPLE RETURNS ===" << std::endl;

    std::vector<int> dataset = {10, 20, 30, 40, 50};

    auto [mean, median, stddev] = calculate_stats(dataset);

    std::cout << "Statistics:" << std::endl;
    std::cout << "  Mean: " << mean << std::endl;
    std::cout << "  Median: " << median << std::endl;
    std::cout << "  Std Dev: " << stddev << std::endl;
```

```
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================
int main() {
    std::cout << "\n================================================" << std::endl;
    std::cout << "    C++17 STRUCTURED BINDINGS EXAMPLES" << std::endl;
    std::cout << "================================================" << std::endl;

    example_basic_structured_bindings();
    example_tuple_bindings();
    example_array_bindings();
    example_struct_bindings();
    example_map_iteration();
    example_reference_bindings();
    example_const_bindings();
    example_function_return_bindings();
    example_nested_bindings();
    example_algorithms_with_bindings();
    example_insert_bindings();
    example_multiple_return_values();

    std::cout << "\n================================================" << std::endl;
    std::cout << "    ALL EXAMPLES COMPLETED" << std::endl;
    std::cout << "================================================\n" << std::endl;

    return 0;
}
```

69 Source Code: SystemInteractionAndParsing.cpp

File: `src/SystemInteractionAndParsing.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// SYSTEM INTERACTION, FILESYSTEM, STRING_VIEW, AND REGEX
// ==================================================================
// This example demonstrates modern C++ for system programming:
//
// TOPICS COVERED:
// 1. Executing external commands (popen, system)
// 2. std::filesystem for file operations (C++17)
// 3. std::string_view for efficient parsing (C++17)
// 4. std::regex for pattern matching (C++11)
// 5. Parsing Linux tool output (ps, df, lsof, etc.)
// 6. Running bash and Python scripts from C++
// 7. File streams (ifstream, ofstream, fstream)
// 8. String streams (ostringstream, istringstream)
// 9. Stream manipulators and formatting
// 10. Binary I/O and stream state handling
// 11. Error handling with std::error_code
// 12. Thread synchronization (mutex, lock_guard, unique_lock)
// 13. File locking for concurrent access
// 14. Best practices for system integration
// 15. Security considerations
// 16. Cross-platform considerations
// ==================================================================

#include <iostream>
#include <iomanip>
#include <string>
#include <string_view>
#include <vector>
#include <map>
#include <regex>
#include <filesystem>
#include <fstream>
#include <sstream>
#include <array>
#include <memory>
#include <algorithm>
#include <cstdio>
#include <cstdlib>
#include <stdexcept>
#include <chrono>
#include <ctime>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <condition_variable>
#include <atomic>

namespace fs = std::filesystem;
```

```

// ==================================================================
// SECTION 1: BASIC COMMAND EXECUTION
// ==================================================================

// Execute command and capture output
std::string execute_command(std::string_view command) {
    std::array<char, 128> buffer;
    std::string result;

    // popen is safer than system() - doesn't invoke shell if not needed
    std::unique_ptr<FILE, decltype(&pclose)> pipe(
        popen(command.data(), "r"), pclose);

    if (!pipe) {
        throw std::runtime_error("popen() failed!");
    }

    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
        result += buffer.data();
    }

    return result;
}

// Execute command with return code
struct CommandResult {
    std::string output;
    int return_code;
    bool success;
};

CommandResult execute_command_with_status(std::string_view command) {
    CommandResult result;

    std::array<char, 256> buffer;
    std::string output;

    FILE* pipe = popen(command.data(), "r");
    if (!pipe) {
        return {.output = "", .return_code = -1, .success = false};
    }

    while (fgets(buffer.data(), buffer.size(), pipe) != nullptr) {
        output += buffer.data();
    }

    int status = pclose(pipe);
    result.output = output;
    result.return_code = WEXITSTATUS(status);
    result.success = (result.return_code == 0);

    return result;
}
```

```

void demonstrate_basic_execution() {
    std::cout << "\n=== 1. BASIC COMMAND EXECUTION ===" << std::endl;

    // Simple command
    std::cout << "\n1.1 Current directory:" << std::endl;
    std::string pwd = execute_command("pwd");
    std::cout << "   " << pwd;

    // Command with pipes
    std::cout << "1.2 Count files in current directory:" << std::endl;
    std::string count = execute_command("ls -1 | wc -l");
    std::cout << "   Files: " << count;

    // With status check
    std::cout << "1.3 Execute with status check:" << std::endl;
    auto result = execute_command_with_status("echo 'Hello from shell'");
    std::cout << "   Output: " << result.output;
    std::cout << "   Return code: " << result.return_code << std::endl;
    std::cout << "   Success: " << std::boolalpha << result.success << std::
        endl;

    std::cout << "\n KEY POINTS:" << std::endl;
    std::cout << " •   Use popen() to capture output" << std::endl;
    std::cout << " •   Check return codes for errors" << std::endl;
    std::cout << " •   pclose() is critical - use RAII" << std::endl;
}

// ==================================================================
// SECTION 2: std::string_view FOR EFFICIENT PARSING
// ==================================================================

// Parse process info using string_view (no copies!)
struct ProcessInfo {
    std::string pid;
    std::string user;
    std::string cpu;
    std::string mem;
    std::string command;
};

std::vector<std::string_view> split_string_view(std::string_view str, char
    delimiter) {
    std::vector<std::string_view> tokens;
    size_t start = 0;

    while (start < str.size()) {
        // Skip leading delimiters
        while (start < str.size() && str[start] == delimiter) {
            ++start;
        }

        if (start >= str.size()) break;

```

```
        // Find end of token
        size_t end = str.find(delimiter, start);
        if (end == std::string_view::npos) {
            end = str.size();
        }

        tokens.push_back(str.substr(start, end - start));
        start = end + 1;
    }

    return tokens;
}

// Efficient parsing with string_view - no memory allocations for substrings!
std::vector<ProcessInfo> parse_ps_output(std::string_view output) {
    std::vector<ProcessInfo> processes;

    size_t line_start = 0;
    size_t line_end = output.find('\n');

    // Skip header line
    if (line_end != std::string_view::npos) {
        line_start = line_end + 1;
    }

    while (line_start < output.size()) {
        line_end = output.find('\n', line_start);
        if (line_end == std::string_view::npos) {
            line_end = output.size();
        }

        std::string_view line = output.substr(line_start, line_end -
            line_start);
        if (line.empty()) {
            line_start = line_end + 1;
            continue;
        }

        auto tokens = split_string_view(line, ' ');
        if (tokens.size() >= 5) {
            ProcessInfo info;
            info.pid = std::string(tokens[0]);
            info.user = std::string(tokens[1]);
            info.cpu = std::string(tokens[2]);
            info.mem = std::string(tokens[3]);

            // Command is rest of line - find position of 5th token in
                original output
            size_t token4_pos = line.find(tokens[4]);
            if (token4_pos != std::string_view::npos) {
                size_t cmd_offset = line_start + token4_pos;
                info.command = std::string(output.substr(cmd_offset, line_end
                    - cmd_offset));
            }
```

```

            processes.push_back(info);
        }

        line_start = line_end + 1;
    }

    return processes;
}

void demonstrate_string_view_parsing() {
    std::cout << "\n=== 2. std::string_view PARSING ===" << std::endl;

    // Get process list
    std::string ps_output = execute_command("ps aux | head -20");

    std::cout << "\n2.1 Parsing ps output with string_view (zero-copy):" <<
        std::endl;
    auto processes = parse_ps_output(ps_output);

    std::cout << "   Found " << processes.size() << " processes" << std::endl;
    std::cout << "\n   Top 5 by CPU:" << std::endl;

    // Sort by CPU
    std::sort(processes.begin(), processes.end(),
              [](const ProcessInfo& a, const ProcessInfo& b) {
                  return std::stod(a.cpu) > std::stod(b.cpu);
              });

    for (size_t i = 0; i < std::min(size_t(5), processes.size()); i++) {
        std::cout << "   " << processes[i].pid << " "
                  << processes[i].cpu << "% CPU "
                  << processes[i].command.substr(0, 50) << std::endl;
    }

    std::cout << "\n string_view benefits:" << std::endl;
    std::cout << " •   No memory allocations during parsing" << std::endl;
    std::cout << " •   Fast substring operations" << std::endl;
    std::cout << " •   Perfect for tokenizing large outputs" << std::endl;
}

// ==================================================================
// SECTION 3: std::regex FOR PATTERN MATCHING
// ==================================================================

// Parse network interface info with regex
struct NetworkInterface {
    std::string name;
    std::string ip_address;
    std::string netmask;
    std::string status;
};

std::vector<NetworkInterface> parse_ifconfig_with_regex(std::string_view
```

```
    output) {
    std::vector<NetworkInterface> interfaces;

    // Regex patterns
    std::regex iface_pattern(R"(^(\w+):\s+flags=.+)");
    std::regex inet_pattern(R"(inet\s+(\d+\.\d+\.\d+\.\d+)\s+netmask\s+(\S+))"
        );
    std::regex status_pattern(R"(status:\s+(\w+))");

    std::string output_str(output);
    std::istringstream iss(output_str);
    std::string line;

    NetworkInterface current_iface;
    bool has_current = false;

    while (std::getline(iss, line)) {
        std::smatch match;

        // Check for interface name
        if (std::regex_search(line, match, iface_pattern)) {
            // Save previous interface
            if (has_current && !current_iface.ip_address.empty()) {
                interfaces.push_back(current_iface);
            }

            // Start new interface
            current_iface = NetworkInterface{};
            current_iface.name = match[1].str();
            has_current = true;
        }
        // Check for IP address
        else if (has_current && std::regex_search(line, match, inet_pattern))
            {
            current_iface.ip_address = match[1].str();
            current_iface.netmask = match[2].str();
        }
        // Check for status
        else if (has_current && std::regex_search(line, match, status_pattern)
            ) {
            current_iface.status = match[1].str();
        }
    }

    // Don't forget last interface
    if (has_current && !current_iface.ip_address.empty()) {
        interfaces.push_back(current_iface);
    }

    return interfaces;
}

// Parse log files with regex
struct LogEntry {
```

```
    std::string timestamp;
    std::string level;
    std::string message;
};

std::vector<LogEntry> parse_log_with_regex(std::string_view log_content) {
    std::vector<LogEntry> entries;

    // Match: [2024-01-15 10:30:45] ERROR: Something went wrong
    std::regex log_pattern(R"(\[(\d{4}-\d{2}-\d{2}\s+\d{2}:\d{2}:\d{2})\]\s+(\w+):\s+(.+))");

    std::string log_str(log_content);
    std::istringstream iss(log_str);
    std::string line;

    while (std::getline(iss, line)) {
        std::smatch match;
        if (std::regex_search(line, match, log_pattern)) {
            entries.push_back({
                .timestamp = match[1].str(),
                .level = match[2].str(),
                .message = match[3].str()
            });
        }
    }

    return entries;
}

void demonstrate_regex_parsing() {
    std::cout << "\n=== 3. std::regex PATTERN MATCHING ===" << std::endl;

    // Network interface parsing
    std::cout << "\n3.1 Parsing network interfaces with regex:" << std::endl;
    std::string ifconfig_output = execute_command("ifconfig 2>/dev/null || ip addr 2>/dev/null || echo 'No network tools available'");

    auto interfaces = parse_ifconfig_with_regex(ifconfig_output);
    for (const auto& iface : interfaces) {
        std::cout << "   Interface: " << iface.name << std::endl;
        std::cout << "      IP: " << iface.ip_address << std::endl;
        std::cout << "      Netmask: " << iface.netmask << std::endl;
        if (!iface.status.empty()) {
            std::cout << "      Status: " << iface.status << std::endl;
        }
    }

    // Create sample log for parsing
    std::cout << "\n3.2 Parsing log files with regex:" << std::endl;
    std::string sample_log =
        "[2024-01-15 10:30:45] INFO: Application started\n"
        "[2024-01-15 10:30:46] DEBUG: Loading configuration\n"
        "[2024-01-15 10:30:47] ERROR: Database connection failed\n"
```

```
        "[2024-01-15 10:30:48] WARN: Retrying connection\n"
        "[2024-01-15 10:30:50] INFO: Connection established\n";

    auto log_entries = parse_log_with_regex(sample_log);

    std::cout << "   Found " << log_entries.size() << " log entries:" << std::endl;
    for (const auto& entry : log_entries) {
        std::cout << "   [" << entry.timestamp << "] "
                  << entry.level << ": " << entry.message << std::endl;
    }

    // Filter errors
    std::cout << "\n   Errors only:" << std::endl;
    for (const auto& entry : log_entries) {
        if (entry.level == "ERROR") {
            std::cout << "     " << entry.message << std::endl;
        }
    }

    std::cout << "\n regex use cases:" << std::endl;
    std::cout << " •   Structured text parsing (logs, config)" << std::endl;
    std::cout << " •   Validation (emails, IPs, dates)" << std::endl;
    std::cout << " •   Extraction from unstructured output" << std::endl;
}

// ==================================================================
// SECTION 4: std::filesystem OPERATIONS
// ==================================================================

void demonstrate_filesystem() {
    std::cout << "\n=== 4. std::filesystem OPERATIONS ===" << std::endl;

    std::error_code ec;

    // Current path
    std::cout << "\n4.1 Current path:" << std::endl;
    fs::path current = fs::current_path(ec);
    std::cout << "   " << current << std::endl;

    // Create temporary test directory
    fs::path test_dir = fs::temp_directory_path(ec) / "cpp_test_dir";
    std::cout << "\n4.2 Creating test directory:" << std::endl;
    std::cout << "   " << test_dir << std::endl;

    if (fs::create_directories(test_dir, ec)) {
        std::cout << "    Created" << std::endl;
    } else {
        std::cout << "     Already exists or error: " << ec.message() << std::endl;
    }

    // Create test files
    std::cout << "\n4.3 Creating test files:" << std::endl;
```

```
    for (int i = 0; i < 3; i++) {
        fs::path file = test_dir / ("test_file_" + std::to_string(i) + ".txt");
        std::ofstream ofs(file);
        ofs << "Test content " << i << "\n";
        ofs << "Line 2 of file " << i << "\n";
        std::cout << "   Created: " << file.filename() << std::endl;
    }

    // Iterate directory
    std::cout << "\n4.4 Directory contents:" << std::endl;
    for (const auto& entry : fs::directory_iterator(test_dir, ec)) {
        std::cout << "   " << entry.path().filename()
                  << " (" << fs::file_size(entry, ec) << " bytes)" << std::endl;
    }

    // File info
    std::cout << "\n4.5 File properties:" << std::endl;
    auto first_file = test_dir / "test_file_0.txt";
    if (fs::exists(first_file, ec)) {
        std::cout << "   Size: " << fs::file_size(first_file, ec) << " bytes" << std::endl;
        std::cout << "   Is regular file: " << fs::is_regular_file(first_file, ec) << std::endl;
        std::cout << "   Is directory: " << fs::is_directory(first_file, ec) << std::endl;

        auto ftime = fs::last_write_time(first_file, ec);
        std::cout << "   Last modified: " << std::chrono::system_clock::to_time_t(
            std::chrono::file_clock::to_sys(ftime)) << " (unix timestamp)" << std::endl;
    }

    // Path operations
    std::cout << "\n4.6 Path operations:" << std::endl;
    fs::path example = "/home/user/documents/file.txt";
    std::cout << "   Full path: " << example << std::endl;
    std::cout << "   Filename: " << example.filename() << std::endl;
    std::cout << "   Stem: " << example.stem() << std::endl;
    std::cout << "   Extension: " << example.extension() << std::endl;
    std::cout << "   Parent: " << example.parent_path() << std::endl;

    // Cleanup
    std::cout << "\n4.7 Cleanup:" << std::endl;
    std::uintmax_t removed = fs::remove_all(test_dir, ec);
    std::cout << "   Removed " << removed << " files/directories" << std::endl;

    std::cout << "\n filesystem features:" << std::endl;
    std::cout << " •   Cross-platform path handling" << std::endl;
    std::cout << " •   Iterator for directory traversal" << std::endl;
    std::cout << " •   Error codes for exception-free errors" << std::endl;
```

```
}

// ==================================================================
// SECTION 5: RUNNING BASH SCRIPTS
// ==================================================================

void create_test_bash_script(const fs::path& script_path) {
    std::ofstream script(script_path);
    script << "#!/bin/bash\n";
    script << "# Test bash script called from C++\n\n";
    script << "echo \"Script started with $# arguments\"\n";
    script << "echo \"Arguments: $@\"\n\n";
    script << "# Process arguments\n";
    script << "for arg in \"$@\"; do\n";
    script << "    echo \"  - Processing: $arg\"\n";
    script << "done\n\n";
    script << "# Return some data\n";
    script << "echo \"Hostname: $(hostname)\"\n";
    script << "echo \"User: $(whoami)\"\n";
    script << "echo \"Date: $(date +%Y-%m-%d)\"\n";
    script << "\n";
    script << "exit 0\n";

    fs::permissions(script_path,
                    fs::perms::owner_all | fs::perms::group_read | fs::perms::
                        others_read,
                    fs::perm_options::add);
}

void demonstrate_bash_scripts() {
    std::cout << "\n=== 5. RUNNING BASH SCRIPTS ===" << std::endl;

    std::error_code ec;
    fs::path temp_dir = fs::temp_directory_path(ec);
    fs::path script_path = temp_dir / "test_script.sh";

    std::cout << "\n5.1 Creating bash script:" << std::endl;
    std::cout << "   " << script_path << std::endl;

    create_test_bash_script(script_path);

    // Execute script with arguments
    std::cout << "\n5.2 Executing script with arguments:" << std::endl;
    std::string command = "bash " + script_path.string() + " arg1 arg2 arg3";
    std::string output = execute_command(command);
    std::cout << output;

    // Parse script output with string_view
    std::cout << "\n5.3 Parsing script output:" << std::endl;
    std::regex hostname_pattern(R"(Hostname:\s+(\S+))");
    std::regex user_pattern(R"(User:\s+(\S+))");
    std::regex date_pattern(R"(Date:\s+(\S+))");

    std::smatch match;
```

```
    std::string output_str(output);

    if (std::regex_search(output_str, match, hostname_pattern)) {
        std::cout << "   Hostname: " << match[1].str() << std::endl;
    }
    if (std::regex_search(output_str, match, user_pattern)) {
        std::cout << "   User: " << match[1].str() << std::endl;
    }
    if (std::regex_search(output_str, match, date_pattern)) {
        std::cout << "   Date: " << match[1].str() << std::endl;
    }

    // Cleanup
    fs::remove(script_path, ec);

    std::cout << "\n Script execution:" << std::endl;
    std::cout << " •   Use bash explicitly for portability" << std::endl;
    std::cout << " •   Pass arguments safely (avoid injection)" << std::endl;
    std::cout << " •   Parse structured output with regex" << std::endl;
}

// ==================================================================
// SECTION 6: RUNNING PYTHON SCRIPTS
// ==================================================================

void create_test_python_script(const fs::path& script_path) {
    std::ofstream script(script_path);
    script << "#!/usr/bin/env python3\n";
    script << "import sys\n";
    script << "import json\n";
    script << "import platform\n\n";
    script << "def main():\n";
    script << "    # Get system info\n";
    script << "    info = {\n";
    script << "        'python_version': sys.version.split()[0],\n";
    script << "        'platform': platform.system(),\n";
    script << "        'architecture': platform.machine(),\n";
    script << "        'arguments': sys.argv[1:]\n";
    script << "    }\n";
    script << "    \n";
    script << "    # Output as JSON for easy parsing\n";
    script << "    print(json.dumps(info, indent=2))\n\n";
    script << "if __name__ == '__main__':\n";
    script << "    main()\n";

    fs::permissions(script_path,
                    fs::perms::owner_all | fs::perms::group_read | fs::perms::
                        others_read,
                    fs::perm_options::add);
}

void demonstrate_python_scripts() {
    std::cout << "\n=== 6. RUNNING PYTHON SCRIPTS ===" << std::endl;

```

```
    std::error_code ec;
    fs::path temp_dir = fs::temp_directory_path(ec);
    fs::path script_path = temp_dir / "test_script.py";

    std::cout << "\n6.1 Creating Python script:" << std::endl;
    std::cout << "   " << script_path << std::endl;

    create_test_python_script(script_path);

    // Execute Python script
    std::cout << "\n6.2 Executing Python script:" << std::endl;
    std::string command = "python3 " + script_path.string() + " data1 data2";
    auto result = execute_command_with_status(command);

    if (result.success) {
        std::cout << "    Script executed successfully" << std::endl;
        std::cout << "\n6.3 JSON Output:\n" << result.output << std::endl;

        // Parse JSON output with regex (simple approach)
        std::cout << "6.4 Extracting data:" << std::endl;
        std::regex version_pattern("\"python_version\"\\s*:\\s*\"([^\"]+)\"");
        std::regex platform_pattern("\"platform\"\\s*:\\s*\"([^\"]+)\"");
        std::regex arch_pattern("\"architecture\"\\s*:\\s*\"([^\"]+)\"");

        std::smatch match;
        if (std::regex_search(result.output, match, version_pattern)) {
            std::cout << "   Python version: " << match[1].str() << std::endl;
        }
        if (std::regex_search(result.output, match, platform_pattern)) {
            std::cout << "   Platform: " << match[1].str() << std::endl;
        }
        if (std::regex_search(result.output, match, arch_pattern)) {
            std::cout << "   Architecture: " << match[1].str() << std::endl;
        }
    } else {
        std::cout << "    Script failed with code: " << result.return_code <<
            std::endl;
    }

    // Cleanup
    fs::remove(script_path, ec);

    std::cout << "\n Python integration:" << std::endl;
    std::cout << " •   Use JSON for structured data exchange" << std::endl;
    std::cout << " •   Check return codes" << std::endl;
    std::cout << " •   Python for quick prototyping/data processing" << std::
        endl;
}

// ==================================================================
// SECTION 7: PARSING LINUX TOOL OUTPUT
// ==================================================================

struct DiskInfo {
```

```
    std::string filesystem;
    std::string size;
    std::string used;
    std::string available;
    std::string use_percent;
    std::string mounted_on;
};

std::vector<DiskInfo> parse_df_output(std::string_view output) {
    std::vector<DiskInfo> disks;

    size_t line_start = 0;
    size_t line_end = output.find('\n');

    // Skip header
    if (line_end != std::string_view::npos) {
        line_start = line_end + 1;
    }

    while (line_start < output.size()) {
        line_end = output.find('\n', line_start);
        if (line_end == std::string_view::npos) {
            line_end = output.size();
        }

        std::string_view line = output.substr(line_start, line_end -
            line_start);
        if (line.empty()) {
            line_start = line_end + 1;
            continue;
        }

        auto tokens = split_string_view(line, ' ');
        if (tokens.size() >= 6) {
            DiskInfo info;
            info.filesystem = std::string(tokens[0]);
            info.size = std::string(tokens[1]);
            info.used = std::string(tokens[2]);
            info.available = std::string(tokens[3]);
            info.use_percent = std::string(tokens[4]);
            info.mounted_on = std::string(tokens[5]);
            disks.push_back(info);
        }

        line_start = line_end + 1;
    }

    return disks;
}

void demonstrate_linux_tools() {
    std::cout << "\n=== 7. PARSING LINUX TOOL OUTPUT ===" << std::endl;

    // df command
```

```
    std::cout << "\n7.1 Disk usage (df -h):" << std::endl;
    std::string df_output = execute_command("df -h 2>/dev/null | head -10");
    auto disks = parse_df_output(df_output);

    for (const auto& disk : disks) {
        if (disk.filesystem.find("/dev/") == 0) {  // Real devices only
            std::cout << "   " << disk.mounted_on
                      << ": " << disk.used << "/" << disk.size
                      << " (" << disk.use_percent << ")" << std::endl;
        }
    }

    // Memory info
    std::cout << "\n7.2 Memory usage:" << std::endl;
    std::string mem_output = execute_command("free -h | grep Mem");
    std::regex mem_pattern(R"(Mem:\s+(\S+)\s+(\S+)\s+(\S+))");
    std::smatch match;
    if (std::regex_search(mem_output, match, mem_pattern)) {
        std::cout << "   Total: " << match[1].str() << std::endl;
        std::cout << "   Used: " << match[2].str() << std::endl;
        std::cout << "   Free: " << match[3].str() << std::endl;
    }

    // CPU info
    std::cout << "\n7.3 CPU information:" << std::endl;
    std::string cpu_output = execute_command("lscpu 2>/dev/null | grep -E '
        Model name|CPU\\(s\\)|Thread'");

    std::istringstream iss(cpu_output);
    std::string line;
    while (std::getline(iss, line)) {
        if (!line.empty()) {
            std::cout << "   " << line << std::endl;
        }
    }

    std::cout << "\n Linux tools:" << std::endl;
    std::cout << " •   Rich ecosystem of system tools" << std::endl;
    std::cout << " •   Standardized output formats" << std::endl;
    std::cout << " •   Combine with grep/awk for filtering" << std::endl;
}

// ==================================================================
// SECTION 8: FILE STREAMS - ifstream, ofstream, fstream
// ==================================================================

void demonstrate_file_streams() {
    std::cout << "\n=== 8. FILE STREAMS (ifstream, ofstream, fstream) ===" <<
        std::endl;

    std::error_code ec;
    fs::path temp_dir = fs::temp_directory_path(ec);
    fs::path test_file = temp_dir / "stream_test.txt";

```

```
    // 8.1 Writing with ofstream
    std::cout << "\n8.1 Writing with ofstream:" << std::endl;
    {
        std::ofstream ofs(test_file);
        if (!ofs) {
            std::cout << "    Failed to open file for writing" << std::endl;
            return;
        }

        ofs << "Line 1: Hello, World!\n";
        ofs << "Line 2: Modern C++ Streams\n";
        ofs << "Line 3: Numbers: " << 42 << ", " << 3.14159 << "\n";
        ofs << "Line 4: Boolean: " << std::boolalpha << true << "\n";

        std::cout << "    Wrote 4 lines to " << test_file.filename() << std::
            endl;
    }  // File automatically closed when ofs goes out of scope

    // 8.2 Reading with ifstream
    std::cout << "\n8.2 Reading with ifstream (line by line):" << std::endl;
    {
        std::ifstream ifs(test_file);
        if (!ifs) {
            std::cout << "    Failed to open file for reading" << std::endl;
            return;
        }

        std::string line;
        int line_num = 1;
        while (std::getline(ifs, line)) {
            std::cout << "   [" << line_num++ << "] " << line << std::endl;
        }
    }

    // 8.3 Reading entire file
    std::cout << "\n8.3 Reading entire file at once:" << std::endl;
    {
        std::ifstream ifs(test_file);
        std::string content((std::istreambuf_iterator<char>(ifs)),
                            std::istreambuf_iterator<char>());
        std::cout << "   File size: " << content.size() << " bytes" << std::
            endl;
        std::cout << "   First 50 chars: " << content.substr(0, 50) << "..."
            << std::endl;
    }

    // 8.4 Appending to file
    std::cout << "\n8.4 Appending with std::ios::app:" << std::endl;
    {
        std::ofstream ofs(test_file, std::ios::app);  // Append mode
        ofs << "Line 5: Appended line\n";
        std::cout << "    Appended one line" << std::endl;
    }

