

# Modern C++ Refresher Course

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

Aung K. Soe

Claude Sonnet 4.5 (AI Assistant)

January 3, 2026

## Contents

<b>1 Modern C++ Examples - Comprehensive Reference</b>	<b>7</b>
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	17
1.5.17 U	17
1.5.18 V	17
1.6 Project Structure	17
1.7 Finding Examples by Topic	18
1.7.1 By C++ Standard	18

1.7.2	By Application Domain . . . . .	18
1.7.3	By Design Pattern . . . . .	19
1.8	<b>Testing &amp; Running Examples</b> . . . . .	19
1.9	<b>Dependencies</b> . . . . .	19
1.10	<b>Contributing</b> . . . . .	19
1.11	<b>License</b> . . . . .	20
1.12	<b>Contact &amp; Support</b> . . . . .	20
1.13	<b>Acknowledgments</b> . . . . .	20
1.14	<b>Additional Resources</b> . . . . .	20
1.14.1	Official Documentation . . . . .	20
1.14.2	Safety Standards . . . . .	20
1.14.3	Books . . . . .	20
<b>2</b>	<b>Source Code: ARMInstructionSets.cpp</b>	<b>21</b>
<b>3</b>	<b>Source Code: AdvancedExceptionHandling.cpp</b>	<b>33</b>
<b>4</b>	<b>Source Code: AsioAndModernCppConcurrency.cpp</b>	<b>48</b>
<b>5</b>	<b>Source Code: AsioMultipleContexts.cpp</b>	<b>61</b>
<b>6</b>	<b>Source Code: BinarySearch.cpp</b>	<b>75</b>
<b>7</b>	<b>Source Code: CRTPvsVirtualFunctions.cpp</b>	<b>77</b>
<b>8</b>	<b>Source Code: CameraModule.cppm</b>	<b>92</b>
<b>9</b>	<b>Source Code: ConceptsExamples.cpp</b>	<b>100</b>
<b>10</b>	<b>Source Code: ConfigLoaderAndChecker.cpp</b>	<b>108</b>
<b>11</b>	<b>Source Code: Cpp11Examples.cpp</b>	<b>121</b>
<b>12</b>	<b>Source Code: Cpp14Examples.cpp</b>	<b>131</b>
<b>13</b>	<b>Source Code: Cpp17Concurrency.cpp</b>	<b>137</b>
<b>14</b>	<b>Source Code: Cpp17Examples.cpp</b>	<b>154</b>
<b>15</b>	<b>Source Code: Cpp20Examples.cpp</b>	<b>163</b>
<b>16</b>	<b>Source Code: Cpp23Examples.cpp</b>	<b>173</b>
<b>17</b>	<b>Source Code: CppWrappingCLibrary.cpp</b>	<b>188</b>
<b>18</b>	<b>Source Code: CreatingCApiFromCpp.cpp</b>	<b>202</b>
<b>19</b>	<b>Source Code: DependencyInjection.cpp</b>	<b>214</b>
<b>20</b>	<b>Source Code: DiamondProblem.cpp</b>	<b>225</b>

<b>21</b> Source Code: EigenSensorFusion.cpp	<b>237</b>
<b>22</b> Source Code: EmbeddedSystemsAvoid.cpp	<b>256</b>
<b>23</b> Source Code: EmbeddedSystemsProgramming.cpp	<b>269</b>
<b>24</b> Source Code: ErrorHandling.cpp	<b>284</b>
<b>25</b> Source Code: ErrorHandlingStroustrup.cpp	<b>297</b>
<b>26</b> Source Code: EventDrivenProgramming_Inheritance.cpp	<b>312</b>
<b>27</b> Source Code: EventDrivenProgramming_Lambdas.cpp	<b>323</b>
<b>28</b> Source Code: ExceptionWithSourceLocation.cpp	<b>334</b>
<b>29</b> Source Code: FindCountOfCommonNodes.cpp	<b>349</b>
<b>30</b> Source Code: FindFirstCommonNode.cpp	<b>352</b>
<b>31</b> Source Code: FindMToLastElement.cpp	<b>355</b>
<b>32</b> Source Code: FindMaxNoOfConsecutiveOnesFromIntArray.cpp	<b>358</b>
<b>33</b> Source Code: FragileBaseClass.cpp	<b>360</b>
<b>34</b> Source Code: FunctionalSafetyISO26262.cpp	<b>374</b>
<b>35</b> Source Code: FuturePromiseAsync.cpp	<b>395</b>
<b>36</b> Source Code: GenericLambdas.cpp	<b>408</b>
<b>37</b> Source Code: InheritanceTypes.cpp	<b>414</b>
<b>38</b> Source Code: InsertAndDeleteNodes.cpp	<b>425</b>
<b>39</b> Source Code: LambdaCaptures.cpp	<b>433</b>
<b>40</b> Source Code: MISRA_Cpp_Demo.cpp	<b>443</b>
<b>41</b> Source Code: MockInterview.cpp	<b>457</b>
<b>42</b> Source Code: MoveSemantics.cpp	<b>477</b>
<b>43</b> Source Code: MoveSemantics_PerfectForwarding.cpp	<b>489</b>
<b>44</b> Source Code: MultiThreadedMicroservices.cpp	<b>502</b>
<b>45</b> Source Code: NVIIdiomTemplateMethod.cpp	<b>520</b>
<b>46</b> Source Code: NlohmannJsonExample.cpp	<b>535</b>
<b>47</b> Source Code: NoexceptBestPractices.cpp	<b>551</b>

<b>48 Source Code: ObjectSlicingCpp20.cpp</b>	<b>565</b>
<b>49 Source Code: ObjectSlicingSmartPtr.cpp</b>	<b>574</b>
<b>50 Source Code: OptionalExamples.cpp</b>	<b>582</b>
<b>51 Source Code: PerfectForwardingAndRequires.cpp</b>	<b>590</b>
<b>52 Source Code: PimplIdiom.cpp</b>	<b>602</b>
<b>53 Source Code: ProtobufExample.cpp</b>	<b>622</b>
<b>54 Source Code: Pybind11Example.cpp</b>	<b>638</b>
<b>55 Source Code: ROMability.cpp</b>	<b>651</b>
<b>56 Source Code: RangesExamples.cpp</b>	<b>664</b>
<b>57 Source Code: RealTimeProgramming.cpp</b>	<b>672</b>
<b>58 Source Code: ResourceLeaks.cpp</b>	<b>701</b>
<b>59 Source Code: RestApiExample.cpp</b>	<b>722</b>
<b>60 Source Code: RuleOf3_5_0.cpp</b>	<b>734</b>
<b>61 Source Code: RuntimePolymorphism.cpp</b>	<b>752</b>
<b>62 Source Code: SOLIDPrinciples.cpp</b>	<b>766</b>
<b>63 Source Code: STLContainersNoHeap.cpp</b>	<b>780</b>
<b>64 Source Code: SafetyCriticalSTLContainers.cpp</b>	<b>793</b>
<b>65 Source Code: SearchAnagramsDictionary.cpp</b>	<b>809</b>
<b>66 Source Code: SinglyLinkedList.cpp</b>	<b>811</b>
<b>67 Source Code: StopTokenExample.cpp</b>	<b>814</b>
<b>68 Source Code: StructuredBindings.cpp</b>	<b>829</b>
<b>69 Source Code: SystemInteractionAndParsing.cpp</b>	<b>836</b>
<b>70 Source Code: TemplatedCameraInterface.cpp</b>	<b>882</b>
<b>71 Source Code: TemplatedCameraModules.cpp</b>	<b>894</b>
<b>72 Source Code: ThreadPoolExamples.cpp</b>	<b>901</b>
<b>73 Source Code: TuplesAndStructuredBindings.cpp</b>	<b>919</b>
<b>74 Source Code: UniversalResourceManager.cpp</b>	<b>930</b>

<b>75 Source Code: VariadicTemplateRecursion.cpp</b>	<b>944</b>
<b>76 Source Code: VirtualFunctionsInTemplates.cpp</b>	<b>964</b>
<b>77 Appendix: Comprehensive Index</b>	<b>977</b>
<b>78 Modern C++ Examples - Comprehensive Index</b>	<b>977</b>
78.1 Table of Contents . . . . .	977
78.2 Interview Preparation . . . . .	977
78.2.1 Complete C++ Interview Guide . . . . .	977
78.3 C++ Standards Features . . . . .	978
78.3.1 C++11 . . . . .	978
78.3.2 C++14 . . . . .	978
78.3.3 C++17 . . . . .	978
78.3.4 C++20 . . . . .	978
78.3.5 C++23 . . . . .	979
78.4 Design Patterns & Idioms . . . . .	979
78.4.1 CRTP (Curiously Recurring Template Pattern) . . . . .	979
78.4.2 Pimpl (Pointer to Implementation) . . . . .	979
78.4.3 NVI (Non-Virtual Interface) . . . . .	979
78.4.4 RAII (Resource Acquisition Is Initialization) . . . . .	979
78.4.5 Dependency Injection . . . . .	979
78.4.6 SOLID Principles . . . . .	980
78.4.7 Factory Pattern . . . . .	980
78.5 Object-Oriented Programming . . . . .	980
78.5.1 Inheritance . . . . .	980
78.5.2 Polymorphism . . . . .	980
78.5.3 Object Slicing . . . . .	980
78.6 Memory Management . . . . .	980
78.6.1 Smart Pointers . . . . .	980
78.6.2 Universal Resource Management with Custom Deleters . . . . .	981
78.6.3 Move Semantics . . . . .	981
78.6.4 Memory Pools & Allocators . . . . .	981
78.6.5 Resource Management . . . . .	981
78.7 Concurrency & Parallelism . . . . .	982
78.7.1 Threading . . . . .	982
78.7.2 Parallel Algorithms . . . . .	982
78.7.3 Future/Promise/Async . . . . .	982
78.7.4 Stop Tokens . . . . .	982
78.7.5 Microservices . . . . .	982
78.8 Real-Time & Embedded Systems . . . . .	982
78.8.1 Real-Time Programming . . . . .	982
78.8.2 Embedded Systems . . . . .	982
78.8.3 ARM & Architecture . . . . .	983
78.8.4 ROM Placement . . . . .	983
78.9 Safety-Critical & Standards . . . . .	983
78.9.1 MISRA C++ . . . . .	983
78.9.2 ISO 26262 (Automotive) . . . . .	983

78.9.3 STL for Safety-Critical . . . . .	983
78.9.4 AUTOSAR . . . . .	983
78.10 STL Containers & Algorithms . . . . .	983
78.10.1 Container Usage . . . . .	983
78.10.2 std::bitset . . . . .	983
78.10.3 std::list::splice() . . . . .	984
78.10.4 Algorithms . . . . .	984
78.11 Template Metaprogramming . . . . .	984
78.11.1 Templates & Concepts . . . . .	984
78.11.2 CRTP (Static Polymorphism) . . . . .	984
78.11.3 Variadic Templates . . . . .	984
78.11.4 Template Interface . . . . .	984
78.12 Error Handling . . . . .	984
78.12.1 Exceptions . . . . .	984
78.12.2 noexcept . . . . .	985
78.12.3 Error Codes & Optional . . . . .	985
78.13 Interoperability . . . . .	985
78.13.1 C Interop . . . . .	985
78.13.2 Python Binding . . . . .	985
78.13.3 Protobuf . . . . .	985
78.13.4 JSON . . . . .	985
78.13.5 REST API . . . . .	985
78.14 Performance & Optimization . . . . .	986
78.14.1 Performance Comparison . . . . .	986
78.14.2 Cache Locality . . . . .	986
78.14.3 Compile-Time Computation . . . . .	986
78.15 Common Problems & Solutions . . . . .	986
78.15.1 Fragile Base Class . . . . .	986
78.15.2 Diamond Problem . . . . .	986
78.15.3 Object Slicing . . . . .	986
78.15.4 Resource Leaks . . . . .	986
78.15.5 Virtual Functions in Templates . . . . .	986
78.16 Keywords & Language Features . . . . .	987
78.16.1 Keywords . . . . .	987
78.17 Event-Driven Programming . . . . .	988
78.17.1 Event Systems . . . . .	988
78.18 Lambda Expressions . . . . .	988
78.18.1 Lambda Basics . . . . .	988
78.19 Data Structures & Algorithms . . . . .	988
78.19.1 Linked Lists . . . . .	988
78.19.2 Search & Sort . . . . .	989
78.19.3 Array Algorithms . . . . .	989
78.20 Tuples & Structured Bindings . . . . .	989
78.20.1 Tuples . . . . .	989
78.20.2 Structured Bindings . . . . .	989
78.21 Configuration & Parsing . . . . .	989
78.21.1 Configuration . . . . .	989
78.21.2 System Interaction . . . . .	989

78.22	Sensor Fusion & Scientific Computing	989
78.22.1	Eigen Library	989
78.23	Quick Reference Tables	990
78.23.1	By Language Version	990
78.23.2	By Use Case	990
78.23.3	By Problem You're Trying to Solve	990
78.24	Additional Resources	991
78.24.1	Documentation Files	991
78.24.2	Build Scripts	991
78.25	How to Use This Index	991
78.26	Contributing	991

## 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:

```
1 # Clone the repository
2 git clone https://github.com/AungKyawSoe-Tech/ModernCppExamples.git
3 cd ModernCppExamples
4
5 # Build all examples
6 mkdir build && cd build
7 cmake ..
8 make -j$(nproc)
9
10 # Run a specific example
11 ./bin/Cpp20Examples
```

#### 1.3.2.2 Windows (PowerShell):

```
1 # Clone the repository
2 git clone https://github.com/AungKyawSoe-Tech/ModernCppExamples.git
3 cd ModernCppExamples
4
5 # Build all examples
6 mkdir build
7 cd build
8 cmake ..
9 cmake --build . --config Release
10
11 # Run a specific example
12 .\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: StackTrace, DatabaseException, TracedException
2. [ARMIInstructionSets.cpp](#)
  - compiled as Thumb
  - Functions: explain\_interworking, explain\_alignment\_issues, explain\_instruction\_sets
3. [AsioAndModernCppConcurrency.cpp](#)
  - Note: This example demonstrates ASIO concepts using standard C++ features.
  - Namespaces: std
  - Classes: HybridServer, SimulatedAsioTimer, EventLoop
4. [AsioMultipleContexts.cpp](#)
  - ...
  - Namespaces: std, SimulatedAsio
  - Classes: io\_context, steady\_timer, executor\_work\_guard

### 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: Image, ImageProcessor, Camera
7. [ConceptsExamples.cpp](#)
  - =====

- Classes: Rectangle, Point2D, Circle
- Functions: example\_basic\_concept, example\_standard\_concepts, print\_sum

## 8. [ConfigLoaderAndChecker.cpp](#)

- ConfigLoaderAndChecker.cpp
- Namespaces: fs, std
- Classes: ConfigManager, ConfigMonitor

## 9. [Cpp11Examples.cpp](#)

- =====
- Classes: Traffic, MoveableResource, Color
- Functions: example\_auto, foo

## 10. [Cpp14Examples.cpp](#)

- =====
- Namespaces: std
- Functions: example\_binary\_literals, example\_lambda\_capture\_initializers, example\_generic\_lambdas

## 11. [Cpp17Concurrency.cpp](#)

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

## 12. [Cpp17Examples.cpp](#)

- =====
- Namespaces: fs, A
- Functions: example\_template\_argument\_deduction, sum\_fold, example\_structured\_bindings

## 13. [Cpp20Examples.cpp](#)

- =====
- Classes: Color
- Functions: example\_designated\_initializers, example\_three\_way\_comparison, example\_concepts

## 14. [Cpp23Examples.cpp](#)

- =====
- Classes: MemoryRegister, SensorError, GPIOPin
- Functions: demonstrate\_byteswap, demonstrate\_expected

## 15. [CppWrappingCLibrary.cpp](#)

- =====
- Classes: SocketError, Socket, TcpServer
- Functions: demonstrate\_raii, demonstrate\_noexcept, demonstrate\_nodiscard

## 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, Base, AnimalBase

#### 1.5.4 D

##### 18. [DependencyInjection.cpp](#)

- =====
- Classes: FileLogger, ILogger, ConsoleLogger
- Functions: example\_traditional\_di, example\_constructor\_injection, example\_interface\_injection

##### 19. [DiamondProblem.cpp](#)

- DiamondProblem.cpp
- Namespaces: diamond\_problem, virtual\_inheritance\_solution
- Classes: Device, OutputDevice, InputDevice

#### 1.5.5 E

##### 20. [EigenSensorFusion.cpp](#)

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

##### 21. [EmbeddedSystemsAvoid.cpp](#)

- =====
- Classes: BaseSensor, SensorBase, TempSensor
- Functions: demonstrate\_vector\_problems, demonstrate\_string\_problems, demonstrate\_heap\_fragmentation

##### 22. [EmbeddedSystemsProgramming.cpp](#)

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

##### 23. [ErrorHandling.cpp](#)

- =====
- Classes: CameraException, runtime\_error, CameraNotConnectedException
- Functions: demonstrate\_basic\_exception\_handling, demonstrate\_custom\_exceptions, demonstrate\_exception\_hierarchy

##### 24. [ErrorHandlingStroustrup.cpp](#)

- =====
- Classes: ConfigParser, FileHandler\_GOOD, FileHandler\_BAD
- Functions: demonstrate\_exceptions, demonstrate\_termination, demonstrate\_return\_values

##### 25. [EventDrivenProgramming\\_Inheritance.cpp](#)

- =====

- Classes: CountingObserver, IObserver, ConsoleObserver
- Functions: example\_event\_hierarchy, example\_inheritance\_observer, example\_listener\_pattern

## 26. [EventDrivenProgramming\\_Lambdas.cpp](#)

- =====
- Classes: Subject, Signal, Button
- Functions: example\_lambda\_observer, example\_signal\_slot, example\_variant\_dispatcher

## 27. [ExceptionWithSourceLocation.cpp](#)

- =====
- Classes: DetailedException, BaseLocationException, SourceLocationException
- Functions: some\_function\_that\_fails, demonstrate\_detailed\_location, demonstrate\_basic\_location

### 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: fragile\_example, composition\_solution
- Classes: Counter\_V2, LoggingCounter\_V1, Counter\_V1

#### 33. [FunctionalSafetyISO26262.cpp](#)

- FunctionalSafetyISO26262.cpp
- Classes: SafetyState, ASILLevel, SafetyResult
- Functions: demonstrate\_redundancy\_voting, demonstrate\_safe\_data\_types, demonstrate\_asil\_levels

#### 34. [FuturePromiseAsync.cpp](#)

- FuturePromiseAsync.cpp

- Namespaces: std, async\_examples
- Classes: RequestProcessor, SimpleThreadPool

### 1.5.7 G

#### 35. [GenericLambdas.cpp](#)

- =====
- Functions: example\_generic\_lambda\_containers, example\_generic\_lambda\_operations, example\_basic\_generic\_lambda

### 1.5.8 I

#### 36. [InheritanceTypes.cpp](#)

- =====
- Classes: Timer, Dog, Animal
- 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: ExampleClass2, Button, ExampleClass
- Functions: example\_capture\_all\_by\_value, example\_capture\_all\_by\_reference, example\_no\_capture

### 1.5.10 M

#### 39. [MISRACppDemo.cpp](#)

- =====
- Namespaces: expressions\_and\_operators, types\_and\_declarations
- Classes: GoodBase, Resource, BadBase

#### 40. [MockInterview.cpp](#)

- =====
- Namespaces: std
- Classes: Data, FileHandler, Parent

#### 41. [MoveSemantics.cpp](#)

- =====
- Classes: MoveOnlyType, LargeObject, Resource
- Functions: demonstrate\_lvalue\_rvalue, demonstrate\_std\_move, demonstrate\_copy\_vs\_move

#### 42. [MoveSemanticsPerfectForwarding.cpp](#)

- =====
- Namespaces: lvalues\_rvalues, move\_semantics
- Classes: Widget, Resource

#### 43. **MultiThreadedMicroservices.cpp**

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

### 1.5.11 N

#### 44. **NlohmannJsonExample.cpp**

- =====
- Functions: example\_custom\_types, example\_arrays, example\_basic\_json

#### 45. **NoexceptBestPractices.cpp**

- =====
- Namespaces: what\_is\_noexcept, always\_use\_noexcept
- Classes: Swappable, Resource, MoveableResource

#### 46. **NVIIIdiomTemplateMethod.cpp**

- NVIIIdiomTemplateMethod.cpp
- Namespaces: basic\_nvi, std
- Classes: DataProcessor, CSVProcessor, JSONProcessor

### 1.5.12 O

#### 47. **ObjectSlicingCpp20.cpp**

- =====
- Classes: Rectangle, Shape, Circle
- Functions: draw\_shape, process\_shape, example\_concepts\_prevent\_slicing

#### 48. **ObjectSlicingSmartPtr.cpp**

- =====
- Classes: Rectangle, Shape, Circle
- Functions: example\_shared\_ptr\_slicing\_wrong, example\_container\_slicing\_wrong, example\_classic\_slicing

#### 49. **OptionalExamples.cpp**

- =====
- Functions: example\_value\_or, example\_optional\_return, example\_basic\_optional

### 1.5.13 P

#### 50. **PerfectForwardingAndRequires.cpp**

- tests
- Classes: Serializable, ThreadSafeQueue, Register

- Functions: call\_with\_forward, call\_by\_value, demonstrate\_perfect\_forwarding

## 51. [PimplIdiom.cpp](#)

- =====
- Namespaces: modern\_cpp\_pimpl, what\_is\_pimpl
- Classes: WidgetWithPimpl, WidgetCppClass98, WidgetNoPimpl

## 52. [ProtobufExample.cpp](#)

- =====
- Functions: example\_binary\_serialization, example\_binary\_deserialization, example\_create\_sensor\_reading

## 53. [Pybind11Example.cpp](#)

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

## 1.5.14 R

### 54. [RangesExamples.cpp](#)

- =====
- Namespaces: views, ranges
- Functions: example\_filter\_view, example\_basic\_ranges, example\_transform\_view

### 55. [RealTimeProgramming.cpp](#)

- RealTimeProgramming.cpp
- Namespaces: big\_o\_notation, thread\_architecture
- Classes: SensorSystem, SensorStatus, RealTimeFlagManager

### 56. [ResourceLeaks.cpp](#)

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

### 57. [RestApiExample.cpp](#)

- =====
- Classes: CurlHandle, RestClient, CurlHeaders
- Functions: example\_create\_post, example\_json\_placeholder\_api, example\_update\_post

### 58. [ROMability.cpp](#)

- ROMability.cpp
- Namespaces: rom\_arrays, const\_vs\_constexpr
- Functions: demonstrate, get\_runtime\_value

### 59. [RuleOf3\\_5\\_0.cpp](#)

- no-op
- Classes: RuleOfThreeExample, RuleOfZeroExample, RuleOfFiveExample

- Functions: example\_rule\_of\_zero, example\_rule\_of\_three, example\_rule\_of\_five

## 60. [RuntimePolymorphism.cpp](#)

- =====
- Classes: Rectangle, IShape, Circle
- Functions: example\_virtual\_with\_default, example\_virtual\_destructor, example\_pure\_virtual

### 1.5.15 S

#### 61. [SafetyCriticalSTLContainers.cpp](#)

- =====
- Namespaces: containers\_to\_avoid, container\_classification
- Classes: SafetyPoolAllocator

#### 62. [SearchAnagramsDictionary.cpp](#)

- Example program
- Functions: main

#### 63. [SinglyLinkedList.cpp](#)

- pHead
- Functions: main, print

#### 64. [SOLIDPrinciples.cpp](#)

- =====
- Namespaces: open\_closed, single\_responsibility
- Classes: TaxCalculator, BadEmployee, Employee

#### 65. [STLContainersNoHeap.cpp](#)

- =====
- Classes: PoolAllocator, StackAllocator, FixedVector
- Functions: example\_std\_array, example\_std\_span, process\_data

#### 66. [StopTokenExample.cpp](#)

- StopTokenExample.cpp
- Namespaces: old\_vs\_new, std
- Classes: OldStyleThread, RESTService, NewStyleThread

#### 67. [StructuredBindings.cpp](#)

- =====
- Functions: example\_array\_bindings, example\_basic\_structured\_bindings, example\_tuple\_bindings

#### 68. [SystemInteractionAndParsing.cpp](#)

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

### 1.5.16 T

#### 69. [TemplatedCameraInterface.cpp](#)

- =====
- Classes: Image, Camera8bit, Camera
- Functions: demonstrate\_float\_camera, demonstrate\_8bit\_camera, demonstrate\_16bit\_camera

#### 70. [TemplatedCameraModules.cpp](#)

- =====
- Functions: demonstrate\_concepts, display\_camera\_info, demonstrate\_camera\_capture

#### 71. [ThreadPoolExamples.cpp](#)

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

#### 72. [TuplesAndStructuredBindings.cpp](#)

- =====
- Functions: demonstrate\_structured\_bindings, demonstrate\_tie, demonstrate\_basic\_tuples

### 1.5.17 U

#### 73. [UniversalResourceManager.cpp](#)

- =====
- Namespaces: std
- Functions: database\_example, file\_example, explain\_the\_problem

### 1.5.18 V

#### 74. [VariadicTemplateRecursion.cpp](#)

- =====
- Classes: GPIO\_Pin
- Functions: print\_recursive, demonstrate\_basic\_recursion

#### 75. [VirtualFunctionsInTemplates.cpp](#)

- VIRTUAL FUNCTIONS IN TEMPLATES: THE CODE BLOAT PROBLEM
- Namespaces: bad\_example\_t80, the\_problem
- Classes: DemoInstantiation, Vector, template

---

## 1.6 Project Structure