```

```
    // 8.5 Random access with fstream
    std::cout << "\n8.5 Random access with fstream:" << std::endl;
    {
        std::fstream fs(test_file, std::ios::in | std::ios::out);

        // Seek to position
        fs.seekg(0, std::ios::end);
        std::streampos file_size = fs.tellg();
        std::cout << "   File size: " << file_size << " bytes" << std::endl;

        // Go back to start
        fs.seekg(0, std::ios::beg);

        // Read first line
        std::string first_line;
        std::getline(fs, first_line);
        std::cout << "   First line: " << first_line << std::endl;

        // Current position
        std::cout << "   Current position: " << fs.tellg() << std::endl;
    }

    // 8.6 Stream state checking
    std::cout << "\n8.6 Stream state checking:" << std::endl;
    {
        std::ifstream ifs(test_file);

        std::cout << "   Initial state:" << std::endl;
        std::cout << "      good(): " << ifs.good() << std::endl;
        std::cout << "      eof():  " << ifs.eof() << std::endl;
        std::cout << "      fail(): " << ifs.fail() << std::endl;
        std::cout << "      bad():  " << ifs.bad() << std::endl;

        // Read until EOF
        std::string line;
        while (std::getline(ifs, line)) {
            // Reading...
        }

        std::cout << "   After reading to EOF:" << std::endl;
        std::cout << "      eof():  " << ifs.eof() << std::endl;
        std::cout << "      good(): " << ifs.good() << std::endl;

        // Clear error state
        ifs.clear();
        std::cout << "   After clear():" << std::endl;
        std::cout << "      good(): " << ifs.good() << std::endl;
    }

    // Cleanup
    fs::remove(test_file, ec);

    std::cout << "\n File stream tips:" << std::endl;
    std::cout << " •   RAII - files auto-close when stream destroyed" << std::
```

```
        endl;
    std::cout << " •   Check state with good(), fail(), bad(), eof()" << std::endl;
    std::cout << " •   Use seekg()/tellg() for reading position" << std::endl;
    std::cout << " •   Use seekp()/tellp() for writing position" << std::endl;
}

// ==================================================================
// SECTION 9: STRING STREAMS - ostringstream, istringstream
// ==================================================================

struct LogRecord {
    std::string timestamp;
    std::string level;
    int thread_id;
    std::string message;
};

void demonstrate_string_streams() {
    std::cout << "\n=== 9. STRING STREAMS (ostringstream, istringstream) ===" << std::endl;

    // 9.1 ostringstream - building strings
    std::cout << "\n9.1 Building strings with ostringstream:" << std::endl;
    {
        std::ostringstream oss;
        oss << "System report:\n";
        oss << "  CPU cores: " << 8 << "\n";
        oss << "  Memory: " << 16.5 << " GB\n";
        oss << "  Load: " << std::fixed << std::setprecision(2) << 0.6789 << "\n";

        std::string report = oss.str();
        std::cout << report;
    }

    // 9.2 istringstream - parsing strings
    std::cout << "\n9.2 Parsing with istringstream:" << std::endl;
    {
        std::string data = "2024-01-15 ERROR 12345 Database connection failed";
        std::istringstream iss(data);

        LogRecord record;
        iss >> record.timestamp >> record.level >> record.thread_id;
        std::getline(iss, record.message);  // Rest is message

        // Trim leading space from message
        if (!record.message.empty() && record.message[0] == ' ') {
            record.message = record.message.substr(1);
        }

        std::cout << "   Parsed log record:" << std::endl;
        std::cout << "      Timestamp: " << record.timestamp << std::endl;
```

```
        std::cout << "      Level: " << record.level << std::endl;
        std::cout << "      Thread: " << record.thread_id << std::endl;
        std::cout << "      Message: " << record.message << std::endl;
    }

    // 9.3 Parsing CSV
    std::cout << "\n9.3 Parsing CSV with istringstream:" << std::endl;
    {
        std::string csv = "John,Doe,30,Engineer\nJane,Smith,28,Designer\nBob,Johnson,35,Manager";
        std::istringstream iss(csv);
        std::string line;

        while (std::getline(iss, line)) {
            std::istringstream line_stream(line);
            std::string first, last, age, role;

            std::getline(line_stream, first, ',');
            std::getline(line_stream, last, ',');
            std::getline(line_stream, age, ',');
            std::getline(line_stream, role, ',');

            std::cout << "   " << first << " " << last
                      << " (" << age << ") - " << role << std::endl;
        }
    }

    // 9.4 Number conversion with stringstream
    std::cout << "\n9.4 Type conversions:" << std::endl;
    {
        // String to number
        std::string num_str = "42.5";
        std::istringstream iss(num_str);
        double value;
        iss >> value;
        std::cout << "   String \"" << num_str << "\" -> double: " << value << std::endl;

        // Number to string (alternative to std::to_string with formatting)
        std::ostringstream oss;
        oss << std::fixed << std::setprecision(3) << 3.14159265;
        std::cout << "   double 3.14159265 -> string: \"" << oss.str() << "\"" << std::endl;

        // Hex conversion
        oss.str("");  // Clear
        oss.clear();  // Clear state
        oss << "0x" << std::hex << 255;
        std::cout << "   int 255 -> hex string: \"" << oss.str() << "\"" << std::endl;
    }

    // 9.5 Building complex formatted strings
    std::cout << "\n9.5 Complex string formatting:" << std::endl;
```

```
    {
        std::ostringstream oss;

        auto now = std::chrono::system_clock::now();
        auto now_time_t = std::chrono::system_clock::to_time_t(now);

        oss << "                    \n";
        oss << "          SYSTEM STATUS REPORT          \n";
        oss << "                    \n";
        oss << "  Time:      " << std::put_time(std::localtime(&now_time_t), "%Y-%m-%d %H:%M:%S") << "           \n";
        oss << "  Status:    " << std::left << std::setw(22) << "Running" << "\n";
        oss << "  Uptime:    " << std::right << std::setw(10) << "99.9%" << "          \n";
        oss << "                    \n";

        std::cout << oss.str();
    }

    std::cout << "\n String stream benefits:" << std::endl;
    std::cout << " •   Type-safe string building" << std::endl;
    std::cout << " •   Parsing with >> operator" << std::endl;
    std::cout << " •   Full stream manipulator support" << std::endl;
    std::cout << " •   Reusable: clear() + str(\"\") to reset" << std::endl;
}

// ==================================================================
// SECTION 10: STREAM MANIPULATORS AND FORMATTING
// ==================================================================

void demonstrate_stream_manipulators() {
    std::cout << "\n=== 10. STREAM MANIPULATORS ===" << std::endl;

    // 10.1 Numeric formatting
    std::cout << "\n10.1 Numeric formatting:" << std::endl;
    {
        double pi = 3.14159265358979;
        int num = 255;

        std::cout << "   Default: " << pi << std::endl;
        std::cout << "   Fixed (6 decimals): " << std::fixed << pi << std::endl;
        std::cout << "   Scientific: " << std::scientific << pi << std::endl;
        std::cout << std::defaultfloat;  // Reset to default

        std::cout << "   Precision 3: " << std::setprecision(3) << pi << std::endl;
        std::cout << "   Precision 10: " << std::setprecision(10) << pi << std::endl;

        std::cout << "   Decimal: " << std::dec << num << std::endl;
        std::cout << "   Hex: 0x" << std::hex << num << std::endl;
        std::cout << "   Octal: 0" << std::oct << num << std::endl;
```

```
        std::cout << std::dec;  // Reset to decimal
    }

    // 10.2 Width and alignment
    std::cout << "\n10.2 Width and alignment:" << std::endl;
    {
        std::cout << "   |" << std::setw(15) << "Right" << "|" << std::endl;
        std::cout << "   |" << std::left << std::setw(15) << "Left" << "|" << std::endl;
        std::cout << "   |" << std::internal << std::setw(15) << -123 << "|" << std::endl;
        std::cout << std::right;  // Reset
    }

    // 10.3 Fill character
    std::cout << "\n10.3 Fill character:" << std::endl;
    {
        std::cout << "   " << std::setfill('*') << std::setw(20) << "Title" << std::setfill(' ') << std::endl;
        std::cout << "   " << std::setfill('0') << std::setw(8) << 42 << std::setfill(' ') << std::endl;
        std::cout << "   " << std::setfill('-') << std::setw(30) << "" << std::setfill(' ') << std::endl;
    }

    // 10.4 Boolean formatting
    std::cout << "\n10.4 Boolean formatting:" << std::endl;
    {
        bool flag = true;
        std::cout << "   Default: " << flag << std::endl;
        std::cout << "   Boolalpha: " << std::boolalpha << flag << std::endl;
        std::cout << std::noboolalpha;  // Reset
    }

    // 10.5 Sign formatting
    std::cout << "\n10.5 Sign formatting:" << std::endl;
    {
        int pos = 42, neg = -42;
        std::cout << "   Default: " << pos << ", " << neg << std::endl;
        std::cout << "   Showpos: " << std::showpos << pos << ", " << neg << std::endl;
        std::cout << std::noshowpos;  // Reset
    }

    // 10.6 Table formatting example
    std::cout << "\n10.6 Table formatting:" << std::endl;
    {
        struct Product {
            std::string name;
            double price;
            int quantity;
        };

        std::vector<Product> products = {
```

```
            {"Widget", 29.99, 15},
            {"Gadget", 149.50, 8},
            {"Doohickey", 9.95, 42}
        };

        std::cout << "                                " << std::endl;
        std::cout << "     " << std::left << std::setw(15) << "Product"
                  << "   " << std::right << std::setw(8) << "Price"
                  << "   " << std::setw(8) << "Qty" << "  " << std::endl;
        std::cout << "                                " << std::endl;

        for (const auto& p : products) {
            std::cout << "     " << std::left << std::setw(15) << p.name
                      << "   $" << std::right << std::setw(7) << std::fixed
                      << std::setprecision(2) << p.price
                      << "   " << std::setw(8) << p.quantity << "  " << std::endl;
        }

        std::cout << "                                " << std::endl;
    }

    std::cout << "\n Manipulator categories:" << std::endl;
    std::cout << " •   Numeric: fixed, scientific, precision, hex, oct, dec" << std::endl;
    std::cout << " •   Alignment: setw, left, right, internal, setfill" << std::endl;
    std::cout << " •   Boolean: boolalpha, noboolalpha" << std::endl;
    std::cout << " •   Sign: showpos, noshowpos, showbase" << std::endl;
}

// ==================================================================
// SECTION 11: BINARY I/O
// ==================================================================

struct BinaryRecord {
    int id;
    double value;
    char name[32];
};

void demonstrate_binary_io() {
    std::cout << "\n=== 11. BINARY I/O ===" << std::endl;

    std::error_code ec;
    fs::path temp_dir = fs::temp_directory_path(ec);
    fs::path binary_file = temp_dir / "binary_data.bin";

    // 11.1 Writing binary data
    std::cout << "\n11.1 Writing binary data:" << std::endl;
    {
        std::ofstream ofs(binary_file, std::ios::binary);

        BinaryRecord records[] = {
```

```
            {1, 3.14159, "Record One"},
            {2, 2.71828, "Record Two"},
            {3, 1.41421, "Record Three"}
        };

        for (const auto& record : records) {
            ofs.write(reinterpret_cast<const char*>(&record), sizeof(
                BinaryRecord));
        }

        std::cout << "     Wrote " << sizeof(records) / sizeof(BinaryRecord)
                  << " binary records (" << sizeof(records) << " bytes)" <<
                      std::endl;
    }

    // 11.2 Reading binary data
    std::cout << "\n11.2 Reading binary data:" << std::endl;
    {
        std::ifstream ifs(binary_file, std::ios::binary);

        // Get file size
        ifs.seekg(0, std::ios::end);
        std::streamsize size = ifs.tellg();
        ifs.seekg(0, std::ios::beg);

        std::cout << "   File size: " << size << " bytes" << std::endl;
        std::cout << "   Records: " << size / sizeof(BinaryRecord) << std::
            endl;

        BinaryRecord record;
        int count = 0;
        while (ifs.read(reinterpret_cast<char*>(&record), sizeof(BinaryRecord)
            )) {
            std::cout << "   Record " << ++count << ": "
                      << "id=" << record.id << ", "
                      << "value=" << record.value << ", "
                      << "name=\"" << record.name << "\"" << std::endl;
        }
    }

    // 11.3 Writing POD types
    std::cout << "\n11.3 Writing POD (Plain Old Data) types:" << std::endl;
    {
        fs::path pod_file = temp_dir / "pod_data.bin";

        std::ofstream ofs(pod_file, std::ios::binary);

        int integer = 42;
        double floating = 3.14159;
        char character = 'X';

        ofs.write(reinterpret_cast<const char*>(&integer), sizeof(integer));
        ofs.write(reinterpret_cast<const char*>(&floating), sizeof(floating));
        ofs.write(reinterpret_cast<const char*>(&character), sizeof(character)
```

```
            );

        ofs.close();

        // Read back
        std::ifstream ifs(pod_file, std::ios::binary);

        int read_int;
        double read_double;
        char read_char;

        ifs.read(reinterpret_cast<char*>(&read_int), sizeof(read_int));
        ifs.read(reinterpret_cast<char*>(&read_double), sizeof(read_double));
        ifs.read(reinterpret_cast<char*>(&read_char), sizeof(read_char));

        std::cout << "   Read back: " << read_int << ", "
                  << read_double << ", " << read_char << std::endl;

        fs::remove(pod_file, ec);
    }

    // Cleanup
    fs::remove(binary_file, ec);

    std::cout << "\n Binary I/O notes:" << std::endl;
    std::cout << " •   Use std::ios::binary flag" << std::endl;
    std::cout << " •   Only for POD types (no pointers, virtual functions)" << std::endl;
    std::cout << " •   Platform-dependent (endianness, padding)" << std::endl;
    std::cout << " •   Use serialization libraries for portability" << std::endl;
}

// ==================================================================
// SECTION 12: CUSTOM STREAM OPERATORS
// ==================================================================

struct Point {
    double x, y, z;
};

// Custom output operator
std::ostream& operator<<(std::ostream& os, const Point& p) {
    os << "Point(" << p.x << ", " << p.y << ", " << p.z << ")";
    return os;
}

// Custom input operator
std::istream& operator>>(std::istream& is, Point& p) {
    char paren, comma1, comma2;
    std::string word;

    // Expected format: Point(1.0, 2.0, 3.0)
    is >> word >> paren >> p.x >> comma1 >> p.y >> comma2 >> p.z >> paren;
```

```

    return is;
}

void demonstrate_custom_operators() {
    std::cout << "\n=== 12. CUSTOM STREAM OPERATORS ===" << std::endl;

    // 12.1 Custom output
    std::cout << "\n12.1 Custom output operator:" << std::endl;
    {
        Point p1{1.5, 2.7, 3.9};
        std::cout << "   " << p1 << std::endl;

        // Works with string streams too
        std::ostringstream oss;
        oss << "Point data: " << p1;
        std::cout << "   Stringified: " << oss.str() << std::endl;
    }

    // 12.2 Custom input
    std::cout << "\n12.2 Custom input operator:" << std::endl;
    {
        std::string input = "Point(4.5, 5.6, 6.7)";
        std::istringstream iss(input);

        Point p2;
        iss >> p2;

        std::cout << "   Parsed from string: " << p2 << std::endl;
    }

    // 12.3 Chaining
    std::cout << "\n12.3 Operator chaining:" << std::endl;
    {
        Point p1{1.0, 2.0, 3.0};
        Point p2{4.0, 5.0, 6.0};

        std::cout << "   Multiple points: " << p1 << " and " << p2 << std::
            endl;
    }

    std::cout << "\ n Custom operator benefits:" << std::endl;
    std::cout << " •   Natural syntax: cout << myObject" << std::endl;
    std::cout << " •   Works with all streams (file, string, cout)" << std::
        endl;
    std::cout << " •   Chainable for multiple operations" << std::endl;
    std::cout << " •   Consistent with standard library" << std::endl;
}

// ==================================================================
// SECTION 13: MUTEXES AND LOCK_GUARD
// ==================================================================

// Shared resource with thread-safe access
```

```cpp
class SharedCounter {
private:
    int counter = 0;
    mutable std::mutex mtx;  // mutable allows locking in const methods

public:
    void increment() {
        std::lock_guard<std::mutex> lock(mtx);  // RAII lock
        ++counter;
    }  // Automatic unlock when lock goes out of scope

    void add(int value) {
        std::lock_guard<std::mutex> lock(mtx);
        counter += value;
    }

    int get() const {
        std::lock_guard<std::mutex> lock(mtx);
        return counter;
    }
};

// File logger with mutex protection
class ThreadSafeLogger {
private:
    std::ofstream log_file;
    std::mutex file_mutex;

public:
    ThreadSafeLogger(const std::string& filename) : log_file(filename, std::ios::app) {}

    void log(const std::string& message) {
        std::lock_guard<std::mutex> lock(file_mutex);

        auto now = std::chrono::system_clock::now();
        auto now_time_t = std::chrono::system_clock::to_time_t(now);

        log_file << "[" << std::put_time(std::localtime(&now_time_t), "%H:%M:%S")
                 << "] " << message << std::endl;
    }

    ~ThreadSafeLogger() {
        if (log_file.is_open()) {
            log_file.close();
        }
    }
};

void demonstrate_mutex_and_lock_guard() {
    std::cout << "\n=== 13. MUTEXES AND LOCK_GUARD ===" << std::endl;

    // 13.1 Basic lock_guard usage
```

```
    std::cout << "\n13.1 std::lock_guard for automatic RAII locking:" << std::endl;
    {
        SharedCounter counter;
        std::vector<std::thread> threads;

        // Launch 10 threads that increment counter
        for (int i = 0; i < 10; ++i) {
            threads.emplace_back([&counter]() {
                for (int j = 0; j < 100; ++j) {
                    counter.increment();
                }
            });
        }

        // Wait for all threads
        for (auto& t : threads) {
            t.join();
        }

        std::cout << "   Final counter value: " << counter.get() << std::endl;
        std::cout << "   Expected: 1000" << std::endl;
        std::cout << "    Thread-safe with lock_guard!" << std::endl;
    }

    // 13.2 Thread-safe file logging
    std::cout << "\n13.2 Thread-safe file logging:" << std::endl;
    {
        std::error_code ec;
        fs::path temp_dir = fs::temp_directory_path(ec);
        fs::path log_file = temp_dir / "threadsafe_log.txt";

        ThreadSafeLogger logger(log_file.string());
        std::vector<std::thread> threads;

        // Multiple threads writing to log
        for (int i = 0; i < 5; ++i) {
            threads.emplace_back([&logger, i]() {
                for (int j = 0; j < 3; ++j) {
                    std::ostringstream oss;
                    oss << "Thread " << i << " message " << j;
                    logger.log(oss.str());
                    std::this_thread::sleep_for(std::chrono::milliseconds(10));
                }
            });
        }

        for (auto& t : threads) {
            t.join();
        }

        std::cout << "    15 log entries written safely" << std::endl;
        std::cout << "   Log file: " << log_file.filename() << std::endl;
```

```

        // Read and display log
        std::ifstream ifs(log_file);
        std::string line;
        int count = 0;
        std::cout << "   First 5 entries:" << std::endl;
        while (std::getline(ifs, line) && count++ < 5) {
            std::cout << "      " << line << std::endl;
        }

        fs::remove(log_file, ec);
    }

    std::cout << "\n lock_guard benefits:" << std::endl;
    std::cout << " •   RAII - automatic unlock on scope exit" << std::endl;
    std::cout << " •   Exception-safe" << std::endl;
    std::cout << " •   Simple and efficient" << std::endl;
    std::cout << " •   Cannot forget to unlock" << std::endl;
}

// ==================================================================
// SECTION 14: UNIQUE_LOCK AND ADVANCED LOCKING
// ==================================================================

class BankAccount {
private:
    double balance = 0.0;
    mutable std::mutex mtx;
    std::condition_variable cv;

public:
    BankAccount(double initial) : balance(initial) {}

    bool withdraw(double amount) {
        std::unique_lock<std::mutex> lock(mtx);

        if (balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

    void deposit(double amount) {
        std::unique_lock<std::mutex> lock(mtx);
        balance += amount;
        cv.notify_all();  // Wake up waiting threads
    }

    // Wait until sufficient funds available
    bool wait_and_withdraw(double amount, std::chrono::milliseconds timeout) {
        std::unique_lock<std::mutex> lock(mtx);

        // Wait with timeout
```

```
        if (!cv.wait_for(lock, timeout, [this, amount]() {
            return balance >= amount;
        })) {
            return false;  // Timeout
        }

        balance -= amount;
        return true;
    }

    double get_balance() const {
        std::unique_lock<std::mutex> lock(mtx);
        return balance;
    }
};

// Reader-writer lock example
class SharedData {
private:
    std::map<std::string, int> data;
    mutable std::shared_mutex rw_mutex;

public:
    // Multiple readers can access simultaneously
    int read(const std::string& key) const {
        std::shared_lock<std::shared_mutex> lock(rw_mutex);
        auto it = data.find(key);
        return (it != data.end()) ? it->second : 0;
    }

    // Only one writer at a time
    void write(const std::string& key, int value) {
        std::unique_lock<std::shared_mutex> lock(rw_mutex);
        data[key] = value;
    }

    size_t size() const {
        std::shared_lock<std::shared_mutex> lock(rw_mutex);
        return data.size();
    }
};

void demonstrate_unique_lock() {
    std::cout << "\n=== 14. UNIQUE_LOCK AND ADVANCED LOCKING ===" << std::endl
        ;

    // 14.1 unique_lock - more flexible than lock_guard
    std::cout << "\n14.1 std::unique_lock features:" << std::endl;
    {
        std::mutex mtx;

        std::cout << " •   Can be locked/unlocked multiple times:" << std::
            endl;
        {
```

```
            std::unique_lock<std::mutex> lock(mtx);
            std::cout << "      Locked" << std::endl;

            lock.unlock();  // Explicit unlock
            std::cout << "      Unlocked (doing non-critical work)" << std::endl;

            lock.lock();  // Relock
            std::cout << "      Locked again" << std::endl;
        }

        std::cout << " •   Deferred locking:" << std::endl;
        {
            std::unique_lock<std::mutex> lock(mtx, std::defer_lock);  // Don't lock yet
            std::cout << "      Created unlocked" << std::endl;

            lock.lock();  // Lock when needed
            std::cout << "      Now locked" << std::endl;
        }
    }

    // 14.2 Condition variable with unique_lock
    std::cout << "\n14.2 Condition variable (wait with timeout):" << std::endl;
    {
        BankAccount account(100.0);

        std::cout << "   Initial balance: $" << account.get_balance() << std::endl;

        // Try to withdraw more than available
        std::thread withdrawal_thread([&account]() {
            std::cout << "   Attempting to withdraw $150 (waiting up to 500ms)..." << std::endl;
            bool success = account.wait_and_withdraw(150.0, std::chrono::milliseconds(500));

            if (success) {
                std::cout << "     Withdrawal successful" << std::endl;
            } else {
                std::cout << "      Timeout - insufficient funds" << std::endl;
            }
        });

        std::this_thread::sleep_for(std::chrono::milliseconds(200));

        // Deposit money to unblock
        std::cout << "   Depositing $75..." << std::endl;
        account.deposit(75.0);

        withdrawal_thread.join();
        std::cout << "   Final balance: $" << account.get_balance() << std::
```

```
            endl;
    }

    // 14.3 Shared mutex for reader-writer locks
    std::cout << "\n14.3 std::shared_mutex (reader-writer lock):" << std::endl;
    {
        SharedData shared;
        std::atomic<int> read_count{0};
        std::atomic<int> write_count{0};

        std::vector<std::thread> threads;

        // 8 readers
        for (int i = 0; i < 8; ++i) {
            threads.emplace_back([&shared, &read_count, i]() {
                for (int j = 0; j < 10; ++j) {
                    shared.read("key" + std::to_string(i % 3));
                    read_count++;
                    std::this_thread::sleep_for(std::chrono::microseconds(100));
                }
            });
        }

        // 2 writers
        for (int i = 0; i < 2; ++i) {
            threads.emplace_back([&shared, &write_count, i]() {
                for (int j = 0; j < 5; ++j) {
                    shared.write("key" + std::to_string(j), i * 100 + j);
                    write_count++;
                    std::this_thread::sleep_for(std::chrono::milliseconds(1));
                }
            });
        }

        for (auto& t : threads) {
            t.join();
        }

        std::cout << "   Completed " << read_count << " reads" << std::endl;
        std::cout << "   Completed " << write_count << " writes" << std::endl;
        std::cout << "   Final data size: " << shared.size() << std::endl;
        std::cout << "    Multiple readers can run concurrently!" << std::endl;
    }

    std::cout << "\n unique_lock vs lock_guard:" << std::endl;
    std::cout << "   lock_guard:  Simple, RAII, no unlock() method" << std::endl;
    std::cout << "   unique_lock: Flexible, can unlock/relock, works with condition_variable" << std::endl;
    std::cout << "\n shared_mutex:" << std::endl;
    std::cout << " •   shared_lock:  Multiple readers simultaneously" << std::
```

```
        endl;
    std::cout << " •   unique_lock:  Exclusive writer access" << std::endl;
    std::cout << " •   Perfect for read-heavy workloads" << std::endl;
}

// ================================================================
// SECTION 15: FILE LOCKING (OS-LEVEL)
// ================================================================

#ifdef __linux__
#include <sys/file.h>  // For flock
#include <fcntl.h>
#include <unistd.h>

class FileLock {
private:
    int fd = -1;
    std::string filename;

public:
    FileLock(const std::string& file) : filename(file) {
        fd = open(filename.c_str(), O_RDWR | O_CREAT, 0666);
        if (fd == -1) {
            throw std::runtime_error("Failed to open file for locking");
        }
    }

    bool try_lock() {
        return flock(fd, LOCK_EX | LOCK_NB) == 0;  // Non-blocking exclusive
            lock
    }

    void lock() {
        if (flock(fd, LOCK_EX) != 0) {  // Blocking exclusive lock
            throw std::runtime_error("Failed to acquire lock");
        }
    }

    void unlock() {
        flock(fd, LOCK_UN);
    }

    ~FileLock() {
        if (fd != -1) {
            unlock();
            close(fd);
        }
    }
};
#endif

void demonstrate_file_locking() {
    std::cout << "\n=== 15. FILE LOCKING (OS-LEVEL) ===" << std::endl;

```

```
#ifdef __linux__
    std::cout << "\n15.1 Advisory file locking with flock():" << std::endl;
    {
        std::error_code ec;
        fs::path temp_dir = fs::temp_directory_path(ec);
        fs::path lock_file = temp_dir / "test.lock";

        try {
            FileLock lock1(lock_file.string());

            std::cout << "   Thread 1: Acquiring lock..." << std::endl;
            lock1.lock();
            std::cout << "   Thread 1:  Lock acquired" << std::endl;

            // Simulate work
            std::this_thread::sleep_for(std::chrono::milliseconds(100));

            // Try to acquire same lock from another thread
            std::thread other_thread([&lock_file]() {
                try {
                    FileLock lock2(lock_file.string());
                    std::cout << "   Thread 2: Trying non-blocking lock..." << std::endl;

                    if (lock2.try_lock()) {
                        std::cout << "   Thread 2:  Lock acquired" << std::endl;
                        lock2.unlock();
                    } else {
                        std::cout << "   Thread 2:   Lock held by another process" << std::endl;
                    }
                } catch (const std::exception& e) {
                    std::cout << "   Thread 2: Error - " << e.what() << std::endl;
                }
            });

            other_thread.join();

            std::cout << "   Thread 1: Releasing lock..." << std::endl;
            lock1.unlock();

        } catch (const std::exception& e) {
            std::cout << "   Error: " << e.what() << std::endl;
        }

        fs::remove(lock_file, ec);
    }

    std::cout << "\n15.2 Preventing concurrent file access:" << std::endl;
    {
        std::error_code ec;
        fs::path temp_dir = fs::temp_directory_path(ec);
```

```
        fs::path shared_file = temp_dir / "shared_data.txt";
        fs::path lock_file_path = temp_dir / "shared_data.txt.lock";

        // Writer with lock
        auto writer = [&shared_file, &lock_file_path](int id) {
            try {
                FileLock lock(lock_file_path.string());
                lock.lock();

                std::cout << "   Writer " << id << ": Writing..." << std::endl;
                std::ofstream ofs(shared_file, std::ios::app);
                ofs << "Line from writer " << id << "\n";
                ofs.close();

                std::this_thread::sleep_for(std::chrono::milliseconds(50));
                lock.unlock();
                std::cout << "   Writer " << id << ": Done" << std::endl;
            } catch (const std::exception& e) {
                std::cout << "   Writer " << id << ": Error - " << e.what() << std::endl;
            }
        };

        std::thread w1(writer, 1);
        std::thread w2(writer, 2);
        std::thread w3(writer, 3);

        w1.join();
        w2.join();
        w3.join();

        // Read result
        std::ifstream ifs(shared_file);
        std::string line;
        int line_count = 0;
        while (std::getline(ifs, line)) {
            line_count++;
        }

        std::cout << "    All writers completed safely" << std::endl;
        std::cout << "   Total lines written: " << line_count << std::endl;

        fs::remove(shared_file, ec);
        fs::remove(lock_file_path, ec);
    }

    std::cout << "\n File locking types:" << std::endl;
    std::cout << " •   Advisory locks: Processes must cooperate (flock)" << std::endl;
    std::cout << " •   Mandatory locks: Enforced by kernel (rare)" << std::endl;
    std::cout << " •   LOCK_EX: Exclusive lock (writer)" << std::endl;
    std::cout << " •   LOCK_SH: Shared lock (reader)" << std::endl;
```

```
    std::cout << " •   LOCK_NB: Non-blocking flag" << std::endl;

#else
    std::cout << "\n      File locking examples require Linux (flock)" << std
        ::endl;
    std::cout << "   Windows equivalent: LockFileEx()" << std::endl;
    std::cout << "   Cross-platform: Use Boost.Interprocess" << std::endl;
#endif
}

// ==================================================================
// SECTION 16: RACE CONDITIONS - PROBLEM AND SOLUTION
// ==================================================================

// Simulated device hotplug manager - WITHOUT proper synchronization (BROKEN!)
class BrokenDeviceManager {
private:
    std::map<std::string, std::string> devices;  // device_id -> status
    int device_count = 0;

public:
    // Called by multiple threads - NO MUTEX!
    void add_device(const std::string& device_id) {
        // RACE CONDITION: Multiple threads can read/write simultaneously
        if (devices.find(device_id) == devices.end()) {
            std::this_thread::sleep_for(std::chrono::microseconds(100));  //
                Simulate processing
            devices[device_id] = "active";
            device_count++;
        }
    }

    void remove_device(const std::string& device_id) {
        // RACE CONDITION: Map modification without synchronization
        if (devices.find(device_id) != devices.end()) {
            std::this_thread::sleep_for(std::chrono::microseconds(100));
            devices.erase(device_id);
            device_count--;
        }
    }

    void process_script_output(const std::string& script_output) {
        // RACE CONDITION: Reading and updating shared state
        std::istringstream iss(script_output);
        std::string line;
        while (std::getline(iss, line)) {
            if (line.find("ADD:") == 0) {
                std::string device = line.substr(4);
                devices[device] = "pending";
            }
        }
    }

    int get_count() const {
```

```
        return device_count;  // RACE CONDITION: Reading without lock
    }

    std::map<std::string, std::string> get_devices() const {
        return devices;  // RACE CONDITION: Copying map without lock
    }
};

// Fixed version - WITH proper synchronization
class SafeDeviceManager {
private:
    std::map<std::string, std::string> devices;
    int device_count = 0;
    mutable std::mutex mtx;  // Protects all shared state
    std::ofstream log_file;

public:
    SafeDeviceManager() : log_file("/tmp/device_manager.log", std::ios::app)
        {}

    void add_device(const std::string& device_id) {
        std::lock_guard<std::mutex> lock(mtx);  // FIXED: Automatic locking

        if (devices.find(device_id) == devices.end()) {
            std::this_thread::sleep_for(std::chrono::microseconds(100));
            devices[device_id] = "active";
            device_count++;

            if (log_file.is_open()) {
                log_file << "[ADD] " << device_id << std::endl;
            }
        }
    }

    void remove_device(const std::string& device_id) {
        std::lock_guard<std::mutex> lock(mtx);  // FIXED: Protected

        if (devices.find(device_id) != devices.end()) {
            std::this_thread::sleep_for(std::chrono::microseconds(100));
            devices.erase(device_id);
            device_count--;

            if (log_file.is_open()) {
                log_file << "[REMOVE] " << device_id << std::endl;
            }
        }
    }

    void process_script_output(const std::string& script_output) {
        std::lock_guard<std::mutex> lock(mtx);  // FIXED: Protected

        std::istringstream iss(script_output);
        std::string line;
        while (std::getline(iss, line)) {
```

```
            if (line.find("ADD:") == 0) {
                std::string device = line.substr(4);
                devices[device] = "pending";
            }
        }
    }

    int get_count() const {
        std::lock_guard<std::mutex> lock(mtx);  // FIXED: Protected read
        return device_count;
    }

    std::map<std::string, std::string> get_devices() const {
        std::lock_guard<std::mutex> lock(mtx);  // FIXED: Protected copy
        return devices;
    }

    ~SafeDeviceManager() {
        if (log_file.is_open()) {
            log_file.close();
        }
    }
};

// Simulate hotplug events by calling external scripts
void simulate_hotplug_script(int event_id) {
    // Create a temporary script that simulates device detection
    std::error_code ec;
    fs::path temp_dir = fs::temp_directory_path(ec);
    fs::path script_path = temp_dir / ("hotplug_" + std::to_string(event_id) +
        ".sh");

    std::ofstream script(script_path);
    script << "#!/bin/bash\n";
    script << "echo 'ADD:device_" << event_id << "'\n";
    script << "sleep 0.01\n";
    script << "echo 'STATUS:online'\n";
    script.close();

    fs::permissions(script_path, fs::perms::owner_all, fs::perm_options::add,
        ec);
}

void demonstrate_race_condition() {
    std::cout << "\n=== 16. RACE CONDITIONS - PROBLEM AND SOLUTION ===" << std
        ::endl;

    // 16.1 Demonstrate the BROKEN version
    std::cout << "\n16.1  BROKEN: Race condition without mutex:" << std::endl
        ;
    {
        BrokenDeviceManager broken_mgr;
        std::vector<std::thread> threads;

```

```
        std::cout << "   Launching 20 threads to add devices..." << std::endl;

        // Multiple threads adding same devices
        for (int i = 0; i < 20; ++i) {
            threads.emplace_back([&broken_mgr, i]() {
                for (int j = 0; j < 10; ++j) {
                    std::string device_id = "dev_" + std::to_string(j);
                    broken_mgr.add_device(device_id);
                }
            });
        }

        for (auto& t : threads) {
            t.join();
        }

        int final_count = broken_mgr.get_count();
        auto devices = broken_mgr.get_devices();

        std::cout << "   Expected: 10 unique devices" << std::endl;
        std::cout << "   Device count variable: " << final_count << std::endl;
        std::cout << "   Actual devices in map: " << devices.size() << std::
            endl;

        if (final_count != 10 || devices.size() != 10) {
            std::cout << "      RACE CONDITION DETECTED!" << std::endl;
            std::cout << "      Count mismatch due to concurrent access" <<
                std::endl;
        } else {
            std::cout << "      (Race condition may not always show - timing
                dependent)" << std::endl;
        }
    }

    // 16.2 Demonstrate the FIXED version
    std::cout << "\n16.2  FIXED: With mutex protection:" << std::endl;
    {
        SafeDeviceManager safe_mgr;
        std::vector<std::thread> threads;

        std::cout << "   Launching 20 threads to add devices..." << std::endl;

        for (int i = 0; i < 20; ++i) {
            threads.emplace_back([&safe_mgr, i]() {
                for (int j = 0; j < 10; ++j) {
                    std::string device_id = "dev_" + std::to_string(j);
                    safe_mgr.add_device(device_id);
                }
            });
        }

        for (auto& t : threads) {
            t.join();
        }
```

```

        int final_count = safe_mgr.get_count();
        auto devices = safe_mgr.get_devices();

        std::cout << "   Expected: 10 unique devices" << std::endl;
        std::cout << "   Device count: " << final_count << std::endl;
        std::cout << "   Devices in map: " << devices.size() << std::endl;

        if (final_count == 10 && devices.size() == 10) {
            std::cout << "    CORRECT: Mutex prevented race condition!" << std::endl;
        }
    }

    // 16.3 Real-world scenario: File updates from multiple sources
    std::cout << "\n16.3 Race condition with file I/O and scripts:" << std::endl;
    {
        std::error_code ec;
        fs::path temp_dir = fs::temp_directory_path(ec);
        fs::path shared_file = temp_dir / "device_registry.txt";

        // BROKEN: Multiple threads writing to file without coordination
        std::cout << "    Without file locking:" << std::endl;
        {
            // Clear file
            std::ofstream(shared_file).close();

            std::vector<std::thread> writers;

            for (int i = 0; i < 5; ++i) {
                writers.emplace_back([&shared_file, i]() {
                    for (int j = 0; j < 3; ++j) {
                        // RACE CONDITION: Multiple threads writing simultaneously
                        std::ofstream ofs(shared_file, std::ios::app);
                        ofs << "Thread_" << i << "_entry_" << j << "\n";
                        ofs.close();
                        std::this_thread::sleep_for(std::chrono::microseconds(100));
                    }
                });
            }

            for (auto& t : writers) {
                t.join();
            }

            // Count lines
            std::ifstream ifs(shared_file);
            int line_count = 0;
            std::string line;
            while (std::getline(ifs, line)) {
                if (!line.empty()) line_count++;
```

```
            }

            std::cout << "      Expected 15 lines, got: " << line_count << std::endl;
            if (line_count != 15) {
                std::cout << "        Some writes may have been lost!" << std::endl;
            }
        }

        // FIXED: With mutex protection
        std::cout << "     With mutex protection:" << std::endl;
        {
            // Clear file
            std::ofstream(shared_file).close();

            std::mutex file_mtx;
            std::vector<std::thread> writers;

            for (int i = 0; i < 5; ++i) {
                writers.emplace_back([&shared_file, &file_mtx, i]() {
                    for (int j = 0; j < 3; ++j) {
                        std::lock_guard<std::mutex> lock(file_mtx);  // FIXED!
                        std::ofstream ofs(shared_file, std::ios::app);
                        ofs << "Thread_" << i << "_entry_" << j << "\n";
                        ofs.close();
                        std::this_thread::sleep_for(std::chrono::microseconds(100));
                    }
                });
            }

            for (auto& t : writers) {
                t.join();
            }

            // Count lines
            std::ifstream ifs(shared_file);
            int line_count = 0;
            std::string line;
            while (std::getline(ifs, line)) {
                if (!line.empty()) line_count++;
            }

            std::cout << "      Expected 15 lines, got: " << line_count << std::endl;
            if (line_count == 15) {
                std::cout << "        All writes preserved!" << std::endl;
            }
        }

        fs::remove(shared_file, ec);
    }

```

```
    // 16.4 Simulating hotplug events with script execution
    std::cout << "\n16.4 Hotplug simulation (scripts + threads):" << std::endl;
    {
        SafeDeviceManager mgr;
        std::vector<std::thread> hotplug_threads;

        std::cout << "   Simulating 3 concurrent hotplug events..." << std::endl;

        for (int i = 0; i < 3; ++i) {
            hotplug_threads.emplace_back([&mgr, i]() {
                // Simulate calling udev script or hotplug handler
                std::string script_output = "ADD:usb_device_" + std::to_string(i) + "\n";
                script_output += "ADD:usb_port_" + std::to_string(i) + "\n";

                mgr.process_script_output(script_output);

                // Simulate device becoming active
                std::this_thread::sleep_for(std::chrono::milliseconds(10));
                mgr.add_device("usb_device_" + std::to_string(i));
                mgr.add_device("usb_port_" + std::to_string(i));
            });
        }

        for (auto& t : hotplug_threads) {
            t.join();
        }

        auto devices = mgr.get_devices();
        std::cout << "   Total devices registered: " << devices.size() << std::endl;
        std::cout << "   Devices:" << std::endl;
        for (const auto& [id, status] : devices) {
            std::cout << "      " << id << " -> " << status << std::endl;
        }
        std::cout << "    All hotplug events handled safely" << std::endl;
    }

    std::cout << "\n Common race condition scenarios:" << std::endl;
    std::cout << "   1. Multiple threads modifying shared container" << std::endl;
    std::cout << "   2. Check-then-act pattern without atomicity" << std::endl;
    std::cout << "   3. Reading and writing without synchronization" << std::endl;
    std::cout << "   4. File I/O from multiple threads/processes" << std::endl;
    std::cout << "   5. Scripts updating shared resources" << std::endl;

    std::cout << "\n Solutions:" << std::endl;
    std::cout << " •   Use std::mutex with lock_guard/unique_lock" << std::endl;
```

```
    std::cout << " •   Use std::atomic for simple counters" << std::endl;
    std::cout << " •   Use file locks for inter-process sync" << std::endl;
    std::cout << " •   Protect ALL access to shared data" << std::endl;
    std::cout << " •   Keep critical sections small" << std::endl;
}

// ==================================================================
// SECTION 17: SECURITY CONSIDERATIONS
// ==================================================================

void explain_security_considerations() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "SECURITY CONSIDERATIONS:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\ n CRITICAL SECURITY ISSUES:" << std::endl;

    std::cout << "\n1. COMMAND INJECTION:" << std::endl;
    std::cout << "     BAD:  system(\"ls \" + user_input);" << std::endl;
    std::cout << "     GOOD: Validate and sanitize input first" << std::endl;
    std::cout << "     GOOD: Use execve() family for direct execution" << std
        ::endl;

    std::cout << "\n2. SHELL METACHARACTERS:" << std::endl;
    std::cout << "   Dangerous: ; | & $ ` \\ \" ' < > ( ) { } [ ] ! #" << std
        ::endl;
    std::cout << "   User input: \"file.txt; rm -rf /\"" << std::endl;
    std::cout << "   Result: DISASTER!" << std::endl;

    std::cout << "\n3. PATH TRAVERSAL:" << std::endl;
    std::cout << "     BAD:  open(\"/data/\" + user_input)" << std::endl;
    std::cout << "   User: \"../../etc/passwd\"" << std::endl;
    std::cout << "     GOOD: Validate paths, use fs::canonical()" << std::endl
        ;

    std::cout << "\n4. ENVIRONMENT VARIABLES:" << std::endl;
    std::cout << "   $PATH, $LD_PRELOAD can be hijacked" << std::endl;
    std::cout << "     Use absolute paths: /usr/bin/python3" << std::endl;
    std::cout << "     Don't trust $PATH from external sources" << std::endl;

    std::cout << "\ n BEST PRACTICES:" << std::endl;
    std::cout << "\n1. INPUT VALIDATION:" << std::endl;
    std::cout << " •   Whitelist allowed characters" << std::endl;
    std::cout << " •   Use regex to validate format" << std::endl;
    std::cout << " •   Reject anything suspicious" << std::endl;

    std::cout << "\n2. AVOID system():" << std::endl;
    std::cout << " •   Use popen() for output capture" << std::endl;
    std::cout << " •   Use fork()+execve() for full control" << std::endl;
    std::cout << " •   Never pass user input directly to shell" << std::endl;

    std::cout << "\n3. LEAST PRIVILEGE:" << std::endl;
    std::cout << " •   Run with minimum necessary permissions" << std::endl;
    std::cout << " •   Drop privileges after initialization" << std::endl;
```

```
    std::cout << " •   Use separate user accounts" << std::endl;

    std::cout << "\n4. ERROR HANDLING:" << std::endl;
    std::cout << " •   Check all return codes" << std::endl;
    std::cout << " •   Don't expose system details in errors" << std::endl;
    std::cout << " •   Log security events" << std::endl;
}

// ==================================================================
// SECTION 18: CROSS-PLATFORM CONSIDERATIONS
// ==================================================================

void explain_cross_platform() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "CROSS-PLATFORM CONSIDERATIONS:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\ n LINUX vs   WINDOWS:" << std::endl;

    std::cout << "\n1. COMMAND EXECUTION:" << std::endl;
    std::cout << "   Linux:   popen(\"/bin/ls\", \"r\")" << std::endl;
    std::cout << "   Windows: _popen(\"dir\", \"r\")" << std::endl;
    std::cout << "   Windows: CreateProcess() for full control" << std::endl;

    std::cout << "\n2. PATH SEPARATORS:" << std::endl;
    std::cout << "   Linux:   /home/user/file.txt" << std::endl;
    std::cout << "   Windows: C:\\Users\\user\\file.txt" << std::endl;
    std::cout << "     Use fs::path - handles both!" << std::endl;

    std::cout << "\n3. LINE ENDINGS:" << std::endl;
    std::cout << "   Linux:   \\n (LF)" << std::endl;
    std::cout << "   Windows: \\r\\n (CRLF)" << std::endl;
    std::cout << "     Open files in text mode for conversion" << std::endl;

    std::cout << "\n4. SYSTEM TOOLS:" << std::endl;
    std::cout << "   Linux:   ps, df, grep, awk, sed" << std::endl;
    std::cout << "   Windows: tasklist, wmic, findstr, PowerShell" << std::
        endl;
    std::cout << "     Check platform and use appropriate tools" << std::endl;

    std::cout << "\ n PORTABLE CODE:" << std::endl;
    std::cout << "\n#ifdef _WIN32" << std::endl;
    std::cout << "    std::string cmd = \"dir\";" << std::endl;
    std::cout << "#else" << std::endl;
    std::cout << "    std::string cmd = \"ls\";" << std::endl;
    std::cout << "#endif" << std::endl;

    std::cout << "\nOr use std::filesystem for platform independence!" << std
        ::endl;
}

// ==================================================================
// SECTION 19: BEST PRACTICES SUMMARY
// ==================================================================
```

```

void explain_best_practices() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "BEST PRACTICES SUMMARY:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\ n DO:" << std::endl;
    std::cout << "\n1. Use std::filesystem for file operations" << std::endl;
    std::cout << " •   Cross-platform" << std::endl;
    std::cout << " •   Exception-safe" << std::endl;
    std::cout << " •   Modern and clean API" << std::endl;

    std::cout << "\n2. Use std::string_view for parsing" << std::endl;
    std::cout << " •   Zero-copy substring operations" << std::endl;
    std::cout << " •   Perfect for tokenizing large outputs" << std::endl;
    std::cout << " •   No memory allocations" << std::endl;

    std::cout << "\n3. Use std::regex for pattern matching" << std::endl;
    std::cout << " •   Structured text parsing" << std::endl;
    std::cout << " •   Validation" << std::endl;
    std::cout << " •   Extraction from unstructured data" << std::endl;

    std::cout << "\n4. Check return codes and use error_code" << std::endl;
    std::cout << " •   filesystem operations can fail" << std::endl;
    std::cout << " •   Commands can fail" << std::endl;
    std::cout << " •   Use std::error_code for exception-free errors" << std::
        endl;

    std::cout << "\n5. Use RAII for resource management" << std::endl;
    std::cout << " •   std::unique_ptr with custom deleter for FILE*" << std::
        endl;
    std::cout << " •   Automatic cleanup on exception" << std::endl;

    std::cout << "\ n DON'T:" << std::endl;
    std::cout << "\n1. Don't use system() with user input" << std::endl;
    std::cout << " •   Command injection vulnerability" << std::endl;
    std::cout << " •   Use execve() family instead" << std::endl;

    std::cout << "\n2. Don't ignore errors" << std::endl;
    std::cout << " •   Check pclose() return value" << std::endl;
    std::cout << " •   Check filesystem operation errors" << std::endl;

    std::cout << "\n3. Don't parse with manual string manipulation" << std::
        endl;
    std::cout << " •   Use string_view for efficiency" << std::endl;
    std::cout << " •   Use regex for complex patterns" << std::endl;

    std::cout << "\n4. Don't assume POSIX everywhere" << std::endl;
    std::cout << " •   Windows is different" << std::endl;
    std::cout << " •   Use std::filesystem for portability" << std::endl;

    std::cout << "\ n GOLDEN RULES:" << std::endl;
    std::cout << "   1. Validate all external input" << std::endl;
    std::cout << "   2. Use modern C++ facilities (filesystem, string_view)"
```

```
        << std::endl;
    std::cout << "   3. Check errors always" << std::endl;
    std::cout << "   4. Think security first!" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "     SYSTEM INTERACTION, FILESYSTEM, STRING_VIEW, AND REGEX
              \n";
    std::cout << "
                                                                        \n";
    std::cout << "  Modern C++ for system programming and external tool
        integration\n";
    std::cout << "                                    \n";

    try {
        demonstrate_basic_execution();
        demonstrate_string_view_parsing();
        demonstrate_regex_parsing();
        demonstrate_filesystem();
        demonstrate_bash_scripts();
        demonstrate_python_scripts();
        demonstrate_linux_tools();
        demonstrate_file_streams();
        demonstrate_string_streams();
        demonstrate_stream_manipulators();
        demonstrate_binary_io();
        demonstrate_custom_operators();
        demonstrate_mutex_and_lock_guard();
        demonstrate_unique_lock();
        demonstrate_file_locking();
        demonstrate_race_condition();
        explain_security_considerations();
        explain_cross_platform();
        explain_best_practices();

        std::cout << "\n" << std::string(70, '=') << std::endl;
        std::cout << "SUMMARY:\n";
        std::cout << std::string(70, '=') << std::endl;

        std::cout << "\n KEY FEATURES DEMONSTRATED:" << std::endl;
        std::cout << "\n1. COMMAND EXECUTION:" << std::endl;
        std::cout << " •   popen() for output capture" << std::endl;
        std::cout << " •   Return code checking" << std::endl;
        std::cout << " •   RAII for resource management" << std::endl;

        std::cout << "\n2. std::string_view (C++17):" << std::endl;
        std::cout << " •   Zero-copy parsing" << std::endl;
        std::cout << " •   Efficient tokenization" << std::endl;
```

```
        std::cout << " •   Perfect for large outputs" << std::endl;

        std::cout << "\n3. std::regex (C++11):" << std::endl;
        std::cout << " •   Pattern matching" << std::endl;
        std::cout << " •   Log parsing" << std::endl;
        std::cout << " •   Network output parsing" << std::endl;

        std::cout << "\n4. std::filesystem (C++17):" << std::endl;
        std::cout << " •   Cross-platform paths" << std::endl;
        std::cout << " •   Directory iteration" << std::endl;
        std::cout << " •   Error code handling" << std::endl;

        std::cout << "\n5. STREAM I/O:" << std::endl;
        std::cout << " •   File streams (ifstream, ofstream, fstream)" << std
            ::endl;
        std::cout << " •   String streams (ostringstream, istringstream)" <<
            std::endl;
        std::cout << " •   Stream manipulators (setw, setprecision, etc.)" <<
            std::endl;
        std::cout << " •   Binary I/O for POD types" << std::endl;
        std::cout << " •   Custom stream operators" << std::endl;

        std::cout << "\n6. SCRIPT INTEGRATION:" << std::endl;
        std::cout << " •   Bash scripts with arguments" << std::endl;
        std::cout << " •   Python scripts with JSON output" << std::endl;
        std::cout << " •   Structured data exchange" << std::endl;

        std::cout << "\n7. THREAD SYNCHRONIZATION:" << std::endl;
        std::cout << " •   std::mutex and std::lock_guard (RAII)" << std::endl
            ;
        std::cout << " •   std::unique_lock (flexible locking)" << std::endl;
        std::cout << " •   std::shared_mutex (reader-writer locks)" << std::
            endl;
        std::cout << " •   std::condition_variable (wait/notify)" << std::endl
            ;
        std::cout << " •   OS-level file locking (flock)" << std::endl;
        std::cout << " •   Race condition examples and fixes" << std::endl;

        std::cout << "\n SECURITY REMINDERS:" << std::endl;
        std::cout << " •   Never pass unsanitized user input to shell" << std
            ::endl;
        std::cout << " •   Validate all paths and arguments" << std::endl;
        std::cout << " •   Use absolute paths for commands" << std::endl;
        std::cout << " •   Check return codes always" << std::endl;

        std::cout << "\n Modern C++: Safe, efficient system programming!\n"
            << std::endl;

    } catch (const std::exception& e) {
        std::cerr << "\n Error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
```

```
}
```

70 Source Code: TemplatedCameraInterface.cpp

File: `src/TemplatedCameraInterface.cpp`
Repository: View on GitHub

```cpp
#include <iostream>
#include <array>
#include <vector>
#include <memory>
#include <cstdint>
#include <type_traits>
#include <algorithm>
#include <cmath>
#include <chrono>

// ====================================================================
// TEMPLATED CAMERA INTERFACE FOR MULTIPLE PIXEL TYPES
// ====================================================================
// Demonstrates C++ templates for camera interfacing where pixel data
// can be various types: uint8_t, uint16_t, float, double
// ====================================================================

// ====================================================================
// 1. BASIC IMAGE CLASS (TEMPLATED BY PIXEL TYPE)
// ====================================================================

template<typename PixelType>
class Image {
private:
    size_t width;
    size_t height;
    std::vector<PixelType> pixels;

public:
    Image(size_t w, size_t h)
        : width(w), height(h), pixels(w * h) {}

    Image(size_t w, size_t h, PixelType initial_value)
        : width(w), height(h), pixels(w * h, initial_value) {}

    // Accessors
    size_t get_width() const { return width; }
    size_t get_height() const { return height; }
    size_t get_size() const { return pixels.size(); }

    // Pixel access
    PixelType& at(size_t x, size_t y) {
        return pixels[y * width + x];
    }

    const PixelType& at(size_t x, size_t y) const {
        return pixels[y * width + x];
    }

```

```
    // Raw data access (for camera hardware interface)
    PixelType* data() { return pixels.data(); }
    const PixelType* data() const { return pixels.data(); }

    // Memory size in bytes
    size_t memory_bytes() const {
        return pixels.size() * sizeof(PixelType);
    }

    // Fill with value
    void fill(PixelType value) {
        std::fill(pixels.begin(), pixels.end(), value);
    }
};

// ==================================================================
// 2. CAMERA INTERFACE (TEMPLATED)
// ==================================================================

template<typename PixelType>
class Camera {
private:
    size_t width;
    size_t height;
    std::string camera_name;

public:
    Camera(const std::string& name, size_t w, size_t h)
        : camera_name(name), width(w), height(h) {}

    virtual ~Camera() = default;

    // Pure virtual: capture image
    virtual Image<PixelType> capture() = 0;

    // Configuration
    size_t get_width() const { return width; }
    size_t get_height() const { return height; }
    const std::string& get_name() const { return camera_name; }

    // Get pixel type information
    static constexpr size_t bits_per_pixel() {
        return sizeof(PixelType) * 8;
    }

    static constexpr bool is_floating_point() {
        return std::is_floating_point_v<PixelType>;
    }

    static constexpr bool is_integer() {
        return std::is_integral_v<PixelType>;
    }
};

```

```
// ==================================================================
// 3. SIMULATED CAMERA IMPLEMENTATIONS
// ==================================================================

// 8-bit grayscale camera (most common)
class Camera8bit : public Camera<uint8_t> {
public:
    Camera8bit(const std::string& name, size_t w, size_t h)
        : Camera<uint8_t>(name, w, h) {}

    Image<uint8_t> capture() override {
        Image<uint8_t> img(get_width(), get_height());

        // Simulate captured gradient pattern
        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                uint8_t value = static_cast<uint8_t>(
                    (x * 255.0 / get_width()) * 0.5 +
                    (y * 255.0 / get_height()) * 0.5
                );
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

// 16-bit camera (scientific/medical imaging)
class Camera16bit : public Camera<uint16_t> {
public:
    Camera16bit(const std::string& name, size_t w, size_t h)
        : Camera<uint16_t>(name, w, h) {}

    Image<uint16_t> capture() override {
        Image<uint16_t> img(get_width(), get_height());

        // Simulate high dynamic range data
        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                uint16_t value = static_cast<uint16_t>(
                    (x * 65535.0 / get_width()) * 0.3 +
                    (y * 65535.0 / get_height()) * 0.7
                );
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

// Float camera (normalized values 0.0-1.0)
class CameraFloat : public Camera<float> {
public:
    CameraFloat(const std::string& name, size_t w, size_t h)
```

```
        : Camera<float>(name, w, h) {}

    Image<float> capture() override {
        Image<float> img(get_width(), get_height());

        // Simulate normalized data with some pattern
        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                float value =
                    0.5f + 0.5f * std::sin(x * 0.1f) * std::cos(y * 0.1f);
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

// Double precision camera (research/astronomy)
class CameraDouble : public Camera<double> {
public:
    CameraDouble(const std::string& name, size_t w, size_t h)
        : Camera<double>(name, w, h) {}

    Image<double> capture() override {
        Image<double> img(get_width(), get_height());

        // Simulate high-precision data
        for (size_t y = 0; y < get_height(); ++y) {
            for (size_t x = 0; x < get_width(); ++x) {
                double value =
                    std::sin(x * 0.05) * std::cos(y * 0.05) +
                    std::exp(-((x - get_width()/2.0) * (x - get_width()/2.0) +
                               (y - get_height()/2.0) * (y - get_height()/2.0)
                               ) / 1000.0);
                img.at(x, y) = value;
            }
        }
        return img;
    }
};

// ==================================================================
// 4. IMAGE PROCESSING ALGORITHMS (TEMPLATED)
// ==================================================================

template<typename PixelType>
class ImageProcessor {
public:
    // Calculate average pixel value
    static double calculate_mean(const Image<PixelType>& img) {
        double sum = 0.0;
        const PixelType* data = img.data();
        size_t size = img.get_size();

```

```
        for (size_t i = 0; i < size; ++i) {
            sum += static_cast<double>(data[i]);
        }

        return sum / size;
    }

    // Find min and max pixel values
    static std::pair<PixelType, PixelType> find_min_max(const Image<PixelType
        >& img) {
        const PixelType* data = img.data();
        size_t size = img.get_size();

        PixelType min_val = data[0];
        PixelType max_val = data[0];

        for (size_t i = 1; i < size; ++i) {
            if (data[i] < min_val) min_val = data[i];
            if (data[i] > max_val) max_val = data[i];
        }

        return {min_val, max_val};
    }

    // Scale pixel values
    static Image<PixelType> scale(const Image<PixelType>& img, double factor)
        {
        Image<PixelType> result(img.get_width(), img.get_height());

        for (size_t y = 0; y < img.get_height(); ++y) {
            for (size_t x = 0; x < img.get_width(); ++x) {
                double scaled = static_cast<double>(img.at(x, y)) * factor;
                result.at(x, y) = static_cast<PixelType>(scaled);
            }
        }

        return result;
    }

    // Threshold operation
    static Image<PixelType> threshold(const Image<PixelType>& img, PixelType
        threshold_value) {
        Image<PixelType> result(img.get_width(), img.get_height());

        for (size_t y = 0; y < img.get_height(); ++y) {
            for (size_t x = 0; x < img.get_width(); ++x) {
                result.at(x, y) = (img.at(x, y) >= threshold_value) ?
                    threshold_value : PixelType(0);
            }
        }

        return result;
    }
};
```

```

// ==================================================================
// 5. TYPE CONVERSION BETWEEN PIXEL TYPES
// ==================================================================

template<typename DestType, typename SrcType>
Image<DestType> convert_image(const Image<SrcType>& src) {
    Image<DestType> dest(src.get_width(), src.get_height());

    // Find source range for proper scaling
    auto [min_val, max_val] = ImageProcessor<SrcType>::find_min_max(src);
    double src_range = static_cast<double>(max_val) - static_cast<double>(
        min_val);

    // Determine destination range
    double dest_min, dest_max;
    if constexpr (std::is_floating_point_v<DestType>) {
        dest_min = 0.0;
        dest_max = 1.0;
    } else if constexpr (std::is_same_v<DestType, uint8_t>) {
        dest_min = 0.0;
        dest_max = 255.0;
    } else if constexpr (std::is_same_v<DestType, uint16_t>) {
        dest_min = 0.0;
        dest_max = 65535.0;
    } else {
        dest_min = 0.0;
        dest_max = 1.0;
    }

    double dest_range = dest_max - dest_min;

    // Convert with proper scaling
    for (size_t y = 0; y < src.get_height(); ++y) {
        for (size_t x = 0; x < src.get_width(); ++x) {
            double normalized = (static_cast<double>(src.at(x, y)) -
                static_cast<double>(min_val)) / src_range;
            double scaled = normalized * dest_range + dest_min;
            dest.at(x, y) = static_cast<DestType>(scaled);
        }
    }

    return dest;
}

// ==================================================================
// 6. GENERIC CAMERA HANDLER (WORKS WITH ANY PIXEL TYPE)
// ==================================================================

template<typename PixelType>
class CameraHandler {
private:
    std::unique_ptr<Camera<PixelType>> camera;