```
1 ModernCppExamples/
2   src/                      # All C++ source files (.cpp, .cppm)
3     Cpp11Examples.cpp        # C++11 features
4     Cpp20Examples.cpp       # C++20 features
5     RealTimeProgramming.cpp
6     SafetyCriticalSTLContainers.cpp
```

```

7  PimplIdiom.cpp
8  ... (70+ examples)
9
10 MarkDownDocuments/      # Detailed documentation
11   CPP20.md
12   EventDrivenProgramming.md
13   MultiThreadedMicroservices.md
14   Protobuf.md
15   Pybind11.md
16   SECURITY.md
17   ...
18
19 scripts/                # Build and utility scripts
20   build.sh                # Main build script
21   build_modules.sh        # C++20 modules build
22   build_protobuf.sh      # Protocol Buffers build
23   build_pybind.sh        # Pybind11 build
24   run_all.sh              # Run all examples
25
26 proto/                  # Protocol Buffer definitions
27   sensor_data.proto
28
29 build/                  # Build output directory
30   bin/                    # Compiled executables
31
32 CMakeLists.txt          # CMake configuration
33 INDEX.md                 # Comprehensive topic index
34 README.md                # This file
35 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](#), [MIS-RACppDemo.cpp](#)

- **Concurrency:** [Cpp17Concurrency.cpp](#), [FuturePromiseAsync.cpp](#), [ThreadPoolExamples.cpp](#)

### 1.7.3 By Design Pattern

- **CRTPO:** [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:

```
1 cd build/bin
2
3 # Run a specific example
4 ./Cpp20Examples
5
6 # Run all examples (Linux/Mac)
7 cd ../..
8 ./scripts/run_all.sh
9
10 # Run all examples (Windows PowerShell)
11 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](mailto: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](#)

```

1 #include <iostream>
2 #include <cstdint>
3 #include <cstring>
4 #include <array>
5
6 // =====
7 // ARM INSTRUCTION SETS: ARM, THUMB, THUMB-2
8 // =====
9 // This example discusses instruction set modes, alignment issues,
10 // and compiler switches for ARM Cortex-M processors.
11 //
12 // IMPORTANT: This is a DEMONSTRATION for understanding concepts.
13 // The actual crashes occur on ARM hardware, not x86/x64.
14 // Compiler flags discussed apply to ARM cross-compilers (arm-none-eabi-gcc).
15 // =====
16 //
17 // =====
18 // 1. ARM INSTRUCTION SET OVERVIEW
19 // =====
20
21 void explain_instruction_sets() {
22     std::cout << "\n== 1. ARM INSTRUCTION SETS OVERVIEW ==" << std::endl;
23
24     std::cout << "\n THREE INSTRUCTION SET MODES:" << std::endl;
25
26     std::cout << "\n1. ARM (A32) - 32-bit instructions:" << std::endl;
27     std::cout << " • Original ARM instruction set" << std::endl;
28     std::cout << " • Fixed 32-bit instruction size" << std::endl;
29     std::cout << " • Most powerful, but code size larger" << std::endl;
30     std::cout << " • Used in: ARM7, ARM9, ARM11, Cortex-A" << std::endl;
31     std::cout << " • Example instruction: ADD r0, r1, r2 (32 bits)" << std
32         ::endl;
33
34     std::cout << "\n2. THUMB (T16) - 16-bit instructions:" << std::endl;
35     std::cout << " • Compact 16-bit instruction subset" << std::endl;
36     std::cout << " • ~30% smaller code size vs ARM" << std::endl;
37     std::cout << " • Limited register access (r0-r7)" << std::endl;
38     std::cout << " • Used in: ARM7TDMI, ARM9, early Cortex-M0" << std::endl;
39     std::cout << " • Example: ADD r0, r1 (16 bits, 2-operand only)" << std
40         ::endl;
41
42     std::cout << "\n3. THUMB-2 (T32) - Mixed 16/32-bit:" << std::endl;
43     std::cout << " • Mix of 16-bit and 32-bit instructions" << std::endl;
44     std::cout << " • Best of both worlds: compact + powerful" << std::endl;
45     std::cout << " • Full register access, conditional execution" << std:

```

```

        std::endl;

46
47     std::cout << "\n MODERN CORTEX-M PROCESSORS:" << std::endl;
48     std::cout << " • Cortex-M0/M0+: Thumb + subset of Thumb-2" << std::endl;
49     std::cout << " • Cortex-M3/M4/M7: Thumb-2 only (no ARM mode)" << std::
50         endl;
50     std::cout << " • Cortex-M33/M55: Thumb-2 + DSP + optional FPU" << std::
50         endl;
51 }
52
53 // =====
54 // 2. ALIGNMENT ISSUES - THE MAIN CAUSE OF CRASHES
55 // =====
56
57 // Simulated aligned data structure
58 struct __attribute__((aligned(4))) AlignedData {
59     uint32_t value1;
60     uint32_t value2;
61     uint32_t value3;
62 };
63
64 // Potentially misaligned data
65 struct UnalignedData {
66     uint8_t byte;      // 1 byte
67     uint32_t value;    // Should be 4-byte aligned, but might not be!
68     uint16_t halfword; // 2 bytes
69 };
70
71 void explain_alignment_issues() {
72     std::cout << "\n==== 2. ALIGNMENT ISSUES (MAIN CRASH CAUSE) ===" << std::
72         endl;
73
74     std::cout << "\n ARM ALIGNMENT REQUIREMENTS:" << std::endl;
75     std::cout << " • uint8_t (byte): 1-byte aligned (any address)" <<
75         std::endl;
76     std::cout << " • uint16_t (halfword): 2-byte aligned (address % 2 == 0)
76         " << std::endl;
77     std::cout << " • uint32_t (word): 4-byte aligned (address % 4 == 0)
77         " << std::endl;
78     std::cout << " • uint64_t (doubleword): 8-byte aligned (address % 8 ==
78         0)" << std::endl;
79
80     std::cout << "\n WHAT HAPPENS ON MISALIGNED ACCESS:" << std::endl;
81     std::cout << " • Cortex-M0/M0+: HARDFAULT (crash)" << std::endl;
82     std::cout << " • Cortex-M3/M4/M7: May work BUT slower (extra cycles)" <<
82         std::endl;
83     std::cout << " • Depends on UNALIGN_TRP bit in CCR register" << std::
83         endl;
84
85     // Demonstrate alignment
86     std::cout << "\n ALIGNMENT EXAMPLES:" << std::endl;
87
88     AlignedData aligned;
89     std::cout << " AlignedData size: " << sizeof(AlignedData) << " bytes

```

```

        " << std::endl;
90    std::cout << "  Address of value1:      " << reinterpret_cast<uintptr_t>(&
91        aligned.value1)
92        << " (mod 4 = " << (reinterpret_cast<uintptr_t>(&aligned.value1)
93        % 4) << ")" << std::endl;
94
95    UnalignedData unaligned;
96    std::cout << "\n  UnalignedData size:      " << sizeof(UnalignedData) << " "
97        bytes" << std::endl;
98    std::cout << "  Address of byte:      " << reinterpret_cast<uintptr_t>(&
99        unaligned.byte) << std::endl;
100   std::cout << "  Address of value:      " << reinterpret_cast<uintptr_t>(&
101        unaligned.value)
102        << " (mod 4 = " << (reinterpret_cast<uintptr_t>(&unaligned.value)
103        % 4) << ")" << std::endl;
104
105   std::cout << "\n PROBLEM: If unaligned.value is at address 0x20000001:"
106        << std::endl;
107   std::cout << "  •  ARM instruction: LDR r0, [r1]  (load word from
108        unaligned address)" << std::endl;
109   std::cout << "  •  Result: HARDFAULT on Cortex-M0, silent corruption on
110        some others" << std::endl;
111 }
112
113 // =====
114 // 3. INTERWORKING ISSUES - STATE SWITCHING
115 // =====
116
117 void explain_interworking() {
118     std::cout << "\n==== 3. INTERWORKING ISSUES (STATE SWITCHING) ===" << std::
119         endl;
120
121     std::cout << "\n HOW PROCESSOR KNOWS WHICH MODE:" << std::endl;
122     std::cout << "  •  Function address LSB (Least Significant Bit):" << std::
123         endl;
124     std::cout << "      - LSB = 0: ARM mode (A32 instructions)" << std::endl;
125     std::cout << "      - LSB = 1: Thumb/Thumb-2 mode (T16/T32 instructions)"
126         << std::endl;
127     std::cout << "  •  BX/BLX instructions switch modes automatically" << std
128         ::endl;
129
130
131     std::cout << "\n CRASH SCENARIOS:" << std::endl;
132     std::cout << "\n  1. WRONG FUNCTION ADDRESS:" << std::endl;
133     std::cout << "      // C code" << std::endl;
134     std::cout << "      void my_function() { /* compiled as Thumb */ }" << std
135         ::endl;
136     std::cout << "      " << std::endl;
137     std::cout << "      // Assembly calling without setting LSB" << std::endl;
138     std::cout << "      LDR r0, =my_function ; Address without LSB=1" << std:
139         endl;
140     std::cout << "      BLX r0                  ; Tries to execute as ARM!" <<
141         std::endl;
142     std::cout << "      " << std::endl;
143     std::cout << "      CORRECT:" << std::endl;
144 }
```

```

127     std::cout << "      LDR r0, =my_function+1 ; Set LSB to indicate Thumb" <<
128         std::endl;
129     std::cout << "      BLX r0                      ; Correctly switches to
130         Thumb" << std::endl;
131
130     std::cout << "\n 2. INTERRUPT SERVICE ROUTINES (ISR):" << std::endl;
131     std::cout << " •      Some ARM7/ARM9 enter exceptions in ARM mode" << std::
132         endl;
132     std::cout << " •      If ISR compiled as Thumb: CRASH!" << std::endl;
133     std::cout << " •      Solution: Force ISR to ARM mode or ensure processor
133         supports Thumb ISRs" << std::endl;
134
135     std::cout << "\n 3. ASSEMBLY CODE MISMATCH:" << std::endl;
136     std::cout << "      // File: startup.s (ARM mode)" << std::endl;
137     std::cout << "      .arm" << std::endl;
138     std::cout << "      Reset_Handler:" << std::endl;
139     std::cout << "      LDR sp, =_estack" << std::endl;
140     std::cout << "      BL main           ; Calls main as ARM!" << std::
140         endl;
141     std::cout << "      " << std::endl;
142     std::cout << "      CORRECT:" << std::endl;
143     std::cout << "      .thumb           ; Switch to Thumb mode" << std::
143         endl;
144     std::cout << "      Reset_Handler:" << std::endl;
145     std::cout << "      LDR sp, =_estack" << std::endl;
146     std::cout << "      BLX main           ; Properly switches to main's
146         mode" << std::endl;
147 }
148
149 // =====
150 // 4. COMPILER SWITCHES AND FLAGS
151 // =====
152
153 void explain_compiler_switches() {
154     std::cout << "\n== 4. COMPILER SWITCHES FOR ARM ==" << std::endl;
155
156     std::cout << "\n ARM-NONE-EABI-GCC FLAGS:" << std::endl;
157
158     std::cout << "\n1. INSTRUCTION SET MODE:" << std::endl;
159     std::cout << "      -marm" << std::endl;
160     std::cout << " •      Generate ARM (A32) instructions" << std::endl;
161     std::cout << " •      Larger code size, faster execution" << std::endl;
162     std::cout << " •      Use for: ARM7, ARM9, Cortex-A" << std::endl;
163     std::cout << " •      Example: arm-none-eabi-gcc -marm -mcpu=arm7tdmi main.
163         c" << std::endl;
164
165     std::cout << "\n      -mthumb" << std::endl;
166     std::cout << " •      Generate Thumb/Thumb-2 instructions" << std::endl;
167     std::cout << " •      Smaller code size, good performance" << std::endl;
168     std::cout << " •      DEFAULT for Cortex-M (they only support Thumb-2)" <<
168         std::endl;
169     std::cout << " •      Example: arm-none-eabi-gcc -mthumb -mcpu=cortex-m4
169         main.c" << std::endl;
170

```

```

171     std::cout << "\n2. CPU/ARCHITECTURE:" << std::endl;
172     std::cout << "    -mcpu=cortex-m0      # Cortex-M0 (Thumb subset)" << std::
173         endl;
173     std::cout << "    -mcpu=cortex-m3      # Cortex-M3 (Thumb-2)" << std::endl;
174     std::cout << "    -mcpu=cortex-m4      # Cortex-M4 (Thumb-2 + DSP)" << std::
175         endl;
175     std::cout << "    -mcpu=cortex-m7      # Cortex-M7 (Thumb-2 + DSP + FPU)"
176         << std::endl;
176
177     std::cout << "\n3. ALIGNMENT OPTIONS:" << std::endl;
178     std::cout << "    -mno-unaligned-access" << std::endl;
179     std::cout << "    •      Disable unaligned memory access" << std::endl;
180     std::cout << "    •      Compiler generates multi-instruction sequences" <<
181         std::endl;
181     std::cout << "    •      Slower but safer on Cortex-M0" << std::endl;
182     std::cout << "    •      Example: arm-none-eabi-gcc -mthumb -mno-unaligned-
182         access" << std::endl;
183
183     std::cout << "\n    -munaligned-access (default on M3/M4/M7)" << std::endl;
184     std::cout << "    •      Allow unaligned memory access" << std::endl;
185     std::cout << "    •      Hardware handles it (with performance penalty)" <<
186         std::endl;
187     std::cout << "    •      Use for: Cortex-M3 and above" << std::endl;
188
189     std::cout << "\n4. INTERWORKING:" << std::endl;
190     std::cout << "    -mthumb-interwork" << std::endl;
191     std::cout << "    •      Generate code that can call between ARM and Thumb" <<
191         std::endl;
192     std::cout << "    •      Required for mixed ARM/Thumb projects" << std::endl;
193     std::cout << "    •      Not needed for Cortex-M (Thumb-2 only)" << std::endl;
194 }
195
196 // =====
197 // 5. PRACTICAL EXAMPLE: ALIGNMENT CRASH SIMULATION
198 // =====
199
200 // This simulates what happens on ARM hardware
201 void demonstrate_alignment_crash() {
202     std::cout << "\n== 5. ALIGNMENT CRASH SIMULATION ==" << std::endl;
203
204     std::cout << "\n NOTE: On x86/x64, unaligned access works (slower)" <<
204         std::endl;
205     std::cout << "    On ARM Cortex-M0: Would cause HARDFAULT" << std::endl;
206
207     // Create a buffer with known alignment
208     alignas(4) uint8_t buffer[16];
209
210     // Write data at aligned address (safe)
211     std::cout << "\n ALIGNED ACCESS (address % 4 == 0):" << std::endl;
212     uint32_t* aligned_ptr = reinterpret_cast<uint32_t*>(&buffer[0]);
213     *aligned_ptr = 0x12345678;
214     std::cout << "    Address: " << reinterpret_cast<uintptr_t>(aligned_ptr)
214         << " (aligned)" << std::endl;
215     std::cout << "    Value written: 0x" << std::hex << *aligned_ptr << std::

```

```

    dec << std::endl;
217  std::cout << "    Result:  SUCCESS (no crash on ARM)" << std::endl;
218
219 // Try unaligned access (dangerous on ARM!)
220 std::cout << "\n UNALIGNED ACCESS (address % 4 != 0):" << std::endl;
221 uint32_t* unaligned_ptr = reinterpret_cast<uint32_t*>(&buffer[1]);
222 std::cout << "    Address: " << reinterpret_cast<uintptr_t>(unaligned_ptr)
223             << " (unaligned, offset by 1)" << std::endl;
224
225 // On x86/x64 this works, on ARM Cortex-M0 it would HARDFAULT
226 std::cout << "    Attempting to write uint32_t at unaligned address..." <<
227             std::endl;
228 *unaligned_ptr = 0xAABBCCDD; // Works on x86, HARDFAULT on Cortex-M0!
229 std::cout << "    Result on x86:  Works (slow)" << std::endl;
230 std::cout << "    Result on ARM Cortex-M0:  HARDFAULT! (crash)" << std::endl;
231 std::cout << "    Result on ARM Cortex-M4:  Works but slow (multiple bus
232             cycles)" << std::endl;
233 }
234
235 // =====
236 // 6. SOLUTION: PORTABLE UNALIGNED ACCESS
237 // =====
238
239 //  SAFE: Portable unaligned read (works on all platforms)
240 uint32_t read_uint32_unaligned(const uint8_t* ptr) {
241     uint32_t value;
242     std::memcpy(&value, ptr, sizeof(value)); // Compiler optimizes this
243     return value;
244 }
245
246 //  SAFE: Portable unaligned write
247 void write_uint32_unaligned(uint8_t* ptr, uint32_t value) {
248     std::memcpy(ptr, &value, sizeof(value));
249 }
250
251 //  UNSAFE: Direct cast (crashes on unaligned ARM)
252 uint32_t read_uint32_unsafe(const uint8_t* ptr) {
253     return *reinterpret_cast<const uint32_t*>(ptr); // HARDFAULT if unaligned
254     !
255 }
256
257 void demonstrate_portable_unaligned() {
258     std::cout << "\n== 6. PORTABLE UNALIGNED ACCESS ==" << std::endl;
259
260     alignas(4) uint8_t buffer[16] = {};
261
262     std::cout << "\n  SAFE METHOD (using memcpy):" << std::endl;
263     std::cout << "    uint32_t read_uint32_unaligned(const uint8_t* ptr) {" <<
264             std::endl;
265     std::cout << "        uint32_t value;" << std::endl;
266     std::cout << "        std::memcpy(&value, ptr, sizeof(value));" << std::endl;
267     std::cout << "        return value;" << std::endl;

```

```

264     std::cout << "    }" << std::endl;
265
266     write_uint32_unaligned(&buffer[1], 0x12345678);
267     uint32_t safe_value = read_uint32_unaligned(&buffer[1]);
268     std::cout << "\n    Written to unaligned address using memcpy" << std::endl
269     ;
270     std::cout << "    Value: 0x" << std::hex << safe_value << std::dec << std::
271     endl;
272     std::cout << "    Works on ALL platforms (x86, ARM, RISC-V)" << std::endl
273     ;
274     std::cout << "    Compiler optimizes memcpy to efficient code" << std::
275     endl;
276
277     std::cout << "\n    UNSAFE METHOD (direct cast):" << std::endl;
278     std::cout << "    uint32_t value = *reinterpret_cast<uint32_t*>(ptr);" <<
279     std::endl;
280     std::cout << "    HARDFAULT on ARM Cortex-M0 if unaligned!" << std::endl;
281 }
282
283 // =====
284 // 7. ASSEMBLY DIRECTIVES FOR MIXED CODE
285 // =====
286
287 void explain_assembly_directives() {
288     std::cout << "\n==== 7. ASSEMBLY DIRECTIVES ===" << std::endl;
289
290     std::cout << "\n    ARM ASSEMBLY DIRECTIVES:" << std::endl;
291
292     std::cout << "\n1. SET INSTRUCTION MODE:" << std::endl;
293     std::cout << "    .arm                      ; Switch to ARM (A32) mode" << std::
294     endl;
295     std::cout << "    .thumb                     ; Switch to Thumb/Thumb-2 (T16/T32)
296     mode" << std::endl;
297
298     std::cout << "\n2. MARK THUMB FUNCTIONS:" << std::endl;
299     std::cout << "    .thumb_func            ; Next symbol is a Thumb function" <<
300     std::endl;
301     std::cout << "    " << std::endl;
302     std::cout << "    Example:" << std::endl;
303     std::cout << "    .thumb" << std::endl;
304     std::cout << "    .thumb_func" << std::endl;
305     std::cout << "    my_function:" << std::endl;
306     std::cout << "        PUSH {lr}" << std::endl;
307     std::cout << "        ; ... function code ..." << std::endl;
308     std::cout << "        POP {pc}" << std::endl;
309
310     std::cout << "\n3. ALIGNMENT DIRECTIVES:" << std::endl;
311     std::cout << "    .align 2                  ; Align to 2^2 = 4 bytes (word)" << std
312     ::endl;
313     std::cout << "    .align 3                  ; Align to 2^3 = 8 bytes (doubleword)" <<
314     std::endl;
315     std::cout << "    " << std::endl;
316     std::cout << "    Example:" << std::endl;
317     std::cout << "    .align 2                  ; Force 4-byte alignment" << std::endl;

```

```

308     std::cout << "    my_data:" << std::endl;
309     std::cout << "          .word 0x12345678" << std::endl;
310 }
311
312 // =====
313 // 8. REAL-WORLD EXAMPLE: PROTOCOL PARSING
314 // =====
315
316 // Common scenario: Parsing network packets with misaligned fields
317 struct __attribute__((packed)) NetworkPacket {
318     uint8_t header;          // 1 byte
319     uint32_t timestamp;      // 4 bytes (might be misaligned!)
320     uint16_t length;         // 2 bytes
321     uint32_t crc;            // 4 bytes (might be misaligned!)
322 };
323
324 void demonstrate_real_world_scenario() {
325     std::cout << "\n==== 8. REAL-WORLD: PROTOCOL PARSING ===" << std::endl;
326
327     std::cout << "\n SCENARIO: Parsing network packet on Cortex-M4" << std::endl;
328
329     // Simulate receiving a packet
330     alignas(4) uint8_t rx_buffer[16] = {
331         0xAA,                                // header (1 byte)
332         0x78, 0x56, 0x34, 0x12,              // timestamp (4 bytes, at offset 1 =
333         // UNALIGNED!)
334         0x10, 0x00,                          // length (2 bytes)
335         0xDD, 0xCC, 0xBB, 0xAA            // crc (4 bytes)
336     };
337
338     std::cout << "\n WRONG WAY (direct struct cast):" << std::endl;
339     std::cout << "    NetworkPacket* pkt = (NetworkPacket*)rx_buffer;" << std::endl;
340     std::cout << "    uint32_t ts = pkt->timestamp; // Unaligned access!" <<
341     std::cout << std::endl;
342     std::cout << "    " << std::endl;
343     std::cout << "    Result: HARDFAULT on Cortex-M0, slow on Cortex-M4" << std::endl;
344
345     std::cout << "\n CORRECT WAY (memcpy for unaligned fields):" << std::endl;
346     std::cout << "    ";
347     uint8_t header = rx_buffer[0];
348     uint32_t timestamp;
349     uint16_t length;
350     uint32_t crc;
351
352     std::memcpy(&timestamp, &rx_buffer[1], sizeof(timestamp));
353     std::memcpy(&length, &rx_buffer[5], sizeof(length));
354     std::memcpy(&crc, &rx_buffer[7], sizeof(crc));
355
356     std::cout << "    Header: 0x" << std::hex << static_cast<int>(header) <<
357     std::endl;
358     std::cout << "    Timestamp: 0x" << timestamp << std::endl;

```

```

355     std::cout << "    Length: 0x" << length << std::endl;
356     std::cout << "    CRC: 0x" << crc << std::dec << std::endl;
357     std::cout << "    Works on ALL ARM processors!" << std::endl;
358
359     std::cout << "\n COMPILER OPTIMIZATIONS:" << std::endl;
360     std::cout << " • With -O2: memcpy() compiles to efficient LDR/STR" <<
361         std::endl;
362     std::cout << " • No function call overhead" << std::endl;
363     std::cout << " • Compiler knows about alignment and handles it" << std::
364         endl;
365 }
366
367 // =====
368 // 9. COMPILER FLAG EXAMPLES
369 // =====
370
371 void show_compiler_examples() {
372     std::cout << "\n== 9. PRACTICAL COMPILER COMMAND EXAMPLES ==" << std::
373         endl;
374
375     std::cout << "\n1. CORTEX-M0 PROJECT (strict alignment):" << std::endl;
376     std::cout << "    CFLAGS = -mcpu=cortex-m0 \\\" << std::endl;
377     std::cout << "                -mthumb \\\" << std::endl;
378     std::cout << "                -mno-unaligned-access \\\" << std::endl;
379     std::cout << "                -O2 -Wall" << std::endl;
380
381     std::cout << "\n2. CORTEX-M4 PROJECT (with FPU):" << std::endl;
382     std::cout << "    CFLAGS = -mcpu=cortex-m4 \\\" << std::endl;
383     std::cout << "                -mthumb \\\" << std::endl;
384     std::cout << "                -mfpu=fpv4-sp-d16 \\\" << std::endl;
385     std::cout << "                -mfloating-abi=hard \\\" << std::endl;
386     std::cout << "                -munaligned-access \\\" << std::endl;
387     std::cout << "                -O2 -Wall" << std::endl;
388
389     std::cout << "\n3. MIXED ARM/THUMB PROJECT (ARM7):" << std::endl;
390     std::cout << "    CFLAGS = -mcpu=arm7tdmi \\\" << std::endl;
391     std::cout << "                -mthumb \\\" << std::endl;
392     std::cout << "                -mthumb-interwork \\\" << std::endl;
393     std::cout << "                -O2 -Wall" << std::endl;
394
395     std::cout << "\n4. ASSEMBLY FILE COMPILATION:" << std::endl;
396     std::cout << "    # For Thumb-2 code" << std::endl;
397     std::cout << "    arm-none-eabi-as -mcpu=cortex-m4 \\\" << std::endl;
398     std::cout << "                -mthumb \\\" << std::endl;
399     std::cout << "                startup.s -o startup.o" << std::endl;
400 }
401
402 // =====
403 // 10. DEBUGGING HARDFAULTS
404 // =====
405 void explain_debugging_hardfaults() {

```

```

406     std::cout << "\n==== 10. DEBUGGING HARDFAULTS ON ARM ===" << std::endl;
407
408     std::cout << "\n WHEN YOU GET A HARDFAULT:" << std::endl;
409
410     std::cout << "\n1. CHECK FAULT STATUS REGISTERS:" << std::endl;
411     std::cout << " • HFSR (HardFault Status Register) at 0xE000ED2C" << std
412         ::endl;
413     std::cout << " • CFSR (Configurable Fault Status) at 0xE000ED28" << std
414         ::endl;
415     std::cout << " • MMFAR (MemManage Fault Address) at 0xE000ED34" << std::
416         endl;
417     std::cout << " • BFAR (BusFault Address) at 0xE000ED38" << std::endl;
418
419     std::cout << "\n2. COMMON FAULT CAUSES:" << std::endl;
420     std::cout << "     IBUSERR (bit 0 of CFSR): Instruction bus error" << std::
421         endl;
422     std::cout << "     → Called function at wrong address or wrong mode" <<
423         std::endl;
424     std::cout << "     " << std::endl;
425     std::cout << "     PRECISERR (bit 1 of CFSR): Data bus error" << std::endl;
426     std::cout << "     → Unaligned access on Cortex-M0" << std::endl;
427     std::cout << "     " << std::endl;
428     std::cout << "     IACCVIOL (bit 0 of CFSR): MPU violation" << std::endl;
429     std::cout << "     → Tried to execute from non-executable memory" << std::
430         endl;
431
432     std::cout << "\n3. DEBUGGER INSPECTION:" << std::endl;
433     std::cout << "     (gdb) info registers # Check PC, LR, SP" << std::endl;
434     std::cout << "     (gdb) x/4x $sp      # Stack contents" << std::endl;
435     std::cout << "     (gdb) bt          # Backtrace" << std::endl;
436     std::cout << "     " << std::endl;
437     std::cout << "     Look for:" << std::endl;
438     std::cout << "     • PC (Program Counter) - where crash occurred" << std::
439         endl;
440     std::cout << "     • LR (Link Register) - return address (LSB = mode)" <<
441         std::endl;
442     std::cout << "     • Unaligned addresses (check if address % 4 != 0)" << std
443         ::endl;
444
445 }

446 // =====
447 // MAIN
448 // =====
449
450 int main() {
451     std::cout << "\n
452     =====
453     std::endl;
454     std::cout << "     ARM INSTRUCTION SETS: ARM, THUMB, THUMB-2" << std::endl;
455     std::cout << "     Alignment Issues and Compiler Switches" << std::endl;
456     std::cout << "
457     =====
458     std::endl;

```

```

447     explain_instruction_sets();
448     explain_alignment_issues();
449     explain_interworking();
450     explain_compiler_switches();
451     demonstrate_alignment_crash();
452     demonstrate_portable_unaligned();
453     explain_assembly_directives();
454     demonstrate_real_world_scenario();
455     show_compiler_examples();
456     explain_debugging_hardfaults();

457
458     std::cout << "\n"
459     ===== " SUMMARY: AVOIDING ARM THUMB-2 CRASHES" << std::endl;
460
461     std::cout << "\n KEY TAKEAWAYS:" << std::endl;
462
463
464     std::cout << "\n1. INSTRUCTION SET MODES:" << std::endl;
465     std::cout << " • ARM (A32): 32-bit, powerful, larger code" << std::endl;
466     std::cout << " • Thumb (T16): 16-bit, compact, limited" << std::endl;
467     std::cout << " • Thumb-2 (T32): 16/32-bit mix, best of both" << std::
468         endl;
469     std::cout << " • Cortex-M: Thumb-2 only (no ARM mode)" << std::endl;
470
471     std::cout << "\n2. ALIGNMENT REQUIREMENTS:" << std::endl;
472     std::cout << " • uint32_t MUST be 4-byte aligned on Cortex-M0" << std::
473         endl;
474     std::cout << " • Cortex-M3/M4/M7 can handle unaligned (but slower)" <<
475         std::endl;
476     std::cout << " • Use std::memcpy() for portable unaligned access" << std
477         ::endl;
478
479
480     std::cout << "\n3. INTERWORKING:" << std::endl;
481     std::cout << " • Function address LSB indicates mode (0=ARM, 1=Thumb)" <<
482         std::endl;
483     std::cout << " • Use BLX instruction for mode switching" << std::endl;
484     std::cout << " • Mark assembly functions with .thumb_func" << std::endl;
485
486
487     std::cout << "\n4. COMPILER FLAGS:" << std::endl;
488     std::cout << "     -mthumb           # Use Thumb-2 instructions" <<
489         std::endl;
490     std::cout << "     -mcpu=cortex-m4      # Specify processor" << std::
491         endl;
492     std::cout << "     -mno-unaligned-access # Disable unaligned (M0)" <<
493         std::endl;
494     std::cout << "     -munaligned-access    # Enable unaligned (M3+)" <<
495         std::endl;
496
497
498     std::cout << "\n5. BEST PRACTICES:" << std::endl;
499     std::cout << "     Use std::memcpy() for unaligned access" << std::endl;

```

```
488     std::cout << "      Align structs with __attribute__((aligned(4)))" << std
489             ::endl;
490     std::cout << "      Use __attribute__((packed)) carefully" << std::endl;
491     std::cout << "      Compile with correct -mcpu flag" << std::endl;
492     std::cout << "      Test on actual hardware (not just simulator)" << std::
493             endl;
494     std::cout << "      Never cast unaligned pointers directly" << std::endl;
495     std::cout << "      Don't mix ARM/Thumb without proper interworking" << std
496             ::endl;
497
498     std::cout << "\n DEBUGGING TIPS:" << std::endl;
499     std::cout << "      1. Enable HardFault handler to print fault registers" <<
500             std::endl;
501     std::cout << "      2. Check CFSR register to identify fault type" << std::
502             endl;
503     std::cout << "      3. Look for unaligned addresses in fault address
504             registers" << std::endl;
505     std::cout << "      4. Verify -mcpu matches your actual hardware" << std::
506             endl;
507     std::cout << "      5. Check linker script for correct alignment" << std::
508             endl;
509
510     std::cout << "\n
511     =====\n" << std::endl;
512
513     return 0;
514 }
```

### 3 Source Code: AdvancedExceptionHandling.cpp

File: src/AdvancedExceptionHandling.cpp

Repository: [View on GitHub](#)

```
1 // AdvancedExceptionHandling.cpp
2 // Comprehensive exception handling with real stack traces, nested exceptions,
3 // and advanced error reporting patterns
4
5 #include <iostream>
6 #include <stdexcept>
7 #include <string>
8 #include <vector>
9 #include <memory>
10 #include <sstream>
11 #include <iomanip>
12 #include <chrono>
13 #include <functional>
14 #include <exception>
15 #include <typeinfo>
16 #include <cstdlib>
17 #include <cstring>
18
19 // Platform-specific includes for stack traces
20 #ifdef __linux__
21     #include <execinfo.h>
22     #include <cxxabi.h>
23     #include <unistd.h>
24 #elif _WIN32
25     #include <windows.h>
26     #include <dbghelp.h>
27     #pragma comment(lib, "dbghelp.lib")
28 #endif
29
30 using namespace std::chrono;
31
32 // =====
33 // SECTION 1: Stack Trace Capture (Platform-Specific)
34 // =====
35
36 class StackTrace {
37 private:
38     static constexpr size_t MAX_FRAMES = 64;
39     std::vector<std::string> frames_;
40
41 public:
42     StackTrace() {
43         capture();
44     }
45 }
```

```

46 void capture() {
47     frames_.clear();
48
49 #ifdef __linux__
50     // Linux: Use backtrace() and backtrace_symbols()
51     void* buffer[MAX_FRAMES];
52     int frame_count = backtrace(buffer, MAX_FRAMES);
53
54     char** symbols = backtrace_symbols(buffer, frame_count);
55     if (symbols) {
56         for (int i = 0; i < frame_count; ++i) {
57             frames_.push_back(demangle_symbol(symbols[i]));
58         }
59         free(symbols);
60     }
61
62 #elif _WIN32
63     // Windows: Use CaptureStackBackTrace() and SymFromAddr()
64     void* buffer[MAX_FRAMES];
65     HANDLE process = GetCurrentProcess();
66     SymInitialize(process, NULL, TRUE);
67
68     WORD frame_count = CaptureStackBackTrace(0, MAX_FRAMES, buffer, NULL);
69
70     SYMBOL_INFO* symbol = (SYMBOL_INFO*)calloc(sizeof(SYMBOL_INFO) + 256 *
71         sizeof(char), 1);
72     if (symbol) {
73         symbol->MaxNameLen = 255;
74         symbol->SizeOfStruct = sizeof(SYMBOL_INFO);
75
76         for (WORD i = 0; i < frame_count; ++i) {
77             if (SymFromAddr(process, (DWORD64)(buffer[i]), 0, symbol)) {
78                 frames_.push_back(std::string(symbol->Name));
79             } else {
80                 std::ostringstream oss;
81                 oss << "0x" << std::hex << (uint64_t)buffer[i];
82                 frames_.push_back(oss.str());
83             }
84         }
85         free(symbol);
86     }
87
88 #else
89     frames_.push_back("[Stack trace not available on this platform]");
90 #endif
91 }
92
93 const std::vector<std::string>& get_frames() const {
94     return frames_;
95 }
96
97 std::string to_string() const {
98     std::ostringstream oss;

```

```

99         oss << "Stack Trace (" << frames_.size() << " frames):\n";
100        for (size_t i = 0; i < frames_.size(); ++i) {
101            oss << " #" << std::setw(2) << i << ":" << frames_[i] << "\n";
102        }
103        return oss.str();
104    }
105
106 private:
107 #ifdef __linux__
108     std::string demangle_symbol(const char* mangled) {
109         std::string result = mangled;
110
111         // Extract the mangled name between '(' and '+'
112         const char* begin = strchr(mangled, '(');
113         const char* end = strchr(mangled, '+');
114
115         if (begin && end && begin < end) {
116             begin++; // Skip '('
117             std::string mangled_name(begin, end - begin);
118
119             int status;
120             char* demangled = abi::__cxa_demangle(mangled_name.c_str(),
121                                         nullptr, nullptr, &status);
122
123             if (status == 0 && demangled) {
124                 result = demangled;
125                 free(demangled);
126             }
127         }
128
129         return result;
130     }
131 #endif
132 };
133
134 void demonstrate_stack_trace_capture() {
135     std::cout << "\n" << std::string(70, '=') << "\n";
136     std::cout << "==== 1. Real Stack Trace Capture ====\n";
137     std::cout << std::string(70, '=') << "\n\n";
138
139     std::cout << "Capturing stack trace from current location...\n\n";
140
141     StackTrace trace;
142     std::cout << trace.to_string();
143
144     std::cout << "\n Stack trace captured using platform-specific APIs\n";
145 #ifdef __linux__
146     std::cout << " Linux: backtrace() + abi::__cxa_demangle()\n";
147 #elif _WIN32
148     std::cout << " Windows: CaptureStackBackTrace() + SymFromAddr()\n";
149 #else
150     std::cout << " Generic fallback (no native stack trace support)\n";
151 #endif
152 }

```

```
152 //  
153 //=====  
154 // SECTION 2: Exception with Stack Trace  
155 //=====  
156  
157 class TracedException : public std::runtime_error {  
158 private:  
159     std::string file_;  
160     int line_;  
161     std::string function_;  
162     StackTrace stack_trace_;  
163     std::string formatted_message_;  
164  
165     std::string format_message() const {  
166         std::ostringstream oss;  
167  
168         // Extract filename only  
169         size_t last_slash = file_.find_last_of("/\\");  
170         std::string filename = (last_slash != std::string::npos)  
171             ? file_.substr(last_slash + 1) : file_;  
172  
173         oss << "\n";  
174         oss << "EXCEPTION THROWN WITH STACK TRACE\n";  
175         oss << "\n";  
176         oss << "Location: " << filename << ":" << line_ << "\n";  
177         oss << "Function: " << function_ << "() \n";  
178         oss << "Message: " << what() << "\n";  
179         oss << "\n";  
180         oss << " " << stack_trace_.to_string();  
181         oss << "\n";  
182  
183     return oss.str();  
184 }  
185  
186 public:  
187     TracedException(const std::string& file, int line, const std::string& func  
188         ,  
189         const std::string& message)  
190         : std::runtime_error(message),  
191         file_(file),  
192         line_(line),  
193         function_(func),  
194         stack_trace_(),  
195         formatted_message_(format_message()) {}  
196  
197     const std::string& get_formatted_message() const {  
198         return formatted_message_;  
199     }  
200  
201     const StackTrace& get_stack_trace() const {
```

```

201     return stack_trace_;
202 }
203
204 const std::string& get_file() const { return file_; }
205 int get_line() const { return line_; }
206 const std::string& get_function() const { return function_; }
207 };
208
209 #define THROW_TRACED(message) \
210     throw TracedException(__FILE__, __LINE__, __FUNCTION__, message)
211
212 void nested_function_3() {
213     std::cout << " [3] About to throw exception...\n";
214     THROW_TRACED("Critical error in nested function");
215 }
216
217 void nested_function_2() {
218     std::cout << " [2] Calling nested_function_3()...\n";
219     nested_function_3();
220 }
221
222 void nested_function_1() {
223     std::cout << " [1] Calling nested_function_2()...\n";
224     nested_function_2();
225 }
226
227 void demonstrate_exception_with_stack_trace() {
228     std::cout << "\n" << std::string(70, '=') << "\n";
229     std::cout << "== 2. Exception with Stack Trace ==\n";
230     std::cout << std::string(70, '=') << "\n\n";
231
232     std::cout << "Calling nested functions to build call stack...\n\n";
233
234     try {
235         nested_function_1();
236     }
237     catch (const TracedException& e) {
238         std::cout << "\n Exception caught!\n";
239         std::cout << e.get_formatted_message();
240     }
241 }
242
243 // =====
244 // SECTION 3: Nested Exceptions (C++11)
245 // =====
246
247 class DatabaseException : public std::runtime_error {
248 public:
249     explicit DatabaseException(const std::string& msg)
250         : std::runtime_error("Database Error: " + msg) {}

```

```
251 };
```

```
252
```

```
253 class ConnectionException : public std::runtime_error {
```

```
254 public:
```

```
255     explicit ConnectionException(const std::string& msg)
```

```
256         : std::runtime_error("Connection Error: " + msg) {}
```

```
257 };
```

```
258
```

```
259 void database_operation() {
```

```
260     std::cout << "    Attempting database query...\n";
```

```
261     throw DatabaseException("Query timeout after 30 seconds");
```

```
262 }
```

```
263
```

```
264 void connect_to_database() {
```

```
265     std::cout << "    Attempting to connect to database...\n";
```

```
266
```

```
267     try {
```

```
268         database_operation();
```

```
269     }
```

```
270     catch (const DatabaseException& e) {
```

```
271         std::cout << "        Database operation failed!\n";
```

```
272         // Wrap the DatabaseException in a ConnectionException
```

```
273         std::throw_with_nested(ConnectionException("Failed to establish
```

```
274             connection"));
```

```
275     }
```

```
276 }
```

```
277
```

```
278 void print_nested_exception(const std::exception& e, int level = 0) {
```

```
279     std::string indent(level * 2, ' '');
```

```
280     std::cout << indent << "    " << e.what() << "\n";
```

```
281
```

```
282     try {
```

```
283         std::rethrow_if_nested(e);
```

```
284     }
```

```
285     catch (const std::exception& nested) {
```

```
286         print_nested_exception(nested, level + 1);
```

```
287     }
```

```
288 }
```

```
289
```

```
290 void demonstrate_nested_exceptions() {
```

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

```
292     std::cout << "==== 3. Nested Exceptions (C++11) ===\n";
```

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

```
294
```

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

```
296
```

```
297     try {
```

```
298         connect_to_database();
```

```
299     }
```

```
300     catch (const std::exception& e) {
```

```
301         std::cout << "\n Exception caught with nested context:\n\n";
```

```
302         print_nested_exception(e);
```

```
303     }
```

```
304     std::cout << "\n Original exception preserved inside wrapper\n";
305     std::cout << "  Use std::throw_with_nested() and std::rethrow_if_nested()\n";
306 }
307 //
308 //=====
309 // SECTION 4: Exception Guarantee Levels
310 //=====
311
312 template<typename T>
313 class Container {
314 private:
315     std::unique_ptr<T[]> data_;
316     size_t size_;
317     size_t capacity_;
318
319 public:
320     Container() : data_(nullptr), size_(0), capacity_(0) {}
321
322     // Basic guarantee: valid state but data may be lost
323     void push_back_basic(const T& value) {
324         if (size_ == capacity_) {
325             size_t new_capacity = (capacity_ == 0) ? 1 : capacity_ * 2;
326             auto new_data = std::make_unique<T[]>(new_capacity);
327
328             // If copy throws here, we're in an inconsistent state
329             for (size_t i = 0; i < size_; ++i) {
330                 new_data[i] = data_[i]; // May throw!
331             }
332
333             data_ = std::move(new_data);
334             capacity_ = new_capacity;
335         }
336
337         data_[size_++] = value;
338     }
339
340     // Strong guarantee: operation succeeds completely or has no effect
341     void push_back_strong(const T& value) {
342         if (size_ == capacity_) {
343             size_t new_capacity = (capacity_ == 0) ? 1 : capacity_ * 2;
344             auto new_data = std::make_unique<T[]>(new_capacity);
345
346             // Copy all elements (may throw)
347             for (size_t i = 0; i < size_; ++i) {
348                 new_data[i] = data_[i];
349             }
350
351             // Only if successful, commit changes (no-throw from here)
352             data_ = std::move(new_data);
353     }
354 }
```

```
353         capacity_ = new_capacity;
354     }
355
356     data_[size_++] = value;
357 }
358
359 // No-throw guarantee: never throws
360 size_t size() const noexcept {
361     return size_;
362 }
363
364 // No-throw guarantee
365 void clear() noexcept {
366     size_ = 0;
367 }
368 };
369
370 void demonstrate_exception_guarantees() {
371     std::cout << "\n" << std::string(70, '=') << "\n";
372     std::cout << "==== 4. Exception Safety Guarantees ===\n";
373     std::cout << std::string(70, '=') << "\n\n";
374
375     std::cout << "1. No-throw guarantee (noexcept):\n";
376     std::cout << " • Function never throws\n";
377     std::cout << " • Example: size(), clear(), swap()\n";
378     std::cout << " • Mark with 'noexcept' keyword\n\n";
379
380     std::cout << "2. Strong guarantee:\n";
381     std::cout << " • Operation succeeds completely OR has no effect\n";
382     std::cout << " • Example: push_back() using copy-and-swap\n";
383     std::cout << " • All or nothing - no partial state\n\n";
384
385     std::cout << "3. Basic guarantee:\n";
386     std::cout << " • Object left in valid state if exception thrown\n";
387     std::cout << " • Example: push_back() that may leave capacity changed\n";
388     ;
389     std::cout << " • No resource leaks, but data may be lost\n\n";
390
391     std::cout << "4. No guarantee:\n";
392     std::cout << " • May leave object in invalid state\n";
393     std::cout << " • AVOID - leads to crashes and undefined behavior\n\n";
394
395     Container<int> cont;
396     cont.push_back_strong(42);
397     std::cout << " Container with strong guarantee used successfully\n";
398 }
399
400 // =====
401 // SECTION 5: RAII and Exception Safety
402 // =====
```

```
402
403 class FileRAII {
404 private:
405     FILE* file_;
406     std::string filename_;
407
408 public:
409     explicit FileRAII(const std::string& filename)
410         : file_(nullptr), filename_(filename) {
411
412         file_ = fopen(filename.c_str(), "w");
413         if (!file_) {
414             throw std::runtime_error("Failed to open file: " + filename);
415         }
416         std::cout << "  [RAII] File opened: " << filename << "\n";
417     }
418
419     ~FileRAII() {
420         if (file_) {
421             fclose(file_);
422             std::cout << "  [RAII] File closed: " << filename_ << "\n";
423         }
424     }
425
426     void write(const std::string& data) {
427         if (!file_) {
428             throw std::runtime_error("File not open");
429         }
430
431         if (data == "ERROR") {
432             throw std::runtime_error("Simulated write error");
433         }
434
435         fprintf(file_, "%s\n", data.c_str());
436     }
437
438     // Delete copy/move to enforce RAII
439     FileRAII(const FileRAII&) = delete;
440     FileRAII& operator=(const FileRAII&) = delete;
441 };
442
443 void demonstrate_raii_exception_safety() {
444     std::cout << "\n" << std::string(70, '=') << "\n";
445     std::cout << "==== 5. RAII and Exception Safety ===\n";
446     std::cout << std::string(70, '=') << "\n\n";
447
448     std::cout << "Without RAII (manual cleanup - leak on exception):\n\n";
449     try {
450         FILE* f = fopen("/tmp/test.txt", "w");
451         std::cout << "  File opened\n";
452         // If exception thrown here, file never closed!
453         throw std::runtime_error("Oops!");
454         fclose(f); // Never executed
455     }
```

```
456     catch (...) {
457         std::cout << "    File leaked - never closed!\n\n";
458     }
459
460     std::cout << "With RAII (automatic cleanup - safe):\n\n";
461     try {
462         FileRAII file("/tmp/test_raii.txt");
463         file.write("Hello");
464         file.write("ERROR"); // Throws exception
465         file.write("World"); // Never executed
466     }
467     catch (const std::exception& e) {
468         std::cout << "    Exception: " << e.what() << "\n";
469         std::cout << "    RAII destructor automatically closed file!\n";
470     }
471
472     std::cout << "\n RAII guarantees resource cleanup even with exceptions\n"
473     ;
474 }
475 // =====
476 // SECTION 6: std::exception_ptr and Thread Exception Propagation
477 // =====
478
479 #include <thread>
480
481 std::exception_ptr worker_exception = nullptr;
482
483 void worker_thread_function() {
484     try {
485         std::cout << " [Worker] Doing some work...\n";
486         std::this_thread::sleep_for(std::chrono::milliseconds(100));
487
488         // Simulate error
489         throw std::runtime_error("Worker thread encountered an error!");
490     }
491     catch (...) {
492         std::cout << " [Worker] Exception caught, storing for main thread\n";
493         worker_exception = std::current_exception();
494     }
495 }
496
497 void demonstrate_exception_ptr() {
498     std::cout << "\n" << std::string(70, '=') << "\n";
499     std::cout << "==== 6. std::exception_ptr (Thread Exception Propagation)
500         ==\n";
501     std::cout << std::string(70, '=') << "\n\n";
502
503     std::cout << "Problem: Exceptions can't cross thread boundaries\n";
504     std::cout << "Solution: std::exception_ptr + std::current_exception()\n\n"
```

```
    ;  
504  
505     std::cout << "Starting worker thread...\n";  
506  
507     worker_exception = nullptr;  
508     std::thread worker(worker_thread_function);  
509     worker.join();  
510  
511     std::cout << "\nMain thread checking for worker exception...\n";  
512  
513     if (worker_exception) {  
514         try {  
515             std::rethrow_exception(worker_exception);  
516         }  
517         catch (const std::exception& e) {  
518             std::cout << " Worker exception received: " << e.what() << "\n";  
519         }  
520     }  
521  
522     std::cout << "\n Exception successfully propagated from worker to main  
523         thread\n";  
524 }  
525 //  
=====  
526 // SECTION 7: Custom Exception Hierarchy  
527 //  
=====  
528  
529 class ApplicationException : public std::runtime_error {  
530 protected:  
531     int error_code_;  
532     system_clock::time_point timestamp_;  
533  
534 public:  
535     ApplicationException(int code, const std::string& msg)  
536         : std::runtime_error(msg),  
537         error_code_(code),  
538         timestamp_(system_clock::now()) {}  
539  
540     int get_error_code() const { return error_code_; }  
541  
542     std::string get_timestamp() const {  
543         auto time = system_clock::to_time_t(timestamp_);  
544         std::ostringstream oss;  
545         oss << std::put_time(std::localtime(&time), "%Y-%m-%d %H:%M:%S");  
546         return oss.str();  
547     }  
548 };  
549  
550 class NetworkError : public ApplicationException {  
551 public:
```