```

```
public:
    CameraHandler(std::unique_ptr<Camera<PixelType>> cam)
        : camera(std::move(cam)) {}

    void display_camera_info() {
        std::cout << "\n  Camera: " << camera->get_name() << std::endl;
        std::cout << "  Resolution: " << camera->get_width() << "x" << camera
            ->get_height() << std::endl;
        std::cout << "  Pixel type: " << typeid(PixelType).name() << std::endl
            ;
        std::cout << "  Bits per pixel: " << Camera<PixelType>::bits_per_pixel
            () << std::endl;
        std::cout << "  Floating point: " << (Camera<PixelType>::
            is_floating_point() ? "Yes" : "No") << std::endl;

        // Calculate memory per frame
        size_t bytes = camera->get_width() * camera->get_height() * sizeof(
            PixelType);
        std::cout << "  Memory per frame: " << bytes << " bytes";
        if (bytes >= 1024 * 1024) {
            std::cout << " (" << bytes / (1024.0 * 1024.0) << " MB)";
        } else if (bytes >= 1024) {
            std::cout << " (" << bytes / 1024.0 << " KB)";
        }
        std::cout << std::endl;
    }

    void capture_and_process() {
        std::cout << "\n  Capturing image..." << std::endl;
        auto start = std::chrono::high_resolution_clock::now();

        Image<PixelType> img = camera->capture();

        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
            end - start);
        std::cout << "  Capture time: " << duration.count() << " µs" << std::
            endl;

        // Process image
        std::cout << "\n  Processing image..." << std::endl;
        double mean = ImageProcessor<PixelType>::calculate_mean(img);
        auto [min_val, max_val] = ImageProcessor<PixelType>::find_min_max(img)
            ;

        std::cout << "  Mean value: " << mean << std::endl;
        std::cout << "  Min value: " << static_cast<double>(min_val) << std::
            endl;
        std::cout << "  Max value: " << static_cast<double>(max_val) << std::
            endl;

        // Sample some pixel values
        std::cout << "\n  Sample pixels:" << std::endl;
        std::cout << "    Top-left (0,0): " << static_cast<double>(img.at(0,
```

```
            0)) << std::endl;
        std::cout << "    Center (" << img.get_width()/2 << "," << img.get_height()/2 << "): "
                  << static_cast<double>(img.at(img.get_width()/2, img.get_height()/2)) << std::endl;
        std::cout << "    Bottom-right (" << img.get_width()-1 << "," << img.get_height()-1 << "): "
                  << static_cast<double>(img.at(img.get_width()-1, img.get_height()-1)) << std::endl;
    }
};

// ==================================================================
// 7. DEMONSTRATION FUNCTIONS
// ==================================================================

void demonstrate_8bit_camera() {
    std::cout << "\n=== 1. 8-BIT GRAYSCALE CAMERA (uint8_t) ===" << std::endl;
    std::cout << "Common in: Webcams, industrial cameras, surveillance" << std::endl;

    CameraHandler<uint8_t> handler(
        std::make_unique<Camera8bit>("Webcam HD", 640, 480)
    );

    handler.display_camera_info();
    handler.capture_and_process();
}

void demonstrate_16bit_camera() {
    std::cout << "\n=== 2. 16-BIT CAMERA (uint16_t) ===" << std::endl;
    std::cout << "Common in: Scientific imaging, medical X-ray, astronomy" << std::endl;

    CameraHandler<uint16_t> handler(
        std::make_unique<Camera16bit>("Scientific CCD", 1024, 1024)
    );

    handler.display_camera_info();
    handler.capture_and_process();
}

void demonstrate_float_camera() {
    std::cout << "\n=== 3. FLOAT CAMERA (32-bit float) ===" << std::endl;
    std::cout << "Common in: HDR imaging, computational photography, depth sensors" << std::endl;

    CameraHandler<float> handler(
        std::make_unique<CameraFloat>("ToF Depth Camera", 320, 240)
    );

    handler.display_camera_info();
    handler.capture_and_process();
}
```

```

void demonstrate_double_camera() {
    std::cout << "\n=== 4. DOUBLE PRECISION CAMERA (64-bit double) ===" << std
        ::endl;
    std::cout << "Common in: Research, astronomical imaging, spectroscopy" <<
        std::endl;

    CameraHandler<double> handler(
        std::make_unique<CameraDouble>("Telescope CCD", 512, 512)
    );

    handler.display_camera_info();
    handler.capture_and_process();
}

void demonstrate_type_conversion() {
    std::cout << "\n=== 5. IMAGE TYPE CONVERSION ===" << std::endl;

    // Capture 8-bit image
    Camera8bit cam8("Source Camera", 320, 240);
    Image<uint8_t> img8 = cam8.capture();

    std::cout << "\nOriginal 8-bit image:" << std::endl;
    std::cout << "  Size: " << img8.memory_bytes() << " bytes" << std::endl;
    auto [min8, max8] = ImageProcessor<uint8_t>::find_min_max(img8);
    std::cout << "  Range: " << (int)min8 << " to " << (int)max8 << std::endl;

    // Convert to 16-bit
    std::cout << "\nConverting to 16-bit..." << std::endl;
    Image<uint16_t> img16 = convert_image<uint16_t>(img8);
    std::cout << "  Size: " << img16.memory_bytes() << " bytes" << std::endl;
    auto [min16, max16] = ImageProcessor<uint16_t>::find_min_max(img16);
    std::cout << "  Range: " << min16 << " to " << max16 << std::endl;

    // Convert to float
    std::cout << "\nConverting to float..." << std::endl;
    Image<float> imgf = convert_image<float>(img8);
    std::cout << "  Size: " << imgf.memory_bytes() << " bytes" << std::endl;
    auto [minf, maxf] = ImageProcessor<float>::find_min_max(imgf);
    std::cout << "  Range: " << minf << " to " << maxf << std::endl;

    // Convert to double
    std::cout << "\nConverting to double..." << std::endl;
    Image<double> imgd = convert_image<double>(img8);
    std::cout << "  Size: " << imgd.memory_bytes() << " bytes" << std::endl;
    auto [mind, maxd] = ImageProcessor<double>::find_min_max(imgd);
    std::cout << "  Range: " << mind << " to " << maxd << std::endl;
}

void demonstrate_processing_algorithms() {
    std::cout << "\n=== 6. IMAGE PROCESSING ALGORITHMS ===" << std::endl;

    Camera8bit cam("Processing Camera", 100, 100);
    Image<uint8_t> img = cam.capture();
```

```

    std::cout << "\nOriginal image statistics:" << std::endl;
    double mean = ImageProcessor<uint8_t>::calculate_mean(img);
    auto [min_val, max_val] = ImageProcessor<uint8_t>::find_min_max(img);
    std::cout << "  Mean: " << mean << std::endl;
    std::cout << "  Min: " << (int)min_val << ", Max: " << (int)max_val << std::endl;

    // Scale by 2x
    std::cout << "\nScaling by 2x:" << std::endl;
    Image<uint8_t> scaled = ImageProcessor<uint8_t>::scale(img, 2.0);
    double mean_scaled = ImageProcessor<uint8_t>::calculate_mean(scaled);
    std::cout << "  Mean after scaling: " << mean_scaled << std::endl;

    // Threshold
    std::cout << "\nThresholding at 128:" << std::endl;
    Image<uint8_t> thresholded = ImageProcessor<uint8_t>::threshold(img, 128);
    size_t above_threshold = 0;
    for (size_t i = 0; i < thresholded.get_size(); ++i) {
        if (thresholded.data()[i] > 0) ++above_threshold;
    }
    std::cout << "  Pixels above threshold: " << above_threshold
              << " (" << (100.0 * above_threshold / thresholded.get_size()) << "%)" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n==================================================================" << std::endl;
    std::cout << "  TEMPLATED CAMERA INTERFACE FOR MULTIPLE PIXEL TYPES" << std::endl;
    std::cout << "==================================================================" << std::endl;

    demonstrate_8bit_camera();
    demonstrate_16bit_camera();
    demonstrate_float_camera();
    demonstrate_double_camera();
    demonstrate_type_conversion();
    demonstrate_processing_algorithms();

    std::cout << "\n==================================================================" << std::endl;
    std::cout << "  TEMPLATE BENEFITS FOR CAMERA INTERFACING" << std::endl;
    std::cout << "==================================================================" << std::endl;
```

```

    std::cout << "\n ADVANTAGES OF TEMPLATES:" << std::endl;
    std::cout << "\n1. TYPE SAFETY" << std::endl;
    std::cout << " •   Compile-time type checking" << std::endl;
    std::cout << " •   No runtime casting needed" << std::endl;
    std::cout << " •   Prevents mixing incompatible pixel types" << std::endl;

    std::cout << "\n2. ZERO OVERHEAD" << std::endl;
    std::cout << " •   No virtual function calls" << std::endl;
    std::cout << " •   All types resolved at compile time" << std::endl;
    std::cout << " •   Fully optimizable by compiler" << std::endl;

    std::cout << "\n3. CODE REUSABILITY" << std::endl;
    std::cout << " •   Write processing algorithms once" << std::endl;
    std::cout << " •   Works with uint8_t, uint16_t, float, double" << std::
        endl;
    std::cout << " •   Easy to add new pixel types" << std::endl;

    std::cout << "\n4. COMPILE-TIME INFORMATION" << std::endl;
    std::cout << " •   sizeof(PixelType) known at compile time" << std::endl;
    std::cout << " •   Type traits (is_floating_point, etc.)" << std::endl;
    std::cout << " •   if constexpr for type-specific code" << std::endl;

    std::cout << "\n MEMORY COMPARISON (640x480 image):" << std::endl;
    size_t pixels = 640 * 480;
    std::cout << "  uint8_t:  " << pixels * sizeof(uint8_t) / 1024 << " KB" <<
         std::endl;
    std::cout << "  uint16_t: " << pixels * sizeof(uint16_t) / 1024 << " KB"
        << std::endl;
    std::cout << "  float:    " << pixels * sizeof(float) / 1024 << " KB" <<
        std::endl;
    std::cout << "  double:   " << pixels * sizeof(double) / 1024 << " KB" <<
        std::endl;

    std::cout << "\n REAL-WORLD APPLICATIONS:" << std::endl;
    std::cout << "\n  uint8_t (8-bit):" << std::endl;
    std::cout << " •    Consumer webcams" << std::endl;
    std::cout << " •    Surveillance cameras" << std::endl;
    std::cout << " •    Industrial inspection" << std::endl;
    std::cout << " •    Video streaming" << std::endl;

    std::cout << "\n  uint16_t (16-bit):" << std::endl;
    std::cout << " •    Medical X-ray imaging" << std::endl;
    std::cout << " •    Scientific CCD cameras" << std::endl;
    std::cout << " •    Astronomy (star imaging)" << std::endl;
    std::cout << " •    High dynamic range photography" << std::endl;

    std::cout << "\n  float (32-bit):" << std::endl;
    std::cout << " •    Time-of-Flight (ToF) depth sensors" << std::endl;
    std::cout << " •    HDR image processing" << std::endl;
    std::cout << " •    Computational photography" << std::endl;
    std::cout << " •    Normalized image data" << std::endl;

    std::cout << "\n  double (64-bit):" << std::endl;
```

```
    std::cout << " •    Radio telescope data" << std::endl;
    std::cout << " •    Spectroscopy" << std::endl;
    std::cout << " •    High-precision research" << std::endl;
    std::cout << " •    Scientific analysis" << std::endl;

    std::cout << "\n TEMPLATE TECHNIQUES DEMONSTRATED:" << std::endl;
    std::cout << " •  Class templates: Image<T>, Camera<T>" << std::endl;
    std::cout << " •  Function templates: convert_image<DestType, SrcType>" << std::endl;
    std::cout << " •  Type traits: std::is_floating_point_v<T>" << std::endl;
    std::cout << " •  if constexpr: Compile-time conditional code" << std::endl;
    std::cout << " •  Template specialization: Type-specific behavior" << std::endl;
    std::cout << " •  constexpr functions: Compile-time computation" << std::endl;

    std::cout << "\n ================================================================\n" << std::endl;

    return 0;
}
```

71 Source Code: TemplatedCameraModules.cpp

File: `src/TemplatedCameraModules.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// C++20 MODULES DEMONSTRATION: IMPORT AND USE CAMERA MODULE
// ==================================================================
// This file demonstrates:
// 1. Importing C++20 modules with "import"
// 2. Using exported templates, classes, and concepts
// 3. C++20 features: concepts, requires clauses, [[nodiscard]]
// ==================================================================

#include <iostream>
#include <iomanip>
#include <chrono>

// C++20 MODULE IMPORT
import camera;

// ==================================================================
// DEMONSTRATION FUNCTIONS
// ==================================================================

void demonstrate_concepts() {
    std::cout << "\n=== C++20 CONCEPTS DEMONSTRATION ===" << std::endl;

    std::cout << "\nPixelType concept checks:" << std::endl;
    std::cout << "  uint8_t is PixelType: " << PixelType<uint8_t> << std::endl;
    std::cout << "  uint16_t is PixelType: " << PixelType<uint16_t> << std::endl;
    std::cout << "  float is PixelType: " << PixelType<float> << std::endl;
    std::cout << "  double is PixelType: " << PixelType<double> << std::endl;
    std::cout << "  int is PixelType: " << PixelType<int> << " (not in allowed list)" << std::endl;

    std::cout << "\nIntegerPixel concept checks:" << std::endl;
    std::cout << "  uint8_t is IntegerPixel: " << IntegerPixel<uint8_t> << std::endl;
    std::cout << "  float is IntegerPixel: " << IntegerPixel<float> << std::endl;

    std::cout << "\nFloatingPixel concept checks:" << std::endl;
    std::cout << "  float is FloatingPixel: " << FloatingPixel<float> << std::endl;
    std::cout << "  uint8_t is FloatingPixel: " << FloatingPixel<uint8_t> << std::endl;
}

template<PixelType T>
void display_camera_info(const Camera<T>& camera) {
    std::cout << "\n  Camera: " << camera.get_name() << std::endl;
```

```
    std::cout << "  Resolution: " << camera.get_width() << "x" << camera.get_height() << std::endl;
    std::cout << "  Bits per pixel: " << Camera<T>::bits_per_pixel() << std::endl;
    std::cout << "  Floating point: " << (Camera<T>::is_floating_point() ? "Yes" : "No") << std::endl;

    size_t bytes = camera.get_width() * camera.get_height() * sizeof(T);
    std::cout << "  Memory per frame: " << bytes << " bytes";
    if (bytes >= 1024 * 1024) {
        std::cout << " (" << std::fixed << std::setprecision(2)
                  << bytes / (1024.0 * 1024.0) << " MB)";
    } else if (bytes >= 1024) {
        std::cout << " (" << std::fixed << std::setprecision(2)
                  << bytes / 1024.0 << " KB)";
    }
    std::cout << std::endl;
}

template<PixelType T>
void demonstrate_camera_capture(CameraHandler<T>& handler) {
    const Camera<T>* cam = handler.get_camera();

    std::cout << "\n  Capturing image..." << std::endl;
    auto start = std::chrono::high_resolution_clock::now();

    Image<T> img = handler.capture();

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    std::cout << "  Capture time: " << duration.count() << " µs" << std::endl;

    std::cout << "\n  Processing image..." << std::endl;
    double mean = ImageProcessor<T>::calculate_mean(img);
    auto [min_val, max_val] = ImageProcessor<T>::find_min_max(img);

    std::cout << "  Mean value: " << mean << std::endl;
    std::cout << "  Min value: " << static_cast<double>(min_val) << std::endl;
    std::cout << "  Max value: " << static_cast<double>(max_val) << std::endl;

    std::cout << "\n  Sample pixels:" << std::endl;
    std::cout << "    Top-left (0,0): " << static_cast<double>(img.at(0, 0)) << std::endl;
    std::cout << "    Center (" << img.get_width()/2 << "," << img.get_height()/2 << "): "
              << static_cast<double>(img.at(img.get_width()/2, img.get_height()/2)) << std::endl;
}

void demonstrate_8bit_camera() {
    std::cout << "\n=== 1. 8-BIT CAMERA (uint8_t) ===" << std::endl;

    CameraHandler<uint8_t> handler(
```

```
        std::make_unique<Camera8bit>("Webcam HD", 640, 480)
    );

    display_camera_info(*handler.get_camera());
    demonstrate_camera_capture(handler);
}

void demonstrate_16bit_camera() {
    std::cout << "\n=== 2. 16-BIT CAMERA (uint16_t) ===" << std::endl;

    CameraHandler<uint16_t> handler(
        std::make_unique<Camera16bit>("Scientific CCD", 1024, 1024)
    );

    display_camera_info(*handler.get_camera());
    demonstrate_camera_capture(handler);
}

void demonstrate_float_camera() {
    std::cout << "\n=== 3. FLOAT CAMERA (32-bit) ===" << std::endl;

    CameraHandler<float> handler(
        std::make_unique<CameraFloat>("ToF Depth Camera", 320, 240)
    );

    display_camera_info(*handler.get_camera());
    demonstrate_camera_capture(handler);
}

void demonstrate_double_camera() {
    std::cout << "\n=== 4. DOUBLE CAMERA (64-bit) ===" << std::endl;

    CameraHandler<double> handler(
        std::make_unique<CameraDouble>("Telescope CCD", 512, 512)
    );

    display_camera_info(*handler.get_camera());
    demonstrate_camera_capture(handler);
}

void demonstrate_type_conversion() {
    std::cout << "\n=== 5. IMAGE TYPE CONVERSION ===" << std::endl;

    Camera8bit cam8("Source Camera", 320, 240);
    Image<uint8_t> img8 = cam8.capture();

    std::cout << "\nOriginal 8-bit image:" << std::endl;
    std::cout << "  Size: " << img8.memory_bytes() << " bytes" << std::endl;
    auto [min8, max8] = ImageProcessor<uint8_t>::find_min_max(img8);
    std::cout << "  Range: " << (int)min8 << " to " << (int)max8 << std::endl;

    std::cout << "\nConverting to 16-bit..." << std::endl;
    Image<uint16_t> img16 = convert_image<uint16_t>(img8);
    std::cout << "  Size: " << img16.memory_bytes() << " bytes" << std::endl;
```

```
    auto [min16, max16] = ImageProcessor<uint16_t>::find_min_max(img16);
    std::cout << "  Range: " << min16 << " to " << max16 << std::endl;

    std::cout << "\nConverting to float..." << std::endl;
    Image<float> imgf = convert_image<float>(img8);
    std::cout << "  Size: " << imgf.memory_bytes() << " bytes" << std::endl;
    auto [minf, maxf] = ImageProcessor<float>::find_min_max(imgf);
    std::cout << "  Range: " << minf << " to " << maxf << std::endl;

    std::cout << "\nConverting to double..." << std::endl;
    Image<double> imgd = convert_image<double>(img8);
    std::cout << "  Size: " << imgd.memory_bytes() << " bytes" << std::endl;
    auto [mind, maxd] = ImageProcessor<double>::find_min_max(imgd);
    std::cout << "  Range: " << mind << " to " << maxd << std::endl;
}

void demonstrate_concept_constrained_operations() {
    std::cout << "\n=== 6. CONCEPT-CONSTRAINED OPERATIONS ===" << std::endl;

    // Integer pixel operations
    std::cout << "\nInteger pixel operations (uint8_t):" << std::endl;
    Camera8bit cam8("Integer Camera", 100, 100);
    Image<uint8_t> img8 = cam8.capture();

    std::cout << "  Original mean: " << ImageProcessor<uint8_t>::
        calculate_mean(img8) << std::endl;

    // Threshold only works with IntegerPixel concept
    Image<uint8_t> thresholded = ImageProcessor<uint8_t>::threshold(img8, 128)
        ;
    size_t above = 0;
    for (size_t i = 0; i < thresholded.get_size(); ++i) {
        if (thresholded.data()[i] > 0) ++above;
    }
    std::cout << "  Pixels above threshold: " << above
              << " (" << (100.0 * above / thresholded.get_size()) << "%)" <<
                  std::endl;

    // Floating point operations
    std::cout << "\nFloating pixel operations (float):" << std::endl;
    CameraFloat camf("Float Camera", 100, 100);
    Image<float> imgf = camf.capture();

    std::cout << "  Original mean: " << ImageProcessor<float>::calculate_mean(
        imgf) << std::endl;

    // Normalize only works with FloatingPixel concept
    Image<float> normalized = ImageProcessor<float>::normalize(imgf);
    std::cout << "  Normalized mean: " << ImageProcessor<float>::
        calculate_mean(normalized) << std::endl;
    auto [minf, maxf] = ImageProcessor<float>::find_min_max(normalized);
    std::cout << "  Normalized range: " << minf << " to " << maxf << std::endl
        ;

```

```
    // Note: The following would NOT compile (concept constraint):
    // Image<float> bad = ImageProcessor<float>::threshold(imgf, 0.5f); // ERROR: requires IntegerPixel
    // Image<uint8_t> bad2 = ImageProcessor<uint8_t>::normalize(img8);  // ERROR: requires FloatingPixel

    std::cout << "\n   Concept constraints prevent invalid operations at compile time!" << std::endl;
}

// ================================================================
// MAIN FUNCTION
// ================================================================

int main() {
    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  C++20 MODULES: TEMPLATED CAMERA INTERFACE" << std::endl;
    std::cout << " ================================================================" << std::endl;

    demonstrate_concepts();
    demonstrate_8bit_camera();
    demonstrate_16bit_camera();
    demonstrate_float_camera();
    demonstrate_double_camera();
    demonstrate_type_conversion();
    demonstrate_concept_constrained_operations();

    std::cout << "\n ================================================================" << std::endl;
    std::cout << "  C++20 FEATURES DEMONSTRATED" << std::endl;
    std::cout << " ================================================================" << std::endl;

    std::cout << "\n MODULES (import/export):" << std::endl;
    std::cout << " •   export module camera - Module interface declaration" << std::endl;
    std::cout << " •   import camera - Import module in main program" << std::endl;
    std::cout << " •   Exported classes: Image<T>, Camera<T>, ImageProcessor<T>" << std::endl;
    std::cout << " •   Exported functions: convert_image<T1, T2>()" << std::endl;
    std::cout << " •   Exported concepts: PixelType, IntegerPixel, FloatingPixel" << std::endl;

    std::cout << "\n CONCEPTS:" << std::endl;
    std::cout << " •   concept PixelType - Constrain allowed pixel types" << std::endl;
```

```
    std::cout << " •   concept IntegerPixel - Only integer pixel types" << std::endl;
    std::cout << " •   concept FloatingPixel - Only floating-point pixel types" << std::endl;
    std::cout << " •   requires clauses on functions (threshold, normalize)" << std::endl;
    std::cout << " •   Compile-time type checking prevents invalid operations" << std::endl;

    std::cout << "\n OTHER C++20 FEATURES:" << std::endl;
    std::cout << " •   [[nodiscard]] - Warn if return value ignored" << std::endl;
    std::cout << " •   Three-way comparison (operator<=>) - Auto-generated comparisons" << std::endl;
    std::cout << " •   noexcept specifications - Exception safety guarantees" << std::endl;

    std::cout << "\n MODULE BENEFITS:" << std::endl;
    std::cout << " •   Faster compilation - No header parsing overhead" << std::endl;
    std::cout << " •   Better encapsulation - Only exported entities visible" << std::endl;
    std::cout << " •   No include guards needed - Modules included once automatically" << std::endl;
    std::cout << " •   Order independent - No macro contamination" << std::endl;
    std::cout << " •   Cleaner code - Explicit export declarations" << std::endl;

    std::cout << "\n COMPILATION NOTES:" << std::endl;
    std::cout << "   GCC 11+:" << std::endl;
    std::cout << "     g++ -std=c++20 -fmodules-ts -xc++-system-header iostream" << std::endl;
    std::cout << "     g++ -std=c++20 -fmodules-ts -c CameraModule.cppm" << std::endl;
    std::cout << "     g++ -std=c++20 -fmodules-ts TemplatedCameraModules.cpp CameraModule.o" << std::endl;

    std::cout << "\n   MSVC 2022+:" << std::endl;
    std::cout << "     cl /std:c++20 /experimental:module /c CameraModule.cppm" << std::endl;
    std::cout << "     cl /std:c++20 /experimental:module TemplatedCameraModules.cpp CameraModule.obj" << std::endl;

    std::cout << "\n   Clang 16+:" << std::endl;
    std::cout << "     clang++ -std=c++20 -fmodules -c CameraModule.cppm" << std::endl;
    std::cout << "     clang++ -std=c++20 -fmodules TemplatedCameraModules.cpp CameraModule.o" << std::endl;

    std::cout << "\n  NOTE: C++20 modules support varies by compiler!" << std::endl;
    std::cout << " •   GCC: Experimental support with -fmodules-ts" << std::endl;
```

```
    std::cout << " •   MSVC: Best support with /std:c++20 and /experimental:module" << std::endl;
    std::cout << " •   Clang: Good support in recent versions (16+)" << std::endl;

    std::cout << "\n==================================================================\n" << std::endl;

    return 0;
}
```

72 Source Code: ThreadPoolExamples.cpp

File: `src/ThreadPoolExamples.cpp`
Repository: View on GitHub

```cpp
// ThreadPoolExamples.cpp
// Comprehensive educational examples of thread pool implementations in Modern C++
// From basic to advanced patterns including work stealing, priority queues, and more

#include <iostream>
#include <thread>
#include <vector>
#include <queue>
#include <functional>
#include <mutex>
#include <condition_variable>
#include <future>
#include <atomic>
#include <chrono>
#include <memory>
#include <type_traits>
#include <deque>
#include <optional>
#include <iomanip>

using namespace std::chrono_literals;

// ============================================================================
// SECTION 1: Basic Thread Pool (Simplest Implementation)
// ============================================================================

class BasicThreadPool {
private:
    std::vector<std::thread> workers_;
    std::queue<std::function<void()>> tasks_;
    std::mutex queue_mutex_;
    std::condition_variable condition_;
    bool stop_;

public:
    explicit BasicThreadPool(size_t num_threads) : stop_(false) {
        std::cout << "  Creating thread pool with " << num_threads << " workers\n";

        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this, i]() {
                std::cout << "    Worker " << i << " started (thread "
                          << std::this_thread::get_id() << ")\n";
```

```

                while (true) {
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(queue_mutex_);

                        // Wait until there's work or we're stopping
                        condition_.wait(lock, [this]() {
                            return stop_ || !tasks_.empty();
                        });

                        if (stop_ && tasks_.empty()) {
                            return;
                        }

                        task = std::move(tasks_.front());
                        tasks_.pop();
                    }

                    task();  // Execute the task
                }
            });
        }
    }

    // Submit a task (no return value)
    void submit(std::function<void()> task) {
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);

            if (stop_) {
                throw std::runtime_error("Cannot submit task to stopped thread pool");
            }

            tasks_.push(std::move(task));
        }
        condition_.notify_one();
    }

    ~BasicThreadPool() {
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);
            stop_ = true;
        }

        condition_.notify_all();

        for (std::thread& worker : workers_) {
            if (worker.joinable()) {
                worker.join();
            }
        }
```

```

        std::cout << "  Thread pool destroyed\n";
    }
};

void demonstrate_basic_thread_pool() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 1. Basic Thread Pool ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Concept: Fixed number of worker threads processing tasks
        from a queue\n\n";

    BasicThreadPool pool(3);

    std::cout << "\nSubmitting 6 tasks...\n\n";

    for (int i = 1; i <= 6; ++i) {
        pool.submit([i]() {
            std::cout << "    Task " << i << " executing on thread "
                      << std::this_thread::get_id() << "\n";
            std::this_thread::sleep_for(200ms);
            std::cout << "    Task " << i << " completed\n";
        });
    }

    std::this_thread::sleep_for(2s);  // Wait for tasks to complete

    std::cout << "\ n All tasks processed by 3 workers\n";
    std::cout << "  Tasks automatically distributed across available threads\n
        ";
}

//
    ==========================================================================
// SECTION 2: Thread Pool with Futures (Return Values)
//
    ==========================================================================

class ThreadPoolWithFutures {
private:
    std::vector<std::thread> workers_;
    std::queue<std::function<void()>> tasks_;
    std::mutex queue_mutex_;
    std::condition_variable condition_;
    bool stop_;

public:
    explicit ThreadPoolWithFutures(size_t num_threads) : stop_(false) {
        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this]() {
                while (true) {
```

```
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(queue_mutex_);
                        condition_.wait(lock, [this]() {
                            return stop_ || !tasks_.empty();
                        });

                        if (stop_ && tasks_.empty()) {
                            return;
                        }

                        task = std::move(tasks_.front());
                        tasks_.pop();
                    }

                    task();
                }
            });
        }
    }

    // Submit task and get future for result
    template<typename F, typename... Args>
    auto submit(F&& f, Args&&... args)
        -> std::future<typename std::invoke_result_t<F, Args...>> {

        using return_type = typename std::invoke_result_t<F, Args...>;

        // Create a packaged_task
        auto task = std::make_shared<std::packaged_task<return_type()>>(
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
        );

        std::future<return_type> result = task->get_future();

        {
            std::lock_guard<std::mutex> lock(queue_mutex_);

            if (stop_) {
                throw std::runtime_error("Cannot submit to stopped pool");
            }

            tasks_.emplace([task]() { (*task)(); });
        }

        condition_.notify_one();
        return result;
    }

    ~ThreadPoolWithFutures() {
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);
            stop_ = true;
```

```
        }

        condition_.notify_all();

        for (std::thread& worker : workers_) {
            if (worker.joinable()) {
                worker.join();
            }
        }
    }
};

void demonstrate_thread_pool_with_futures() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 2. Thread Pool with Futures (Return Values) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Benefit: Can return results from tasks using std::future\n\n
        ";

    ThreadPoolWithFutures pool(4);

    std::vector<std::future<int>> results;

    std::cout << "Submitting computation tasks...\n";

    for (int i = 1; i <= 8; ++i) {
        results.push_back(pool.submit([i]() {
            std::cout << "  Task " << i << ": Computing " << i << "²\n";
            std::this_thread::sleep_for(100ms);
            return i * i;
        }));
    }

    std::cout << "\nCollecting results...\n";

    for (size_t i = 0; i < results.size(); ++i) {
        int result = results[i].get();  // Blocks until result is ready
        std::cout << "  Result " << (i + 1) << ": " << result << "\n";
    }

    std::cout << "\ n All tasks returned results via std::future\n";
}

//
    ==========================================================================
// SECTION 3: Thread Pool with Priority Queue
//
    ==========================================================================

enum class TaskPriority {
    LOW = 0,
```

```
    NORMAL = 1,
    HIGH = 2,
    CRITICAL = 3
};

struct PrioritizedTask {
    std::function<void()> func;
    TaskPriority priority;
    int sequence;  // For FIFO within same priority

    bool operator<(const PrioritizedTask& other) const {
        if (priority != other.priority) {
            return priority < other.priority;  // Higher priority first
        }
        return sequence > other.sequence;  // FIFO for same priority
    }
};

class PriorityThreadPool {
private:
    std::vector<std::thread> workers_;
    std::priority_queue<PrioritizedTask> tasks_;
    std::mutex queue_mutex_;
    std::condition_variable condition_;
    bool stop_;
    std::atomic<int> sequence_counter_;

public:
    explicit PriorityThreadPool(size_t num_threads)
        : stop_(false), sequence_counter_(0) {

        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this]() {
                while (true) {
                    PrioritizedTask task{{}, TaskPriority::NORMAL, 0};

                    {
                        std::unique_lock<std::mutex> lock(queue_mutex_);
                        condition_.wait(lock, [this]() {
                            return stop_ || !tasks_.empty();
                        });

                        if (stop_ && tasks_.empty()) {
                            return;
                        }

                        task = tasks_.top();
                        tasks_.pop();
                    }

                    task.func();
                }
            });
        }
```

```
    }

    void submit(std::function<void()> task, TaskPriority priority =
        TaskPriority::NORMAL) {
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);

            if (stop_) {
                throw std::runtime_error("Cannot submit to stopped pool");
            }

            tasks_.push({std::move(task), priority, sequence_counter_++});
        }

        condition_.notify_one();
    }

    ~PriorityThreadPool() {
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);
            stop_ = true;
        }

        condition_.notify_all();

        for (std::thread& worker : workers_) {
            if (worker.joinable()) {
                worker.join();
            }
        }
    }
};

void demonstrate_priority_thread_pool() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 3. Priority Thread Pool ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Concept: Higher priority tasks execute before lower priority
        ones\n\n";

    PriorityThreadPool pool(2);

    std::cout << "Submitting tasks with different priorities...\n\n";

    // Submit in mixed order
    pool.submit([]() {
        std::cout << "  [NORMAL] Task 1\n";
        std::this_thread::sleep_for(100ms);
    }, TaskPriority::NORMAL);

    pool.submit([]() {
        std::cout << "  [CRITICAL] Urgent task!\n";
        std::this_thread::sleep_for(100ms);
```

```
    }, TaskPriority::CRITICAL);

    pool.submit([]() {
        std::cout << "  [LOW] Background task\n";
        std::this_thread::sleep_for(100ms);
    }, TaskPriority::LOW);

    pool.submit([]() {
        std::cout << "  [HIGH] Important task\n";
        std::this_thread::sleep_for(100ms);
    }, TaskPriority::HIGH);

    pool.submit([]() {
        std::cout << "  [NORMAL] Task 2\n";
        std::this_thread::sleep_for(100ms);
    }, TaskPriority::NORMAL);

    std::this_thread::sleep_for(1s);

    std::cout << "\n Tasks executed by priority: CRITICAL → HIGH → NORMAL →
        LOW\n";
}