```
552     NetworkError(int code, const std::string& msg)
553         : ApplicationException(code, msg) {}
554     };
555
556     class FileSystemError : public ApplicationException {
557     public:
558         FileSystemError(int code, const std::string& msg)
559             : ApplicationException(code, msg) {}
560     };
561
562     class BusinessLogicError : public ApplicationException {
563     public:
564         BusinessLogicError(int code, const std::string& msg)
565             : ApplicationException(code, msg) {}
566     };
567
568     void demonstrate_exception_hierarchy() {
569         std::cout << "\n" << std::string(70, '=') << "\n";
570         std::cout << "==== 7. Custom Exception Hierarchy ====\n";
571         std::cout << std::string(70, '=') << "\n\n";
572
573         std::cout << "Exception Hierarchy:\n";
574         std::cout << "    ApplicationException (base)\n";
575         std::cout << "        NetworkError\n";
576         std::cout << "        FileSystemError\n";
577         std::cout << "        BusinessLogicError\n\n";
578
579     try {
580         throw NetworkError(503, "Service unavailable");
581     }
582     catch (const NetworkError& e) {
583         std::cout << "Caught NetworkError:\n";
584         std::cout << "    Code:      " << e.get_error_code() << "\n";
585         std::cout << "    Message:   " << e.what() << "\n";
586         std::cout << "    Timestamp: " << e.get_timestamp() << "\n";
587     }
588     catch (const ApplicationException& e) {
589         std::cout << "Caught generic ApplicationException\n";
590     }
591
592     std::cout << "\n Specific catch before generic catch\n";
593     std::cout << "  Polymorphic exception handling\n";
594 }
595
596 /**
597 // SECTION 8: Function Try Blocks (Constructor Exception Handling)
598 /**
599
600 class Resource {
601 public:
```

```
602     Resource(int id) {
603         std::cout << "      [Resource " << id << "] Constructed\n";
604         if (id == 2) {
605             throw std::runtime_error("Resource 2 construction failed!");
606         }
607     }
608
609     ~Resource() {
610         std::cout << "      [Resource] Destroyed\n";
611     }
612 };
613
614 class ComponentWithResources {
615 private:
616     Resource res1_;
617     Resource res2_;
618     Resource res3_;
619
620 public:
621     // Function try block for constructor
622     ComponentWithResources()
623     try : res1_(1), res2_(2), res3_(3) // res2 will throw!
624     {
625         std::cout << "  Constructor body executed\n";
626     }
627     catch (const std::exception& e) {
628         std::cout << "  Constructor caught: " << e.what() << "\n";
629         std::cout << "  res1_ will be automatically destroyed\n";
630         // Note: Exception is automatically rethrown after this catch
631     }
632 };
633
634 void demonstrate_function_try_blocks() {
635     std::cout << "\n" << std::string(70, '=') << "\n";
636     std::cout << "==== 8. Function Try Blocks (Constructor Exception Handling)
637         ==\n";
638     std::cout << std::string(70, '=') << "\n\n";
639
640     std::cout << "Attempting to construct ComponentWithResources...\n\n";
641
642     try {
643         ComponentWithResources component;
644     }
645     catch (const std::exception& e) {
646         std::cout << "\n Exception propagated to caller\n";
647         std::cout << "  Partially constructed members automatically cleaned up
648             \n";
649     }
650 }
651
652 // =====
```

```
652 //  
=====  
  
653  
654 void demonstrate_best_practices() {  
655     std::cout << "\n" << std::string(70, '=') << "\n";  
656     std::cout << "== 9. Exception Handling Best Practices ==\n";  
657     std::cout << std::string(70, '=') << "\n\n";  
658  
659     std::cout << "  DO:\n";  
660     std::cout << "  \n";  
661     std::cout << "    1. Use RAII for resource management\n";  
662     std::cout << "    2. Catch by const reference: catch (const Ex& e)\n";  
663     std::cout << "    3. Throw by value: throw MyException(...)\n";  
664     std::cout << "    4. Inherit from std::exception hierarchy\n";  
665     std::cout << "    5. Mark functions noexcept when appropriate\n";  
666     std::cout << "    6. Use specific exceptions before generic ones\n";  
667     std::cout << "    7. Provide meaningful error messages\n";  
668     std::cout << "    8. Document what exceptions functions can throw\n";  
669     std::cout << "    9. Use stack traces for debugging\n";  
670     std::cout << "   10. Test exception paths thoroughly\n\n";  
671  
672     std::cout << "  DON'T:\n";  
673     std::cout << "  \n";  
674     std::cout << "    1. Throw from destructors (noexcept by default in C++11)\n";  
675     std::cout << "    2. Catch (...) without rethrowing or logging\n";  
676     std::cout << "    3. Use exceptions for normal control flow\n";  
677     std::cout << "    4. Return error codes for exceptional situations\n";  
678     std::cout << "    5. Ignore exceptions silently\n";  
679     std::cout << "    6. Throw pointers: throw new Ex() \n";  
680     std::cout << "    7. Catch by value (causes slicing)\n";  
681     std::cout << "    8. Mix exceptions and error codes inconsistently\n";  
682     std::cout << "    9. Throw from noexcept functions\n";  
683     std::cout << "   10. Forget to clean up resources (use RAII!)\n\n";  
684  
685     std::cout << "EXCEPTION vs ERROR CODES:\n";  
686     std::cout << "  \n";  
687     std::cout << "Use Exceptions when:\n";  
688     std::cout << "  • Error is truly exceptional\n";  
689     std::cout << "  • Can't continue execution\n";  
690     std::cout << "  • Need to propagate through many layers\n";  
691     std::cout << "  • RAII cleanup is needed\n\n";  
692  
693     std::cout << "Use Error Codes when:\n";  
694     std::cout << "  • Expected/recoverable errors\n";  
695     std::cout << "  • Performance-critical hot paths\n";  
696     std::cout << "  • C API compatibility\n";  
697     std::cout << "  • Safety-critical systems (ISO 26262)\n";  
698 }  
699  
700 //  
=====
```

```
701 // MAIN FUNCTION
702 //
703 =====
704 int main() {
705     std::cout << "\n";
706     std::cout << "                                         \n";
707     std::cout << "         Advanced Exception Handling with Stack Traces
708             \n";
709     std::cout << "                                         \n";
710     std::cout << "                                         \n";
711
712     demonstrate_stack_trace_capture();
713     demonstrate_exception_with_stack_trace();
714     demonstrate_nested_exceptions();
715     demonstrate_exception_guarantees();
716     demonstrate_raii_exception_safety();
717     demonstrate_exception_ptr();
718     demonstrate_exception_hierarchy();
719     demonstrate_function_try_blocks();
720     demonstrate_best_practices();
721
722     std::cout << "\n" << std::string(70, '=') << "\n";
723     std::cout << "All exception handling demonstrations completed!\n";
724     std::cout << "\nKEY TAKEAWAYS:\n";
725     std::cout << " 1. Real stack traces available on Linux/Windows\n";
726     std::cout << " 2. RAII guarantees cleanup even with exceptions\n";
727     std::cout << " 3. Nested exceptions preserve error context\n";
728     std::cout << " 4. std::exception_ptr for thread exception propagation\n";
729     std::cout << " 5. Always catch by const reference\n";
730     std::cout << " 6. Use noexcept for no-throw guarantees\n";
731     std::cout << std::string(70, '=') << "\n\n";
732
733     return 0;
734 }
```

## 4 Source Code: AsioAndModernCppConcurrency.cpp

File: src/AsioAndModernCppConcurrency.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <thread>
3 #include <future>
4 #include <chrono>
5 #include <vector>
6 #include <string>
7 #include <mutex>
8 #include <memory>
9 #include <functional>
10
11 // Note: This example demonstrates ASIO concepts using standard C++ features.
12 // For actual ASIO usage, install: https://think-async.com/Asio/
13 // Standalone ASIO: #include <asio.hpp>
14 // Boost.ASIO: #include <boost/asio.hpp>
15
16 // =====
17 // ASIO AND MODERN C++ CONCURRENCY COMPARISON
18 // =====
19
20 // This example shows the concepts and patterns without requiring ASIO
21 // installation
22 // It demonstrates when to use ASIO vs C++ standard library features
23
24 using namespace std::chrono_literals;
25
26 // =====
27 // 1. THE FUNDAMENTAL DIFFERENCE: I/O vs CPU
28 // =====
29
30 void example_io_vs_cpu_bound() {
31     std::cout << "\n== 1. I/O-BOUND vs CPU-BOUND OPERATIONS ==" << std::endl
32     ;
33
34     std::cout << "\n--- I/O-BOUND OPERATIONS (ASIO's domain) ---" << std::endl
35     ;
36     std::cout << "• Waiting for network data (recv/send)" << std::endl;
37     std::cout << "• Reading/writing files" << std::endl;
38     std::cout << "• Waiting for timers to expire" << std::endl;
39     std::cout << "• Database queries" << std::endl;
40     std::cout << "• HTTP requests/responses" << std::endl;
41     std::cout << "\nCharacteristics:" << std::endl;
42     std::cout << " → Thread spends most time WAITING" << std::endl;
43     std::cout << " → CPU is mostly idle" << std::endl;
44     std::cout << " → One thread can handle thousands of operations" << std::endl;
45
46     std::cout << "\n--- CPU-BOUND OPERATIONS (std::async/thread domain) ---"
47     << std::endl;
```

```
45     std::cout << "• Image processing" << std::endl;
46     std::cout << "• Video encoding" << std::endl;
47     std::cout << "• Mathematical computations" << std::endl;
48     std::cout << "• Data compression" << std::endl;
49     std::cout << "• Machine learning inference" << std::endl;
50     std::cout << "\nCharacteristics:" << std::endl;
51     std::cout << " → Thread spends most time COMPUTING" << std::endl;
52     std::cout << " → CPU is fully utilized" << std::endl;
53     std::cout << " → Need multiple threads/cores for parallelism" << std::endl;
54     std::cout << " → Thread count    CPU core count for best performance" << std::endl;
55 }
56
57 // =====
58 // 2. SIMULATED ASIO-STYLE TIMER vs std::this_thread::sleep_for
59 // =====
60
61 // Simulate ASIO timer callback pattern
62 class SimulatedAsioTimer {
63 private:
64     std::chrono::milliseconds duration;
65     std::function<void()> callback;
66     std::jthread timer_thread;
67
68 public:
69     SimulatedAsioTimer(std::chrono::milliseconds ms) : duration(ms) {}
70
71     // ASIO-style async_wait (non-blocking)
72     void async_wait(std::function<void()> handler) {
73         callback = std::move(handler);
74
75         // Start timer in background (simulates event loop)
76         timer_thread = std::jthread([this]() {
77             std::this_thread::sleep_for(duration);
78             if (callback) {
79                 callback();
80             }
81         });
82     }
83
84     void cancel() {
85         // In real ASIO, this would cancel the pending operation
86         timer_thread.request_stop();
87     }
88 };
89
90 void example_async_timer_patterns() {
91     std::cout << "\n== 2. ASYNC TIMER PATTERNS ==" << std::endl;
92
93     std::cout << "\n-- Pattern 1: Blocking (std::this_thread::sleep_for) --"
94     << std::endl;
95     auto start = std::chrono::steady_clock::now();
96     std::cout << "Starting blocking sleep..." << std::endl;
```

```
96     std::this_thread::sleep_for(100ms);
97     std::cout << "Sleep completed (thread was blocked)" << std::endl;
98     auto end = std::chrono::steady_clock::now();
99     std::cout << "Time: " << std::chrono::duration_cast<std::chrono::
100       milliseconds>(end - start).count() << "ms" << std::endl;
101
102     std::cout << "\n--- Pattern 2: Async Timer (ASIO-style) ---" << std::endl;
103     start = std::chrono::steady_clock::now();
104     std::cout << "Starting async timer..." << std::endl;
105
106     SimulatedAsioTimer timer(100ms);
107     timer.async_wait([start]() {
108       auto end = std::chrono::steady_clock::now();
109       std::cout << "Timer callback executed (thread was NOT blocked)" << std
110         ::endl;
111       std::cout << "Time: " << std::chrono::duration_cast<std::chrono::
112         milliseconds>(end - start).count() << "ms" << std::endl;
113     });
114
115     std::cout << "async_wait returned immediately (non-blocking)" << std::endl
116       ;
117     std::cout << "Main thread can do other work..." << std::endl;
118
119     // Wait for timer to complete
120     std::this_thread::sleep_for(150ms);
121
122   }
123
124 // =====
125 // 3. CALLBACK-BASED vs FUTURE-BASED ASYNC
126 // =====
127
128 // Simulate async network read with callback (ASIO pattern)
129 void async_read_callback(const std::string& data, std::function<void(const std
130   ::string&)> callback) {
131   std::jthread([data, callback = std::move(callback)]() {
132     std::this_thread::sleep_for(50ms); // Simulate I/O delay
133     callback(data);
134   }).detach();
135 }
136
137 // Future-based async read (std::async pattern)
138 std::future<std::string> async_read_future(const std::string& data) {
139   return std::async(std::launch::async, [data]() {
140     std::this_thread::sleep_for(50ms); // Simulate I/O delay
141     return data;
142   });
143 }
```

```
142 }
143
144 void example_callback_vs_future() {
145     std::cout << "\n== 3. CALLBACK-BASED vs FUTURE-BASED ASYNC ==" << std::endl;
146
147     std::cout << "\n--- Callback-Based (ASIO pattern) ---" << std::endl;
148     std::cout << "Initiating async read with callback..." << std::endl;
149
150     async_read_callback("Data from network", [](const std::string& result) {
151         std::cout << "Callback received: " << result << std::endl;
152     });
153
154     std::cout << "async_read_callback returned immediately" << std::endl;
155     std::cout << "Main thread continues..." << std::endl;
156     std::this_thread::sleep_for(100ms); // Wait for callback
157
158     std::cout << "\n--- Future-Based (std::async pattern) ---" << std::endl;
159     std::cout << "Initiating async read with future..." << std::endl;
160
161     auto future = async_read_future("Data from computation");
162     std::cout << "async_read_future returned immediately" << std::endl;
163     std::cout << "Main thread can do work before getting result..." << std::endl;
164
165     // Get result (blocks until ready)
166     std::string result = future.get();
167     std::cout << "Future result: " << result << std::endl;
168
169     std::cout << "\n TRADE-OFFS:" << std::endl;
170     std::cout << "  - Callbacks (ASIO):" << std::endl;
171     std::cout << "    + Very efficient for I/O (no thread per operation)" << std::endl;
172     std::cout << "    - Composable (chain callbacks)" << std::endl;
173     std::cout << "    - Callback hell (deep nesting)" << std::endl;
174     std::cout << "    - Error handling more complex" << std::endl;
175     std::cout << "\n  Futures (std::async):" << std::endl;
176     std::cout << "    + Simpler synchronous-style code" << std::endl;
177     std::cout << "    + Easier error handling (exceptions)" << std::endl;
178     std::cout << "    - Creates threads (expensive for many I/O ops)" << std::endl;
179     std::cout << "    - Doesn't scale for thousands of concurrent operations"
180         << std::endl;
181
182 // =====
183 // 4. EVENT LOOP CONCEPT (CORE OF ASIO)
184 // =====
185
186 // Simplified event loop to demonstrate the concept
187 class EventLoop {
188 private:
189     std::vector<std::function<void()>> callbacks;
190     std::mutex mutex;
```

```
191     bool running = false;
192
193     public:
194         void post(std::function<void()> callback) {
195             std::lock_guard<std::mutex> lock(mutex);
196             callbacks.push_back(std::move(callback));
197         }
198
199         void run() {
200             running = true;
201             std::cout << "Event loop started" << std::endl;
202
203             while (running) {
204                 std::vector<std::function<void()>> work;
205
206                 {
207                     std::lock_guard<std::mutex> lock(mutex);
208                     work.swap(callbacks);
209                 }
210
211                 for (auto& callback : work) {
212                     callback();
213                 }
214
215                 if (work.empty() && !running) {
216                     break;
217                 }
218
219                 std::this_thread::sleep_for(10ms); // Poll interval
220             }
221
222             std::cout << "Event loop stopped" << std::endl;
223         }
224
225         void stop() {
226             running = false;
227         }
228     };
229
230     void example_event_loop() {
231         std::cout << "\n==== 4. EVENT LOOP CONCEPT (ASIO's HEART) ===" << std::endl
232         ;
233
234         EventLoop loop;
235
236         // Run event loop in background thread
237         std::jthread loop_thread([&loop]() {
238             loop.run();
239         });
240
241         std::cout << "\nPosting work to event loop..." << std::endl;
242
243         // Post work items (like ASIO handlers)
244         loop.post([]() {
```

```

244     std::cout << " → Handler 1 executed" << std::endl;
245 }
246
247 loop.post([]() {
248     std::cout << " → Handler 2 executed" << std::endl;
249 })
250
251 loop.post([]() {
252     std::cout << " → Handler 3 executed" << std::endl;
253 })
254
255 std::this_thread::sleep_for(50ms);
256
257 loop.post([]() {
258     std::cout << " → Handler 4 executed (posted later)" << std::endl;
259 })
260
261 std::this_thread::sleep_for(50ms);
262 loop.stop();
263
264 std::cout << "\n EVENT LOOP BENEFITS:" << std::endl;
265 std::cout << " • Single thread processes all callbacks sequentially" <<
266     std::endl;
267 std::cout << " • No race conditions within event loop" << std::endl;
268 std::cout << " • Can handle thousands of I/O operations efficiently" <<
269     std::endl;
270 std::cout << " • Real ASIO: Uses OS-level primitives (epoll/IOCP)" << std
271     ::endl;
272 }
273
274 // =====
275 // 5. WHEN TO USE WHAT: DECISION MATRIX
276 // =====
277
278 void example_decision_matrix() {
279     std::cout << "\n== 5. DECISION MATRIX: ASIO vs C++ STD ==" << std::endl;
280
281     std::cout << "\n" << std::endl;
282     std::cout << " Use Case" << " ASIO" << " C++ Std" << " "
283         << std::endl;
284     std::cout << " TCP/UDP networking" << std::endl;
285     std::cout << " HTTP server/client" << std::endl;
286     std::cout << " WebSocket" << std::endl;
287     std::cout << " Async timers" << std::endl;
288     std::cout << " Serial port I/O" << std::endl;
289     std::cout << " Many concurrent connections" << std::endl;
290     std::cout << " Image processing" << std::endl;

```

```

288     std::endl;
289     std::cout << "  Parallel computation" <<
290     std::endl;
291     std::cout << "  Thread synchronization" <<
292     std::endl;
293     std::cout << "  Protect shared data" << (mutex) <<
294     std::endl;
295     std::cout << "  Wait for task completion" << (future) <<
296     std::endl;
297     std::cout << "" << std::endl;
298
299     std::cout << "\n COMBINED USAGE (BEST PRACTICE):" << std::endl;
300     std::cout << "" << std::endl;
301     std::cout << "  Component" << Solution
302     std::cout << "" << std::endl;
303     std::cout << "  Network I/O" << ASIO (io_context)
304     std::cout << "" << std::endl;
305     std::cout << "  CPU-heavy processing" << std::endl;
306     std::cout << "  Shared cache" << std::endl;
307     std::cout << "  Thread coordination" << std::endl;
308     std::cout << "  Rate limiting" << std::endl;
309     std::cout << "" << std::endl;
310 }
311
312 // =====
313 // 6. SCALABILITY COMPARISON
314 // =====
315
316 void example_scalability() {
317     std::cout << "\n== 6. SCALABILITY: ASIO vs std::thread ==" << std::endl;
318
319     std::cout << "\n--- Scenario: Handle 10,000 concurrent connections ---" <<
320             std::endl;
321
322     std::cout << "\n APPROACH 1: One thread per connection (std::thread)" <<
323             std::endl;
324     std::cout << "  • Need 10,000 threads" << std::endl;
325     std::cout << "  • Each thread: ~1MB stack = 10GB memory" << std::endl;
326     std::cout << "  • Context switching overhead" << std::endl;
327     std::cout << "  • OS thread limit (~32k on Linux)" << std::endl;
328     std::cout << "  • Result: System collapse" << std::endl;
329
330     std::cout << "\n APPROACH 2: Event-driven with ASIO" << std::endl;
331     std::cout << "  • 1-4 threads (typically)" << std::endl;
332     std::cout << "  • Event multiplexing (epoll/IOCP)" << std::endl;
333     std::cout << "  • Each connection: ~few KB state" << std::endl;
334     std::cout << "  • Total memory: <100MB" << std::endl;
335     std::cout << "  • Result: Handles load easily" << std::endl;
336
337 }
```

```

329     std::cout << "\n PERFORMANCE COMPARISON:" << std::endl;
330     std::cout << "    Connections | std::thread | ASIO      " << std::endl;
331     std::cout << "    -----|-----|-----" << std::endl;
332     std::cout << "    10      | OK      | OK      " << std::endl;
333     std::cout << "    100     | Struggling | OK      " << std::endl;
334     std::cout << "    1,000    | Failing   | OK      " << std::endl;
335     std::cout << "    10,000   | Impossible | Good    " << std::endl;
336     std::cout << "    100,000  | N/A      | Possible " << std::endl;
337 }
338
339 // =====
340 // 7. INTEGRATION PATTERNS
341 // =====
342
343 // Pattern: ASIO for I/O + std::async for CPU work
344 class HybridServer {
345 private:
346     std::mutex shared_cache_mutex;
347     std::vector<std::string> shared_cache;
348
349 public:
350     // Simulated: ASIO receives network request
351     void on_request_received(const std::string& data) {
352         std::cout << "  [ASIO] Received request: " << data << std::endl;
353
354         // CPU-intensive work: offload to std::async
355         auto future = std::async(std::launch::async, [this, data]() {
356             return process_data(data);
357         });
358
359         std::cout << "  [ASIO] Offloaded to worker thread, can handle more
360             requests" << std::endl;
361
362         // Get result (in real app, would be callback)
363         std::string result = future.get();
364
365         // ASIO would send response here
366         std::cout << "  [ASIO] Sending response: " << result << std::endl;
367     }
368
369 private:
370     std::string process_data(const std::string& data) {
371         // Simulate CPU-intensive work
372         std::this_thread::sleep_for(50ms);
373         std::string result = "Processed: " + data;
374
375         // Update shared cache (use mutex for thread safety)
376         {
377             std::lock_guard<std::mutex> lock(shared_cache_mutex);
378             shared_cache.push_back(result);
379         }
380
381         return result;
382     }

```

```
382 };
```

```
383
```

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

```
386
```

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

```
388
```

```
389     HybridServer server;
```

```
390
```

```
391     // Simulate multiple requests
392     server.on_request_received("Request-1");
393     server.on_request_received("Request-2");
394     server.on_request_received("Request-3");

395     std::cout << "\n HYBRID APPROACH:" << std::endl;
396     std::cout << " 1. ASIO handles network I/O (event loop)" << std::endl;
397     std::cout << " 2. Offload CPU work to std::async" << std::endl;
398     std::cout << " 3. Use std::mutex for shared data" << std::endl;
399     std::cout << " 4. ASIO remains responsive during CPU work" << std::endl;
400 }
```

```
401
```

```
402 // =====
403 // 8. C++20 COROUTINES + ASIO (THE FUTURE)
404 // =====
```

```
405
```

```
406
```

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

```
409
```

```
410     std::cout << "\n--- Without Coroutines (Callback Hell) ---" << std::endl;
411     std::cout << "async_read(socket, buffer, [](error, bytes) {" << std::endl;
412     std::cout << "    async_write(socket, data, [](error, bytes) {" << std::endl;
413     std::cout << "        async_read(socket, response, [](error, bytes) {" << std::endl;
414     std::cout << "            // 3 levels deep already!" << std::endl;
415     std::cout << "        });" << std::endl;
416     std::cout << "    });" << std::endl;
417     std::cout << "});" << std::endl;
```

```
418
```

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

```
428
```

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

```

427 std::cout << " • Write async code that looks synchronous" << std::endl;
428 std::cout << " • No callback nesting (no hell)" << std::endl;
429 std::cout << " • Exception handling works naturally (try/catch)" << std::endl;
430 std::cout << " • Still non-blocking (efficient as callbacks)" << std::endl;
431 std::cout << " • ASIO has full coroutine support (co_await)" << std::endl;
432 ;
433
434 // =====
435 // 9. REAL-WORLD ARCHITECTURE EXAMPLE
436 // =====
437
438 void example_architecture() {
439     std::cout << "\n== 9. REAL-WORLD ARCHITECTURE: WEB SERVER ==" << std::endl;
440
441     std::cout << "\n" << std::endl;
442     std::cout << "          High-Performance Web Server" << std::endl;
443     std::cout << "          " << std::endl;
444     std::cout << "          " << std::endl;
445     std::cout << "          " << std::endl;
446     std::cout << "          " << std::endl;
447     std::cout << "          " << std::endl;
448     std::cout << "          " << std::endl;
449     std::cout << "      ASIO          C++ Std          C++ Std" << std::endl;
450     std::cout << "          " << std::endl;
451     std::cout << "          " << std::endl;
452     std::cout << "          " << std::endl;
453
454     std::cout << "\n ASIO LAYER:" << std::endl;
455     std::cout << " • io_context (1-4 threads)" << std::endl;
456     std::cout << " • Accept connections (10k+ concurrent)" << std::endl;
457     std::cout << " • Parse HTTP requests" << std::endl;
458     std::cout << " • Send HTTP responses" << std::endl;
459     std::cout << " • Timers for keepalive/timeout" << std::endl;
460
461     std::cout << "\n CPU PROCESSING LAYER:" << std::endl;
462     std::cout << " • std::jthread pool (N = CPU cores)" << std::endl;
463     std::cout << " • Process business logic" << std::endl;
464     std::cout << " • Image/video processing" << std::endl;
465     std::cout << " • Compute-intensive tasks" << std::endl;
466     std::cout << " • Post results back to ASIO" << std::endl;
467
468     std::cout << "\n SYNCHRONIZATION LAYER:" << std::endl;
469     std::cout << " • std::mutex: Protect shared cache" << std::endl;

```

```
470     std::cout << " • std::shared_mutex: Read-write locks" << std::endl;
471     std::cout << " • std::atomic: Lock-free counters" << std::endl;
472     std::cout << " • std::semaphore: Rate limiting" << std::endl;
473     std::cout << " • std::latch: Wait for initialization" << std::endl;
474
475     std::cout << "\n BENEFITS OF HYBRID APPROACH:" << std::endl;
476     std::cout << "    Scales to 100k+ connections (ASIO)" << std::endl;
477     std::cout << "    Utilizes all CPU cores (std::jthread)" << std::endl;
478     std::cout << "    Thread-safe shared state (std::mutex)" << std::endl;
479     std::cout << "    Best of both worlds!" << std::endl;
480 }
481
482 // =====
483 // 10. SUMMARY AND RECOMMENDATIONS
484 // =====
485
486 void example_summary() {
487     std::cout << "\n== 10. SUMMARY: ASIO vs C++ STANDARD LIBRARY ==" << std
488         ::endl;
489
490     std::cout << "\n KEY TAKEAWAYS:" << std::endl;
491     std::cout << "\n1. DIFFERENT PROBLEMS, DIFFERENT TOOLS:" << std::endl;
492     std::cout << " • ASIO: Asynchronous I/O (network, timers, file I/O)" <<
493         std::endl;
494     std::cout << " • C++ Std: Threading, CPU parallelism, synchronization"
495         << std::endl;
496     std::cout << " • They COMPLEMENT each other, not compete!" << std::endl;
497
498     std::cout << "\n2. WHEN TO USE ASIO:" << std::endl;
499     std::cout << "    Building network servers/clients" << std::endl;
500     std::cout << "    Need to handle thousands of concurrent connections" <<
501         std::endl;
502     std::cout << "    I/O-bound operations (waiting > computing)" << std::
503         endl;
504     std::cout << "    Event-driven architecture" << std::endl;
505     std::cout << "    Cross-platform async I/O" << std::endl;
506
507     std::cout << "\n3. WHEN TO USE C++ STANDARD LIBRARY:" << std::endl;
508     std::cout << "    CPU-bound parallel computations" << std::endl;
509     std::cout << "    Thread-safe data structures" << std::endl;
510     std::cout << "    Simple threading needs" << std::endl;
511     std::cout << "    Waiting for computation results (futures)" << std::endl
512         ;
513     std::cout << "    Thread coordination (semaphores, latches)" << std::endl
514         ;
515
516     std::cout << "\n4. BEST PRACTICES (2026):" << std::endl;
517     std::cout << "    Use ASIO for I/O multiplexing" << std::endl;
518     std::cout << "    Use std::async/thread_pool for CPU work" << std::endl;
519     std::cout << "    Use std::mutex for thread-safe shared data" << std::
520         endl;
521     std::cout << "    Use C++20 coroutines with ASIO (co_await)" << std::endl
522         ;
523     std::cout << "    Combine: ASIO event loop + std threading primitives" <<
```

```
        std::endl;

515    std::cout << "\n5. GETTING STARTED:" << std::endl;
516    std::cout << " • Install: Standalone ASIO (https://think-async.com/Asio/)" << std::endl;
517    std::cout << " • Or: Boost.ASIO (apt install libboost-all-dev)" << std::endl;
518    std::cout << " • Learn: Start with timers, then sockets" << std::endl;
519    std::cout << " • Pattern: Event loop in 1-4 threads" << std::endl;
520    std::cout << " • Modern: Use ASIO with C++20 coroutines" << std::endl;
521
522
523    std::cout << "\n6. COMMON MISTAKES:" << std::endl;
524    std::cout << "     Using std::thread for thousands of connections" << std::endl;
525    std::cout << "     Using ASIO for CPU-intensive work" << std::endl;
526    std::cout << "     Blocking ASIO event loop with long operations" << std::endl;
527    std::cout << "     Not protecting shared data between ASIO and worker
528          threads" << std::endl;
529
530    std::cout << "\n GOLDEN RULE:" << std::endl;
531    std::cout << "     \"ASIO for waiting, std::thread for computing\"" << std::endl;
532
533 // =====
534 // MAIN FUNCTION
535 // =====
536
537 int main() {
538     std::cout << "\n
539         =====
540         ASIO AND MODERN C++ CONCURRENCY: COMPREHENSIVE GUIDE" <<
541         std::endl;
542     std::cout << "
543         =====
544         Note: This demonstrates concepts using standard C++
545             features." << std::endl;
546     std::cout << "For actual ASIO usage, install from: https://think-async.com/Asio/" << std::endl;
547
548     example_io_vs_cpu_bound();
549     example_async_timer_patterns();
550     example_callback_vs_future();
551     example_event_loop();
552     example_decision_matrix();
553     example_scalability();
554     example_integration_patterns();
555     example_coroutines_concept();
556     example_architecture();
557     example_summary();
558 }
```

```

555     std::cout << "\n"
556     ======" <<
557     std::endl;
558     std::cout << "  FINAL VERDICT: ASIO vs C++ STANDARD LIBRARY" << std::endl;
559     std::cout << "
560     ======" <<
561     std::endl;
562     std::cout << "\n WINNER: BOTH (They're Partners, Not Competitors!)" <<
563     std::endl;
564
565     std::cout << "\n COMPARISON TABLE:" << std::endl;
566     std::cout << "\n" << std::endl;
567     std::cout << "  Feature           ASIO          C++ Std      " << std
568     ::endl;
569     std::cout << "           " << std::endl;
570     std::cout << "  Network I/O          " << std::endl;
571     std::cout << "  Async Timers          " << std::endl;
572     std::cout << "  Scalability           " << std::endl;
573     std::cout << "  CPU Parallelism        " << std::endl;
574     std::cout << "  Thread Safety          " << std::endl;
575     std::cout << "  Ease of Use            " << std::endl;
576     std::cout << "  Learning Curve          Steep        Moderate      " << std
577     ::endl;
578     std::cout << "  Standardization        Not yet      Standard      " << std
579     ::endl;
580     std::cout << "  Maturity                20+ years   Modern C++      " << std
581     ::endl;
582     std::cout << "           " << std::endl;
583
584     std::cout << "\n LEARNING PATH:" << std::endl;
585     std::cout << "  1. Master C++ std threading first (mutex, thread, async)" <<
586     std::endl;
587     std::cout << "  2. Understand I/O vs CPU bound problems" << std::endl;
588     std::cout << "  3. Learn ASIO basics (timers, then sockets)" << std::endl;
589     std::cout << "  4. Study ASIO examples and patterns" << std::endl;
590     std::cout << "  5. Explore ASIO + C++20 coroutines" << std::endl;
591     std::cout << "  6. Build hybrid systems (ASIO + std)" << std::endl;
592
593     std::cout << "\n"
594     ======" <<
595     std::endl;
596
597     return 0;
598 }
```

## 5 Source Code: AsioMultipleContexts.cpp

File: src/AsioMultipleContexts.cpp

Repository: [View on GitHub](#)

```
1 // AsioMultipleContexts.cpp
2 // Comprehensive educational example of using multiple io_context objects in
3 // standalone ASIO
4 // Demonstrates LAN/WAN separation, thread pooling, and priority-based I/O
5 // handling
6
7 // Note: This example uses simulated ASIO patterns for educational purposes
8 // For actual ASIO usage, install standalone ASIO: https://think-async.com/Asio/
9 // Then: #include <asio.hpp>
10 // And replace SimulatedAsio classes with real asio::io_context, asio::steady_timer, etc.
11
12 using namespace std::chrono_literals;
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
```

```
=====
// SIMULATED ASIO CLASSES (for demonstration without requiring ASIO
installtion)
=====

// In real code, replace these with actual ASIO classes

namespace SimulatedAsio {
    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
    };
}
```

```
41
42     public:
43         explicit io_context(const std::string& name = "io_context") : name_(name)
44             {}
45
46         // Main event loop - blocks until no more work
47         void run() {
48             std::cout << "[" << name_ << "] Thread " << std::this_thread::get_id()
49             << " calling run()\n";
50
51             while (!stopped_) {
52                 std::function<void()> work;
53
54                 {
55                     std::unique_lock<std::mutex> lock(work_mutex_);
56                     work_cv_.wait_for(lock, 50ms, [this] {
57                         return !pending_work_.empty() || stopped_;
58                     });
59
60                     if (stopped_ && pending_work_.empty()) {
61                         break;
62                     }
63
64                     if (!pending_work_.empty()) {
65                         work = std::move(pending_work_.back());
66                         pending_work_.pop_back();
67                     }
68
69                     if (work) {
70                         work();
71                         --work_count_;
72                     }
73
74                     // If no more work, exit
75                     if (work_count_ == 0 && pending_work_.empty()) {
76                         break;
77                     }
78                 }
79
80                 std::cout << "[" << name_ << "] Thread " << std::this_thread::get_id()
81                 << " exiting run()\n";
82             }
83
84             // Post work to the io_context
85             void post(std::function<void()> handler) {
86                 {
87                     std::lock_guard<std::mutex> lock(work_mutex_);
88                     pending_work_.push_back(std::move(handler));
89                     ++work_count_;
90                 }
91                 work_cv_.notify_one();
92             }
93 }
```

```

94 // Stop the io_context
95 void stop() {
96     stopped_ = true;
97     work_cv_.notify_all();
98 }
99
100 const std::string& name() const { return name_; }
101 };
102
103 // Simulate asio::steady_timer
104 class steady_timer {
105 private:
106     io_context& io_;
107     std::chrono::milliseconds duration_;
108     std::jthread timer_thread_;
109
110 public:
111     steady_timer(io_context& io, std::chrono::milliseconds ms)
112         : io_(io), duration_(ms) {}
113
114     void async_wait(std::function<void()> handler) {
115         // Start timer in background, then post handler to io_context
116         timer_thread_ = std::jthread([this, h = std::move(handler)]() {
117             std::this_thread::sleep_for(duration_);
118             io_.post(h); // Post to io_context when timer expires
119         });
120     }
121 };
122
123 // Work guard to keep io_context alive
124 template<typename Executor>
125 class executor_work_guard {
126 private:
127     io_context& io_;
128
129 public:
130     explicit executor_work_guard(io_context& io) : io_(io) {}
131     ~executor_work_guard() = default;
132 };
133
134 } // namespace SimulatedAsio
135
136 // =====
137 // SECTION 1: Basic Multiple io_context Pattern
138 // =====
139
140 void demonstrate_basic_multiple_contexts() {
141     std::cout << "\n" << std::string(70, '=') << "\n";
142     std::cout << "==== 1. Basic Multiple io_context Pattern ===\n";
143     std::cout << std::string(70, '=') << "\n\n";

```

```

144
145     std::cout << "Concept: Two separate io_context instances, each with its
146         own thread\n";
147     std::cout << "Use case: Isolate different types of I/O operations\n\n";
148
149     SimulatedAsio::io_context io1("io_context_1");
150     SimulatedAsio::io_context io2("io_context_2");
151
152     // Post work to both contexts
153     io1.post([]() {
154         std::cout << "  Work executing on io_context_1\n";
155     });
156
157     io2.post([]() {
158         std::cout << "  Work executing on io_context_2\n";
159     });
160
161     // Run each io_context in a separate thread
162     std::thread t1([&io1](){ io1.run(); });
163     std::thread t2([&io2](){ io2.run(); });
164
165     t1.join();
166     t2.join();
167
168     std::cout << "\n Both contexts completed independently\n";
169     std::cout << "  Each thread called run() on its assigned io_context\n";
170
171 }
172
173 // =====
174 // SECTION 2: Thread Pool Pattern (Multiple threads, one io_context)
175 // =====
176
177 void demonstrate_thread_pool_pattern() {
178     std::cout << "\n" << std::string(70, '=') << "\n";
179     std::cout << "==== 2. Thread Pool Pattern (Multiple threads share one
180         io_context) ===\n";
181     std::cout << std::string(70, '=') << "\n\n";
182
183     std::cout << "Concept: Multiple threads call run() on the SAME io_context\n";
184     std::cout << "Benefit: Work is automatically distributed across threads\n";
185
186     SimulatedAsio::io_context io("thread_pool");
187
188     // Post 6 tasks
189     for (int i = 1; i <= 6; ++i) {
190         io.post([i](){
191             std::cout << "  Task " << i << " executing on thread "
192                 << std::this_thread::get_id() << "\n";
193     });
194 }

```

```
190         std::this_thread::sleep_for(100ms); // Simulate work
191     });
192 }
193
194 // Create thread pool: 3 threads all calling run() on the same io_context
195 std::vector<std::thread> thread_pool;
196 for (int i = 0; i < 3; ++i) {
197     thread_pool.emplace_back([&io]() {
198         io.run();
199     });
200 }
201
202 // Wait for all threads
203 for (auto& t : thread_pool) {
204     t.join();
205 }
206
207 std::cout << "\n 6 tasks distributed across 3 threads automatically\n";
208 std::cout << " This is the standard ASIO thread pool pattern\n";
209 }
210
211 // =====
212 // SECTION 3: LAN vs WAN Separation (The Main Example!)
213 // =====
214
215 // Simulate a network connection
216 class Connection {
217 private:
218     std::string name_;
219     std::string type_; // "LAN" or "WAN"
220     int request_count_;
221
222 public:
223     Connection(const std::string& name, const std::string& type)
224         : name_(name), type_(type), request_count_(0) {}
225
226     void handle_request(int request_id) {
227         ++request_count_;
228         std::cout << " [" << type_ << " - " << name_ << "] Processing request
229         #"
230             << request_id << " (total: " << request_count_ << ")\n";
231
232         // Simulate processing time
233         if (type_ == "LAN") {
234             std::this_thread::sleep_for(50ms); // Fast LAN processing
235         } else {
236             std::this_thread::sleep_for(150ms); // Slower WAN processing
237         }
238     }
239 }
```

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

```
240 };
241
242 void demonstrate_lan_wan_separation() {
243     std::cout << "\n" << std::string(70, '=') << "\n";
244     std::cout << "==== 3. LAN vs WAN Separation (Dual io_context) ====\n";
245     std::cout << std::string(70, '=') << "\n\n";
246
247     std::cout << "Scenario: Web server handling both LAN and WAN traffic\n";
248     std::cout << "Architecture:\n";
249     std::cout << " • io_lan → Fast local network clients (prioritized)\n";
250     std::cout << " • io_wan → Slower internet clients (lower priority)\n";
251     std::cout << "Benefits:\n";
252     std::cout << " • LAN traffic never blocked by slow WAN connections\n";
253     std::cout << " • Can apply different rate limits per context\n";
254     std::cout << " • Independent thread pools for each network\n\n";
255
256 // Create two separate io_context instances
257 SimulatedAsio::io_context io_lan("io_LAN");
258 SimulatedAsio::io_context io_wan("io_WAN");
259
260 // Create connections
261 auto lan_client1 = std::make_shared<Connection>("InternalAPI", "LAN");
262 auto lan_client2 = std::make_shared<Connection>("Dashboard", "LAN");
263 auto wan_client1 = std::make_shared<Connection>("PublicAPI", "WAN");
264 auto wan_client2 = std::make_shared<Connection>("MobileApp", "WAN");
265
266 std::cout << "Posting work to both contexts...\n\n";
267
268 // Post LAN work (fast, high priority)
269 for (int i = 1; i <= 3; ++i) {
270     io_lan.post([lan_client1, i]() {
271         lan_client1->handle_request(i);
272     });
273
274     io_lan.post([lan_client2, i]() {
275         lan_client2->handle_request(i);
276     });
277 }
278
279 // Post WAN work (slower, lower priority)
280 for (int i = 1; i <= 3; ++i) {
281     io_wan.post([wan_client1, i]() {
282         wan_client1->handle_request(i);
283     });
284
285     io_wan.post([wan_client2, i]() {
286         wan_client2->handle_request(i);
287     });
288 }
289
290 // Run LAN with 2 threads (higher capacity for fast traffic)
291 std::vector<std::thread> lan_threads;
292 lan_threads.emplace_back([&io_lan]() { io_lan.run(); });
```

```

293     lan_threads.emplace_back([&io_lan]() { io_lan.run(); });
294
295     // Run WAN with 1 thread (lower capacity, prevent resource exhaustion)
296     std::thread wan_thread([&io_wan]() { io_wan.run(); });
297
298     // Wait for completion
299     for (auto& t : lan_threads) {
300         t.join();
301     }
302     wan_thread.join();
303
304     std::cout << "\n LAN traffic processed with 2 threads (fast completion)\n"
305             "";
306     std::cout << " WAN traffic processed with 1 thread (controlled rate)\n";
307     std::cout << " Neither context blocked the other\n";
308 }
309
310 // SECTION 4: Priority-Based io_context (Critical vs Normal)
311 // =====
312
313 class PriorityServer {
314 private:
315     SimulatedAsio::io_context io_critical_;
316     SimulatedAsio::io_context io_normal_;
317     std::atomic<int> critical_count_{0};
318     std::atomic<int> normal_count_{0};
319
320 public:
321     PriorityServer() : io_critical_("io_CRITICAL"), io_normal_("io_NORMAL") {}
322
323     void handle_critical_request(const std::string& request) {
324         io_critical_.post([this, request]() {
325             int id = ++critical_count_;
326             std::cout << " [CRITICAL] Request " << id << ":" << request
327                         << " (thread " << std::this_thread::get_id() << ") \n";
328             std::this_thread::sleep_for(30ms); // Fast processing
329         });
330     }
331
332     void handle_normal_request(const std::string& request) {
333         io_normal_.post([this, request]() {
334             int id = ++normal_count_;
335             std::cout << " [NORMAL] Request " << id << ":" << request
336                         << " (thread " << std::this_thread::get_id() << ") \n";
337             std::this_thread::sleep_for(80ms); // Slower processing
338         });
339     }
340
341     void run() {

```

```
342     // Critical requests get dedicated thread
343     std::thread critical_thread([this]() {
344         io_critical_.run();
345     });
346
347     // Normal requests share a thread pool
348     std::thread normal_thread([this]() {
349         io_normal_.run();
350     });
351
352     critical_thread.join();
353     normal_thread.join();
354 }
355 };
356
357 void demonstrate_priority_separation() {
358     std::cout << "\n" << std::string(70, '=') << "\n";
359     std::cout << "==== 4. Priority-Based Separation (Critical vs Normal) ===\n";
360     std::cout << std::string(70, '=') << "\n\n";
361
362     std::cout << "Scenario: System with time-critical and regular operations\n";
363     std::cout << "Examples:\n";
364     std::cout << " • Critical: Heartbeats, alarms, safety-critical commands\n";
365     std::cout << " • Normal: Data logging, statistics, background tasks\n\n";
366
367     PriorityServer server;
368
369     // Mix critical and normal requests
370     server.handle_critical_request("Emergency Stop");
371     server.handle_normal_request("Log statistics");
372     server.handle_critical_request("Heartbeat");
373     server.handle_normal_request("Update dashboard");
374     server.handle_critical_request("Safety check");
375     server.handle_normal_request("Backup data");
376
377     std::cout << "\nProcessing requests...\n\n";
378     server.run();
379
380     std::cout << "\n Critical requests never delayed by normal traffic\n";
381     std::cout << " Each context can have different threading strategies\n";
382 }
383
384 // =====
385 // SECTION 5: Service-Type Separation (Clients, Database, Logging)
386 // =====
```

```
388 void demonstrate_service_type_separation() {
389     std::cout << "\n" << std::string(70, '=') << "\n";
390     std::cout << "==== 5. Service-Type Separation ===\n";
391     std::cout << std::string(70, '=') << "\n\n";
392
393     std::cout << "Architecture: One io_context per service type\n\n";
394
395     SimulatedAsio::io_context io_clients("io_CLIENTS");
396     SimulatedAsio::io_context io_database("io_DATABASE");
397     SimulatedAsio::io_context io_logging("io_LOGGING");
398
399     // Client connections (high concurrency)
400     for (int i = 1; i <= 4; ++i) {
401         io_clients.post([i]() {
402             std::cout << " [CLIENTS] Handling client " << i << " connection\n";
403             std::this_thread::sleep_for(60ms);
404         });
405     }
406
407     // Database operations (controlled concurrency)
408     for (int i = 1; i <= 2; ++i) {
409         io_database.post([i]() {
410             std::cout << " [DATABASE] Executing query " << i << "\n";
411             std::this_thread::sleep_for(100ms);
412         });
413     }
414
415     // Logging operations (background, low priority)
416     for (int i = 1; i <= 3; ++i) {
417         io_logging.post([i]() {
418             std::cout << " [LOGGING] Writing log entry " << i << "\n";
419             std::this_thread::sleep_for(40ms);
420         });
421     }
422
423     std::cout << "Starting service threads...\n\n";
424
425     // Each service gets its own thread configuration
426     std::thread client_thread([&]() { io_clients.run(); });
427     std::thread db_thread([&]() { io_database.run(); });
428     std::thread log_thread([&]() { io_logging.run(); });
429
430     client_thread.join();
431     db_thread.join();
432     log_thread.join();
433
434     std::cout << "\n Services isolated from each other\n";
435     std::cout << " Database load doesn't impact client connections\n";
436     std::cout << " Logging doesn't block critical services\n";
437 }
438
439 //
```

```
440 // SECTION 6: When to Call run() - Summary
441 //
442 =====
443 void demonstrate_run_call_patterns() {
444     std::cout << "\n" << std::string(70, '=') << "\n";
445     std::cout << "==== 6. When to Call io_context.run() - Complete Guide ===\n";
446     std::cout << std::string(70, '=') << "\n\n";
447
448     std::cout << "PATTERN 1: Single Thread per io_context\n";
449     std::cout << "                                \n";
450     std::cout << "    asio::io_context io1, io2;\n";
451     std::cout << "    std::thread t1([&]() { io1.run(); }); // Thread 1 → io1\n";
452     std::cout << "    std::thread t2([&]() { io2.run(); }); // Thread 2 → io2\n";
453     std::cout << "    Use: Separate concerns (LAN/WAN, Client/DB)\n\n";
454
455     std::cout << "PATTERN 2: Thread Pool (Multiple threads, one io_context)\n";
456     std::cout << "                                \n";
457     std::cout << "    asio::io_context io;\n";
458     std::cout << "    std::thread t1([&]() { io.run(); }); // Thread 1 → io\n";
459     std::cout << "    std::thread t2([&]() { io.run(); }); // Thread 2 → io\n";
460     std::cout << "    std::thread t3([&]() { io.run(); }); // Thread 3 → io\n";
461     std::cout << "    Use: Distribute work across threads automatically\n\n";
462
463     std::cout << "PATTERN 3: Main Thread runs io_context\n";
464     std::cout << "                                \n";
465     std::cout << "    asio::io_context io;\n";
466     std::cout << "    // Post all work...\n";
467     std::cout << "    io.run(); // Main thread blocks here\n";
468     std::cout << "    Use: Simple single-threaded servers\n\n";
469
470     std::cout << "PATTERN 4: Keep io_context Alive\n";
471     std::cout << "                                \n";
472     std::cout << "    asio::io_context io;\n";
473     std::cout << "    auto guard = asio::make_work_guard(io);\n";
474     std::cout << "    std::thread t([&]() { io.run(); }); // Won't exit
475     std::cout << "        immediately\n";
476     std::cout << "        // ... post work later ... \n";
477     std::cout << "        guard.reset(); // Allow run() to exit when work done\n";
478     std::cout << "    Use: Long-running services with dynamic work\n\n";
479
480     std::cout << "KEY RULES:\n";
481     std::cout << "    \n";
482     std::cout << "    1. run() BLOCKS until no more work (or stopped)\n";
483     std::cout << "    2. Multiple threads CAN call run() on SAME io_context (
484         thread pool)\n";
485     std::cout << "    3. One thread should NOT call run() on MULTIPLE
486         io_contexts\n";
```

```
484     std::cout << " 4. run() returns when work queue is empty (use work_guard  
485         to prevent)\n";  
486     std::cout << " 5. Call stop() to force run() to exit early\n\n";  
487 }  
488 //  
489 // SECTION 7: Real-World Example - Microservice Gateway  
490 //  
491 //  
492 void demonstrate_microservice_gateway() {  
493     std::cout << "\n" << std::string(70, '=') << "\n";  
494     std::cout << "== 7. Real-World Example: Microservice API Gateway ==\n";  
495     std::cout << std::string(70, '=') << "\n\n";  
496  
497     std::cout << "Architecture:\n";  
498     std::cout << "           \n";  
499     std::cout << "           io_frontend ← Public internet clients (rate limited  
500         )\n";  
500     std::cout << "           (1 thread)    \n";  
501     std::cout << "           \n";  
502     std::cout << "           ↓           \n";  
503     std::cout << "           \n";  
504     std::cout << "           io_internal ← Internal microservices (fast LAN)\n";  
505     std::cout << "           (4 threads)    \n";  
506     std::cout << "           \n";  
507     std::cout << "           ↓           \n";  
508     std::cout << "           \n";  
509     std::cout << "           io_database ← Database connections (controlled)\n";  
510     std::cout << "           (2 threads)    \n";  
511     std::cout << "           \n\n";  
512  
513     SimulatedAsio::io_context io_frontend("FRONTEND");  
514     SimulatedAsio::io_context io_internal("INTERNAL");  
515     SimulatedAsio::io_context io_database("DATABASE");  
516  
517     // Simulate incoming requests  
518     std::cout << "Processing requests...\n\n";  
519  
520     io_frontend.post([]() {  
521         std::cout << "  [FRONTEND] Public client → Authentication\n";  
522         std::this_thread::sleep_for(50ms);  
523     });  
524  
525     io_internal.post([]() {  
526         std::cout << "  [INTERNAL] Calling user-service microservice\n";  
527         std::this_thread::sleep_for(30ms);  
528     });  
529  
530     io_internal.post([]() {  
531         std::cout << "  [INTERNAL] Calling order-service microservice\n";  
532     });
```

```
532     std::this_thread::sleep_for(30ms);
533 );
534
535     io_database.post([]() {
536         std::cout << "[DATABASE] Query user preferences\n";
537         std::this_thread::sleep_for(80ms);
538 );
539
540 // Thread allocation reflects importance and load
541 std::thread frontend_thread([]() { io_frontend.run(); });
542
543 std::vector<std::thread> internal_threads;
544 for (int i = 0; i < 2; ++i) { // 2 threads for internal services
545     internal_threads.emplace_back([]() { io_internal.run(); });
546 }
547
548 std::thread db_thread([]() { io_database.run(); });
549
550 frontend_thread.join();
551 for (auto& t : internal_threads) { t.join(); }
552 db_thread.join();
553
554 std::cout << "\n Frontend protected from internal service overload\n";
555 std::cout << " Internal services have high concurrency (4 threads)\n";
556 std::cout << " Database access controlled (2 threads max)\n";
557 std::cout << " Each layer can be scaled independently\n";
558 }
559
560 // =====
561 // SECTION 8: Common Pitfalls and Best Practices
562 // =====
563
564 void demonstrate_best_practices() {
565     std::cout << "\n" << std::string(70, '=') << "\n";
566     std::cout << "==== 8. Common Pitfalls and Best Practices ===\n";
567     std::cout << std::string(70, '=') << "\n\n";
568
569     std::cout << " PITFALL 1: run() exits immediately (no work posted)\n";
570     std::cout << "     asio::io_context io;\n";
571     std::cout << "     std::thread t([]() { io.run(); }); // ← Exits
572         immediately!\n";
573     std::cout << "     io.post([]() { /* ... */ }); // ← Too late, thread
574         already exited\n\n";
575     std::cout << " FIX: Post work BEFORE starting thread, or use work_guard\n
576         \n";
577
578     std::cout << " PITFALL 2: One thread calling run() on multiple contexts\n
579         ";
580     std::cout << "     io1.run(); // Blocks until io1 is done\n";
581     std::cout << "     io2.run(); // ← io2 never runs until io1 finishes!\n\n";
```

```
578     std::cout << "  FIX: Use separate threads for each io_context\n\n";
579
580     std::cout << "  PITFALL 3: Forgetting to join threads\n";
581     std::cout << "    std::thread t([&]() { io.run(); });\n";
582     std::cout << "    // ← main() exits, std::terminate() called!\n\n";
583     std::cout << "  FIX: Always join or detach threads\n\n";
584
585     std::cout << "  PITFALL 4: Sharing mutable state without synchronization\n";
586     std::cout << "    ";
587     std::cout << "    int counter = 0; // ← Multiple threads accessing!\n";
588     std::cout << "    io.post([&]() { ++counter; }); // ← RACE CONDITION\n\n";
589     std::cout << "  FIX: Use std::atomic or std::mutex for shared state\n\n";
590
591     std::cout << "BEST PRACTICES:\n";
592     std::cout << "    \n";
593     std::cout << "    1. Use multiple io_contexts to SEPARATE concerns (not for
594         parallelism)\n";
595     std::cout << "    2. For parallelism, use THREAD POOL (multiple threads, one
596         io_context)\n";
597     std::cout << "    3. Number of threads    number of CPU cores (for CPU-bound
598         work)\n";
599     std::cout << "    4. For I/O-bound work, can have many more threads than
600         cores\n";
601     std::cout << "    5. Always use strand for sequential execution across
602         threads\n";
603     std::cout << "    6. Profile before optimizing - one io_context is often
604         sufficient\n\n";
605
606 }
607
608 // -----
609 // MAIN FUNCTION
610 // -----
611
612 int main() {
613     std::cout << "\n";
614     std::cout << "                                \n";
615     std::cout << "                                ASIO Multiple io_context - Educational Examples
616                                \n";
617     std::cout << "                                \n";
618
619     ;
620     std::cout << "    Learn when and how to use multiple io_context objects in
621         ASIO      \n";
622     std::cout << "                                \n";
623
624     demonstrate_basic_multiple_contexts();
625     demonstrate_thread_pool_pattern();
626     demonstrate_lan_wan_separation();           // ← Your requested example!
627     demonstrate_priority_separation();
628     demonstrate_service_type_separation();
```

```
617 demonstrate_run_call_patterns();
618 demonstrate_microservice_gateway();
619 demonstrate_best_practices();
620
621 std::cout << "\n" << std::string(70, '=') << "\n";
622 std::cout << "All demonstrations completed!\n";
623 std::cout << "\nKEY TAKEAWAY:\n";
624 std::cout << "  Multiple io_context = SEPARATION (isolate different
625   concerns)\n";
626 std::cout << "  Multiple threads on one io_context = PARALLELISM (
627   distribute work)\n";
628 std::cout << std::string(70, '=') << "\n\n";
629
630 }
```