//
    ==========================================================================
// SECTION 4: Work-Stealing Thread Pool (Advanced)
//
    ==========================================================================

class WorkStealingThreadPool {
private:
    struct WorkerThread {
        std::deque<std::function<void()>> local_queue;
        std::mutex queue_mutex;
        std::thread thread;
    };

    std::vector<std::unique_ptr<WorkerThread>> workers_;
    std::atomic<bool> stop_;
    std::atomic<size_t> next_worker_;  // Round-robin submission

public:
    explicit WorkStealingThreadPool(size_t num_threads)
        : stop_(false), next_worker_(0) {

        std::cout << "  Creating work-stealing pool with " << num_threads << "
            workers\n";

        for (size_t i = 0; i < num_threads; ++i) {
            auto worker = std::make_unique<WorkerThread>();

            worker->thread = std::thread([this, i, &w = *worker]() {
```

```
                std::cout << "    Worker " << i << " started\n";

                while (!stop_) {
                    std::function<void()> task;

                    // Try to get task from own queue first
                    {
                        std::lock_guard<std::mutex> lock(w.queue_mutex);
                        if (!w.local_queue.empty()) {
                            task = std::move(w.local_queue.front());
                            w.local_queue.pop_front();
                        }
                    }

                    // If no local task, try to steal from other workers
                    if (!task) {
                        task = try_steal_work(i);
                    }

                    if (task) {
                        task();
                    } else {
                        std::this_thread::sleep_for(10ms);  // Prevent busy-waiting
                    }
                }
            });

            workers_.push_back(std::move(worker));
        }
    }

    void submit(std::function<void()> task) {
        // Round-robin distribution
        size_t worker_idx = next_worker_++ % workers_.size();

        std::lock_guard<std::mutex> lock(workers_[worker_idx]->queue_mutex);
        workers_[worker_idx]->local_queue.push_back(std::move(task));
    }

    ~WorkStealingThreadPool() {
        stop_ = true;

        for (auto& worker : workers_) {
            if (worker->thread.joinable()) {
                worker->thread.join();
            }
        }

        std::cout << "  Work-stealing pool destroyed\n";
    }

private:
    std::function<void()> try_steal_work(size_t my_index) {
```

```
        // Try to steal from other workers
        for (size_t i = 0; i < workers_.size(); ++i) {
            if (i == my_index) continue;

            std::lock_guard<std::mutex> lock(workers_[i]->queue_mutex);
            if (!workers_[i]->local_queue.empty()) {
                // Steal from the back (different from local FIFO)
                auto task = std::move(workers_[i]->local_queue.back());
                workers_[i]->local_queue.pop_back();
                return task;
            }
        }

        return nullptr;
    }
};

void demonstrate_work_stealing_pool() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 4. Work-Stealing Thread Pool ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Concept: Idle workers steal tasks from busy workers' queues\n";
    std::cout << "Benefit: Better load balancing for uneven task durations\n\n";

    WorkStealingThreadPool pool(4);

    std::cout << "Submitting 12 tasks with varying durations...\n\n";

    for (int i = 1; i <= 12; ++i) {
        pool.submit([i]() {
            int duration = (i % 3 + 1) * 100;  // 100ms, 200ms, or 300ms
            std::cout << "  Task " << i << " (duration: " << duration
                      << "ms) on thread " << std::this_thread::get_id() << "\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(duration));
        });
    }

    std::this_thread::sleep_for(2s);

    std::cout << "\n Work stealing enabled better load distribution\n";
    std::cout << "  Idle workers helped busy workers finish faster\n";
}

// ==========================================================================
// SECTION 5: Dynamic Thread Pool (Auto-scaling)
// ==========================================================================
```

```

class DynamicThreadPool {
private:
    std::vector<std::thread> workers_;
    std::queue<std::function<void()>> tasks_;
    std::mutex queue_mutex_;
    std::condition_variable condition_;
    bool stop_;

    size_t min_threads_;
    size_t max_threads_;
    std::atomic<size_t> active_threads_;
    std::atomic<size_t> idle_threads_;

public:
    DynamicThreadPool(size_t min_threads, size_t max_threads)
        : stop_(false), min_threads_(min_threads), max_threads_(max_threads),
          active_threads_(0), idle_threads_(0) {

        std::cout << "  Creating dynamic pool (min: " << min_threads
                  << ", max: " << max_threads << ")\n";

        // Start with minimum threads
        for (size_t i = 0; i < min_threads_; ++i) {
            add_worker();
        }
    }

    void submit(std::function<void()> task) {
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);

            if (stop_) {
                throw std::runtime_error("Cannot submit to stopped pool");
            }

            tasks_.push(std::move(task));

            // Auto-scale: add worker if all busy and below max
            if (idle_threads_ == 0 && workers_.size() < max_threads_) {
                std::cout << "    Scaling up: adding worker (total: "
                          << (workers_.size() + 1) << ")\n";
                add_worker();
            }
        }

        condition_.notify_one();
    }

    ~DynamicThreadPool() {
        {
            std::lock_guard<std::mutex> lock(queue_mutex_);
            stop_ = true;
        }
```

```

        condition_.notify_all();

        for (std::thread& worker : workers_) {
            if (worker.joinable()) {
                worker.join();
            }
        }

        std::cout << "  Dynamic pool destroyed\n";
    }

private:
    void add_worker() {
        workers_.emplace_back([this]() {
            while (true) {
                std::function<void()> task;

                {
                    std::unique_lock<std::mutex> lock(queue_mutex_);

                    ++idle_threads_;

                    condition_.wait(lock, [this]() {
                        return stop_ || !tasks_.empty();
                    });

                    --idle_threads_;

                    if (stop_ && tasks_.empty()) {
                        return;
                    }

                    if (!tasks_.empty()) {
                        task = std::move(tasks_.front());
                        tasks_.pop();
                    }
                }

                if (task) {
                    ++active_threads_;
                    task();
                    --active_threads_;
                }
            }
        });
    }
};

void demonstrate_dynamic_thread_pool() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 5. Dynamic Thread Pool (Auto-scaling) ===\n";
    std::cout << std::string(70, '=') << "\n\n";

```

```
    std::cout << "Concept: Pool grows/shrinks based on workload\n\n";

    DynamicThreadPool pool(2, 6);

    std::cout << "Phase 1: Light load (2 tasks)\n\n";

    for (int i = 1; i <= 2; ++i) {
        pool.submit([i]() {
            std::cout << "  Light task " << i << "\n";
            std::this_thread::sleep_for(200ms);
        });
    }

    std::this_thread::sleep_for(500ms);

    std::cout << "\nPhase 2: Heavy load (10 tasks)\n\n";

    for (int i = 1; i <= 10; ++i) {
        pool.submit([i]() {
            std::cout << "  Heavy task " << i << "\n";
            std::this_thread::sleep_for(300ms);
        });
        std::this_thread::sleep_for(50ms);  // Gradual submission
    }

    std::this_thread::sleep_for(2s);

    std::cout << "\ n Pool automatically scaled up during heavy load\n";
}

// ==========================================================================
// SECTION 6: Thread Pool Best Practices and Anti-Patterns
// ==========================================================================

void demonstrate_best_practices() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 6. Thread Pool Best Practices ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  BEST PRACTICES:\n";
    std::cout << "        \n\n";

    std::cout << "1. Pool Size Selection:\n";
    std::cout << "   CPU-bound: num_threads   std::thread::hardware_concurrency()\n";
    std::cout << "   I/O-bound: num_threads > CPU cores (2x-4x)\n";
    std::cout << "   Mixed:     Profile and tune based on workload\n\n";

    std::cout << "2. Task Granularity:\n";
    std::cout << "     Tasks should run for at least 1ms (avoid overhead)\n";
```

```
    std::cout << "     Don't submit trivial tasks (e.g., single addition)\n";
    std::cout << "     Batch small operations into larger tasks\n\n";

    std::cout << "3. Exception Handling:\n";
    std::cout << "     Tasks should catch their own exceptions\n";
    std::cout << "     Use std::future to propagate exceptions\n";
    std::cout << "     Uncaught exceptions terminate worker threads\n\n";

    std::cout << "4. Shutdown:\n";
    std::cout << "     Signal stop before destroying pool\n";
    std::cout << "     Join all worker threads in destructor\n";
    std::cout << "     Consider draining pending tasks gracefully\n\n";

    std::cout << "5. Task Dependencies:\n";
    std::cout << "     DEADLOCK RISK: Task waiting for another task in same pool\n";
    std::cout << "     Use separate pools for dependent tasks\n";
    std::cout << "     Or ensure pool size > max dependency depth\n\n";

    std::cout << "  ANTI-PATTERNS:\n";
    std::cout << "        \n\n";

    std::cout << "1. Too many pools: Creates thread explosion\n";
    std::cout << "     One pool per operation type (100+ pools)\n";
    std::cout << "     One global pool or few specialized pools\n\n";

    std::cout << "2. Blocking in tasks:\n";
    std::cout << "     Task calls blocking I/O (starves other tasks)\n";
    std::cout << "     Use async I/O or separate I/O thread pool\n\n";

    std::cout << "3. Long-running tasks:\n";
    std::cout << "     Task runs for minutes/hours (blocks worker)\n";
    std::cout << "     Break into smaller chunks or use dedicated thread\n\n";

    std::cout << "4. Unbounded queue growth:\n";
    std::cout << "     Submit faster than processing (memory exhaustion)\n";
    std::cout << "     Use bounded queue with backpressure\n\n";
}

// ==========================================================================
// SECTION 7: Real-World Example - Image Processing Pipeline
// ==========================================================================

struct Image {
    int id;
    std::string name;
    int processing_stage;  // 0=load, 1=filter, 2=resize, 3=save
};

void demonstrate_real_world_example() {
```

```
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 7. Real-World Example: Image Processing Pipeline ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Pipeline: Load → Filter → Resize → Save\n";
    std::cout << "Strategy: Use thread pool to parallelize each stage\n\n";

    ThreadPoolWithFutures pool(std::thread::hardware_concurrency());

    std::vector<Image> images = {
        {1, "photo1.jpg", 0},
        {2, "photo2.jpg", 0},
        {3, "photo3.jpg", 0},
        {4, "photo4.jpg", 0}
    };

    std::cout << "Processing " << images.size() << " images...\n\n";

    // Stage 1: Load images
    std::vector<std::future<Image>> loaded;
    for (const auto& img : images) {
        loaded.push_back(pool.submit([img]() {
            std::cout << "  [LOAD] " << img.name << "\n";
            std::this_thread::sleep_for(100ms);
            Image result = img;
            result.processing_stage = 1;
            return result;
        }));
    }

    // Stage 2: Apply filters
    std::vector<std::future<Image>> filtered;
    for (auto& future : loaded) {
        Image img = future.get();
        filtered.push_back(pool.submit([img]() {
            std::cout << "  [FILTER] " << img.name << "\n";
            std::this_thread::sleep_for(150ms);
            Image result = img;
            result.processing_stage = 2;
            return result;
        }));
    }

    // Stage 3: Resize
    std::vector<std::future<Image>> resized;
    for (auto& future : filtered) {
        Image img = future.get();
        resized.push_back(pool.submit([img]() {
            std::cout << "  [RESIZE] " << img.name << "\n";
            std::this_thread::sleep_for(100ms);
            Image result = img;
            result.processing_stage = 3;
            return result;
        }));
```

```
    }

    // Stage 4: Save
    std::vector<std::future<void>> saved;
    for (auto& future : resized) {
        Image img = future.get();
        saved.push_back(pool.submit([img]() {
            std::cout << "  [SAVE] " << img.name << " → output/" << img.name << "\n";
            std::this_thread::sleep_for(80ms);
        }));
    }

    // Wait for all to complete
    for (auto& future : saved) {
        future.get();
    }

    std::cout << "\n Pipeline completed for all images\n";
    std::cout << "  Each stage parallelized using thread pool\n";
}

// ============================================================================
// SECTION 8: Comparison - When to Use What
// ============================================================================

void demonstrate_comparison() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "=== 8. Thread Pool Comparison - When to Use What ===\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "                                        \n";
    std::cout << "  Pool Type             Use Case              Pros/Cons                \n";
    std::cout << "                                        \n";
    std::cout << "  Basic                 Simple tasks          + Easy to implement  \n";
    std::cout << "                        Fire-and-forget       - No return values   \n";
    std::cout << "                                        \n";
    std::cout << "  With Futures          Computational work    + Return values      \n";
    std::cout << "                        Need results          + Type-safe          \n";
    std::cout << "                                        \n";
    std::cout << "  Priority              Mixed importance      + Predictable order  \n";
    std::cout << "                        Real-time systems     - More overhead      \n";
    std::cout << "                                        \n";
```

```
    std::cout << "  Work-Stealing         Uneven task sizes     + Better load balance\n";
    std::cout << "                        Recursive tasks       - More complex \n";
    std::cout << "                                              \n";
    std::cout << "  Dynamic               Variable workload     + Resource efficient  \n";
    std::cout << "                        Bursty traffic        - Scale-up latency    \n";
    std::cout << "                                              \n\n";

    std::cout << "ALTERNATIVES TO THREAD POOLS:\n";
    std::cout << "                \n\n";

    std::cout << "1. std::async:\n";
    std::cout << "   Use when: One-off async operations, automatic thread management\n";
    std::cout << "   Avoid when: High frequency task submission (creates many threads)\n\n";

    std::cout << "2. std::thread:\n";
    std::cout << "   Use when: Long-running background tasks, dedicated worker \n";
    std::cout << "   Avoid when: Many short-lived tasks (thread creation overhead)\n\n";

    std::cout << "3. ASIO/Boost.Asio:\n";
    std::cout << "   Use when: I/O-bound operations, async networking\n";
    std::cout << "   Avoid when: CPU-bound computations\n\n";

    std::cout << "4. C++20 Coroutines:\n";
    std::cout << "   Use when: Async I/O, cooperative multitasking\n";
    std::cout << "   Avoid when: Heavy CPU computations\n\n";

    std::cout << "5. TBB (Threading Building Blocks):\n";
    std::cout << "   Use when: Need mature, production-ready parallel algorithms\n";
    std::cout << "   Benefit: Work-stealing, parallel_for, parallel_reduce, etc.\n\n";
}

// ==========================================================================
// MAIN FUNCTION
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "            Thread Pool Implementations - Educational Examples        \n";
```

```
    std::cout << "                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                
    std::cout << "   From basic to advanced: Complete guide to C++ thread pools        \n";
    std::cout << "                                   \n";

    std::cout << "\nSystem Info: " << std::thread::hardware_concurrency()
              << " hardware threads available\n";

    demonstrate_basic_thread_pool();
    demonstrate_thread_pool_with_futures();
    demonstrate_priority_thread_pool();
    demonstrate_work_stealing_pool();
    demonstrate_dynamic_thread_pool();
    demonstrate_best_practices();
    demonstrate_real_world_example();
    demonstrate_comparison();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "All thread pool demonstrations completed!\n";
    std::cout << "\nKEY TAKEAWAYS:\n";
    std::cout << "  1. Thread pools reuse threads → avoid creation overhead\n";
    std::cout << "  2. Choose pool type based on workload characteristics\n";
    std::cout << "  3. Use futures for tasks that return values\n";
    std::cout << "  4. Profile before optimizing thread count\n";
    std::cout << "  5. Watch for deadlocks with task dependencies\n";
    std::cout << std::string(70, '=') << "\n\n";

    return 0;
}
```

73 Source Code: TuplesAndStructuredBindings.cpp

File: `src/TuplesAndStructuredBindings.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// COMPREHENSIVE TUPLES AND STRUCTURED BINDINGS
// ==================================================================
// This example demonstrates modern C++ tuple features and best practices.
//
// TOPICS COVERED:
// 1. Basic tuple operations (C++11)
// 2. std::tie and unpacking
// 3. Structured bindings (C++17)
// 4. std::apply (C++17)
// 5. Tuple comparison and algorithms
// 6. Tuple concatenation
// 7. std::tuple_cat
// 8. Real-world use cases
// 9. Performance considerations
// 10. When to use tuples vs structs
// ==================================================================

#include <iostream>
#include <string>
#include <tuple>
#include <vector>
#include <map>
#include <algorithm>
#include <numeric>

// ==================================================================
// SECTION 1: BASIC TUPLE OPERATIONS
// ==================================================================

void demonstrate_basic_tuples() {
    std::cout << "\n=== 1. BASIC TUPLE OPERATIONS ===" << std::endl;

    // Creating tuples
    std::tuple<int, std::string, double> person1{25, "Alice", 5.6};
    auto person2 = std::make_tuple(30, "Bob", 6.0);

    // Accessing elements with std::get
    std::cout << "\nPerson 1: " << std::get<0>(person1) << " years, "
              << std::get<1>(person1) << ", "
              << std::get<2>(person1) << "ft" << std::endl;

    // Accessing by type (C++14) - only if type is unique!
    std::cout << "Name: " << std::get<std::string>(person1) << std::endl;

    // Modifying elements
    std::get<0>(person1) = 26;
    std::cout << "After birthday: " << std::get<0>(person1) << " years" << std
        ::endl;
```

```

    // Tuple size
    std::cout << "\nTuple size: " << std::tuple_size<decltype(person1)>::value << std::endl;

    std::cout << "\n KEY POINTS:" << std::endl;
    std::cout << " •   std::get<N>(tuple) - access by index (compile-time)" << std::endl;
    std::cout << " •   std::get<Type>(tuple) - access by type (if unique)" << std::endl;
    std::cout << " •   Zero runtime overhead vs struct" << std::endl;
}

// ==================================================================
// SECTION 2: std::tie AND UNPACKING
// ==================================================================

std::tuple<int, int, int> divide_full(int dividend, int divisor) {
    return {dividend / divisor, dividend % divisor, dividend};
}

void demonstrate_tie() {
    std::cout << "\n=== 2. std::tie AND UNPACKING ===" << std::endl;

    // Old way: extract tuple elements
    auto result = divide_full(17, 5);
    int quotient = std::get<0>(result);
    int remainder = std::get<1>(result);
    int original = std::get<2>(result);

    std::cout << "\nOld way: 17 / 5 = " << quotient
              << " remainder " << remainder << std::endl;

    // Better: std::tie (C++11)
    int q, r, o;
    std::tie(q, r, o) = divide_full(17, 5);
    std::cout << "With tie: 17 / 5 = " << q << " remainder " << r << std::endl;

    // Ignore some values with std::ignore
    std::tie(q, std::ignore, std::ignore) = divide_full(20, 3);
    std::cout << "Quotient only: 20 / 3 = " << q << std::endl;

    // Using tie for swap
    int a = 10, b = 20;
    std::cout << "\nBefore swap: a=" << a << ", b=" << b << std::endl;
    std::tie(a, b) = std::tie(b, a);
    std::cout << "After swap: a=" << a << ", b=" << b << std::endl;

    std::cout << "\n std::tie benefits:" << std::endl;
    std::cout << " •   Unpack multiple return values" << std::endl;
    std::cout << " •   Use std::ignore for unwanted values" << std::endl;
    std::cout << " •   Create reference tuples" << std::endl;
}
```

```

// ==================================================================
// SECTION 3: STRUCTURED BINDINGS (C++17)
// ==================================================================

struct Sensor {
    int id;
    double temperature;
    double humidity;
    std::string location;
};

std::tuple<double, double, double> compute_statistics(const std::vector<double
    >& values) {
    double sum = std::accumulate(values.begin(), values.end(), 0.0);
    double mean = sum / values.size();
    double min = *std::min_element(values.begin(), values.end());
    double max = *std::max_element(values.begin(), values.end());
    return {mean, min, max};
}

void demonstrate_structured_bindings() {
    std::cout << "\n=== 3. STRUCTURED BINDINGS (C++17) ===" << std::endl;

    // Structured bindings - most modern and readable!
    auto [quotient, remainder, original] = divide_full(25, 7);
    std::cout << "\n25 / 7 = " << quotient << " remainder " << remainder <<
        std::endl;

    // Works with pairs
    std::map<std::string, int> scores{{"Alice", 95}, {"Bob", 87}};
    for (const auto& [name, score] : scores) {
        std::cout << name << ": " << score << " points" << std::endl;
    }

    // Works with structs
    Sensor sensor{101, 22.5, 65.0, "Lab-A"};
    auto [id, temp, humid, loc] = sensor;
    std::cout << "\nSensor " << id << " at " << loc
              << ": " << temp << "°C, " << humid << "% humidity" << std::endl;

    // With functions
    std::vector<double> temps{20.1, 22.5, 21.8, 23.0, 19.5};
    auto [mean, min, max] = compute_statistics(temps);
    std::cout << "\nTemperature stats: mean=" << mean
              << ", min=" << min << ", max=" << max << std::endl;

    std::cout << "\n BEST PRACTICE:" << std::endl;
    std::cout << " •   Use structured bindings (C++17) over std::tie" << std::
        endl;
    std::cout << " •   More readable, less verbose" << std::endl;
    std::cout << " •   Works with tuples, pairs, structs, arrays" << std::endl
        ;
}
```

```

// ==================================================================
// SECTION 4: std::apply (C++17)
// ==================================================================

int add_three(int a, int b, int c) {
    return a + b + c;
}

double multiply(double x, double y, double z) {
    return x * y * z;
}

void demonstrate_apply() {
    std::cout << "\n=== 4. std::apply (C++17) ===" << std::endl;

    // std::apply - call function with tuple as arguments
    std::tuple<int, int, int> args{10, 20, 30};
    int sum = std::apply(add_three, args);
    std::cout << "\nsum of (10, 20, 30) = " << sum << std::endl;

    std::tuple<double, double, double> factors{2.0, 3.0, 4.0};
    double product = std::apply(multiply, factors);
    std::cout << "product of (2.0, 3.0, 4.0) = " << product << std::endl;

    // With lambdas
    auto printer = [](auto... args) {
        std::cout << "Values: ";
        ((std::cout << args << " "), ...);
        std::cout << std::endl;
    };

    std::apply(printer, std::make_tuple(1, "hello", 3.14, 'X'));

    std::cout << "\n std::apply use cases:" << std::endl;
    std::cout << " •   Call function with tuple arguments" << std::endl;
    std::cout << " •   Forward arguments stored in tuple" << std::endl;
    std::cout << " •   Unpack tuple into function call" << std::endl;
}

// ==================================================================
// SECTION 5: TUPLE COMPARISON AND ALGORITHMS
// ==================================================================

void demonstrate_tuple_comparison() {
    std::cout << "\n=== 5. TUPLE COMPARISON ===" << std::endl;

    std::tuple<int, std::string> t1{1, "apple"};
    std::tuple<int, std::string> t2{1, "banana"};
    std::tuple<int, std::string> t3{2, "apple"};

    // Lexicographic comparison (left to right)
    std::cout << "\nComparisons:" << std::endl;
    std::cout << "   (1, 'apple') < (1, 'banana'): "
```

```
              << std::boolalpha << (t1 < t2) << std::endl;
    std::cout << "   (1, 'apple') < (2, 'apple'): " << (t1 < t3) << std::endl;
    std::cout << "   (1, 'banana') < (2, 'apple'): " << (t2 < t3) << std::endl
        ;

    // Sorting with tuples
    std::vector<std::tuple<int, std::string, double>> students{
        {85, "Charlie", 3.2},
        {92, "Alice", 3.8},
        {85, "Bob", 3.5},
        {92, "David", 3.6}
    };

    std::cout << "\nBefore sort:" << std::endl;
    for (const auto& [score, name, gpa] : students) {
        std::cout << "   " << name << ": " << score << ", GPA " << gpa << std
            ::endl;
    }

    // Sort by tuple (score desc, then name asc)
    std::sort(students.begin(), students.end(),
              [](const auto& a, const auto& b) {
                  return std::tie(std::get<0>(b), std::get<1>(a)) <
                         std::tie(std::get<0>(a), std::get<1>(b));
              });

    std::cout << "\nAfter sort (score desc, name asc):" << std::endl;
    for (const auto& [score, name, gpa] : students) {
        std::cout << "   " << name << ": " << score << ", GPA " << gpa << std
            ::endl;
    }

    std::cout << "\ n Tuple comparison:" << std::endl;
    std::cout << " •   Lexicographic (element by element)" << std::endl;
    std::cout << " •   Perfect for sorting by multiple keys" << std::endl;
    std::cout << " •   Use std::tie for custom comparison" << std::endl;
}

// ==================================================================
// SECTION 6: std::tuple_cat AND CONCATENATION
// ==================================================================

void demonstrate_tuple_cat() {
    std::cout << "\n=== 6. std::tuple_cat ===" << std::endl;

    std::tuple<int, std::string> t1{42, "answer"};
    std::tuple<double, char> t2{3.14, 'X'};

    // Concatenate tuples
    auto combined = std::tuple_cat(t1, t2);
    auto [num, text, pi, letter] = combined;

    std::cout << "\nCombined tuple: " << num << ", " << text
              << ", " << pi << ", " << letter << std::endl;
```

```

    // Combine multiple tuples
    std::tuple<int> id{100};
    std::tuple<std::string, std::string> name{"John", "Doe"};
    std::tuple<int> age{30};

    auto person = std::tuple_cat(id, name, age);
    std::cout << "\nPerson: ID=" << std::get<0>(person)
              << ", Name=" << std::get<1>(person) << " " << std::get<2>(person
                )
              << ", Age=" << std::get<3>(person) << std::endl;

    std::cout << "\n std::tuple_cat:" << std::endl;
    std::cout << " •   Concatenate multiple tuples" << std::endl;
    std::cout << " •   All done at compile-time" << std::endl;
    std::cout << " •   Zero runtime overhead" << std::endl;
}

// ================================================================
// SECTION 7: REAL-WORLD USE CASE - PARTICLE SYSTEM
// ================================================================

using State = std::tuple<double, double, double, double, double>;
using ParticleWeight = double;
using Particle = std::map<ParticleWeight, State>;
using Particles = std::vector<Particle>;

double compute_total_weight(const Particles& particles) {
    double total = 0.0;
    for (const auto& particle : particles) {
        for (const auto& [weight, state] : particle) {
            total += weight;
        }
    }
    return total;
}

void print_particle_state(const State& state) {
    auto [x, y, z, vx, vy] = state;
    std::cout << "   Position(" << x << "," << y << "," << z << ") "
              << "Velocity(" << vx << "," << vy << ")" << std::endl;
}

void demonstrate_particle_system() {
    std::cout << "\n=== 7. REAL-WORLD: PARTICLE SYSTEM ===" << std::endl;

    Particles particles;

    // Create particles with weighted states
    for (int i = 0; i < 4; i++) {
        State state = std::make_tuple(i*1.0, i*1.1, i*1.2, i*0.5, i*0.6);
        Particle particle;
        particle.emplace(5.0 + i, state);
        particles.push_back(particle);
```

```
    }

    std::cout << "\nParticle states:" << std::endl;
    for (size_t i = 0; i < particles.size(); i++) {
        for (const auto& [weight, state] : particles[i]) {
            std::cout << "Particle " << i << " (weight=" << weight << "):" << std::endl;
            print_particle_state(state);
        }
    }

    double total_weight = compute_total_weight(particles);
    std::cout << "\nTotal weight: " << total_weight << std::endl;

    std::cout << "\n Why use tuples here:" << std::endl;
    std::cout << " •   Simple data aggregation" << std::endl;
    std::cout << " •   No need for named struct" << std::endl;
    std::cout << " •   Easy to decompose with structured bindings" << std::endl;
}

// ==================================================================
// SECTION 8: TUPLE VS STRUCT - WHEN TO USE EACH
// ==================================================================

void explain_tuple_vs_struct() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "TUPLE VS STRUCT - WHEN TO USE EACH:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n USE TUPLES WHEN:" << std::endl;
    std::cout << "\n1. RETURNING MULTIPLE VALUES:" << std::endl;
    std::cout << "     std::tuple<int, int> divide(int a, int b) { ... }" << std::endl;
    std::cout << " •   Quick and easy multiple returns" << std::endl;
    std::cout << " •   Use structured bindings to unpack" << std::endl;

    std::cout << "\n2. TEMPORARY DATA GROUPING:" << std::endl;
    std::cout << " •   Local scope only" << std::endl;
    std::cout << " •   No need for named type" << std::endl;
    std::cout << " •   One-off data combinations" << std::endl;

    std::cout << "\n3. GENERIC PROGRAMMING:" << std::endl;
    std::cout << " •   Template metaprogramming" << std::endl;
    std::cout << " •   Variadic template arguments" << std::endl;
    std::cout << " •   Type computations" << std::endl;

    std::cout << "\n4. QUICK PROTOTYPING:" << std::endl;
    std::cout << " •   Experimenting with data structures" << std::endl;
    std::cout << " •   Before defining proper types" << std::endl;

    std::cout << "\n USE STRUCTS WHEN:" << std::endl;
    std::cout << "\n1. NAMED SEMANTICS MATTER:" << std::endl;
    std::cout << "     struct Person { int age; string name; double height; };
```

```
        " << std::endl;
    std::cout << " •   Clear what each field means" << std::endl;
    std::cout << " •   Self-documenting code" << std::endl;

    std::cout << "\n2. MEMBER FUNCTIONS NEEDED:" << std::endl;
    std::cout << " •   Methods to manipulate data" << std::endl;
    std::cout << " •   Encapsulation" << std::endl;
    std::cout << " •   Invariants to maintain" << std::endl;

    std::cout << "\n3. PUBLIC API:" << std::endl;
    std::cout << " •   Library interfaces" << std::endl;
    std::cout << " •   Long-term maintainability" << std::endl;
    std::cout << " •   Clear documentation" << std::endl;

    std::cout << "\n4. COMPLEX TYPES:" << std::endl;
    std::cout << " •   Many fields (>5)" << std::endl;
    std::cout << " •   Constructors needed" << std::endl;
    std::cout << " •   Relationships with other types" << std::endl;

    std::cout << "\n ANTI-PATTERNS:" << std::endl;
    std::cout << "\n DON'T: tuple<int, int, int, int, int, int, int>" << std
        ::endl;
    std::cout << "   Too many fields - use struct!" << std::endl;
    std::cout << "\n DON'T: Use tuples in public API" << std::endl;
    std::cout << "   Hard to understand: get<2>(result) means what?" << std::
        endl;
    std::cout << "\n DON'T: Store tuples long-term" << std::endl;
    std::cout << "   Use proper named types for persistence" << std::endl;

    std::cout << "\n GOLDEN RULE:" << std::endl;
    std::cout << "   Tuples for quick local returns," << std::endl;
    std::cout << "   Structs for everything else!" << std::endl;
}

// ==================================================================
// SECTION 9: PERFORMANCE CONSIDERATIONS
// ==================================================================

struct Point3D {
    double x, y, z;
};

void demonstrate_performance() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "PERFORMANCE CONSIDERATIONS:\n";
    std::cout << std::string(70, '=') << std::endl;

    using TuplePoint = std::tuple<double, double, double>;

    std::cout << "\n MEMORY LAYOUT:" << std::endl;
    std::cout << "   struct Point3D:  " << sizeof(Point3D) << " bytes" << std
        ::endl;
    std::cout << "   tuple<d,d,d>:    " << sizeof(TuplePoint) << " bytes" <<
        std::endl;
```

```
    std::cout << "     SAME SIZE - no overhead!" << std::endl;

    std::cout << "\n RUNTIME PERFORMANCE:" << std::endl;
    std::cout << "   Struct: point.x (direct access)" << std::endl;
    std::cout << "   Tuple:  std::get<0>(point) (compile-time index)" << std::
        endl;
    std::cout << "     SAME SPEED - both inline to same code!" << std::endl;

    std::cout << "\n COMPILATION TIME:" << std::endl;
    std::cout << "   Struct: Fast - simple type" << std::endl;
    std::cout << "   Tuple:  Slower - template instantiation" << std::endl;
    std::cout << "     Tuples increase compile time slightly" << std::endl;

    std::cout << "\n PERFORMANCE TIPS:" << std::endl;
    std::cout << " •   Tuples have ZERO runtime overhead" << std::endl;
    std::cout << " •   std::get<N> is compile-time constant" << std::endl;
    std::cout << " •   Use references to avoid copies" << std::endl;
    std::cout << " •   Move semantics work perfectly" << std::endl;
}

// ==================================================================
// SECTION 10: ADVANCED TECHNIQUES
// ==================================================================

// Tuple element type at index
template<size_t I, typename Tuple>
using tuple_element_t = typename std::tuple_element<I, Tuple>::type;

void demonstrate_advanced() {
    std::cout << "\n=== 10. ADVANCED TECHNIQUES ===" << std::endl;

    // Type introspection
    using MyTuple = std::tuple<int, double, std::string>;

    std::cout << "\nTuple type information:" << std::endl;
    std::cout << "   Size: " << std::tuple_size_v<MyTuple> << std::endl;

    // Get element types
    using FirstType = tuple_element_t<0, MyTuple>;
    using SecondType = tuple_element_t<1, MyTuple>;

    std::cout << "   First element is int: "
              << std::boolalpha << std::is_same_v<FirstType, int> << std::endl
                ;
    std::cout << "   Second element is double: "
              << std::is_same_v<SecondType, double> << std::endl;

    // Forward as tuple (perfect forwarding)
    auto forward_example = [](auto&&... args) {
        auto tuple = std::forward_as_tuple(args...);
        std::cout << "\n   Created reference tuple of size "
                  << std::tuple_size_v<decltype(tuple)> << std::endl;
    };