## 6 Source Code: BinarySearch.cpp

**File:** src/BinarySearch.cpp

**Repository:** [View on GitHub](#)

```
47 int main () {
48     Solution testSolution;
49     int n = 3;
50     std::vector<int> searchArray(1); // initialize vector of n elements with 0
51
52     // push_back another n - 1 elements
53     for (int i = 1; i < n; i++)
54         searchArray.push_back(i);
55
56     // n element in vector now
57     for (std::vector<int>::const_iterator vecIT = searchArray.begin(); vecIT <
58         searchArray.end(); vecIT++) {
59         std::cout << *vecIT << std::endl;
60     }
61     std::cout << std::endl;
62
63     std::cout << std::distance(searchArray.begin(), searchArray.end()) << "\n"
64         << std::endl;
65
66     for (int i = 0; i <= n; i++) {
67         std::cout << testSolution.binarySearch(searchArray, i) << std::endl;
68         std::cout << testSolution.binarySearchRecursion(searchArray, 0, n, i)
69             << std::endl;
70     }
71
72     return 0;
73 }
```

## 7 Source Code: `CRTPvsVirtualFunctions.cpp`

**File:** `src/CRTPvsVirtualFunctions.cpp`

**Repository:** [View on GitHub](#)

```
1  /*
2  *  CRTP vs Virtual Functions: Static vs Dynamic Polymorphism
3  *
4  *  This example demonstrates:
5  *  1. How virtual functions work (vtable mechanism)
6  *  2. Performance overhead of virtual functions
7  *  3. CRTP (Curiously Recurring Template Pattern) as zero-overhead alternative
8  *  4. When to use each approach
9  *  5. Benchmarking and trade-offs
10 */
11
12 #include <iostream>
13 #include <string>
14 #include <vector>
15 #include <memory>
16 #include <chrono>
17 #include <iomanip>
18
19 /**
20  =====
21 // SECTION 1: VIRTUAL FUNCTIONS AND VTABLE MECHANISM
22 // =====
23
24 namespace virtual_functions {
25
26 /**
27 * VTABLE (Virtual Function Table) EXPLANATION:
28 *
29 * When you use virtual functions, the compiler creates:
30 *
31 * 1. VTABLE (per class):
32 *      - Static array of function pointers
33 *      - One vtable per polymorphic class
34 *      - Stored in read-only memory
35 *
36 * 2. VPTR (per object):
37 *      - Hidden pointer member in each object
38 *      - Points to the class's vtable
39 *      - Added automatically by compiler
40 *      - Size overhead: 8 bytes (64-bit system)
41 *
42 * MEMORY LAYOUT EXAMPLE:
43 *
44 * class Base {
45 *     virtual void foo();
46 *     int data;
```

```
46 *  };
47 *
48 * Object memory:  [vptr][data]
49 *                  |
50 *                  v
51 * Vtable:          [&Base::foo][type_info*]
52 *
53 * CALL MECHANISM:
54 * obj.foo() becomes:
55 *   1. Load vptr from object
56 *   2. Index into vtable
57 *   3. Load function pointer
58 *   4. Call through pointer
59 *
60 * PERFORMANCE COSTS:
61 * • 2-3 extra memory loads (vptr + function pointer)
62 * • Cannot inline (compiler doesn't know target at compile-time)
63 * • Prevents devirtualization optimizations
64 * • Cache misses (vtable might not be in cache)
65 * • Branch prediction harder (indirect call)
66 */
67
68 class AnimalBase {
69 public:
70     virtual ~AnimalBase() = default;
71
72     // Virtual function - uses vtable dispatch
73     virtual void speak() const = 0;
74     virtual std::string getName() const = 0;
75
76     // Virtual function with implementation
77     virtual void describe() const {
78         std::cout << "    I am a " << getName() << "\n";
79     }
80 };
81
82 class Dog : public AnimalBase {
83 public:
84     void speak() const override {
85         std::cout << "    Woof! Woof!\n";
86     }
87
88     std::string getName() const override {
89         return "Dog";
90     }
91 };
92
93 class Cat : public AnimalBase {
94 public:
95     void speak() const override {
96         std::cout << "    Meow! Meow!\n";
97     }
98
99     std::string getName() const override {
```

```
100         return "Cat";
101     }
102 }
103
104 class Bird : public AnimalBase {
105 public:
106     void speak() const override {
107         std::cout << "      Tweet! Tweet!\n";
108     }
109
110     std::string getName() const override {
111         return "Bird";
112     }
113 }
114
115 void demonstrate() {
116     std::cout << "\n" << std::string(70, '=') << "\n";
117     std::cout << "SECTION 1: VIRTUAL FUNCTIONS (DYNAMIC POLYMORPHISM)\n";
118     std::cout << std::string(70, '=') << "\n\n";
119
120     std::cout << "  VTABLE MECHANISM:\n";
121     std::cout << "  •  Each object has hidden vptr (8 bytes overhead)\n";
122     std::cout << "  •  vptr points to class vtable\n";
123     std::cout << "  •  Function calls: load vptr -> index vtable -> call\n";
124     std::cout << "  •  Runtime overhead: 2-3 memory loads per virtual call\n\n";
125
126 // Demonstrate dynamic polymorphism
127 std::vector<std::unique_ptr<AnimalBase>> animals;
128 animals.push_back(std::make_unique<Dog>());
129 animals.push_back(std::make_unique<Cat>());
130 animals.push_back(std::make_unique<Bird>());
131
132 std::cout << "  ADVANTAGE: Runtime polymorphism (heterogeneous containers)\n";
133 for (const auto& animal : animals) {
134     animal->describe();
135     animal->speak();
136 }
137
138 std::cout << "  SIZE OVERHEAD:\n";
139 std::cout << "    sizeof(AnimalBase*) = " << sizeof(AnimalBase*) << " bytes\n";
140 std::cout << "    sizeof(Dog) = " << sizeof(Dog) << " bytes (includes vptr)\n";
141 std::cout << "    sizeof(Cat) = " << sizeof(Cat) << " bytes (includes vptr)\n";
142 }
143
144 } // namespace virtual_functions
145
146 // =====
```

```
147 // SECTION 2: STATIC MEMBER FUNCTIONS AND VTABLES
148 //
149 //=====
150
151 namespace static_functions {
152 /*
153 * STATIC MEMBER FUNCTIONS AND VTABLES:
154 *
155 * KEY FACT: Static member functions CANNOT be virtual!
156 *
157 * WHY?
158 * • Static functions don't have 'this' pointer
159 * • No object instance to determine which vtable to use
160 * • Called via class name, not object
161 * • Resolved at compile-time, not runtime
162 *
163 * IMPLICATION:
164 * • Static functions don't appear in vtable
165 * • Only ONE implementation exists (class-level, not object-level)
166 * • Cannot be overridden in derived classes
167 * • Zero runtime overhead (direct call)
168 *
169 * WHEN TO USE:
170 * • Factory functions
171 * • Utility functions that don't need object state
172 * • Performance-critical code that doesn't need polymorphism
173 */
174
175 class Base {
176 public:
177     virtual ~Base() = default;
178
179     // Virtual function - in vtable
180     virtual void instanceMethod() const {
181         std::cout << "    Base::instanceMethod() - uses vtable\n";
182     }
183
184     // Static function - NOT in vtable, cannot be virtual
185     static void staticMethod() {
186         std::cout << "    Base::staticMethod() - no vtable, direct call\n";
187     }
188
189     // Non-virtual function - NOT in vtable
190     void nonVirtualMethod() const {
191         std::cout << "    Base::nonVirtualMethod() - no vtable, direct call\n";
192     }
193 };
194
195 class Derived : public Base {
196 public:
197     // Overrides virtual function - has entry in Derived's vtable
198     void instanceMethod() const override {
```

```
199     std::cout << "    Derived::instanceMethod() - uses vtable\n";
200 }
201
202 // CANNOT override static function (compile error if we tried)
203 // This is a NEW function, not an override
204 static void staticMethod() {
205     std::cout << "    Derived::staticMethod() - different function!\n";
206 }
207
208 // Hides base class function (NOT override, no virtual)
209 void nonVirtualMethod() const {
210     std::cout << "    Derived::nonVirtualMethod() - hides base, no vtable\n"
211     "";
212 }
213
214 void demonstrate() {
215     std::cout << "\n" << std::string(70, '=') << "\n";
216     std::cout << "SECTION 2: STATIC FUNCTIONS AND VTABLES\n";
217     std::cout << std::string(70, '=') << "\n\n";
218
219     std::cout << "    STATIC FUNCTIONS:\n";
220     std::cout << "    •    Cannot be virtual (no 'this' pointer)\n";
221     std::cout << "    •    NOT in vtable (only ONE implementation)\n";
222     std::cout << "    •    Called via class name, resolved at compile-time\n";
223     std::cout << "    •    Zero runtime overhead\n\n";
224
225     Base* ptr = new Derived();
226
227     std::cout << "    VIRTUAL FUNCTION (uses vtable):\n";
228     ptr->instanceMethod(); // Calls Derived::instanceMethod via vtable
229
230     std::cout << "\n    STATIC FUNCTION (no vtable, direct call):\n";
231     Base::staticMethod(); // Calls Base::staticMethod
232     Derived::staticMethod(); // Calls Derived::staticMethod (different
233     // function!)
234
235     std::cout << "\n    NON-VIRTUAL FUNCTION (no override, no vtable):\n";
236     ptr->nonVirtualMethod(); // Calls Base::nonVirtualMethod (no polymorphism
237     !
238
239     std::cout << "\n    KEY INSIGHT:\n";
240     std::cout << "    •    Virtual functions: ONE entry in vtable per class\n";
241     std::cout << "    •    Static functions: ONE function for entire class
242     hierarchy\n";
243     std::cout << "    •    Static means 'belongs to class, not object'\n";
244
245     delete ptr;
246 }
247 } // namespace static_functions
248
249 // =====
```

```
248 // SECTION 3: CRTP - STATIC POLYMORPHISM (ZERO OVERHEAD)
249 // =====
250
251 namespace crtp_pattern {
252
253 /*
254 * CRTP: Curiously Recurring Template Pattern
255 *
256 * HOW IT WORKS:
257 * • Base class is template parameterized by derived class
258 * • Base uses static_cast to call derived methods
259 * • Compiler resolves calls at compile-time
260 * • NO vtable, NO vptr, NO runtime overhead
261 *
262 * CALL MECHANISM:
263 * obj.speak() becomes:
264 * 1. Compiler knows exact type at compile-time
265 * 2. Direct function call (no indirection)
266 * 3. Can inline the function
267 * 4. Zero runtime overhead
268 *
269 * BENEFITS:
270 * • Zero runtime overhead (no vtable lookup)
271 * • Functions can be inlined
272 * • No memory overhead (no vptr)
273 * • Compiler can optimize aggressively
274 *
275 * DRAWBACKS:
276 * • Cannot use heterogeneous containers (no common base type)
277 * • Type known at compile-time (no runtime polymorphism)
278 * • More complex code (template metaprogramming)
279 * • Longer compile times
280 */
281
282 // CRTP Base class
283 template <typename Derived>
284 class Animal {
285 public:
286     void speak() const {
287         // Static cast to derived type - resolved at compile-time
288         static_cast<const Derived*>(this)->speak_impl();
289     }
290
291     std::string getName() const {
292         return static_cast<const Derived*>(this)->getName_impl();
293     }
294
295     void describe() const {
296         std::cout << " I am a " << getName() << "\n";
297     }
298 }
```

```
299 // Optional: Provide default implementation
300 void defaultBehavior() const {
301     std::cout << "    All animals need food and water\n";
302 }
303 };
304
305 class Dog : public Animal<Dog> {
306 public:
307     void speak_impl() const {
308         std::cout << "    Woof! Woof!\n";
309     }
310
311     std::string getName_impl() const {
312         return "Dog";
313     }
314 };
315
316 class Cat : public Animal<Cat> {
317 public:
318     void speak_impl() const {
319         std::cout << "    Meow! Meow!\n";
320     }
321
322     std::string getName_impl() const {
323         return "Cat";
324     }
325 };
326
327 class Bird : public Animal<Bird> {
328 public:
329     void speak_impl() const {
330         std::cout << "    Tweet! Tweet!\n";
331     }
332
333     std::string getName_impl() const {
334         return "Bird";
335     }
336 };
337
338 // Template function that works with any Animal<T>
339 template <typename T>
340 void makeAnimalSpeak(const Animal<T>& animal) {
341     animal.describe();
342     animal.speak();
343 }
344
345 void demonstrate() {
346     std::cout << "\n" << std::string(70, '=') << "\n";
347     std::cout << "SECTION 3: CRTP - STATIC POLYMORPHISM (ZERO OVERHEAD)\n";
348     std::cout << std::string(70, '=') << "\n\n";
349
350     std::cout << "    CRTP MECHANISM:\n";
351     std::cout << "    •    Compile-time polymorphism (no runtime overhead)\n";
352     std::cout << "    •    No vtable, no vptr (zero memory overhead)\n";
353 }
```

```
353     std::cout << " • Direct function calls (can be inlined)\n";
354     std::cout << " • Compiler knows exact type at compile-time\n\n";
355
356     Dog d;
357     Cat c;
358     Bird b;
359
360     std::cout << " DIRECT CALLS (statically resolved):\n";
361     d.speak(); // Compiler knows this is Dog::speak_impl at compile-time
362     c.speak(); // Compiler knows this is Cat::speak_impl at compile-time
363     b.speak(); // Compiler knows this is Bird::speak_impl at compile-time
364
365     std::cout << "\n TEMPLATE FUNCTION (works with any Animal<T>):\n";
366     makeAnimalSpeak(d);
367     makeAnimalSpeak(c);
368     makeAnimalSpeak(b);
369
370     std::cout << "\n SIZE COMPARISON:\n";
371     std::cout << " sizeof(Dog) = " << sizeof(Dog) << " bytes (NO vptr!)\n";
372     std::cout << " sizeof(Cat) = " << sizeof(Cat) << " bytes (NO vptr!)\n";
373     std::cout << " Compare to virtual_functions::Dog = "
374             << sizeof(virtual_functions::Dog) << " bytes (with vptr)\n";
375
376     std::cout << "\n LIMITATION: Cannot create heterogeneous container:\n";
377     std::cout << " // std::vector<Animal<??>> animals; // Won't work!\n";
378     std::cout << " // Dog and Cat are DIFFERENT types (Animal<Dog> vs Animal
379             <Cat>)\n";
380 }
381 } // namespace crtp_pattern
382 //
383 // =====
384 // SECTION 4: PERFORMANCE COMPARISON
385 // =====
386
387 namespace performance_comparison {
388
389 // Virtual function version
390 class VirtualAnimal {
391 public:
392     virtual ~VirtualAnimal() = default;
393     virtual int compute(int x) const = 0;
394 };
395
396 class VirtualDog : public VirtualAnimal {
397 public:
398     int compute(int x) const override {
399         return x * x + x; // Simple computation
400     }
401 };
```

```
402 // CRTP version
403 template <typename Derived>
404 class CRTPAnimal {
405     public:
406         int compute(int x) const {
407             return static_cast<const Derived*>(this)>compute_<b>impl</b>(x);
408         }
409     };
410 };
411
412 class CRTPDog : public CRTPAnimal<CRTPDog> {
413     public:
414         int compute_<b>impl</b>(int x) const {
415             return x * x + x; // Same computation
416         }
417     };
418
419     void demonstrate() {
420         std::cout << "\n" << std::string(70, '=') << "\n";
421         std::cout << "SECTION 4: PERFORMANCE BENCHMARKING\n";
422         std::cout << std::string(70, '=') << "\n\n";
423
424         constexpr int ITERATIONS = 10'000'000;
425
426         // Benchmark virtual functions
427         auto start = std::chrono::high_resolution_clock::now();
428     {
429         VirtualDog dog;
430         VirtualAnimal* ptr = &dog;
431         long long sum = 0;
432         for (int i = 0; i < ITERATIONS; ++i) {
433             sum += ptr->compute(i % 100); // Virtual call
434         }
435         // Use sum to prevent optimization
436         if (sum > 0) {}
437     }
438     auto end = std::chrono::high_resolution_clock::now();
439     auto virtual_time = std::chrono::duration_cast<std::chrono::milliseconds>(
440         end - start).count();
441
442     // Benchmark CRTP
443     start = std::chrono::high_resolution_clock::now();
444     {
445         CRTPDog dog;
446         long long sum = 0;
447         for (int i = 0; i < ITERATIONS; ++i) {
448             sum += dog.compute(i % 100); // Static call
449         }
450         // Use sum to prevent optimization
451         if (sum > 0) {}
452     }
453     end = std::chrono::high_resolution_clock::now();
454     auto crtp_time = std::chrono::duration_cast<std::chrono::milliseconds>(end
455         - start).count();
```

```

454
455     std::cout << " BENCHMARK: " << ITERATIONS << " function calls\n\n";
456     std::cout << "     Virtual functions: " << std::setw(5) << virtual_time <<
457         " ms\n";
458     std::cout << "     CRTP (static): " << std::setw(5) << crtp_time << " ms\n";
459     std::cout << "     Speedup: " << std::setw(5) << std::fixed <<
460         std::setprecision(2)
461             << (double)virtual_time / crtp_time << "x\n\n";
462
463     std::cout << " WHY CRTP IS FASTER:\n";
464     std::cout << " •     No vtable lookup (2-3 memory loads eliminated)\n";
465     std::cout << " •     Function can be inlined (compiler sees implementation)\n";
466     std::cout << " •     Better cache locality (no vtable indirection)\n";
467     std::cout << " •     Better branch prediction (direct call)\n";
468     std::cout << " •     Enables other compiler optimizations\n";
469 }
470
471 } // namespace performance_comparison
472
473 // =====
474 // SECTION 5: WHEN TO USE EACH APPROACH
475 // =====
476
477 namespace when_to_use {
478
479     void demonstrate() {
480         std::cout << "\n" << std::string(70, '=') << "\n";
481         std::cout << "SECTION 5: WHEN TO USE VIRTUAL FUNCTIONS VS CRTP\n";
482         std::cout << std::string(70, '=') << "\n\n";
483
484         std::cout << "     USE VIRTUAL FUNCTIONS WHEN:\n\n";
485         std::cout << "         1 RUNTIME POLYMORPHISM NEEDED:\n";
486         std::cout << "             •     Heterogeneous containers (vector<Base*>)\n";
487         std::cout << "             •     Plugin systems (load types at runtime)\n";
488         std::cout << "             •     Factory patterns\n";
489         std::cout << "             •     Type not known until runtime\n\n";
490
491         std::cout << "     2 INTERFACE-BASED DESIGN:\n";
492         std::cout << "             •     Defining abstract interfaces\n";
493         std::cout << "             •     Separating interface from implementation\n";
494         std::cout << "             •     Dependency injection\n\n";
495
496         std::cout << "     3 BINARY COMPATIBILITY:\n";
497         std::cout << "             •     DLLs/shared libraries\n";
498         std::cout << "             •     ABI stability requirements\n";
499         std::cout << "             •     Plugin systems across compilation units\n\n";
500
501         std::cout << "     4 CODE SIMPLICITY:\n";
502     }
503 }

```

```

500 std::cout << " • Simpler code (no template metaprogramming)\n";
501 std::cout << " • Faster compile times\n";
502 std::cout << " • More familiar to developers\n\n";
503
504 std::cout << " 5 PERFORMANCE NOT CRITICAL:\n";
505 std::cout << " • UI code, configuration, initialization\n";
506 std::cout << " • Virtual call overhead is negligible\n";
507 std::cout << " • Clarity more important than speed\n\n";
508
509 std::cout << std::string(70, '=') << "\n\n";
510
511 std::cout << " USE CRTP (STATIC POLYMORPHISM) WHEN:\n\n";
512 std::cout << " 1 PERFORMANCE CRITICAL:\n";
513 std::cout << " • Hot loops (inner loops, tight iterations)\n";
514 std::cout << " • Real-time systems (low latency required)\n";
515 std::cout << " • Game engines (per-frame calculations)\n";
516 std::cout << " • High-frequency trading systems\n\n";
517
518 std::cout << " 2 MEMORY CONSTRAINED:\n";
519 std::cout << " • Embedded systems (every byte counts)\n";
520 std::cout << " • Large arrays of objects (vptr overhead * N)\n";
521 std::cout << " • Cache-sensitive code\n\n";
522
523 std::cout << " 3 COMPILE-TIME POLYMORPHISM SUFFICIENT:\n";
524 std::cout << " • Type known at compile-time\n";
525 std::cout << " • Template containers (vector<Dog>, vector<Cat>)\n";
526 std::cout << " • Generic algorithms\n\n";
527
528 std::cout << " 4 INLINING REQUIRED:\n";
529 std::cout << " • Small functions that must be inlined\n";
530 std::cout << " • Zero-overhead abstractions needed\n";
531 std::cout << " • Maximum compiler optimization wanted\n\n";
532
533 std::cout << " 5 TEMPLATE-BASED LIBRARIES:\n";
534 std::cout << " • STL-style libraries (iterators, algorithms)\n";
535 std::cout << " • Expression templates\n";
536 std::cout << " • Policy-based design\n\n";
537
538 std::cout << std::string(70, '=') << "\n\n";
539
540 std::cout << " TRADE-OFFS SUMMARY:\n\n";
541 std::cout << " VIRTUAL FUNCTIONS:\n";
542 std::cout << " • Runtime polymorphism\n";
543 std::cout << " • Heterogeneous containers\n";
544 std::cout << " • Simpler code\n";
545 std::cout << " • Faster compile times\n";
546 std::cout << " • Runtime overhead (vtable lookup)\n";
547 std::cout << " • Memory overhead (vptr per object)\n";
548 std::cout << " • Cannot inline\n\n";
549
550 std::cout << " CRTP (STATIC POLYMORPHISM):\n";
551 std::cout << " • Zero runtime overhead\n";
552 std::cout << " • Zero memory overhead\n";
553 std::cout << " • Can inline functions\n";

```

```
554     std::cout << "      Maximum performance\n";
555     std::cout << "      No runtime polymorphism\n";
556     std::cout << "      No heterogeneous containers\n";
557     std::cout << "      More complex code\n";
558     std::cout << "      Longer compile times\n\n";
559
560     std::cout << "  GOLDEN RULE:\n";
561     std::cout << "      'Use virtual functions by default for flexibility.\n";
562     std::cout << "      Use CRTP only when performance profiling shows\n";
563     std::cout << "      virtual function overhead is a bottleneck.\n\n";
564
565     std::cout << "  PREMATURE OPTIMIZATION WARNING:\n";
566     std::cout << "  •  Don't use CRTP everywhere \"just in case\"\n";
567     std::cout << "  •  Profile first, optimize later\n";
568     std::cout << "  •  Virtual function overhead is often negligible\n";
569     std::cout << "  •  Code clarity often more valuable than tiny speedup\n";
570 }
571
572 } // namespace when_to_use
573
574 // =====
575 // SECTION 6: HYBRID APPROACH - BEST OF BOTH WORLDS
576 // =====
577
578 namespace hybrid_approach {
579
580 /*
581 * HYBRID APPROACH: Combine virtual functions with CRTP
582 *
583 * STRATEGY:
584 *  • Use virtual functions for high-level interfaces
585 *  • Use CRTP for performance-critical inner operations
586 *  • Get both flexibility and performance
587 */
588
589 // Virtual base for runtime polymorphism
590 class RenderableBase {
591 public:
592     virtual ~RenderableBase() = default;
593     virtual void render() const = 0;
594 };
595
596 // CRTP for performance-critical operations
597 template <typename Derived>
598 class FastOperations {
599 public:
600     void processPixel(int x, int y) const {
601         static_cast<const Derived*>(this)->processPixel_impl(x, y);
602     }
603 };
```

```
604 // Concrete class uses both
605 class Sprite : public RenderableBase, public FastOperations<Sprite> {
606 private:
607     mutable int pixel_count = 0;
608
609 public:
610     // Virtual function for high-level interface
611     void render() const override {
612         std::cout << "    [Sprite] Rendering...\n";
613         // Performance-critical inner loop uses CRTP
614         for (int y = 0; y < 100; ++y) {
615             for (int x = 0; x < 100; ++x) {
616                 processPixel(x, y); // CRTP call (zero overhead)
617             }
618         }
619         std::cout << "    [Sprite] Processed " << pixel_count << " pixels\n";
620     }
621
622     // CRTP implementation (inlined, zero overhead)
623     void processPixel_impl(int x, int y) const {
624         // Performance-critical pixel processing
625         ++pixel_count;
626         // Actual pixel operations would go here
627         (void)x; (void)y; // Suppress unused warnings
628     }
629 };
630
631 void demonstrate() {
632     std::cout << "\n" << std::string(70, '=') << "\n";
633     std::cout << "SECTION 6: HYBRID APPROACH - BEST OF BOTH WORLDS\n";
634     std::cout << std::string(70, '=') << "\n\n";
635
636     std::cout << "    STRATEGY:\n";
637     std::cout << "    •    Virtual functions for high-level interface\n";
638     std::cout << "    •    CRTP for performance-critical inner loops\n";
639     std::cout << "    •    Get both flexibility AND performance\n\n";
640
641     std::cout << "    DEMONSTRATION:\n";
642     Sprite sprite;
643     RenderableBase* ptr = &sprite;
644
645     // Virtual call for high-level operation
646     ptr->render(); // Virtual call (small overhead, called once)
647             // Inner loop uses CRTP (zero overhead, called 10000x)
648
649     std::cout << "\n BENEFIT:\n";
650     std::cout << "    •    Can store in vector<RenderableBase*> (runtime
651             polymorphism)\n";
652     std::cout << "    •    Inner loop has zero overhead (CRTP inlining)\n";
653     std::cout << "    •    Best of both worlds!\n";
654 }
655
656 } // namespace hybrid_approach
```

```
657 //  
658 //=====  
659 // MAIN  
660 //=====  
661  
662 int main() {  
663     std::cout << "\n";  
664     std::cout << "  
665         std::cout << "           CRTP vs VIRTUAL FUNCTIONS IN MODERN C++  
666             \n";  
667         std::cout << "           Static vs Dynamic Polymorphism Trade-offs  
668             \n";  
669         std::cout << "           \n";  
670     virtual_functions::demonstrate();  
671     static_functions::demonstrate();  
672     crtp_pattern::demonstrate();  
673     performance_comparison::demonstrate();  
674     when_to_use::demonstrate();  
675     hybrid_approach::demonstrate();  
676  
677     std::cout << "\n" << std::string(70, '=') << "\n";  
678     std::cout << "KEY TAKEAWAYS:\n";  
679     std::cout << std::string(70, '=') << "\n\n";  
680  
681     std::cout << " 1 VIRTUAL FUNCTIONS:\n";  
682     std::cout << "  • Use vtable (per class) and vptr (per object)\n";  
683     std::cout << "  • Runtime overhead: 2-3 memory loads per call\n";  
684     std::cout << "  • Memory overhead: 8 bytes per object (vptr)\n";  
685     std::cout << "  • Cannot inline virtual calls\n";  
686     std::cout << "  • Enable runtime polymorphism\n\n";  
687  
688     std::cout << " 2 STATIC MEMBER FUNCTIONS:\n";  
689     std::cout << "  • Cannot be virtual (no 'this' pointer)\n";  
690     std::cout << "  • Only ONE function per class hierarchy\n";  
691     std::cout << "  • Zero runtime overhead (direct call)\n";  
692     std::cout << "  • Not in vtable\n\n";  
693  
694     std::cout << " 3 CRTP (CURIOSLY RECURRING TEMPLATE PATTERN):\n";  
695     std::cout << "  • Zero runtime overhead (compile-time resolution)\n";  
696     std::cout << "  • Zero memory overhead (no vptr)\n";  
697     std::cout << "  • Functions can be inlined\n";  
698     std::cout << "  • No heterogeneous containers\n";  
699     std::cout << "  • More complex code\n\n";  
700  
701     std::cout << " 4 WHEN TO USE WHAT:\n";  
702     std::cout << "  • Virtual: Flexibility, runtime polymorphism, simplicity\n";  
703     std::cout << "  • CRTP: Performance-critical code, compile-time known  
704         types\n";
```

```
703     std::cout << " • Hybrid: High-level virtual + low-level CRTP\n\n";
704
705     std::cout << " 5 PERFORMANCE:\n";
706     std::cout << " • Virtual overhead often negligible in real applications\
707           \n";
708     std::cout << " • CRTP can be 2-10x faster in tight loops\n";
709     std::cout << " • Profile before optimizing!\n\n";
710
711     std::cout << "                               \n";
712     std::cout << "           ALL CONCEPTS DEMONSTRATED SUCCESSFULLY!
713           \n";
714
715     std::cout << "                               \n\n";
716
717     return 0;
718 }
```

## 8 Source Code: CameraModule.cppm

File: src/CameraModule.cppm

Repository: [View on GitHub](#)

```
1 // =====
2 // C++20 MODULE: CAMERA INTERFACE WITH TEMPLATES
3 // =====
4 // Demonstrates C++20 modules with export/import
5 // Also includes concepts for type constraints
6 // =====
7
8 module;
9
10 // Global module fragment - for #include directives
11 #include <vector>
12 #include <memory>
13 #include <string>
14 #include <cstdint>
15 #include <type_traits>
16 #include <algorithm>
17 #include <cmath>
18 #include <utility>
19 #include <concepts>
20
21 export module camera;
22
23 // =====
24 // C++20 CONCEPTS FOR PIXEL TYPES
25 // =====
26
27 export template<typename T>
28 concept PixelType = std::is_arithmetic_v<T> && (
29     std::is_same_v<T, uint8_t> ||
30     std::is_same_v<T, uint16_t> ||
31     std::is_same_v<T, uint32_t> ||
32     std::is_same_v<T, float> ||
33     std::is_same_v<T, double>
34 );
35
36 export template<typename T>
37 concept IntegerPixel = PixelType<T> && std::is_integral_v<T>;
38
39 export template<typename T>
40 concept FloatingPixel = PixelType<T> && std::is_floating_point_v<T>;
41
42 // =====
43 // IMAGE CLASS (EXPORTED)
44 // =====
45
46 export template<PixelType T>
47 class Image {
48 private:
49     size_t width;
```

```
50     size_t height;
51     std::vector<T> pixels;
52
53 public:
54     Image(size_t w, size_t h)
55         : width(w), height(h), pixels(w * h) {}
56
57     Image(size_t w, size_t h, T initial_value)
58         : width(w), height(h), pixels(w * h, initial_value) {}
59
60     // Accessors
61     [[nodiscard]] size_t get_width() const noexcept { return width; }
62     [[nodiscard]] size_t get_height() const noexcept { return height; }
63     [[nodiscard]] size_t get_size() const noexcept { return pixels.size(); }
64
65     // Pixel access
66     T& at(size_t x, size_t y) {
67         return pixels[y * width + x];
68     }
69
70     const T& at(size_t x, size_t y) const {
71         return pixels[y * width + x];
72     }
73
74     // Raw data access
75     [[nodiscard]] T* data() noexcept { return pixels.data(); }
76     [[nodiscard]] const T* data() const noexcept { return pixels.data(); }
77
78     // Memory size
79     [[nodiscard]] size_t memory_bytes() const noexcept {
80         return pixels.size() * sizeof(T);
81     }
82
83     // Fill with value
84     void fill(T value) {
85         std::fill(pixels.begin(), pixels.end(), value);
86     }
87
88     // C++20: Three-way comparison
89     auto operator<=>(const Image&) const = default;
90 };
91
92 // =====
93 // CAMERA INTERFACE (EXPORTED)
94 // =====
95
96 export template<PixelType T>
97 class Camera {
98     private:
99         size_t width;
100        size_t height;
101        std::string camera_name;
102
103 public:
```

```
104     Camera(const std::string& name, size_t w, size_t h)
105         : camera_name(name), width(w), height(h) {}
106
107     virtual ~Camera() = default;
108
109     // Pure virtual: capture image
110     virtual Image<T> capture() = 0;
111
112     // Configuration
113     [[nodiscard]] size_t get_width() const noexcept { return width; }
114     [[nodiscard]] size_t get_height() const noexcept { return height; }
115     [[nodiscard]] const std::string& get_name() const noexcept { return
116         camera_name; }
117
118     // Get pixel type information
119     [[nodiscard]] static constexpr size_t bits_per_pixel() noexcept {
120         return sizeof(T) * 8;
121     }
122
123     [[nodiscard]] static constexpr bool is_floating_point() noexcept {
124         return std::is_floating_point_v<T>;
125     }
126
127     [[nodiscard]] static constexpr bool is_integer() noexcept {
128         return std::is_integral_v<T>;
129     };
130
131 // =====
132 // IMAGE PROCESSOR (EXPORTED)
133 // =====
134
135 export template<PixelType T>
136 class ImageProcessor {
137 public:
138     // Calculate average pixel value
139     [[nodiscard]] static double calculate_mean(const Image<T>& img) {
140         double sum = 0.0;
141         const T* data = img.data();
142         size_t size = img.get_size();
143
144         for (size_t i = 0; i < size; ++i) {
145             sum += static_cast<double>(data[i]);
146         }
147
148         return sum / size;
149     }
150
151     // Find min and max pixel values
152     [[nodiscard]] static std::pair<T, T> find_min_max(const Image<T>& img) {
153         const T* data = img.data();
154         size_t size = img.get_size();
155
156         T min_val = data[0];
```

```
157     T max_val = data[0];
158
159     for (size_t i = 1; i < size; ++i) {
160         if (data[i] < min_val) min_val = data[i];
161         if (data[i] > max_val) max_val = data[i];
162     }
163
164     return {min_val, max_val};
165 }
166
167 // Scale pixel values
168 [[nodiscard]] static Image<T> scale(const Image<T>& img, double factor) {
169     Image<T> result(img.get_width(), img.get_height());
170
171     for (size_t y = 0; y < img.get_height(); ++y) {
172         for (size_t x = 0; x < img.get_width(); ++x) {
173             double scaled = static_cast<double>(img.at(x, y)) * factor;
174             result.at(x, y) = static_cast<T>(scaled);
175         }
176     }
177
178     return result;
179 }
180
181 // Threshold operation - using concept constraint
182 [[nodiscard]] static Image<T> threshold(const Image<T>& img, T
183     threshold_value)
184     requires IntegerPixel<T>
185 {
186     Image<T> result(img.get_width(), img.get_height());
187
188     for (size_t y = 0; y < img.get_height(); ++y) {
189         for (size_t x = 0; x < img.get_width(); ++x) {
190             result.at(x, y) = (img.at(x, y) >= threshold_value) ?
191                 threshold_value : T(0);
192         }
193     }
194
195     return result;
196 }
197
198 // Normalize operation - only for floating point images
199 [[nodiscard]] static Image<T> normalize(const Image<T>& img)
200     requires FloatingPixel<T>
201 {
202     auto [min_val, max_val] = find_min_max(img);
203     T range = max_val - min_val;
204
205     if (range == T(0)) return img;
206
207     Image<T> result(img.get_width(), img.get_height());
208
209     for (size_t y = 0; y < img.get_height(); ++y) {
210         for (size_t x = 0; x < img.get_width(); ++x) {
```

```
210         result.at(x, y) = (img.at(x, y) - min_val) / range;
211     }
212 }
213
214     return result;
215 }
216 };
217
218 // =====
219 // TYPE CONVERSION (EXPORTED)
220 // =====
221
222 export template<PixelType DestType, PixelType SrcType>
223 [[nodiscard]] Image<DestType> convert_image(const Image<SrcType>& src) {
224     Image<DestType> dest(src.get_width(), src.get_height());
225
226     // Find source range
227     auto [min_val, max_val] = ImageProcessor<SrcType>::find_min_max(src);
228     double src_range = static_cast<double>(max_val) - static_cast<double>(min_val);
229
230     // Determine destination range
231     double dest_min, dest_max;
232     if constexpr (std::is_floating_point_v<DestType>) {
233         dest_min = 0.0;
234         dest_max = 1.0;
235     } else if constexpr (std::is_same_v<DestType, uint8_t>) {
236         dest_min = 0.0;
237         dest_max = 255.0;
238     } else if constexpr (std::is_same_v<DestType, uint16_t>) {
239         dest_min = 0.0;
240         dest_max = 65535.0;
241     } else {
242         dest_min = 0.0;
243         dest_max = 1.0;
244     }
245
246     dest_range = dest_max - dest_min;
247
248     // Convert with proper scaling
249     if (src_range == 0.0) {
250         dest.fill(static_cast<DestType>(dest_min));
251         return dest;
252     }
253
254     for (size_t y = 0; y < src.get_height(); ++y) {
255         for (size_t x = 0; x < src.get_width(); ++x) {
256             double normalized = (static_cast<double>(src.at(x, y)) -
257                             static_cast<double>(min_val)) / src_range;
258             double scaled = normalized * dest_range + dest_min;
259             dest.at(x, y) = static_cast<DestType>(scaled);
260         }
261     }
262 }
```

```
262     return dest;
263 }
264
265 // =====
266 // CAMERA IMPLEMENTATIONS (EXPORTED)
267 // =====
268
269 export class Camera8bit : public Camera<uint8_t> {
270 public:
271     Camera8bit(const std::string& name, size_t w, size_t h)
272         : Camera<uint8_t>(name, w, h) {}
273
274     Image<uint8_t> capture() override {
275         Image<uint8_t> img(get_width(), get_height());
276
277         for (size_t y = 0; y < get_height(); ++y) {
278             for (size_t x = 0; x < get_width(); ++x) {
279                 uint8_t value = static_cast<uint8_t>(
280                     (x * 255.0 / get_width()) * 0.5 +
281                     (y * 255.0 / get_height()) * 0.5
282                 );
283                 img.at(x, y) = value;
284             }
285         }
286         return img;
287     }
288 };
289
290 export class Camera16bit : public Camera<uint16_t> {
291 public:
292     Camera16bit(const std::string& name, size_t w, size_t h)
293         : Camera<uint16_t>(name, w, h) {}
294
295     Image<uint16_t> capture() override {
296         Image<uint16_t> img(get_width(), get_height());
297
298         for (size_t y = 0; y < get_height(); ++y) {
299             for (size_t x = 0; x < get_width(); ++x) {
300                 uint16_t value = static_cast<uint16_t>(
301                     (x * 65535.0 / get_width()) * 0.3 +
302                     (y * 65535.0 / get_height()) * 0.7
303                 );
304                 img.at(x, y) = value;
305             }
306         }
307         return img;
308     }
309 };
310
311 export class CameraFloat : public Camera<float> {
312 public:
313     CameraFloat(const std::string& name, size_t w, size_t h)
314         : Camera<float>(name, w, h) {}
```

```
316     Image<float> capture() override {
317         Image<float> img(get_width(), get_height());
318
319         for (size_t y = 0; y < get_height(); ++y) {
320             for (size_t x = 0; x < get_width(); ++x) {
321                 float value =
322                     0.5f + 0.5f * std::sin(x * 0.1f) * std::cos(y * 0.1f);
323                 img.at(x, y) = value;
324             }
325         }
326         return img;
327     }
328 };
329
330 export class CameraDouble : public Camera<double> {
331 public:
332     CameraDouble(const std::string& name, size_t w, size_t h)
333         : Camera<double>(name, w, h) {}
334
335     Image<double> capture() override {
336         Image<double> img(get_width(), get_height());
337
338         for (size_t y = 0; y < get_height(); ++y) {
339             for (size_t x = 0; x < get_width(); ++x) {
340                 double value =
341                     std::sin(x * 0.05) * std::cos(y * 0.05) +
342                     std::exp(-((x - get_width()/2.0) * (x - get_width()/2.0) +
343                               (y - get_height()/2.0) * (y - get_height()/2.0)
344                               ) / 1000.0);
345                 img.at(x, y) = value;
346             }
347         }
348         return img;
349     }
350 };
351 // =====
352 // CAMERA HANDLER (EXPORTED)
353 // =====
354
355 export template<PixelType T>
356 class CameraHandler {
357 private:
358     std::unique_ptr<Camera<T>> camera;
359
360 public:
361     CameraHandler(std::unique_ptr<Camera<T>> cam)
362         : camera(std::move(cam)) {}
363
364     [[nodiscard]] const Camera<T>* get_camera() const noexcept {
365         return camera.get();
366     }
367
368     [[nodiscard]] Image<T> capture() {
```

```
369     return camera->capture();
370 }
371
372 [[nodiscard]] static constexpr size_t pixel_size() noexcept {
373     return sizeof(T);
374 }
375
376 [[nodiscard]] static constexpr size_t bits_per_pixel() noexcept {
377     return sizeof(T) * 8;
378 }
379 };
```

## 9 Source Code: ConceptsExamples.cpp

File: src/ConceptsExamples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <concepts>
5 #include <type_traits>
6 #include <algorithm>
7 #include <iterator>
8
9 // =====
10 // 1. BASIC CONCEPT DEFINITION
11 // =====
12 template<typename T>
13 concept Numeric = std::is_arithmetic_v<T>;
14
15 template<Numeric T>
16 T add(T a, T b) {
17     return a + b;
18 }
19
20 void example_basic_concept() {
21     std::cout << "\n==== 1. BASIC CONCEPT DEFINITION ===" << std::endl;
22
23     std::cout << "add(10, 20) = " << add(10, 20) << std::endl;
24     std::cout << "add(3.14, 2.86) = " << add(3.14, 2.86) << std::endl;
25
26     // This would fail to compile:
27     // add(std::string("hello"), std::string("world"));
28 }
29
30 // =====
31 // 2. STANDARD LIBRARY CONCEPTS
32 // =====
33 template<std::integral T>
34 T multiply(T a, T b) {
35     return a * b;
36 }
37
38 template<std::floating_point T>
39 T divide(T a, T b) {
40     return a / b;
41 }
42
43 void example_standard_concepts() {
44     std::cout << "\n==== 2. STANDARD LIBRARY CONCEPTS ===" << std::endl;
45
46     std::cout << "multiply(5, 6) = " << multiply(5, 6) << std::endl;
47     std::cout << "divide(10.0, 3.0) = " << divide(10.0, 3.0) << std::endl;
48
49     // multiply(3.14, 2.0); // ERROR: requires integral type
```

```
50     // divide(10, 3);           // ERROR: requires floating point type
51 }
52
53 // =====
54 // 3. COMPOUND CONCEPTS
55 // =====
56 template<typename T>
57 concept Addable = requires(T a, T b) {
58     { a + b } -> std::convertible_to<T>;
59 }
60
61 template<typename T>
62 concept Printable = requires(T t) {
63     { std::cout << t } -> std::convertible_to<std::ostream&>;
64 }
65
66 template<typename T>
67 concept AddableAndPrintable = Addable<T> && Printable<T>;
68
69 template<AddableAndPrintable T>
70 void print_sum(T a, T b) {
71     std::cout << a << " + " << b << " = " << (a + b) << std::endl;
72 }
73
74 void example_compound_concepts() {
75     std::cout << "\n==== 3. COMPOUND CONCEPTS ===" << std::endl;
76
77     print_sum(10, 20);
78     print_sum(3.14, 2.86);
79     print_sum(std::string("Hello "), std::string("World"));
80 }
81
82 // =====
83 // 4. REQUIRES CLAUSE
84 // =====
85 template<typename T>
86 T max_value(T a, T b) requires std::totally_ordered<T> {
87     return (a > b) ? a : b;
88 }
89
90 template<typename T>
91 requires std::integral<T>
92 T factorial(T n) {
93     if (n <= 1) return 1;
94     return n * factorial(n - 1);
95 }
96
97 void example_requires_clause() {
98     std::cout << "\n==== 4. REQUIRES CLAUSE ===" << std::endl;
99
100    std::cout << "max(10, 20) = " << max_value(10, 20) << std::endl;
101    std::cout << "max(3.14, 2.71) = " << max_value(3.14, 2.71) << std::endl;
102
103    std::cout << "factorial(5) = " << factorial(5) << std::endl;
```

```
104 }
105
106 // =====
107 // 5. CONCEPTS WITH MULTIPLE REQUIREMENTS
108 // =====
109 template<typename T>
110 concept Container = requires(T t) {
111     typename T::value_type;
112     typename T::iterator;
113     { t.begin() } -> std::same_as<typename T::iterator>;
114     { t.end() } -> std::same_as<typename T::iterator>;
115     { t.size() } -> std::convertible_to<std::size_t>;
116 };
117
118 template<Container C>
119 void print_container(const C& container, const std::string& name) {
120     std::cout << name << " (size " << container.size() << ")";
121     for (const auto& elem : container) {
122         std::cout << elem << " ";
123     }
124     std::cout << std::endl;
125 }
126
127 void example_container_concept() {
128     std::cout << "\n==== 5. CONCEPTS WITH MULTIPLE REQUIREMENTS ===" << std::endl;
129
130     std::vector<int> vec = {1, 2, 3, 4, 5};
131     print_container(vec, "Vector");
132
133     std::vector<std::string> words = {"Hello", "Modern", "C++"};
134     print_container(words, "Strings");
135 }
136
137 // =====
138 // 6. CONCEPT SUBSUMPTION
139 // =====
140 template<typename T>
141 concept Number = std::is_arithmetic_v<T>;
142
143 template<typename T>
144 concept Integer = Number<T> && std::is_integral_v<T>;
145
146 template<Number T>
147 void process(T value) {
148     std::cout << "Processing number: " << value << std::endl;
149 }
150
151 template<Integer T>
152 void process(T value) {
153     std::cout << "Processing integer: " << value << " (doubled: " << value * 2
154     << ")" << std::endl;
155 }
```