```

```
    int x = 10;
    double y = 20.5;
    forward_example(x, y, std::string("test"));

    std::cout << "\n Advanced features:" << std::endl;
    std::cout << " •   std::tuple_element - get element type" << std::endl;
    std::cout << " •   std::tuple_size - get tuple size" << std::endl;
    std::cout << " •   std::forward_as_tuple - create reference tuple" << std::endl;
    std::cout << " •   Perfect for template metaprogramming" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "          COMPREHENSIVE TUPLES AND STRUCTURED BINDINGS                    \n";
    std::cout << "                                                                          \n";
    std::cout << "   Modern C++ techniques for heterogeneous data                           \n";
    std::cout << "                                    \n";

    demonstrate_basic_tuples();
    demonstrate_tie();
    demonstrate_structured_bindings();
    demonstrate_apply();
    demonstrate_tuple_comparison();
    demonstrate_tuple_cat();
    demonstrate_particle_system();
    explain_tuple_vs_struct();
    demonstrate_performance();
    demonstrate_advanced();

    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "SUMMARY:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n KEY TAKEAWAYS:" << std::endl;
    std::cout << "\n1. MODERN SYNTAX:" << std::endl;
    std::cout << " •   Prefer structured bindings (C++17) over std::tie" << std::endl;
    std::cout << " •   Use std::apply for tuple-to-function unpacking" << std::endl;
    std::cout << " •   auto [a, b, c] = func(); // Most readable!" << std::endl;

    std::cout << "\n2. WHEN TO USE:" << std::endl;
    std::cout << "     Multiple return values" << std::endl;
    std::cout << "     Temporary data grouping" << std::endl;
```

```
    std::cout << "     Generic programming" << std::endl;
    std::cout << "     Public APIs (use structs)" << std::endl;
    std::cout << "     Many fields (>5) (use structs)" << std::endl;

    std::cout << "\n3. PERFORMANCE:" << std::endl;
    std::cout << " •   Zero runtime overhead vs struct" << std::endl;
    std::cout << " •   std::get<N> is compile-time" << std::endl;
    std::cout << " •   Slightly slower compilation" << std::endl;

    std::cout << "\n4. BEST PRACTICES:" << std::endl;
    std::cout << " •   Keep tuples small (<= 5 elements)" << std::endl;
    std::cout << " •   Use for local/temporary data" << std::endl;
    std::cout << " •   Switch to struct when semantics matter" << std::endl;
    std::cout << " •   Document what tuple fields mean!" << std::endl;

    std::cout << "\n Tuples: Quick and powerful for the right use cases!\n"
        << std::endl;

    return 0;
}
```

74 Source Code: VariadicTemplateRecursion.cpp

File: `src/VariadicTemplateRecursion.cpp`
Repository: View on GitHub

```cpp
// ==================================================================
// VARIADIC TEMPLATES WITH COMPILE-TIME RECURSION
// ==================================================================
// This example demonstrates how variadic templates enable compile-time
// recursion and metaprogramming, eliminating runtime overhead.
//
// TOPICS COVERED:
// 1. Basic variadic template recursion
// 2. Compile-time computation (constexpr)
// 3. Type manipulation at compile-time
// 4. Fold expressions (C++17)
// 5. Practical use cases
// 6. Embedded systems applications
//
// KEY CONCEPT: Template recursion happens during compilation,
// resulting in ZERO runtime cost - perfect for embedded systems!
// ==================================================================

#include <iostream>
#include <string>
#include <type_traits>
#include <array>
#include <tuple>
#include <cstdint>
#include <limits>
#include <bitset>

// ==================================================================
// SECTION 1: BASIC VARIADIC RECURSION - COMPILE-TIME
// ==================================================================

// Base case: no arguments
void print_recursive() {
    std::cout << std::endl;
}

// Recursive case: process first argument, recurse on rest
template<typename First, typename... Rest>
void print_recursive(First first, Rest... rest) {
    std::cout << first << " ";
    print_recursive(rest...);  // Compile-time recursion
}

void demonstrate_basic_recursion() {
    std::cout << "\n=== 1. BASIC VARIADIC RECURSION ===" << std::endl;

    std::cout << "Printing: ";
    print_recursive(1, "hello", 3.14, 'X', "world");

```

```
    std::cout << "\n KEY POINT:" << std::endl;
    std::cout << " •   Compiler generates 5 functions at compile-time" << std::endl;
    std::cout << " •   Each function handles one argument type" << std::endl;
    std::cout << " •   Zero runtime overhead - all resolved at compile-time" << std::endl;
    std::cout << " •   No loops, no vtables, no dynamic dispatch" << std::endl;
}

// ==================================================================
// SECTION 2: COMPILE-TIME COMPUTATION - SUM
// ==================================================================

// Base case
constexpr int sum() {
    return 0;
}

// Recursive case
template<typename First, typename... Rest>
constexpr auto sum(First first, Rest... rest) {
    return first + sum(rest...);
}

// Compile-time maximum
constexpr int max_value() {
    return -2147483648;  // INT_MIN
}

template<typename First, typename... Rest>
constexpr auto max_value(First first, Rest... rest) {
    auto rest_max = max_value(rest...);
    return (first > rest_max) ? first : rest_max;
}

void demonstrate_compile_time_computation() {
    std::cout << "\n=== 2. COMPILE-TIME COMPUTATION ===" << std::endl;

    // These are computed at COMPILE TIME!
    constexpr int total = sum(1, 2, 3, 4, 5);
    constexpr int max_val = max_value(10, 25, 15, 30, 5);

    std::cout << "sum(1,2,3,4,5) = " << total << " (computed at compile-time!)" << std::endl;
    std::cout << "max(10,25,15,30,5) = " << max_val << " (computed at compile-time!)" << std::endl;

    std::cout << "\n COST ANALYSIS:" << std::endl;
    std::cout << "   Runtime cost: ZERO" << std::endl;
    std::cout << "   Binary size: Just the constant values" << std::endl;
    std::cout << "   CPU cycles: ZERO (values are literals in binary)" << std::endl;

```

```
    // Verify with static_assert (compile-time only!)
    static_assert(sum(1, 2, 3, 4, 5) == 15, "Sum should be 15");
    static_assert(max_value(10, 25, 15, 30, 5) == 30, "Max should be 30");
    std::cout << "     static_assert passed - verified at compile-time!" << std::endl;
}

// ==================================================================
// SECTION 3: TYPE MANIPULATION - COMPILE-TIME
// ==================================================================

// Check if all types are integral
template<typename... Types>
struct all_integral;

// Base case: empty parameter pack
template<>
struct all_integral<> : std::true_type {};

// Recursive case
template<typename First, typename... Rest>
struct all_integral<First, Rest...> {
    static constexpr bool value =
        std::is_integral_v<First> && all_integral<Rest...>::value;
};

// Helper variable template (C++17)
template<typename... Types>
inline constexpr bool all_integral_v = all_integral<Types...>::value;

// Check if all types are the same
template<typename... Types>
struct all_same;

template<typename T>
struct all_same<T> : std::true_type {};

template<typename First, typename Second, typename... Rest>
struct all_same<First, Second, Rest...> {
    static constexpr bool value =
        std::is_same_v<First, Second> && all_same<Second, Rest...>::value;
};

template<typename... Types>
inline constexpr bool all_same_v = all_same<Types...>::value;

// Get size of largest type
template<typename... Types>
struct max_sizeof;

template<typename T>
struct max_sizeof<T> {
    static constexpr size_t value = sizeof(T);
};
```

```cpp

template<typename First, typename... Rest>
struct max_sizeof<First, Rest...> {
    static constexpr size_t value =
        (sizeof(First) > max_sizeof<Rest...>::value)
            ? sizeof(First)
            : max_sizeof<Rest...>::value;
};

template<typename... Types>
inline constexpr size_t max_sizeof_v = max_sizeof<Types...>::value;

void demonstrate_type_manipulation() {
    std::cout << "\n=== 3. TYPE MANIPULATION AT COMPILE-TIME ===" << std::endl;

    std::cout << "\nall_integral_v:" << std::endl;
    std::cout << "   <int, long, short> = "
              << std::boolalpha << all_integral_v<int, long, short> << std::endl;
    std::cout << "   <int, double, char> = "
              << all_integral_v<int, double, char> << std::endl;

    std::cout << "\nall_same_v:" << std::endl;
    std::cout << "   <int, int, int> = "
              << all_same_v<int, int, int> << std::endl;
    std::cout << "   <int, long, int> = "
              << all_same_v<int, long, int> << std::endl;

    std::cout << "\nmax_sizeof_v:" << std::endl;
    std::cout << "   <char, int, long> = "
              << max_sizeof_v<char, int, long> << " bytes" << std::endl;
    std::cout << "   <double, int, float> = "
              << max_sizeof_v<double, int, float> << " bytes" << std::endl;

    std::cout << "\n ALL COMPUTED AT COMPILE-TIME!" << std::endl;
    std::cout << "   No runtime type checking needed" << std::endl;
    std::cout << "   Perfect for template constraints" << std::endl;
}

// ==================================================================
// SECTION 4: FOLD EXPRESSIONS (C++17) - SIMPLIFIED RECURSION
// ==================================================================

// Old way: explicit recursion
template<typename... Args>
constexpr auto sum_old_way(Args... args) {
    return (args + ...);  // Fold expression!
}

// Fold expressions for logical operations
template<typename... Args>
constexpr bool all_positive(Args... args) {
    return ((args > 0) && ...);  // Fold with &&
```

```
}

template<typename... Args>
constexpr bool any_negative(Args... args) {
    return ((args < 0) || ...);  // Fold with ||
}

// Fold for comma operator (call function for each)
template<typename... Args>
void print_all_fold(Args... args) {
    ((std::cout << args << " "), ...);
    std::cout << std::endl;
}

void demonstrate_fold_expressions() {
    std::cout << "\n=== 4. FOLD EXPRESSIONS (C++17) ===" << std::endl;

    std::cout << "\nArithmetic folds:" << std::endl;
    std::cout << "   sum(1,2,3,4,5) = " << sum_old_way(1, 2, 3, 4, 5) << std::
        endl;

    std::cout << "\nLogical folds:" << std::endl;
    std::cout << "   all_positive(1,2,3) = "
              << std::boolalpha << all_positive(1, 2, 3) << std::endl;
    std::cout << "   all_positive(1,-2,3) = "
              << all_positive(1, -2, 3) << std::endl;
    std::cout << "   any_negative(1,2,3) = "
              << any_negative(1, 2, 3) << std::endl;
    std::cout << "   any_negative(1,-2,3) = "
              << any_negative(1, -2, 3) << std::endl;

    std::cout << "\nPrint with fold: ";
    print_all_fold(1, "hello", 3.14, "world");

    std::cout << "\n FOLD EXPRESSIONS:" << std::endl;
    std::cout << " •   Simpler than explicit recursion" << std::endl;
    std::cout << " •   Still compile-time only" << std::endl;
    std::cout << " •   More readable code" << std::endl;
    std::cout << " •   Zero runtime cost" << std::endl;
}

// ==================================================================
// SECTION 5: PRACTICAL USE CASE - TYPE-SAFE PRINTF
// ==================================================================

// Validate format string at compile-time
template<typename... Args>
void safe_printf(const char* format, Args... args) {
    // Count format specifiers
    int format_count = 0;
    for (const char* p = format; *p; ++p) {
        if (*p == '%' && *(p+1) != '%') {
            ++format_count;
        }
```

```
    }

    int arg_count = sizeof...(args);

    if (format_count != arg_count) {
        std::cout << "ERROR: Format string has " << format_count
                  << " specifiers but " << arg_count << " arguments!" << std::endl;
        return;
    }

    printf(format, args...);
}

void demonstrate_safe_printf() {
    std::cout << "\n=== 5. TYPE-SAFE PRINTF ===" << std::endl;

    std::cout << "\n Correct usage:" << std::endl;
    safe_printf("Integer: %d, String: %s, Float: %.2f\n", 42, "hello", 3.14);

    std::cout << "\n Incorrect usage (caught at runtime):" << std::endl;
    safe_printf("Two specifiers: %d %s\n", 42);  // Missing argument!

    std::cout << "\n BETTER APPROACH:" << std::endl;
    std::cout << "   Use C++20 std::format for complete type safety!" << std::endl;
    std::cout << "   Or compile-time format string validation" << std::endl;
}

// ==================================================================
// SECTION 6: COMPILE-TIME ARRAY INITIALIZATION
// ==================================================================

// Create array with compile-time computed values
template<typename... Args>
constexpr auto make_array(Args... args) {
    return std::array<std::common_type_t<Args...>, sizeof...(Args)>{args...};
}

// Fibonacci at compile-time
constexpr int fib(int n) {
    return (n <= 1) ? n : fib(n-1) + fib(n-2);
}

// Generate array of first N fibonacci numbers
template<size_t... Is>
constexpr auto fib_array_impl(std::index_sequence<Is...>) {
    return std::array<int, sizeof...(Is)>{fib(Is)...};
}

template<size_t N>
constexpr auto fib_array() {
    return fib_array_impl(std::make_index_sequence<N>{});
}
```

```cpp

void demonstrate_compile_time_array() {
    std::cout << "\n=== 6. COMPILE-TIME ARRAY INITIALIZATION ===" << std::endl;

    // Array created at compile-time!
    constexpr auto arr = make_array(10, 20, 30, 40, 50);
    std::cout << "\nArray: ";
    for (auto val : arr) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Fibonacci array computed at compile-time!
    constexpr auto fibs = fib_array<10>();
    std::cout << "\nFirst 10 Fibonacci numbers (compile-time): ";
    for (auto val : fibs) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    std::cout << "\ n PERFORMANCE:" << std::endl;
    std::cout << " •   Arrays are in .rodata section (read-only)" << std::endl;
    std::cout << " •   No initialization code at runtime" << std::endl;
    std::cout << " •   Values embedded in binary" << std::endl;
}

// ================================================================
// SECTION 7: EMBEDDED SYSTEMS - REGISTER CONFIGURATION
// ================================================================

// Compile-time register bit field computation
template<uint32_t... Bits>
struct RegisterBits {
    static constexpr uint32_t value = ((1u << Bits) | ...);
};

// Compile-time register configuration
enum class GPIO_Pin : uint8_t {
    Pin0 = 0, Pin1 = 1, Pin2 = 2, Pin3 = 3,
    Pin4 = 4, Pin5 = 5, Pin6 = 6, Pin7 = 7
};

template<GPIO_Pin... Pins>
struct GPIO_Config {
    static constexpr uint8_t mask = ((1u << static_cast<uint8_t>(Pins)) | ...);

    static void set_output() {
        // This would write to actual hardware register
        std::cout << "   Setting pins 0x" << std::hex
                  << static_cast<int>(mask) << std::dec
                  << " as output" << std::endl;
```

```
    }
};

// Compile-time validation of peripheral configuration
template<uint32_t ClockFreq, uint32_t DesiredBaud>
struct UART_Divider {
    static constexpr uint32_t divider = ClockFreq / (16 * DesiredBaud);
    static constexpr uint32_t actual_baud = ClockFreq / (16 * divider);
    static constexpr uint32_t error_percent =
        (actual_baud > DesiredBaud)
            ? ((actual_baud - DesiredBaud) * 100 / DesiredBaud)
            : ((DesiredBaud - actual_baud) * 100 / DesiredBaud);

    static_assert(divider > 0 && divider < 65536,
                  "UART divider out of range!");
    static_assert(error_percent < 3,
                  "Baud rate error exceeds 3%!");
};

void demonstrate_embedded_usage() {
    std::cout << "\n=== 7. EMBEDDED SYSTEMS APPLICATIONS ===" << std::endl;

    std::cout << "\n Register bit configuration (compile-time):" << std::endl;
    constexpr uint32_t bits = RegisterBits<0, 2, 4, 7>::value;
    std::cout << "   Bits 0,2,4,7 = 0x" << std::hex << bits << std::dec
              << " (0b" << std::bitset<8>(bits) << ")" << std::endl;

    std::cout << "\n GPIO configuration (compile-time):" << std::endl;
    using MyGPIO = GPIO_Config<GPIO_Pin::Pin0, GPIO_Pin::Pin3, GPIO_Pin::Pin7>;
    std::cout << "   GPIO mask: 0x" << std::hex
              << static_cast<int>(MyGPIO::mask) << std::dec << std::endl;
    MyGPIO::set_output();

    std::cout << "\n UART configuration (compile-time validated):" << std::endl;
    using UART = UART_Divider<16000000, 9600>;
    std::cout << "   Clock: 16MHz, Baud: 9600" << std::endl;
    std::cout << "   Divider: " << UART::divider
              << " (computed at compile-time)" << std::endl;

    std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
    std::cout << "    Configuration errors caught at compile-time" << std::endl;
    std::cout << "    Zero runtime overhead" << std::endl;
    std::cout << "    No magic numbers - everything is named" << std::endl;
    std::cout << "    Hardware constraints validated at compile-time" << std::endl;
}

// ==================================================================
// SECTION 8: TUPLE OPERATIONS - COMPILE-TIME RECURSION
// ==================================================================
```

```

// Print tuple recursively at compile-time
template<size_t Index = 0, typename... Types>
void print_tuple(const std::tuple<Types...>& t) {
    if constexpr (Index < sizeof...(Types)) {
        std::cout << std::get<Index>(t);
        if constexpr (Index + 1 < sizeof...(Types)) {
            std::cout << ", ";
        }
        print_tuple<Index + 1>(t);
    }
}

// Sum all tuple elements (if numeric)
template<size_t Index = 0, typename... Types>
auto sum_tuple(const std::tuple<Types...>& t) {
    if constexpr (Index < sizeof...(Types)) {
        if constexpr (Index + 1 < sizeof...(Types)) {
            return std::get<Index>(t) + sum_tuple<Index + 1>(t);
        } else {
            return std::get<Index>(t);
        }
    } else {
        return 0;
    }
}

void demonstrate_tuple_operations() {
    std::cout << "\n=== 8. TUPLE OPERATIONS WITH RECURSION ===" << std::endl;

    auto mixed_tuple = std::make_tuple(1, "hello", 3.14, 'X');
    std::cout << "\nTuple: ";
    print_tuple(mixed_tuple);
    std::cout << std::endl;

    auto numeric_tuple = std::make_tuple(10, 20, 30, 40, 50);
    std::cout << "\nNumeric tuple: ";
    print_tuple(numeric_tuple);
    std::cout << std::endl;

    auto total = sum_tuple(numeric_tuple);
    std::cout << "Sum: " << total << std::endl;

    std::cout << "\n if constexpr:" << std::endl;
    std::cout << " •   Compile-time conditional" << std::endl;
    std::cout << " •   Dead branches eliminated by compiler" << std::endl;
    std::cout << " •   Perfect for template recursion termination" << std::
        endl;
}

// ==================================================================
// SECTION 9: WHEN TO USE COMPILE-TIME RECURSION
// ==================================================================

```

```
void explain_when_to_use() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "WHEN TO USE COMPILE-TIME RECURSION:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\ n USE FOR:" << std::endl;
    std::cout << "\n1. TYPE COMPUTATIONS:" << std::endl;
    std::cout << " •   Finding largest type in parameter pack" << std::endl;
    std::cout << " •   Type trait combinations" << std::endl;
    std::cout << " •   Template metaprogramming" << std::endl;

    std::cout << "\n2. COMPILE-TIME CONSTANTS:" << std::endl;
    std::cout << " •   Mathematical computations (Fibonacci, factorial)" << std::endl;
    std::cout << " •   Configuration values" << std::endl;
    std::cout << " •   Lookup tables" << std::endl;

    std::cout << "\n3. HETEROGENEOUS COLLECTIONS:" << std::endl;
    std::cout << " •   Tuple operations" << std::endl;
    std::cout << " •   Variant handling" << std::endl;
    std::cout << " •   Type-safe function wrappers" << std::endl;

    std::cout << "\n4. EMBEDDED SYSTEMS:" << std::endl;
    std::cout << " •   Hardware register configuration" << std::endl;
    std::cout << " •   Peripheral setup validation" << std::endl;
    std::cout << " •   Zero-overhead abstractions" << std::endl;
    std::cout << " •   Compile-time constraint checking" << std::endl;

    std::cout << "\n5. CODE GENERATION:" << std::endl;
    std::cout << " •   Unrolling loops at compile-time" << std::endl;
    std::cout << " •   Generating specialized functions" << std::endl;
    std::cout << " •   Avoiding runtime dispatch" << std::endl;

    std::cout << "\ n DON'T USE FOR:" << std::endl;
    std::cout << "\n1. RUNTIME DATA:" << std::endl;
    std::cout << " •   User input processing" << std::endl;
    std::cout << " •   Dynamic collections" << std::endl;
    std::cout << " •   Data-dependent logic" << std::endl;

    std::cout << "\n2. EXCESSIVE RECURSION:" << std::endl;
    std::cout << " •   Deep recursion slows compilation" << std::endl;
    std::cout << " •   Can cause compiler errors (recursion limits)" << std::endl;
    std::cout << " •   Use fold expressions when possible" << std::endl;

    std::cout << "\n3. SIMPLE CASES:" << std::endl;
    std::cout << " •   Use standard library when available" << std::endl;
    std::cout << " •   Don't reinvent the wheel" << std::endl;

    std::cout << "\ n MODERN C++ ALTERNATIVES:" << std::endl;
    std::cout << " •   C++17: Fold expressions (replace simple recursion)" << std::endl;
    std::cout << " •   C++17: if constexpr (cleaner recursion)" << std::endl;
    std::cout << " •   C++20: Concepts (constrain templates)" << std::endl;
```

```
    std::cout << " •   C++20: consteval (force compile-time evaluation)" << std::endl;
}

// ==================================================================
// SECTION 10: PERFORMANCE COMPARISON
// ==================================================================

// Runtime version
int sum_runtime(int* values, size_t count) {
    int total = 0;
    for (size_t i = 0; i < count; ++i) {
        total += values[i];
    }
    return total;
}

void demonstrate_performance_comparison() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "PERFORMANCE: COMPILE-TIME VS RUNTIME:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n COMPILE-TIME (variadic template):" << std::endl;
    constexpr int compile_time_sum = sum(1, 2, 3, 4, 5);
    std::cout << "   Result: " << compile_time_sum << std::endl;
    std::cout << "   Binary: MOV EAX, 15  (single instruction!)" << std::endl;
    std::cout << "   Cost: 0 CPU cycles at runtime" << std::endl;
    std::cout << "   Size: ~4 bytes (just the constant)" << std::endl;

    std::cout << "\n RUNTIME (loop):" << std::endl;
    int values[] = {1, 2, 3, 4, 5};
    int runtime_sum = sum_runtime(values, 5);
    std::cout << "   Result: " << runtime_sum << std::endl;
    std::cout << "   Binary: ~20+ instructions (loop, increment, add)" << std::endl;
    std::cout << "   Cost: ~10+ CPU cycles" << std::endl;
    std::cout << "   Size: ~40+ bytes (loop code)" << std::endl;

    std::cout << "\n SPEEDUP: Infinite (0 vs ~10 cycles)" << std::endl;
    std::cout << "  SIZE: 10x smaller (4 vs 40+ bytes)" << std::endl;

    std::cout << "\n WHEN COMPILE-TIME WINS:" << std::endl;
    std::cout << " •   Values known at compile-time" << std::endl;
    std::cout << " •   Constant configuration" << std::endl;
    std::cout << " •   Type computations" << std::endl;
    std::cout << " •   Safety-critical embedded systems" << std::endl;
}

// ==================================================================
// SECTION 11: DRAWBACKS AND PITFALLS
// ==================================================================

// Pitfall 1: Deep recursion causes slow compilation
template<int N>
```

```
struct Fibonacci {
    static constexpr int value = Fibonacci<N-1>::value + Fibonacci<N-2>::value
        ;
};

template<> struct Fibonacci<0> { static constexpr int value = 0; };
template<> struct Fibonacci<1> { static constexpr int value = 1; };

// Pitfall 2: Exponential template instantiations
template<typename... Args>
struct InstantiationExplosion {
    // Each call creates multiple instantiations
    using type = std::tuple<Args..., Args...>;  // Doubles the types!
};

// Pitfall 3: Cryptic error messages
template<typename... Args>
constexpr auto bad_function(Args... args) {
    // This will produce a HORRIBLE error message if types don't support
        operator+
    return (args + ...);
}

// Pitfall 4: Code bloat - each instantiation generates code
template<typename T>
T generic_sort(T value) {
    // Even though this does nothing different, compiler generates
    // separate code for EACH type instantiation
    std::cout << "Sorting single value: " << value << std::endl;
    return value;
}

// Pitfall 5: Hidden recursion limits
template<int N>
struct DeepRecursion {
    static constexpr int value = DeepRecursion<N-1>::value + 1;
};
template<> struct DeepRecursion<0> { static constexpr int value = 0; };

void demonstrate_drawbacks_and_pitfalls() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "DRAWBACKS AND POTENTIAL PITFALLS:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n PITFALL 1: SLOW COMPILATION" << std::endl;
    std::cout << "   Problem: Deep template recursion is SLOW to compile" <<
        std::endl;
    std::cout << "   Example: Fibonacci<40>::value takes FOREVER to compile!"
        << std::endl;

    // Safe to compute small values
    constexpr int fib10 = Fibonacci<10>::value;
    std::cout << "   Fibonacci<10> = " << fib10 << " (tolerable)" << std::endl
        ;
```

```

    // Fibonacci<40> would take minutes to compile!
    // constexpr int fib40 = Fibonacci<40>::value;  // DON'T DO THIS!

    std::cout << "\n     Solution:" << std::endl;
    std::cout << " •      Use constexpr functions instead of template recursion" << std::endl;
    std::cout << " •      constexpr int fib(int n) { return (n<=1) ? n : fib(n-1)+fib(n-2); }" << std::endl;
    std::cout << " •      Much faster compilation, same runtime performance" << std::endl;

    std::cout << "\n PITFALL 2: CRYPTIC ERROR MESSAGES" << std::endl;
    std::cout << "   Problem: Template errors are EXTREMELY hard to read" << std::endl;
    std::cout << "   Example error (trying to add strings with +):" << std::endl;
    std::cout << "      error: invalid operands to binary expression" << std::endl;
    std::cout << "      ('std::string' and 'std::string')" << std::endl;
    std::cout << "      in instantiation of function template specialization" << std::endl;
    std::cout << "      'bad_function<std::string, std::string>' requested here" << std::endl;
    std::cout << "      ... 50 more lines of template instantiation backtrace ..." << std::endl;

    std::cout << "\n     Solution:" << std::endl;
    std::cout << " •      Use C++20 concepts to constrain templates EARLY" << std::endl;
    std::cout << " •      Add static_assert with clear messages" << std::endl;
    std::cout << " •      Use std::enable_if with meaningful names" << std::endl;

    std::cout << "\n PITFALL 3: CODE BLOAT" << std::endl;
    std::cout << "   Problem: Each template instantiation generates SEPARATE code" << std::endl;

    // Each call generates separate function in binary
    generic_sort(10);
    generic_sort(20.5);
    generic_sort(std::string("hello"));

    std::cout << "\n   Impact on binary size:" << std::endl;
    std::cout << " •      generic_sort<int>     : ~50 bytes" << std::endl;
    std::cout << " •      generic_sort<double>  : ~50 bytes" << std::endl;
    std::cout << " •      generic_sort<string>  : ~80 bytes" << std::endl;
    std::cout << " •      Total                 : ~180 bytes for same logic!" << std::endl;

    std::cout << "\n     Solution:" << std::endl;
    std::cout << " •      Use type erasure for runtime polymorphism" << std::endl;
    std::cout << " •      Extract common code into non-template functions" <<
```

```
    std::endl;
    std::cout << " •      Use extern template to prevent duplicate instantiations" << std::endl;

    std::cout << "\n PITFALL 4: COMPILER RECURSION LIMITS" << std::endl;
    std::cout << "   Problem: Compilers have hard limits on template depth" << std::endl;
    std::cout << "   Typical limit: 256-1024 instantiation depth" << std::endl;

    // This works (small depth)
    constexpr int depth_100 = DeepRecursion<100>::value;
    std::cout << "   DeepRecursion<100> = " << depth_100 << " (works)" << std::endl;

    // DeepRecursion<2000> would hit the limit!
    // constexpr int depth_2000 = DeepRecursion<2000>::value;  // ERROR!

    std::cout << "\n   Error you'd see:" << std::endl;
    std::cout << "      fatal error: recursive template instantiation exceeded" << std::endl;
    std::cout << "      maximum depth of 1024" << std::endl;

    std::cout << "\n     Solution:" << std::endl;
    std::cout << " •      Use constexpr functions instead" << std::endl;
    std::cout << " •      Use fold expressions (no recursion)" << std::endl;
    std::cout << " •      Increase limit: -ftemplate-depth=2000 (not recommended!)" << std::endl;

    std::cout << "\n PITFALL 5: DEBUGGING NIGHTMARE" << std::endl;
    std::cout << "   Problem: Can't step through template code" << std::endl;
    std::cout << " •   Debugger shows instantiated code, not template" << std::endl;
    std::cout << " •   Type names are mangled and unreadable" << std::endl;
    std::cout << " •   Call stack filled with template instantiation frames" << std::endl;

    std::cout << "\n     Solution:" << std::endl;
    std::cout << " •      Use std::cout for compile-time debugging" << std::endl;
    std::cout << " •      Use static_assert to validate assumptions" << std::endl;
    std::cout << " •      Test with simple types first" << std::endl;
    std::cout << " •      Use __PRETTY_FUNCTION__ to see instantiated types" << std::endl;

    std::cout << "\n PITFALL 6: EXCESSIVE MEMORY DURING COMPILATION" << std::endl;
    std::cout << "   Problem: Complex templates use LOTS of compiler memory" << std::endl;
    std::cout << "   Impact:" << std::endl;
    std::cout << " •      Simple project: ~500MB compiler memory" << std::endl;
    std::cout << " •      Heavy template use: 4GB+ compiler memory" << std::
```

```
        endl;
    std::cout << " •      Can cause out-of-memory errors in CI/CD" << std::endl;

    std::cout << "\n     Solution:" << std::endl;
    std::cout << " •      Split template-heavy code across files" << std::endl;
    std::cout << " •      Use forward declarations" << std::endl;
    std::cout << " •      Use precompiled headers for common instantiations" << std::endl;
    std::cout << " •      Consider compilation time in design decisions" << std::endl;
}

// ==================================================================
// SECTION 12: REAL-WORLD EXAMPLES OF THINGS GOING WRONG
// ==================================================================

// Example 1: Accidental O(2^N) compilation complexity
template<int N, typename... Args>
struct BadMetaFunction {
    // Each level DOUBLES the work - exponential explosion!
    using result = typename BadMetaFunction<N-1, Args..., Args...>::result;
};

template<typename... Args>
struct BadMetaFunction<0, Args...> {
    using result = std::tuple<Args...>;
};

// Example 2: Forgetting base case - infinite recursion
// template<typename T, typename... Rest>
// struct ForgetBaseCase {
//     static constexpr int value = 1 + ForgetBaseCase<Rest...>::value;
//     // OOPS! No base case for empty pack!
// };

// Example 3: Ambiguous overload resolution
template<typename T>
void overload_problem(T value) {
    std::cout << "Generic version" << std::endl;
}

template<typename T, typename... Args>
void overload_problem(T first, Args... rest) {
    std::cout << "Variadic version" << std::endl;
}

// overload_problem(42);  // Ambiguous! Which one to call?

// Example 4: Type deduction failure
template<typename... Args>
auto broken_deduction(Args... args) {
    // This fails if args is empty!
```

```
    // auto first = args...;  // ERROR: can't expand to nothing
    return 0;
}

void demonstrate_real_world_failures() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "REAL-WORLD EXAMPLES OF FAILURES:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\n FAILURE 1: EXPONENTIAL COMPILATION TIME" << std::endl;
    std::cout << "   Code: BadMetaFunction<N, Types...>" << std::endl;
    std::cout << "   Problem: Each recursion DOUBLES the type list" << std::endl;
    std::cout << "   Complexity: O(2^N) template instantiations!" << std::endl;
    std::cout << "   Result:" << std::endl;
    std::cout << " •      N=5:  32 instantiations   (0.1s)" << std::endl;
    std::cout << " •      N=10: 1024 instantiations (5s)" << std::endl;
    std::cout << " •      N=15: 32768 instantiations (MINUTES!)" << std::endl;
    std::cout << " •      N=20: Out of memory / compiler crash" << std::endl;

    std::cout << "\n FAILURE 2: MISSING BASE CASE" << std::endl;
    std::cout << "   Error message:" << std::endl;
    std::cout << "      fatal error: template instantiation depth exceeds maximum" << std::endl;
    std::cout << "      note: use -ftemplate-depth= to increase the maximum" << std::endl;
    std::cout << "   Root cause: Forgot to specialize for empty parameter pack" << std::endl;
    std::cout << "   Fix: Always provide template<> specialization" << std::endl;

    std::cout << "\n FAILURE 3: AMBIGUOUS OVERLOADS" << std::endl;
    std::cout << "   Problem: overload_problem(42)" << std::endl;
    std::cout << "   Both match:" << std::endl;
    std::cout << " •      void overload_problem(T)           // T=int" << std::endl;
    std::cout << " •      void overload_problem(T, Args...)  // T=int, Args=<>" << std::endl;
    std::cout << "   Solution: Use SFINAE or concepts to disambiguate" << std::endl;

    std::cout << "\n FAILURE 4: EMPTY PARAMETER PACK" << std::endl;
    std::cout << "   Problem: broken_deduction() called with no arguments" << std::endl;
    std::cout << "   Can't expand empty pack in certain contexts" << std::endl;
    std::cout << "   Solution: Always check sizeof...(Args) or provide base case" << std::endl;

    std::cout << "\n FAILURE 5: TYPE DEDUCTION CONFUSION" << std::endl;
    std::cout << "   Problem: auto deduction with parameter packs" << std::endl;
    std::cout << "   Example:" << std::endl;
```

```
    std::cout << "      template<typename... Args>" << std::endl;
    std::cout << "      auto func(Args... args) {" << std::endl;
    std::cout << "          return args;  // Which one? ERROR!" << std::endl;
    std::cout << "      }" << std::endl;
    std::cout << "   Solution: Be explicit about which argument to return" << std::endl;
}

// ==================================================================
// SECTION 13: BEST PRACTICES TO AVOID PITFALLS
// ==================================================================

void explain_best_practices() {
    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "BEST PRACTICES TO AVOID PITFALLS:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\ n DO:" << std::endl;

    std::cout << "\n1. USE FOLD EXPRESSIONS (C++17+):" << std::endl;
    std::cout << "     BAD:  Explicit recursion for sum" << std::endl;
    std::cout << "      template<typename T>" << std::endl;
    std::cout << "      T sum(T t) { return t; }" << std::endl;
    std::cout << "      template<typename T, typename... Args>" << std::endl;
    std::cout << "      T sum(T t, Args... args) { return t + sum(args...); }" << std::endl;
    std::cout << "\n     GOOD: Fold expression" << std::endl;
    std::cout << "      template<typename... Args>" << std::endl;
    std::cout << "      auto sum(Args... args) { return (args + ...); }" << std::endl;

    std::cout << "\n2. USE CONSTEXPR FUNCTIONS OVER TEMPLATE RECURSION:" << std::endl;
    std::cout << "     BAD:  template<int N> struct Fib { ... };" << std::endl;
    std::cout << "     GOOD: constexpr int fib(int n) { ... }" << std::endl;
    std::cout << "   Benefits: Faster compilation, clearer code, easier to debug" << std::endl;

    std::cout << "\n3. USE CONCEPTS TO CONSTRAIN EARLY (C++20):" << std::endl;
    std::cout << "     BAD:  template<typename T> void func(T t) { t + t; }" << std::endl;
    std::cout << "     GOOD: template<std::integral T> void func(T t) { t + t; }" << std::endl;
    std::cout << "   Benefits: Clear error messages, compile-time validation" << std::endl;

    std::cout << "\n4. ALWAYS PROVIDE BASE CASE:" << std::endl;
    std::cout << "     BAD:  Only recursive case" << std::endl;
    std::cout << "     GOOD: template<> struct Base { }; + recursive case" << std::endl;
    std::cout << "   Benefits: Avoid infinite recursion errors" << std::endl;

    std::cout << "\n5. USE static_assert WITH CLEAR MESSAGES:" << std::endl;
```

```
    std::cout << "     GOOD: static_assert(sizeof...(Args) > 0, \"Need at least one argument\");" << std::endl;
    std::cout << "   Benefits: Catch errors early with helpful messages" << std::endl;

    std::cout << "\n6. TEST WITH SIMPLE TYPES FIRST:" << std::endl;
    std::cout << " •   Start with int, double before complex types" << std::endl;
    std::cout << " •   Verify logic works before adding complexity" << std::endl;
    std::cout << " •   Use small parameter counts during development" << std::endl;

    std::cout << "\n7. MEASURE COMPILATION TIME:" << std::endl;
    std::cout << " •   Use -ftime-report (GCC/Clang)" << std::endl;
    std::cout << " •   Track build times in CI/CD" << std::endl;
    std::cout << " •   Set reasonable limits (e.g., <10s per file)" << std::endl;

    std::cout << "\n8. DOCUMENT CONSTRAINTS:" << std::endl;
    std::cout << " •   Specify required type traits" << std::endl;
    std::cout << " •   Document recursion depth limits" << std::endl;
    std::cout << " •   Provide usage examples" << std::endl;

    std::cout << "\ n DON'T:" << std::endl;
    std::cout << "\n1. DON'T use template recursion for deep computations" << std::endl;
    std::cout << "2. DON'T instantiate with many types unless necessary" << std::endl;
    std::cout << "3. DON'T ignore compilation time during development" << std::endl;
    std::cout << "4. DON'T use variadic templates for runtime data" << std::endl;
    std::cout << "5. DON'T nest template recursion deeply" << std::endl;

    std::cout << "\ n GOLDEN RULE:" << std::endl;
    std::cout << "   If it takes more than 1 second to compile," << std::endl;
    std::cout << "   you're probably doing it wrong!" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "        VARIADIC TEMPLATES & COMPILE-TIME RECURSION                    \n";
    std::cout << "                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                
    std::cout << "                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                
    std::cout << "        GOOD: static_assert(sizeof...(Args) > 0, \"Need at least one argument\");" << std::endl;
    std::cout << "   Benefits: Catch errors early with helpful messages" << std::endl;

    std::cout << "\n6. TEST WITH SIMPLE TYPES FIRST:" << std::endl;
    std::cout << " •   Start with int, double before complex types" << std::endl;
    std::cout << " •   Verify logic works before adding complexity" << std::endl;
    std::cout << " •   Use small parameter counts during development" << std::endl;

    std::cout << "\n7. MEASURE COMPILATION TIME:" << std::endl;
    std::cout << " •   Use -ftime-report (GCC/Clang)" << std::endl;
    std::cout << " •   Track build times in CI/CD" << std::endl;
    std::cout << " •   Set reasonable limits (e.g., <10s per file)" << std::endl;

    std::cout << "\n8. DOCUMENT CONSTRAINTS:" << std::endl;
    std::cout << " •   Specify required type traits" << std::endl;
    std::cout << " •   Document recursion depth limits" << std::endl;
    std::cout << " •   Provide usage examples" << std::endl;

    std::cout << "\ n DON'T:" << std::endl;
    std::cout << "\n1. DON'T use template recursion for deep computations" << std::endl;
    std::cout << "2. DON'T instantiate with many types unless necessary" << std::endl;
    std::cout << "3. DON'T ignore compilation time during development" << std::endl;
    std::cout << "4. DON'T use variadic templates for runtime data" << std::endl;
    std::cout << "5. DON'T nest template recursion deeply" << std::endl;

    std::cout << "\ n GOLDEN RULE:" << std::endl;
    std::cout << "   If it takes more than 1 second to compile," << std::endl;
    std::cout << "   you're probably doing it wrong!" << std::endl;
}

// ==================================================================
// MAIN FUNCTION
// ==================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "        VARIADIC TEMPLATES & COMPILE-TIME RECURSION                    \n";
    std::cout << "                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                
    std::cout << "                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                
    std::cout << "   Zero runtime cost • Type safety • Perfect for embedded                    \n";
    std::cout << "                                    \n";
```

```

    demonstrate_basic_recursion();
    demonstrate_compile_time_computation();
    demonstrate_type_manipulation();
    demonstrate_fold_expressions();
    demonstrate_safe_printf();
    demonstrate_compile_time_array();
    demonstrate_embedded_usage();
    demonstrate_tuple_operations();
    explain_when_to_use();
    demonstrate_performance_comparison();
    demonstrate_drawbacks_and_pitfalls();
    demonstrate_real_world_failures();
    explain_best_practices();

    std::cout << "\n" << std::string(70, '=') << std::endl;
    std::cout << "SUMMARY:\n";
    std::cout << std::string(70, '=') << std::endl;

    std::cout << "\ n KEY TAKEAWAYS:" << std::endl;
    std::cout << "\n1. COMPILE-TIME RECURSION:" << std::endl;
    std::cout << " •   All computation happens during compilation" << std::
        endl;
    std::cout << " •   Zero runtime cost" << std::endl;
    std::cout << " •   Results embedded in binary as constants" << std::endl;

    std::cout << "\n2. VARIADIC TEMPLATES:" << std::endl;
    std::cout << " •   Handle any number of arguments" << std::endl;
    std::cout << " •   Type-safe heterogeneous processing" << std::endl;
    std::cout << " •   Perfect for metaprogramming" << std::endl;

    std::cout << "\n3. EMBEDDED SYSTEMS:" << std::endl;
    std::cout << " •   Validate hardware configurations at compile-time" <<
        std::endl;
    std::cout << " •   Zero-overhead abstractions" << std::endl;
    std::cout << " •   Catch errors before deployment" << std::endl;
    std::cout << " •   Minimal binary size impact" << std::endl;

    std::cout << "\n4. MODERN FEATURES:" << std::endl;
    std::cout << " •   Use fold expressions (C++17) when possible" << std::
        endl;
    std::cout << " •   Use if constexpr for cleaner recursion" << std::endl;
    std::cout << " •   Use constexpr/consteval for clarity" << std::endl;
    std::cout << " •   Combine with concepts for constraints" << std::endl;

    std::cout << "\ n CRITICAL PITFALLS TO AVOID:" << std::endl;
    std::cout << "\n1. COMPILATION TIME:" << std::endl;
    std::cout << " •   Template recursion can be VERY slow" << std::endl;
    std::cout << " •   Prefer constexpr functions over template recursion" <<
        std::endl;
    std::cout << " •   Use fold expressions to eliminate recursion" << std::
        endl;

    std::cout << "\n2. CODE BLOAT:" << std::endl;
```

```
    std::cout << " •   Each instantiation = separate code in binary" << std::endl;
    std::cout << " •   Can increase binary size significantly" << std::endl;
    std::cout << " •   Extract common code to non-template functions" << std::endl;

    std::cout << "\n3. ERROR MESSAGES:" << std::endl;
    std::cout << " •   Template errors are cryptic and verbose" << std::endl;
    std::cout << " •   Use concepts (C++20) for better errors" << std::endl;
    std::cout << " •   Add static_assert with clear messages" << std::endl;

    std::cout << "\n4. RECURSION LIMITS:" << std::endl;
    std::cout << " •   Compilers limit template depth (256-1024)" << std::endl;
    std::cout << " •   Don't increase limits - redesign instead!" << std::endl;
    std::cout << " •   Always provide proper base cases" << std::endl;

    std::cout << "\n Compile-Time = Type Safety + Zero Cost!\n" << std::endl;
    std::cout << "  But use responsibly - compilation time matters!\n" << std::endl;

    return 0;
}
```

75 Source Code: VirtualFunctionsInTemplates.cpp

File: `src/VirtualFunctionsInTemplates.cpp`
Repository: View on GitHub

```cpp
/*
 * VIRTUAL FUNCTIONS IN TEMPLATES: THE CODE BLOAT PROBLEM
 *
 * This file demonstrates WHY you should AVOID virtual functions in class templates
 * and shows the severe code bloat that results from naive templatization.
 *
 * KEY PRINCIPLE: Virtual functions in templates are instantiated EVERY TIME,
 * even if never called, causing exponential code bloat.
 *
 * References:
 * - C++ Core Guidelines T.80: Do not naively templatize a class hierarchy
 * - C++ Core Guidelines T.5: Combine generic and OO techniques judiciously
 * - C++ Core Guidelines T.84: Use a non-template core implementation
 */

#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <algorithm>

// ============================================================================
// SECTION 1: THE PROBLEM - WHY VIRTUAL FUNCTIONS CAUSE BLOAT
// ============================================================================

namespace the_problem {

/*
 * THE CORE ISSUE:
 *
 * In a class template:
 * - NON-VIRTUAL functions: Only instantiated if explicitly called (lazy)
 * - VIRTUAL functions: ALWAYS instantiated when class is instantiated (eager)
 *
 * WHY?
 * Virtual functions MUST be in the vtable, and the vtable is constructed
 * at compile-time during template instantiation. Therefore, ALL virtual
 * functions must be compiled, even if your code never calls them.
 *
 * This is the root cause of code bloat in template hierarchies.
 */

template<typename T>
class DemoInstantiation {
```

```
public:
    // VIRTUAL: Compiled when class is instantiated
    virtual void always_compiled() {
        std::cout << "Virtual: always compiled for " << typeid(T).name() << "\n";
    }

    virtual void never_called_but_still_compiled() {
        std::cout << "Virtual: wasting space for " << typeid(T).name() << "\n";
    }

    virtual ~DemoInstantiation() = default;

    // NON-VIRTUAL: Only compiled if actually called
    void only_if_used() {
        std::cout << "Non-virtual: only compiled if called for " << typeid(T).name() << "\n";
    }

    void also_only_if_used() {
        std::cout << "Non-virtual: lazy instantiation for " << typeid(T).name() << "\n";
    }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "THE PROBLEM: Virtual Functions Cause Template Bloat\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Creating DemoInstantiation<int>:\n\n";
    DemoInstantiation<int> demo;

    std::cout << "What gets compiled?\n";
    std::cout << "     always_compiled() - YES (virtual, in vtable)\n";
    std::cout << "     never_called_but_still_compiled() - YES (virtual, in vtable)\n";
    std::cout << "     only_if_used() - NO (not called yet)\n";
    std::cout << "     also_only_if_used() - NO (not called yet)\n\n";

    demo.always_compiled();

    std::cout << "\n THE BLOAT:\n";
    std::cout << " •   never_called_but_still_compiled() is compiled\n";
    std::cout << " •   It's in the binary taking up space\n";
    std::cout << " •   But we never call it!\n";
    std::cout << " •   This is WASTED CODE\n\n";

    std::cout << "Now imagine:\n";
    std::cout << " •   10 virtual functions\n";
    std::cout << " •   10 template instantiations\n";
    std::cout << " •   = 100 compiled functions\n";
    std::cout << " •   If you only use 2-3 per type = 70-80 wasted functions!\
```

```
        n";
}

} // namespace the_problem

// ==========================================================================
// SECTION 2: THE BAD EXAMPLE - C++ Core Guidelines T.80
// ==========================================================================

namespace bad_example_t80 {

/*
 *   BAD DESIGN: Template with virtual functions
 *
 * This is the EXACT example from C++ Core Guidelines T.80 showing
 * what NOT to do when designing template hierarchies.
 */

template<typename T>
struct Container {          // an interface
    virtual T* get(int i) = 0;
    virtual T* first() = 0;
    virtual T* next() = 0;
    virtual void sort() = 0;
    virtual ~Container() = default;
};

template<typename T>
class Vector : public Container<T> {
private:
    std::vector<T> data_;
    typename std::vector<T>::iterator current_;

public:
    Vector() : current_(data_.begin()) {}

    T* get(int i) override {
        return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
            nullptr;
    }

    T* first() override {
        current_ = data_.begin();
        return current_ != data_.end() ? &(*current_) : nullptr;
    }

    T* next() override {
        if (current_ != data_.end()) ++current_;
        return current_ != data_.end() ? &(*current_) : nullptr;
    }
```

```

    void sort() override {
        std::sort(data_.begin(), data_.end());
    }

    void add(const T& item) { data_.push_back(item); }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "BAD EXAMPLE: C++ Core Guidelines T.80\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Code:\n";
    std::cout << "   template<typename T>\n";
    std::cout << "   struct Container {\n";
    std::cout << "       virtual T* get(int i);\n";
    std::cout << "       virtual T* first();\n";
    std::cout << "       virtual T* next();\n";
    std::cout << "       virtual void sort();\n";
    std::cout << "   };\n\n";

    std::cout << "Creating two instantiations:\n";
    std::cout << "   Vector<int> vi;\n";
    std::cout << "   Vector<string> vs;\n\n";

    Vector<int> vi;
    Vector<std::string> vs;

    std::cout << "   WHAT GETS COMPILED:\n\n";

    std::cout << "Vector<int>:\n";
    std::cout << " •   get(int) - compiled\n";
    std::cout << " •   first() - compiled\n";
    std::cout << " •   next() - compiled\n";
    std::cout << " •   sort() - compiled\n";
    std::cout << "   Even if you only use get()!\n\n";

    std::cout << "Vector<string>:\n";
    std::cout << " •   get(int) - compiled AGAIN\n";
    std::cout << " •   first() - compiled AGAIN\n";
    std::cout << " •   next() - compiled AGAIN\n";
    std::cout << " •   sort() - compiled AGAIN\n";
    std::cout << "   Complete duplication!\n\n";

    std::cout << "  CODE BLOAT CALCULATION:\n";
    std::cout << " •   2 types × 4 virtual functions = 8 functions\n";
    std::cout << " •   10 types × 10 virtual functions = 100 functions\n";
    std::cout << " •   Average function: 50-100 bytes\n";
    std::cout << " •   100 functions × 75 bytes = 7.5 KB of bloat\n";
    std::cout << " •   In large projects: MEGABYTES of wasted code\n\n";

    std::cout << "  WHY THIS IS BAD:\n";
    std::cout << "   1. Binary size increases unnecessarily\n";
```

```
    std::cout << "   2. Longer compilation times\n";
    std::cout << "   3. More pressure on instruction cache\n";
    std::cout << "   4. Harder to maintain\n";
    std::cout << "   5. Embedded systems: wastes precious flash memory\n";
}

} // namespace bad_example_t80

// ==========================================================================
// SECTION 3: WHY IT HAPPENS - THE VTABLE REQUIREMENT
// ==========================================================================

namespace why_it_happens {

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "WHY VIRTUAL FUNCTIONS MUST BE INSTANTIATED\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  THE MECHANISM:\n\n";

    std::cout << "1. VTABLE CONSTRUCTION:\n";
    std::cout << " •   Each class with virtual functions has a vtable\n";
    std::cout << " •   vtable is built at COMPILE-TIME\n";
    std::cout << " •   vtable contains pointers to ALL virtual functions\n";
    std::cout << " •   Therefore, all virtual functions MUST exist\n\n";

    std::cout << "2. TEMPLATE INSTANTIATION:\n";
    std::cout << " •   Template<int> creates a new class type\n";
    std::cout << " •   This class needs its own vtable\n";
    std::cout << " •   Compiler must compile ALL virtual functions\n";
    std::cout << " •   Even those never called\n\n";

    std::cout << "3. COMPARISON:\n\n";

    std::cout << "   Non-virtual function:\n";
    std::cout << "     Compiler sees call site\n";
    std::cout << "     \"Oh, this function is used, compile it\"\n";
    std::cout << "     No call = no compilation (lazy)\n\n";

    std::cout << "   Virtual function:\n";
    std::cout << "     Compiler instantiates class\n";
    std::cout << "     \"Must build vtable for this class\"\n";
    std::cout << "     \"Need ALL virtual function addresses\"\n";
    std::cout << "     ALL compiled regardless of use (eager)\n\n";

    std::cout << "  STANDARD LIBRARY MISTAKE:\n";
    std::cout << " •   std::locale facets (std::ctype<T>) made this mistake\n";
    std::cout << " •   ~15 virtual functions per facet\n";
```

```
    std::cout << " •   Most programs use 2-3 of them\n";
    std::cout << " •   But all 15 are instantiated\n";
    std::cout << " •   Acknowledged in C++ Core Guidelines as historical error
        \n";
}

} // namespace why_it_happens

// ============================================================================
// SOLUTION 1: NON-TEMPLATE BASE CLASS (TYPE ERASURE)
// ============================================================================

namespace solution_type_erasure {

/*
 *   GOOD: Move virtual interface to non-template base
 *
 * Virtual functions compiled ONCE, not per template instantiation.
 * This is the T.84 guideline: "Use a non-template core implementation"
 */

// Non-template base - compiled ONCE
class ContainerBase {
public:
    virtual ~ContainerBase() = default;
    virtual void* get_impl(int i) = 0;
    virtual void* first_impl() = 0;
    virtual void* next_impl() = 0;
    virtual void sort_impl() = 0;
};

// Template provides type safety
template<typename T>
class Vector : public ContainerBase {
private:
    std::vector<T> data_;
    typename std::vector<T>::iterator current_;

public:
    Vector() : current_(data_.begin()) {}

    void* get_impl(int i) override {
        return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
            nullptr;
    }

    void* first_impl() override {
        current_ = data_.begin();
        return current_ != data_.end() ? static_cast<void*>(&(*current_)) :
            nullptr;
```

```
    }

    void* next_impl() override {
        if (current_ != data_.end()) ++current_;
        return current_ != data_.end() ? static_cast<void*>(&(*current_)) :
            nullptr;
    }

    void sort_impl() override {
        std::sort(data_.begin(), data_.end());
    }

    // Type-safe wrappers
    T* get(int i) { return static_cast<T*>(get_impl(i)); }
    T* first() { return static_cast<T*>(first_impl()); }
    T* next() { return static_cast<T*>(next_impl()); }
    void sort() { sort_impl(); }
    void add(const T& item) { data_.push_back(item); }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SOLUTION 1: Non-Template Base (Type Erasure)\n";
    std::cout << std::string(70, '=') << "\n\n";

    Vector<int> vi;
    Vector<std::string> vs;

    std::cout << "  FIXED DESIGN:\n\n";

    std::cout << "ContainerBase (non-template):\n";
    std::cout << " •   Virtual functions compiled ONCE\n";
    std::cout << " •   get_impl(), first_impl(), next_impl(), sort_impl()\n";
    std::cout << " •   Shared across ALL instantiations\n\n";

    std::cout << "Vector<T> (template):\n";
    std::cout << " •   Overrides virtual functions (minimal code)\n";
    std::cout << " •   Type-specific logic only\n";
    std::cout << " •   Type-safe wrapper functions\n\n";

    std::cout << "  BLOAT REDUCTION:\n";
    std::cout << " •   10 types × 4 virtuals = 4 base + 10 overrides\n";
    std::cout << " •   vs Bad design: 40 separate functions\n";
    std::cout << " •   Reduction: 50-80% less code\n\n";

    std::cout << "   TRADE-OFFS:\n";
    std::cout << "     Massive code size reduction\n";
    std::cout << "     Runtime polymorphism preserved\n";
    std::cout << "      Slight runtime overhead (void* casts)\n";
    std::cout << "      Loss of type information in base\n";
}

} // namespace solution_type_erasure

```

```
//
    ==========================================================================
// SOLUTION 2: CRTP - NO VIRTUAL FUNCTIONS
//
    ==========================================================================

namespace solution_crtp {

/*
 *   BETTER: CRTP for compile-time polymorphism
 *
 * No vtable = No forced instantiation = No bloat
 * Zero runtime overhead
 */

template<typename Derived>
class ContainerCRTP {
public:
    auto* get(int i) { return derived()->get_impl(i); }
    auto* first() { return derived()->first_impl(); }
    auto* next() { return derived()->next_impl(); }
    void sort() { derived()->sort_impl(); }

private:
    Derived* derived() { return static_cast<Derived*>(this); }
};

template<typename T>
class Vector : public ContainerCRTP<Vector<T>> {
private:
    std::vector<T> data_;
    typename std::vector<T>::iterator current_;

public:
    Vector() : current_(data_.begin()) {}

    T* get_impl(int i) {
        return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
            nullptr;
    }

    T* first_impl() {
        current_ = data_.begin();
        return current_ != data_.end() ? &(*current_) : nullptr;
    }

    T* next_impl() {
        if (current_ != data_.end()) ++current_;
        return current_ != data_.end() ? &(*current_) : nullptr;
    }

    void sort_impl() {
```

```
        std::sort(data_.begin(), data_.end());
    }

    void add(const T& item) { data_.push_back(item); }
};

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SOLUTION 2: CRTP (Static Polymorphism)\n";
    std::cout << std::string(70, '=') << "\n\n";

    Vector<int> vi;
    vi.add(5); vi.add(3); vi.add(9);

    std::cout << "  NO VIRTUAL FUNCTIONS:\n\n";

    std::cout << "Benefits:\n";
    std::cout << " •   No vtable\n";
    std::cout << " •   No vptr (8 bytes saved per object)\n";
    std::cout << " •   Functions ONLY compiled if called\n";
    std::cout << " •   Can be fully inlined\n";
    std::cout << " •   Zero runtime overhead\n\n";

    vi.sort();

    std::cout << "What got compiled:\n";
    std::cout << "     sort_impl<int> - used\n";
    std::cout << "     add - used\n";
    std::cout << "     get_impl - NOT compiled (not used)\n";
    std::cout << "     first_impl - NOT compiled (not used)\n";
    std::cout << "     next_impl - NOT compiled (not used)\n\n";

    std::cout << "  BLOAT ELIMINATION:\n";
    std::cout << " •   Only used functions compiled\n";
    std::cout << " •   Lazy instantiation = minimal code\n";
    std::cout << " •   Perfect for performance-critical code\n\n";

    std::cout << "   TRADE-OFFS:\n";
    std::cout << "     Zero code bloat\n";
    std::cout << "     Zero runtime overhead\n";
    std::cout << "     No runtime polymorphism\n";
    std::cout << "     Cannot use heterogeneous containers\n";
}

} // namespace solution_crtp

// ==========================================================================
// SOLUTION 3: C++20 CONCEPTS
// ==========================================================================

```

```
namespace solution_concepts {

#if __cplusplus >= 202002L

template<typename T>
concept Container = requires(T c, int i) {
    { c.get(i) };
    { c.first() };
    { c.next() };
    { c.sort() } -> std::same_as<void>;
};

template<typename T>
class Vector {
private:
    std::vector<T> data_;
    typename std::vector<T>::iterator current_;

public:
    Vector() : current_(data_.begin()) {}

    T* get(int i) {
        return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
            nullptr;
    }

    T* first() {
        current_ = data_.begin();
        return current_ != data_.end() ? &(*current_) : nullptr;
    }

    T* next() {
        if (current_ != data_.end()) ++current_;
        return current_ != data_.end() ? &(*current_) : nullptr;
    }

    void sort() { std::sort(data_.begin(), data_.end()); }
    void add(const T& item) { data_.push_back(item); }
};

template<Container C>
void process(C& cont) {
    cont.sort();
}

#endif

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "SOLUTION 3: C++20 Concepts\n";
    std::cout << std::string(70, '=') << "\n\n";

#if __cplusplus >= 202002L
    Vector<int> vi;
```

```
    vi.add(5);
    process(vi);

    std::cout << "  NO INHERITANCE, NO VIRTUAL:\n\n";

    std::cout << "Concepts provide:\n";
    std::cout << " •   Compile-time interface checking\n";
    std::cout << " •   Duck typing with type safety\n";
    std::cout << " •   No base class needed\n";
    std::cout << " •   Better error messages\n";
    std::cout << " •   Zero runtime overhead\n\n";

    std::cout << "  BLOAT ELIMINATION:\n";
    std::cout << " •   No virtual functions at all\n";
    std::cout << " •   Only used functions instantiated\n";
    std::cout << " •   Cleanest solution\n";
#else
    std::cout << "   C++20 concepts not available.\n";
    std::cout << "    Compile with -std=c++20 to see this solution.\n";
#endif
}

} // namespace solution_concepts

// ==========================================================================
// GUIDELINES AND RECOMMENDATIONS
// ==========================================================================

namespace guidelines {

void demonstrate() {
    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "GUIDELINES: AVOID VIRTUAL FUNCTIONS IN TEMPLATES\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  AVOID THIS:\n\n";
    std::cout << "    template<typename T>\n";
    std::cout << "    class Container {\n";
    std::cout << "        virtual void operation();  // BAD!\n";
    std::cout << "    };\n\n";

    std::cout << "  PROBLEMS:\n";
    std::cout << "    1. CODE BLOAT: All virtuals instantiated every time\n";
    std::cout << "    2. COMPILATION: Slower, more memory\n";
    std::cout << "    3. BINARY SIZE: Can add megabytes in large projects\n";
    std::cout << "    4. PERFORMANCE: More pressure on I-cache\n";
    std::cout << "    5. MAINTENANCE: Harder to track what's compiled\n\n";

    std::cout << "   WHEN IT'S A PROBLEM:\n";
    std::cout << " •   Many virtual functions (5+)\n";
```

```
    std::cout << " •   Many template instantiations (10+)\n";
    std::cout << " •   Large codebase\n";
    std::cout << " •   Embedded systems (limited flash)\n";
    std::cout << " •   Header-only libraries\n\n";

    std::cout << "  SOLUTIONS (IN ORDER OF PREFERENCE):\n\n";

    std::cout << "1. C++20 CONCEPTS (best)\n";
    std::cout << " •   No inheritance, no virtual\n";
    std::cout << " •   Compile-time duck typing\n";
    std::cout << " •   Zero overhead\n\n";

    std::cout << "2. CRTP (for static polymorphism)\n";
    std::cout << " •   No virtual functions\n";
    std::cout << " •   Lazy instantiation\n";
    std::cout << " •   Zero runtime overhead\n\n";

    std::cout << "3. NON-TEMPLATE BASE (for runtime polymorphism)\n";
    std::cout << " •   Virtual compiled once\n";
    std::cout << " •   Massive bloat reduction\n";
    std::cout << " •   Runtime flexibility preserved\n\n";

    std::cout << "  WHEN VIRTUAL IN TEMPLATES IS OK:\n";
    std::cout << " •   Very few virtual functions (1-2)\n";
    std::cout << " •   Few instantiations (2-3 types)\n";
    std::cout << " •   Virtual functions actually needed\n";
    std::cout << " •   Code clarity more important than size\n\n";

    std::cout << "  RELATED GUIDELINES:\n";
    std::cout << " •   T.5: Combine generic and OO judiciously\n";
    std::cout << " •   T.80: Do not naively templatize a hierarchy\n";
    std::cout << " •   T.83: Do not declare member function template virtual\n";
    std::cout << " •   T.84: Use a non-template core implementation\n";
}

} // namespace guidelines

// ==========================================================================
// MAIN
// ==========================================================================

int main() {
    std::cout << "\n";
    std::cout << "                                    \n";
    std::cout << "       VIRTUAL FUNCTIONS IN TEMPLATES: THE CODE BLOAT PROBLEM        \n";
    std::cout << "                                                                      \n";
```

```
    std::cout << "           WHY TO AVOID & BETTER ALTERNATIVES
                                \n";
    std::cout << "                                    \n";

    the_problem::demonstrate();
    bad_example_t80::demonstrate();
    why_it_happens::demonstrate();
    solution_type_erasure::demonstrate();
    solution_crtp::demonstrate();
    solution_concepts::demonstrate();
    guidelines::demonstrate();

    std::cout << "\n" << std::string(70, '=') << "\n";
    std::cout << "KEY TAKEAWAYS\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "  THE PROBLEM:\n";
    std::cout << " •   Virtual functions in templates = code bloat\n";
    std::cout << " •   ALL virtuals instantiated for EVERY type\n";
    std::cout << " •   Even if never called\n";
    std::cout << " •   Can add megabytes to binary size\n\n";

    std::cout << "  WHY IT HAPPENS:\n";
    std::cout << " •   vtable must be built at compile-time\n";
    std::cout << " •   vtable needs ALL virtual function addresses\n";
    std::cout << " •   Compiler must compile them all\n";
    std::cout << " •   Non-virtual functions use lazy instantiation\n\n";

    std::cout << "  SOLUTIONS:\n";
    std::cout << "    1. C++20 Concepts - no inheritance needed\n";
    std::cout << "    2. CRTP - compile-time polymorphism\n";
    std::cout << "    3. Non-template base - runtime polymorphism\n\n";

    std::cout << "  RECOMMENDATION:\n";
    std::cout << "    AVOID virtual functions in class templates\n";
    std::cout << "    unless you have a compelling reason and few
        instantiations.\n\n";

    std::cout << "                                    \n";
    std::cout << "         Follow C++ Core Guidelines T.80 and T.84
                        \n";
    std::cout << "                                    \n\n";

    return 0;
}
```

76 Appendix: Comprehensive Index

77 Modern C++ Examples - Comprehensive Index

Last Updated: January 3, 2026

This index maps C++ concepts, keywords, features, design patterns, problems, and standards to specific example files in this repository. Use this as a quick reference to find examples for specific topics.

77.1 Table of Contents

- Interview Preparation
- C++ Standards Features
- Design Patterns & Idioms
- Object-Oriented Programming
- Memory Management
- Concurrency & Parallelism
- Real-Time & Embedded Systems
- Safety-Critical & Standards
- STL Containers & Algorithms
- Template Metaprogramming
- Error Handling
- Interoperability
- Performance & Optimization
- Common Problems & Solutions
- Keywords & Language Features

77.2 Interview Preparation

77.2.1 Complete C++ Interview Guide

- **File:** MockInterview.cpp
- **Description:** Comprehensive collection of C++ interview questions with detailed, runnable answers
- **Topics Covered:**
  - **Fundamentals:** Pointers vs references, const correctness, RAII pattern
  - **Memory Management:** Smart pointers (unique_ptr, shared_ptr, weak_ptr), custom deleters, aliasing constructor, enable_shared_from_this, memory alignment, custom allocators
  - **OOP & Design Patterns:** Virtual dispatch mechanics, vtable internals, multiple inheritance, CRTP with mixins, PIMPL idiom, compile-time polymorphism
  - **Templates & Metaprogramming:** SFINAE techniques, enable_if, detection idiom, variadic templates, fold expressions, compile-time computation, constexpr/consteval
  - **Concurrency:** Thread-safe singleton (Meyer's, call_once), producer-consumer pattern, condition variables, atomic operations, memory ordering

  - **Performance:** Move semantics, copy elision (RVO/NRVO), cache-friendly design, branch prediction optimization
- **Common Interview Questions:**
  - What's the difference between pointers and references?
  - Explain smart pointers and when to use each type
  - How does virtual dispatch work internally?
  - What is SFINAE and how is it used?
  - How do you implement a thread-safe singleton?
  - What's the difference between RVO and move semantics?
  - Explain memory ordering in atomic operations
  - How does CRTP provide compile-time polymorphism?
- **Format:** Each section includes working code examples with detailed explanations and best practices

Back to Top

77.3 C++ Standards Features

77.3.1 C++11

- **File:** Cpp11Examples.cpp
- **Topics:** `auto`, `nullptr`, range-based for, lambda expressions, smart pointers, move semantics, `constexpr`, `static_assert`, uniform initialization, `decltype`, `std::array`, `std::thread`, `std::mutex`

77.3.2 C++14

- **File:** Cpp14Examples.cpp
- **Topics:** Generic lambdas, `auto` return type deduction, binary literals, digit separators, `std::make_unique`, relaxed `constexpr`, variable templates
- **File:** GenericLambdas.cpp
- **Topics:** Generic lambda expressions with `auto` parameters

77.3.3 C++17

- **File:** Cpp17Examples.cpp
- **Topics:** Structured bindings, `if constexpr`, fold expressions, `std::optional`, `std::variant`, `std::any`, `std::string_view`, parallel algorithms, `std::filesystem`
- **File:** Cpp17Concurrency.cpp
- **Topics:** Parallel STL algorithms, execution policies
- **File:** StructuredBindings.cpp
- **Topics:** Structured bindings syntax and use cases
- **File:** OptionalExamples.cpp
- **Topics:** `std::optional` usage patterns

77.3.4 C++20

- **File:** Cpp20Examples.cpp

- **Topics:** Concepts, ranges, coroutines, modules, three-way comparison (`<=>`), designated initializers, `std::span`, `consteval`, `constinit`
- **File:** ConceptsExamples.cpp
- **Topics:** Concepts, constraints, `requires` clauses
- **File:** RangesExamples.cpp
- **Topics:** Ranges library, views, adaptors, pipelines
- **File:** CameraModule.cppm / TemplatedCameraModules.cpp
- **Topics:** C++20 modules

77.3.5 C++23

- **File:** Cpp23Examples.cpp
- **Topics:** `std::expected`, deducing `this`, `if consteval`, multidimensional subscript operator, `std::print`, `std::stacktrace`

Back to Top

77.4 Design Patterns & Idioms

77.4.1 CRTP (Curiously Recurring Template Pattern)

- **File:** CRTPvsVirtualFunctions.cpp
- **Topics:** CRTP vs virtual functions, static polymorphism, performance comparison, vtable overhead

77.4.2 Pimpl (Pointer to Implementation)

- **File:** PimplIdiom.cpp
- **Topics:** Pimpl idiom, ABI stability, compilation firewall, real-time concerns, cache locality, FastPimpl alternative

77.4.3 NVI (Non-Virtual Interface)

- **File:** NVIIdiomTemplateMethod.cpp
- **Topics:** Non-virtual interface idiom, template method pattern, Herb Sutter's virtuality guidelines

77.4.4 RAII (Resource Acquisition Is Initialization)

- **File:** RuleOf3_5_0.cpp
- **Topics:** Rule of 3/5/0, RAII, copy/move semantics, special member functions
- **File:** ResourceLeaks.cpp
- **Topics:** Preventing resource leaks with RAII

77.4.5 Dependency Injection

- **File:** DependencyInjection.cpp
- **Topics:** Constructor injection, setter injection, interface injection, composition over inheritance

77.4.6 SOLID Principles

- **File:** SOLIDPrinciples.cpp
- **Topics:** Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion

77.4.7 Factory Pattern

- **File:** CreatingCApiFromCpp.cpp
- **Topics:** Factory functions for C API

Back to Top

77.5 Object-Oriented Programming

77.5.1 Inheritance

- **File:** InheritanceTypes.cpp
- **Topics:** Public, private, protected inheritance, IS-A vs implemented-in-terms-of
- **File:** DiamondProblem.cpp
- **Topics:** Diamond problem, virtual inheritance, composition alternative
- **File:** FragileBaseClass.cpp
- **Topics:** Fragile base class problem, ABI stability, solutions

77.5.2 Polymorphism

- **File:** RuntimePolymorphism.cpp
- **Topics:** Virtual functions, pure virtual functions, abstract classes, dynamic polymorphism
- **File:** VirtualFunctionsInTemplates.cpp
- **Topics:** Why virtual functions in templates cause code bloat, type erasure, CRTP, concepts as alternatives
- **File:** CRTPvsVirtualFunctions.cpp
- **Topics:** Static vs dynamic polymorphism comparison

77.5.3 Object Slicing

- **File:** ObjectSlicingCpp20.cpp
- **Topics:** Object slicing problem, C++20 solutions
- **File:** ObjectSlicingSmartPtr.cpp
- **Topics:** Preventing object slicing with smart pointers

Back to Top

77.6 Memory Management

77.6.1 Smart Pointers

- **File:** RuleOf3_5_0.cpp
- **Topics:** `unique_ptr`, `shared_ptr`, `weak_ptr`, ownership semantics

- **File:** ResourceLeaks.cpp
- **Topics:** Smart pointers for preventing leaks
- **File:** ObjectSlicingSmartPtr.cpp
- **Topics:** Smart pointers with polymorphism

77.6.2 Move Semantics

- **File:** MoveSemantics.cpp
- **Topics:** Rvalue references, move constructors, move assignment, perfect forwarding
- **File:** MoveSematicsPerfectForwarding.cpp
- **Topics:** Move semantics and perfect forwarding patterns
- **File:** PerfectForwardingAndRequires.cpp
- **Topics:** Perfect forwarding with `std::forward`, requires clauses

77.6.3 Memory Pools & Allocators

- **File:** RealTimeProgramming.cpp
- **Topics:** Memory pools, custom allocators, pre-allocation strategies
- **File:** STLContainersNoHeap.cpp
- **Topics:** Custom allocators, stack allocators, pool allocators

77.6.4 Resource Management

- **File:** ResourceLeaks.cpp
- **Topics:** Common resource leak patterns and prevention

Back to Top

77.7 Concurrency & Parallelism

77.7.1 Threading

- **File:** AsioAndModernCppConcurrency.cpp
- **Topics:** ASIO, thread pools, async operations, io_context
- **File:** AsioMultipleContexts.cpp
- **Topics:** Multiple io_context patterns, thread distribution
- **File:** ThreadPoolExamples.cpp
- **Topics:** Thread pool implementation patterns

77.7.2 Parallel Algorithms

- **File:** Cpp17Concurrency.cpp
- **Topics:** Parallel STL, execution policies (`std::execution::par`)

77.7.3 Future/Promise/Async

- **File:** FuturePromiseAsync.cpp
- **Topics:** `std::future`, `std::promise`, `std::async`, `std::packaged_task`, ASIO relationship

77.7.4 Stop Tokens

- **File:** StopTokenExample.cpp
- **Topics:** `std::stop_token`, `std::stop_source`, cooperative cancellation (C++20)

77.7.5 Microservices

- **File:** MultiThreadedMicroservices.cpp
- **Topics:** Multi-threaded microservice architecture

Back to Top

77.8 Real-Time & Embedded Systems

77.8.1 Real-Time Programming

- **File:** RealTimeProgramming.cpp
- **Topics:** Deterministic behavior, WCET, Big O notation, memory pre-allocation, `std::bitset`, `std::list::splice()`, real-time patterns

77.8.2 Embedded Systems

- **File:** EmbeddedSystemsProgramming.cpp
- **Topics:** Best practices for embedded systems, constexpr, fixed-size containers
- **File:** EmbeddedSystemsAvoid.cpp
- **Topics:** What to avoid in embedded systems (exceptions, dynamic allocation, iostream)

77.8.3 ARM & Architecture

- **File:** ARMInstructionSets.cpp
- **Topics:** ARM instruction sets, NEON, SVE, architecture-specific optimization

77.8.4 ROM Placement

- **File:** ROMability.cpp
- **Topics:** `constexpr`, `consteval`, `constinit`, ROM placement strategies

Back to Top

77.9 Safety-Critical & Standards

77.9.1 MISRA C++

- **File:** MISRACppDemo.cpp
- **Topics:** MISRA C++ rules, safety-critical coding guidelines

77.9.2 ISO 26262 (Automotive)

- **File:** FunctionalSafetyISO26262.cpp
- **Topics:** ISO 26262 functional safety, ASIL levels, automotive safety

77.9.3 STL for Safety-Critical

- **File:** SafetyCriticalSTLContainers.cpp
- **Topics:** Which STL containers to avoid/use in safety-critical systems, MISRA compliance, AUTOSAR, heap allocation issues

77.9.4 AUTOSAR

- **File:** SafetyCriticalSTLContainers.cpp
- **Topics:** AUTOSAR C++14 guidelines, container usage

Back to Top

77.10 STL Containers & Algorithms

77.10.1 Container Usage

- **File:** STLContainersNoHeap.cpp
- **Topics:** Using STL containers without heap, custom allocators, `std::array`, `std::span`
- **File:** SafetyCriticalSTLContainers.cpp
- **Topics:** Container safety analysis, forbidden containers (list, map, set, unordered_*)

77.10.2 std::bitset

- **File:** RealTimeProgramming.cpp
- **Topics:** `std::bitset` vs `std::vector<bool>`, real-time usage

77.10.3 std::list::splice()

- **File:** RealTimeProgramming.cpp
- **Topics:** O(1) element movement, splice operations, real-time patterns

77.10.4 Algorithms

- **File:** BinarySearch.cpp
- **Topics:** Binary search algorithms, STL algorithm usage

Back to Top

77.11 Template Metaprogramming

77.11.1 Templates & Concepts

- **File:** ConceptsExamples.cpp
- **Topics:** C++20 concepts, type constraints, `requires` clauses
- **File:** PerfectForwardingAndRequires.cpp
- **Topics:** Perfect forwarding with concepts

77.11.2 CRTP (Static Polymorphism)

- **File:** CRTPvsVirtualFunctions.cpp
- **Topics:** Curiously Recurring Template Pattern, compile-time polymorphism

77.11.3 Variadic Templates

- **File:** VariadicTemplateRecursion.cpp
- **Topics:** Variadic templates, template recursion, parameter packs

77.11.4 Template Interface

- **File:** TemplatedCameraInterface.cpp
- **Topics:** Generic templated interfaces

Back to Top

77.12 Error Handling

77.12.1 Exceptions

- **File:** ErrorHandling.cpp
- **Topics:** Exception handling patterns, custom exceptions
- **File:** AdvancedExceptionHandling.cpp
- **Topics:** Advanced exception techniques, exception safety guarantees
- **File:** ExceptionWithSourceLocation.cpp
- **Topics:** `std::source_location` (C++20), exception context
- **File:** ErrorHandlingStroustrup.cpp
- **Topics:** Bjarne Stroustrup's error handling guidelines

77.12.2 noexcept

- **File:** NoexceptBestPractices.cpp
- **Topics:** `noexcept` specifier, move operations, best practices

77.12.3 Error Codes & Optional

- **File:** OptionalExamples.cpp
- **Topics:** `std::optional` for error handling
- **File:** Cpp23Examples.cpp
- **Topics:** `std::expected` for error handling (C++23)

Back to Top

77.13 Interoperability

77.13.1 C Interop

- **File:** CppWrappingCLibrary.cpp

- **Topics:** Wrapping C libraries in C++, RAII for C resources
- **File:** CreatingCApiFromCpp.cpp
- **Topics:** Creating C API from C++ code, `extern "C"`, opaque pointers

77.13.2 Python Binding

- **File:** Pybind11Example.cpp
- **Topics:** pybind11, exposing C++ to Python, automatic binding generation

77.13.3 Protobuf

- **File:** ProtobufExample.cpp
- **Topics:** Protocol Buffers, serialization, cross-language data exchange

77.13.4 JSON

- **File:** NlohmannJsonExample.cpp
- **Topics:** nlohmann/json library, JSON parsing and serialization

77.13.5 REST API

- **File:** RestApiExample.cpp
- **Topics:** REST API client, HTTP requests, CURL integration

Back to Top

77.14 Performance & Optimization

77.14.1 Performance Comparison

- **File:** CRTPvsVirtualFunctions.cpp
- **Topics:** CRTP vs virtual functions performance, vtable overhead, benchmarking
- **File:** PimplIdiom.cpp
- **Topics:** Pimpl performance impact, cache locality, indirection overhead

77.14.2 Cache Locality

- **File:** PimplIdiom.cpp
- **Topics:** Cache-friendly design, avoiding pointer indirection
- **File:** RealTimeProgramming.cpp
- **Topics:** Memory layout for real-time systems

77.14.3 Compile-Time Computation

- **File:** ROMability.cpp
- **Topics:** `constexpr`, `consteval`, compile-time evaluation

Back to Top

77.15 Common Problems & Solutions

77.15.1 Fragile Base Class

- **File:** FragileBaseClass.cpp
- **Topics:** Fragile base class problem, ABI breaks, composition over inheritance

77.15.2 Diamond Problem

- **File:** DiamondProblem.cpp
- **Topics:** Multiple inheritance diamond problem, virtual inheritance solution

77.15.3 Object Slicing

- **File:** ObjectSlicingCpp20.cpp
- **Topics:** Object slicing, prevention strategies
- **File:** ObjectSlicingSmartPtr.cpp
- **Topics:** Smart pointers to prevent slicing

77.15.4 Resource Leaks

- **File:** ResourceLeaks.cpp
- **Topics:** Memory leaks, file handle leaks, RAII solutions

77.15.5 Virtual Functions in Templates

- **File:** VirtualFunctionsInTemplates.cpp
- **Topics:** Code bloat from virtual functions in templates, C++ Core Guidelines T.80, T.83, T.84

Back to Top

77.16 Keywords & Language Features

77.16.1 Keywords

77.16.1.1 `auto`

- Cpp11Examples.cpp - C++11 auto keyword
- Cpp14Examples.cpp - C++14 auto return type
- GenericLambdas.cpp - Generic lambdas with auto

77.16.1.2 `constexpr` / `consteval` / `constinit`

- Cpp11Examples.cpp - C++11 constexpr
- Cpp14Examples.cpp - C++14 relaxed constexpr
- Cpp20Examples.cpp - C++20 consteval, constinit
- ROMability.cpp - ROM placement with const* keywords

77.16.1.3 `virtual`

- RuntimePolymorphism.cpp - Virtual functions basics
- CRTPvsVirtualFunctions.cpp - Virtual vs CRTP
- VirtualFunctionsInTemplates.cpp - Virtual in templates problem
- NVIIdiomTemplateMethod.cpp - Virtual in NVI pattern

77.16.1.4 `noexcept`

- NoexceptBestPractices.cpp - noexcept best practices
- MoveSemantics.cpp - noexcept move operations

77.16.1.5 `requires` (Concepts)

- ConceptsExamples.cpp - Requires clauses
- PerfectForwardingAndRequires.cpp - Requires with forwarding

77.16.1.6 `final`

- FragileBaseClass.cpp - Final keyword usage

77.16.1.7 `override`

- RuntimePolymorphism.cpp - Override specifier

77.16.1.8 `nullptr`

- Cpp11Examples.cpp - nullptr vs NULL

77.16.1.9 `decltype`

- Cpp11Examples.cpp - decltype usage
- PerfectForwardingAndRequires.cpp - decltype with forwarding

Back to Top

77.17 Event-Driven Programming

77.17.1 Event Systems

- **File:** EventDrivenProgramming_Lambdas.cpp
- **Topics:** Signals/slots with lambdas, observer pattern
- **File:** EventDrivenProgramming_Inheritance.cpp
- **Topics:** Event systems with inheritance

Back to Top

77.18 Lambda Expressions

77.18.1 Lambda Basics

- **File:** Cpp11Examples.cpp
- **Topics:** Basic lambda syntax (C++11)
- **File:** GenericLambdas.cpp
- **Topics:** Generic lambdas with auto parameters (C++14)
- **File:** LambdaCaptures.cpp
- **Topics:** Lambda capture modes, mutable lambdas

Back to Top

77.19 Data Structures & Algorithms

77.19.1 Linked Lists

- **File:** SinglyLinkedList.cpp
- **Topics:** Singly linked list implementation
- **File:** InsertAndDeleteNodes.cpp
- **Topics:** Node insertion and deletion
- **File:** FindFirstCommonNode.cpp
- **Topics:** Finding common node in linked lists
- **File:** FindCountOfCommonNodes.cpp
- **Topics:** Counting common nodes
- **File:** FindMToLastElement.cpp
- **Topics:** Finding m-th to last element

77.19.2 Search & Sort

- **File:** BinarySearch.cpp
- **Topics:** Binary search implementation
- **File:** SearchAnagramsDictionary.cpp
- **Topics:** Anagram searching in dictionary

77.19.3 Array Algorithms

- **File:** FindMaxNoOfConsecutiveOnesFromIntArray.cpp
- **Topics:** Maximum consecutive elements

Back to Top

77.20 Tuples & Structured Bindings

77.20.1 Tuples

- **File:** TuplesAndStructuredBindings.cpp
- **Topics:** `std::tuple`, tuple operations, structured bindings

77.20.2 Structured Bindings

- **File:** StructuredBindings.cpp
- **Topics:** Structured bindings syntax and patterns

Back to Top

77.21 Configuration & Parsing

77.21.1 Configuration

- **File:** ConfigLoaderAndChecker.cpp
- **Topics:** Configuration file loading and validation

77.21.2 System Interaction

- **File:** SystemInteractionAndParsing.cpp
- **Topics:** System calls, command execution, parsing

Back to Top

77.22 Sensor Fusion & Scientific Computing

77.22.1 Eigen Library

- **File:** EigenSensorFusion.cpp
- **Topics:** Eigen library, Kalman filter, sensor fusion, matrix operations

Back to Top

77.23 Quick Reference Tables

77.23.1 By Language Version

C++11	C++14	C++17	C++20	C++23
Cpp11Examples.cpp	Cpp14Examples.cpp	Cpp17Examples.cpp	Cpp20Examples.cpp	Cpp23Examples.cpp
MoveSemantics.cpp	GenericLambdas.cpp	StructuredBindings.cpp	ConceptsExamples.cpp	-
LambdaCaptures.cpp	-	OptionalExamples.cpp	RangesExamples.cpp	-
-	-	Cpp17Concurrency.cpp	StopTokenExample.cpp	-

77.23.2 By Use Case

Use Case	Recommended Files
Learning Modern C++	Start with Cpp11Examples.cpp, then progress through C++14, 17, 20, 23 examples

Use Case	Recommended Files
Real-Time Systems	RealTimeProgramming.cpp, SafetyCriticalSTLContainers.cpp, PimplIdiom.cpp
Embedded Systems	EmbeddedSystemsProgramming.cpp, EmbeddedSystemsAvoid.cpp, STLContainersNoHeap.cpp
Safety-Critical	MISRACppDemo.cpp, FunctionalSafetyISO26262.cpp, SafetyCriticalSTLContainers.cpp
Performance	CRTPvsVirtualFunctions.cpp, PimplIdiom.cpp, RealTimeProgramming.cpp
OOP Design	InheritanceTypes.cpp, DependencyInjection.cpp, SOLIDPrinciples.cpp
Memory Management	RuleOf3_5_0.cpp, MoveSemantics.cpp, ResourceLeaks.cpp
Concurrency	AsioAndModernCppConcurrency.cpp, FuturePromiseAsync.cpp, ThreadPoolExamples.cpp

77.23.3 By Problem You're Trying to Solve

Problem	Solution Files
"My code has memory leaks"	ResourceLeaks.cpp, RuleOf3_5_0.cpp
"Objects are getting sliced"	ObjectSlicingCpp20.cpp, ObjectSlicingSmartPtr.cpp
"Base class changes break derived classes"	FragileBaseClass.cpp
"Multiple inheritance is causing problems"	DiamondProblem.cpp
"Virtual functions are too slow"	CRTPvsVirtualFunctions.cpp, VirtualFunctionsInTemplates.cpp
"Pimpl hurts performance"	PimplIdiom.cpp
"Need containers without heap"	STLContainersNoHeap.cpp, SafetyCriticalSTLContainers.cpp
"Timing is non-deterministic"	RealTimeProgramming.cpp
"Need C interop"	CppWrappingCLibrary.cpp, CreatingCApiFromCpp.cpp

Back to Top

77.24 Additional Resources

77.24.1 Documentation Files

- MarkDownDocuments/ - Comprehensive markdown documentation
  - CPP11.md, CPP14.md, CPP17.md, CPP20.md, CPP23.md
  - Cpp20Modules.md, Cpp20ModulesQuickRef.md
  - CppCInterop.md
  - EventDrivenProgramming.md
  - MultiThreadedMicroservices.md

  - NlohmannJson.md
  - Protobuf.md
  - Pybind11.md
  - RestApi.md
  - SECURITY.md
  - TemplatedCameraInterface.md
  - VirtualFunctions.md

77.24.2 Build Scripts

- scripts/ - Build and execution scripts
- build_modules.ps1 - PowerShell build script

Back to Top

77.25 How to Use This Index

1. **Find by Topic**: Search for a keyword (e.g., "lambda", "virtual", "real-time")
2. **Find by Standard**: Look under C++ Standards Features for version-specific examples
3. **Find by Problem**: Check "Common Problems & Solutions" section
4. **Find by Use Case**: See "By Use Case" table for curated file lists

---

77.26 Contributing

When adding new example files: 1. Update this index with appropriate mappings 2. Add to relevant sections 3. Update quick reference tables 4. Add problem-solution mapping if applicable

---

Repository Structure:

```
ModernCppExamples/
 src/                  # Source files (.cpp, .cppm)
 MarkDownDocuments/    # Markdown documentation
 scripts/              # Build scripts
 proto/                # Protocol buffer definitions
 CMakeLists.txt        # CMake build configuration
 INDEX.md              # This file
```

Back to Top

Last updated: January 3, 2026