```
156 void example_concept_subsumption() {
157     std::cout << "\n==== 6. CONCEPT SUBSUMPTION ===" << std::endl;
158
159     process(42);           // Calls Integer version
160     process(3.14);        // Calls Number version
161 }
162
163 // =====
164 // 7. CUSTOM CONCEPT FOR RANGES
165 // =====
166 template<typename T>
167 concept Sortable = requires(T container) {
168     { container.begin() } -> std::input_or_output_iterator;
169     { container.end() } -> std::input_or_output_iterator;
170     requires std::sortable<decltype(container.begin())>;
171 };
172
173 template<Sortable C>
174 void sort_and_print(C& container) {
175     std::sort(container.begin(), container.end());
176     std::cout << "Sorted: ";
177     for (const auto& elem : container) {
178         std::cout << elem << " ";
179     }
180     std::cout << std::endl;
181 }
182
183 void example_sortable_concept() {
184     std::cout << "\n==== 7. CUSTOM CONCEPT FOR RANGES ===" << std::endl;
185
186     std::vector<int> numbers = {5, 2, 8, 1, 9};
187     sort_and_print(numbers);
188 }
189
190 // =====
191 // 8. CONCEPT WITH CLASS TEMPLATE
192 // =====
193 template<typename T>
194 concept Multipliable = requires(T a, T b) {
195     { a * b } -> std::convertible_to<T>;
196 };
197
198 template<Multipliable T>
199 class Point2D {
200 private:
201     T x, y;
202
203 public:
204     Point2D(T x_val, T y_val) : x(x_val), y(y_val) {}
205
206     T dot_product(const Point2D& other) const {
207         return x * other.x + y * other.y;
208     }
209 }
```

```

210     void print() const {
211         std::cout << "Point(" << x << ", " << y << ")" << std::endl;
212     }
213 };
214
215 void example_concept_class_template() {
216     std::cout << "\n==== 8. CONCEPT WITH CLASS TEMPLATE ===" << std::endl;
217
218     Point2D<int> p1(3, 4);
219     Point2D<int> p2(5, 6);
220
221     p1.print();
222     p2.print();
223     std::cout << "Dot product: " << p1.dot_product(p2) << std::endl;
224 }
225
226 // =====
227 // 9. CONCEPTS WITH LOGICAL OPERATORS
228 // =====
229 template<typename T>
230 concept SignedIntegral = std::integral<T> && std::is_signed_v<T>;
231
232 template<typename T>
233 concept UnsignedIntegral = std::integral<T> && std::is_unsigned_v<T>;
234
235 template<SignedIntegral T>
236 T abs_value(T value) {
237     return value < 0 ? -value : value;
238 }
239
240 template<UnsignedIntegral T>
241 T abs_value(T value) {
242     return value; // Already positive
243 }
244
245 void example_logical_concepts() {
246     std::cout << "\n==== 9. CONCEPTS WITH LOGICAL OPERATORS ===" << std::endl;
247
248     std::cout << "abs(-42) = " << abs_value(-42) << std::endl;
249     std::cout << "abs(42u) = " << abs_value(42u) << std::endl;
250 }
251
252 // =====
253 // 10. CONCEPT FOR CALLABLE TYPES
254 // =====
255 template<typename F, typename T>
256 concept Predicate = std::predicate<F, T>;
257
258 template<typename C, typename Pred>
259 requires Container<C> && Predicate<Pred, typename C::value_type>
260 void filter_and_print(const C& container, Pred pred, const std::string& name)
261 {
262     std::cout << name << ":" ;
263     for (const auto& elem : container) {

```

```
263     if (pred(elem)) {
264         std::cout << elem << " ";
265     }
266 }
267 std::cout << std::endl;
268 }
269
270 void example_predicate_concept() {
271     std::cout << "\n== 10. CONCEPT FOR CALLABLE TYPES ==" << std::endl;
272
273     std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
274
275     filter_and_print(numbers, [](int n) { return n % 2 == 0; }, "Even numbers");
276     filter_and_print(numbers, [](int n) { return n > 5; }, "Greater than 5");
277 }
278
279 // =====
280 // 11. CONCEPT FOR CUSTOM TYPES
281 // =====
282 template<typename T>
283 concept Drawable = requires(T obj) {
284     { obj.draw() } -> std::same_as<void>;
285     { obj.get_name() } -> std::convertible_to<std::string>;
286 };
287
288 class Circle {
289 public:
290     void draw() const {
291         std::cout << "Drawing a circle" << std::endl;
292     }
293
294     std::string get_name() const {
295         return "Circle";
296     }
297 };
298
299 class Rectangle {
300 public:
301     void draw() const {
302         std::cout << "Drawing a rectangle" << std::endl;
303     }
304
305     std::string get_name() const {
306         return "Rectangle";
307     }
308 };
309
310 template<Drawable T>
311 void render(const T& shape) {
312     std::cout << "Rendering: " << shape.get_name() << std::endl;
313     shape.draw();
314 }
```

```
316 void example_custom_type_concept() {
317     std::cout << "\n==== 11. CONCEPT FOR CUSTOM TYPES ===" << std::endl;
318
319     Circle circle;
320     Rectangle rect;
321
322     render(circle);
323     render(rect);
324 }
325
326 // =====
327 // 12. ADVANCED CONCEPT WITH NESTED REQUIREMENTS
328 // =====
329 template<typename T>
330 concept Arithmetic = requires(T a, T b) {
331     { a + b } -> std::convertible_to<T>;
332     { a - b } -> std::convertible_to<T>;
333     { a * b } -> std::convertible_to<T>;
334     { a / b } -> std::convertible_to<T>;
335 };
336
337 template<Arithmetic T>
338 class Calculator {
339 private:
340     T value;
341
342 public:
343     Calculator(T val) : value(val) {}
344
345     T add(T other) { return value + other; }
346     T subtract(T other) { return value - other; }
347     T multiply(T other) { return value * other; }
348     T divide(T other) { return value / other; }
349
350     void print() const {
351         std::cout << "Calculator value: " << value << std::endl;
352     }
353 };
354
355 void example_arithmetic_concept() {
356     std::cout << "\n==== 12. ADVANCED CONCEPT WITH NESTED REQUIREMENTS ===" <<
357             std::endl;
358
359     Calculator<int> calc_int(100);
360     calc_int.print();
361     std::cout << "100 + 50 = " << calc_int.add(50) << std::endl;
362     std::cout << "100 * 2 = " << calc_int.multiply(2) << std::endl;
363
364     Calculator<double> calc_double(10.5);
365     calc_double.print();
366     std::cout << "10.5 / 2.0 = " << calc_double.divide(2.0) << std::endl;
367 }
368
369 // =====
```

```
369 // MAIN FUNCTION
370 // =====
371 int main() {
372     std::cout << "\n====="
373     std::cout << "      C++20 CONCEPTS EXAMPLES" << std::endl;
374     std::cout << "====="
375     std::endl;
376     example_basic_concept();
377     example_standard_concepts();
378     example_compound_concepts();
379     example_requires_clause();
380     example_container_concept();
381     example_concept_subsumption();
382     example_sortable_concept();
383     example_concept_class_template();
384     example_logical_concepts();
385     example_predicate_concept();
386     example_custom_type_concept();
387     example_arithmetic_concept();
388
389     std::cout << "\n====="
390     std::cout << "      ALL EXAMPLES COMPLETED" << std::endl;
391     std::cout << "=====\\n"
392     std::endl;
393
394     return 0;
395 }
```

## 10 Source Code: ConfigLoaderAndChecker.cpp

File: src/ConfigLoaderAndChecker.cpp

Repository: [View on GitHub](#)

```
1 // ConfigLoaderAndChecker.cpp
2 // Demonstrates dynamic configuration loading and monitoring using std::
3 //     unordered_multimap
4 //
5 // FEATURES:
6 // 1. Load key-value pairs from JSON config file into unordered_multimap
7 // 2. Monitor config.json for changes (file modification time)
8 // 3. Automatically reload and update configuration on changes
9 // 4. Handle new keys, modified values, and deleted keys
10 // 5. Thread-safe configuration access
11 // 6. JSON validation and error handling
12 // 7. Real-time configuration updates without restart
13
14 #include <iostream>
15 #include <fstream>
16 #include <string>
17 #include <unordered_map>
18 #include <vector>
19 #include <set>
20 #include <chrono>
21 #include <thread>
22 #include <mutex>
23 #include <atomic>
24 #include <iomanip>
25 #include <filesystem>
26 #include <nlohmann/json.hpp>
27
28 namespace fs = std::filesystem;
29 using namespace std::chrono;
30 using namespace std::chrono_literals;
31
32 // SECTION 1: Using nlohmann/json library for JSON parsing
33 // =====
34
35 using json = nlohmann::json;
36
37 // =====
38 // SECTION 2: Thread-Safe Configuration Manager with unordered_multimap
39 // =====
```

```
41 class ConfigManager {
42 private:
43     std::unordered_multimap<std::string, std::string> config_data_;
44     std::unordered_multimap<std::string, std::string> previous_config_data_;
45     mutable std::mutex mutex_;
46     fs::file_time_type last_write_time_;
47     std::string config_file_path_;
48     std::atomic<int> version_{0};
49
50 public:
51     explicit ConfigManager(const std::string& config_file)
52         : config_file_path_(config_file) {
53
54         if (fs::exists(config_file_path_)) {
55             last_write_time_ = fs::last_write_time(config_file_path_);
56         }
57     }
58
59     // Load configuration from file
60     bool load() {
61         try {
62             std::ifstream file(config_file_path_);
63             if (!file.is_open()) {
64                 std::cerr << " Cannot open config file: " <<
65                 config_file_path_ << "\n";
66                 return false;
67             }
68
69             // Parse JSON using nlohmann/json
70             json parsed_json;
71             file >> parsed_json;
72             file.close();
73
74             // Update configuration
75             std::lock_guard<std::mutex> lock(mutex_);
76
77             // Save previous config for change detection
78             previous_config_data_ = config_data_;
79             config_data_.clear();
80
81             // Support both flat objects and nested structures
82             for (auto& [key, value] : parsed_json.items()) {
83                 // Convert value to string representation
84                 std::string value_str;
85                 if (value.is_string()) {
86                     value_str = value.get<std::string>();
87                 } else if (value.is_number()) {
88                     value_str = value.dump();
89                 } else if (value.is_boolean()) {
90                     value_str = value.get<bool>() ? "true" : "false";
91                 } else if (value.is_array()) {
92                     // For arrays, insert multiple entries (true multimap
93                     // usage)
94                     for (const auto& item : value) {
```

```
93         if (item.is_string()) {
94             config_data_.insert({key, item.get<std::string>()})
95             });
96         } else {
97             config_data_.insert({key, item.dump()});
98         }
99         continue; // Skip the insert below
100    } else {
101        value_str = value.dump();
102    }
103
104    config_data_.insert({key, value_str});
105}
106
107    version_++;
108
109    std::cout << " Loaded " << config_data_.size()
110                << " configuration entries (version " << version_ << ")"
111                << "\n";
112
113    return true;
114}
115
116    } catch (const std::exception& e) {
117        std::cerr << " Failed to load config: " << e.what() << "\n";
118        return false;
119    }
120
121    // Reload if file has changed
122    bool reload_if_changed() {
123        if (!fs::exists(config_file_path_)) {
124            std::cerr << " Config file not found: " << config_file_path_ <<
125            "\n";
126            return false;
127        }
128
129        auto current_write_time = fs::last_write_time(config_file_path_);
130
131        if (current_write_time != last_write_time_) {
132            std::cout << "\n Config file modified, reloading...\n";
133            last_write_time_ = current_write_time;
134            bool loaded = load();
135            if (loaded) {
136                print_changes();
137            }
138            return loaded;
139        }
140
141        return false;
142    }
143
144    // Get value by key (returns first match for multimap)
145    std::string get(const std::string& key, const std::string& default_value =
```

```
        "") const {
144     std::lock_guard<std::mutex> lock(mutex_);
145     auto it = config_data_.find(key);
146     if (it != config_data_.end()) {
147         return it->second;
148     }
149     return default_value;
150 }
151
152 // Get all values for a key (multimap can have multiple values per key)
153 std::vector<std::string> get_all(const std::string& key) const {
154     std::lock_guard<std::mutex> lock(mutex_);
155     std::vector<std::string> values;
156
157     auto range = config_data_.equal_range(key);
158     for (auto it = range.first; it != range.second; ++it) {
159         values.push_back(it->second);
160     }
161
162     return values;
163 }
164
165 // Check if key exists
166 bool has_key(const std::string& key) const {
167     std::lock_guard<std::mutex> lock(mutex_);
168     return config_data_.find(key) != config_data_.end();
169 }
170
171 // Get all keys
172 std::vector<std::string> get_all_keys() const {
173     std::lock_guard<std::mutex> lock(mutex_);
174     std::vector<std::string> keys;
175
176     for (const auto& [key, value] : config_data_) {
177         keys.push_back(key);
178     }
179
180     return keys;
181 }
182
183 // Get configuration count
184 size_t size() const {
185     std::lock_guard<std::mutex> lock(mutex_);
186     return config_data_.size();
187 }
188
189 // Display all configuration
190 void display() const {
191     std::lock_guard<std::mutex> lock(mutex_);
192
193     std::cout << "\n";
194     std::cout << "  Current Configuration (Version " << version_ << ")";
195     std::cout << std::string(20 - std::to_string(version_.load()).length()
196     , ' ') << "\n";
```

```
196     std::cout << "\n";
197
198     if (config_data_.empty()) {
199         std::cout << " (empty)\n";
200     } else {
201         for (const auto& [key, value] : config_data_) {
202             std::string line = " " + key + ": " + value;
203             line += std::string(58 - line.length(), ' ') + " ";
204             std::cout << line << "\n";
205         }
206     }
207
208     std::cout << "\n\n";
209 }
210
211 int get_version() const {
212     return version_.load();
213 }
214
215 std::string get_file_path() const {
216     return config_file_path_;
217 }
218
219 void print_changes() {
220     std::lock_guard<std::mutex> lock(mutex_);
221
222     std::cout << "\n Configuration changes detected:\n";
223
224     // Track unique keys
225     std::set<std::string> all_keys;
226     for (const auto& [key, _] : previous_config_data_) all_keys.insert(key);
227     for (const auto& [key, _) : config_data_) all_keys.insert(key);
228
229     for (const auto& key : all_keys) {
230         auto prev_range = previous_config_data_.equal_range(key);
231         auto curr_range = config_data_.equal_range(key);
232
233         std::vector<std::string> prev_values;
234         for (auto it = prev_range.first; it != prev_range.second; ++it) {
235             prev_values.push_back(it->second);
236         }
237
238         std::vector<std::string> curr_values;
239         for (auto it = curr_range.first; it != curr_range.second; ++it) {
240             curr_values.push_back(it->second);
241         }
242
243         if (prev_values.empty() && !curr_values.empty()) {
244             std::cout << " ADDED to unordered_multimap: '" << key << "'";
245             for (size_t i = 0; i < curr_values.size(); ++i) {
246                 std::cout << "\\" << curr_values[i] << "\\";
247             }
248         }
249     }
250 }
```

```

247         if (i < curr_values.size() - 1) std::cout << ", ";
248     }
249     if (curr_values.size() > 1) {
250         std::cout << "(" << curr_values.size() << " entries)";
251     }
252     std::cout << "\n";
253 } else if (!prev_values.empty() && curr_values.empty()) {
254     std::cout << "    REMOVED from unordered_multimap: '" << key
255     << "' (was: ";
256     for (size_t i = 0; i < prev_values.size(); ++i) {
257         std::cout << "\\" << prev_values[i] << "\\";
258         if (i < prev_values.size() - 1) std::cout << ", ";
259     }
260     std::cout << ")\n";
261 } else if (prev_values != curr_values) {
262     std::cout << "    MODIFIED in unordered_multimap: '" << key <<
263     "'\n";
264     std::cout << "    Old: ";
265     for (size_t i = 0; i < prev_values.size(); ++i) {
266         std::cout << "\\" << prev_values[i] << "\\";
267         if (i < prev_values.size() - 1) std::cout << ", ";
268     }
269     std::cout << "\n    New: ";
270     for (size_t i = 0; i < curr_values.size(); ++i) {
271         std::cout << "\\" << curr_values[i] << "\\";
272         if (i < curr_values.size() - 1) std::cout << ", ";
273     }
274     std::cout << "\n";
275 }
276 }
277 }
278 /**
279 =====
280 // SECTION 3: Configuration Monitor (Background Thread)
281 /**
282 =====
283
284 class ConfigMonitor {
285 private:
286     ConfigManager& config_manager_;
287     std::atomic<bool> running_{false};
288     std::thread monitor_thread_;
289     std::chrono::milliseconds check_interval_;
290
291 public:
292     ConfigMonitor(ConfigManager& manager, std::chrono::milliseconds interval =
293                 600000ms)
294         : config_manager_(manager), check_interval_(interval) {}

```

```
294     ~ConfigMonitor() {
295         stop();
296     }
297
298     void start() {
299         if (running_) {
300             std::cout << "    Monitor already running\n";
301             return;
302         }
303
304         running_ = true;
305         monitor_thread_ = std::thread(&ConfigMonitor::monitor_loop, this);
306
307         std::cout << "    Configuration monitor started (checking every "
308             << check_interval_.count() << "ms)\n";
309     }
310
311     void stop() {
312         if (!running_) return;
313
314         running_ = false;
315         if (monitor_thread_.joinable()) {
316             monitor_thread_.join();
317         }
318
319         std::cout << "    Configuration monitor stopped\n";
320     }
321
322 private:
323     void monitor_loop() {
324         while (running_) {
325             config_manager_.reload_if_changed();
326             std::this_thread::sleep_for(check_interval_);
327         }
328     }
329 };
330 /**
331 =====
332 // SECTION 4: Demonstration Application
333 /**
334 =====
335
336 void demonstrate_basic_loading() {
337     std::cout << "\n" << std::string(70, '=') << "\n";
338     std::cout << "==== Demo 1: Basic Configuration Loading ===\n";
339     std::cout << std::string(70, '=') << "\n\n";
340
341     ConfigManager config("../config.json");
342
343     std::cout << "Loading initial configuration...\n";
344     if (config.load()) {
```

```
344     config.display();
345
346     // Access individual values
347     std::cout << "Accessing specific values:\n";
348     std::cout << "  server_host: " << config.get("server_host", "not found"
349             " ") << "\n";
350     std::cout << "  server_port: " << config.get("server_port", "not found"
351             " ") << "\n";
352     std::cout << "  debug_mode: " << config.get("debug_mode", "not found"
353             " ") << "\n";
354     std::cout << "  max_connections: " << config.get("max_connections", "not
355             found") << "\n";
356
357     // Check if keys exist
358     std::cout << "\nKey existence checks:\n";
359     std::cout << "  has 'server_host': " << (config.has_key("server_host")
360             ? "Yes" : "No") << "\n";
361     std::cout << "  has 'nonexistent': " << (config.has_key("nonexistent")
362             ? "Yes" : "No") << "\n";
363
364     // Get all keys
365     std::cout << "\nAll keys: ";
366     auto keys = config.get_all_keys();
367     for (size_t i = 0; i < keys.size(); ++i) {
368         std::cout << keys[i];
369         if (i < keys.size() - 1) std::cout << ", ";
370     }
371     std::cout << "\n";
372 }
373
374 void demonstrate_live_monitoring() {
375     std::cout << "\n" << std::string(70, '=') << "\n";
376     std::cout << "==== Demo 2: Live Configuration Monitoring ===\n";
377     std::cout << std::string(70, '=') << "\n\n";
378
379     ConfigManager config("../config.json");
380     std::cout << "  Reading config from: " << fs::absolute("../config.json")
381             << "\n\n";
382     config.load();
383     config.display();
384
385     // Start monitoring in background - use 10 seconds for demo (production:
386     // 10 minutes)
387     ConfigMonitor monitor(config, 10000ms); // Check every 10 seconds for
388     // demo
389     monitor.start();
390
391     // Read demoTime from config (default to 120 seconds)
392     int demo_time = 120;
393     std::string demo_time_str = config.get("demoTime");
394     if (!demo_time_str.empty()) {
395         try {
396             demo_time = std::stoi(demo_time_str);
397         }
398     }
399 }
```

```
389         std::cout << " Using demoTime from config: " << demo_time << "
390             seconds\n";
391     } catch (...) {
392         std::cout << " Invalid demoTime in config, using default: 120
393             seconds\n";
394     }
395     std::cout << "\n Instructions:\n";
396     std::cout << " 1. Edit config.json file while this program is running\n";
397     std::cout << " 2. Add new key-value pairs: \"new_key\": \"new_value\"\n";
398     std::cout << " 3. Modify existing values: \"server_port\": \"9090\"\n";
399     std::cout << " 4. Add arrays: \"allowed_ips\": [\"192.168.1.1\",
400             \"10.0.0.1\"]\n";
401     std::cout << " 5. Change \"demoTime\": 60 to adjust monitoring duration
402             dynamically\n";
403     std::cout << " 6. Save the file and watch for automatic reload!\n";
404     std::cout << "\n App will run for " << demo_time << " seconds (checks
405             config every 10 seconds)...\n";
406     std::cout << " (In production: use 10 minutes / 600000ms)\n\n";
407
408     // Simulate application running and accessing config
409     int elapsed = 0;
410     while (elapsed < demo_time) {
411         std::this_thread::sleep_for(1s);
412         elapsed++;
413
414         // Periodically show current config version
415         if (elapsed % 10 == 0) {
416             std::cout << " " << elapsed << "s - Config version: " << config.
417                 get_version()
418                     << " | Entries: " << config.size();
419
420         // Check if demoTime has changed
421         std::string current_demo_time_str = config.get("demoTime");
422         if (!current_demo_time_str.empty()) {
423             try {
424                 int new_demo_time = std::stoi(current_demo_time_str);
425                 if (new_demo_time != demo_time) {
426                     std::cout << " | demoTime changed: " << demo_time << "
427                         →" << new_demo_time << "s";
428                     demo_time = new_demo_time;
429                     std::cout << "\n Runtime adjusted! Will now run until
430                         " << demo_time << " seconds\n";
431                 } else {
432                     std::cout << "\n";
433                 }
434             } catch (...) {
435                 std::cout << "\n";
436             }
437         } else {
438             std::cout << "\n";
439         }
440     }
441 }
```

```
435     }
436
437     monitor.stop();
438
439     std::cout << "\nFinal configuration state:\n";
440     config.display();
441 }
442
443 void demonstrate_error_handling() {
444     std::cout << "\n" << std::string(70, '=') << "\n";
445     std::cout << "==== Demo 3: Error Handling ===\n";
446     std::cout << std::string(70, '=') << "\n\n";
447
448     // Test with non-existent file
449     std::cout << "1. Loading non-existent file:\n";
450     ConfigManager config1("nonexistent.json");
451     config1.load();
452
453     // Test with invalid JSON
454     std::cout << "\n2. Creating and loading invalid JSON:\n";
455     std::ofstream bad_file("bad_config.json");
456     bad_file << "{ invalid json without quotes }";
457     bad_file.close();
458
459     ConfigManager config2("bad_config.json");
460     config2.load();
461
462     // Cleanup
463     fs::remove("bad_config.json");
464
465     std::cout << "\n Error handling demonstrated\n";
466     std::cout << " • File not found: gracefully handled\n";
467     std::cout << " • Invalid JSON: exception caught and reported\n";
468 }
469
470 void demonstrate_multimap_features() {
471     std::cout << "\n" << std::string(70, '=') << "\n";
472     std::cout << "==== Demo 4: unordered_multimap Features ===\n";
473     std::cout << std::string(70, '=') << "\n\n";
474
475     std::cout << "Concept: unordered_multimap allows multiple values per key\n";
476     std::cout << "Use case: Configuration with arrays or multiple values\n\n";
477
478     std::cout << "Example JSON (if extended):\n";
479     std::cout << R"({
480     "server": "localhost",
481     "allowed_ip": "192.168.1.1",
482     "allowed_ip": "192.168.1.2",
483     "allowed_ip": "10.0.0.1"
484 })" << "\n\n";
485
486     std::cout << "With unordered_multimap, you can store multiple 'allowed_ip' values\n";
```

```
487     std::cout << "and retrieve them all using get_all('allowed_ip')\n\n";
488
489     std::cout << "Current implementation: Simple key-value pairs\n";
490     std::cout << "If duplicate keys in JSON: last one wins (map behavior)\n";
491     std::cout << "To enable true multimap: parse JSON arrays into separate
492         entries\n\n";
493
494     ConfigManager config("../config.json");
495     config.load();
496
497     std::cout << "Accessing all values for a key:\n";
498     auto values = config.get_all("server_host");
499     std::cout << " 'server_host' has " << values.size() << " value(s):\n";
500     for (const auto& val : values) {
501         std::cout << "     - " << val << "\n";
502     }
503
504 void create_sample_config_if_not_exists() {
505     if (!fs::exists("config.json")) {
506         std::cout << " Creating sample config.json...\n";
507         std::ofstream config_file("config.json");
508         config_file << R"({
509             \"server_host\": \"localhost\",
510             \"server_port\": \"8080\",
511             \"debug_mode\": \"true\",
512             \"max_connections\": \"100\",
513             \"timeout_seconds\": \"30\",
514             \"database_url\": \"mongodb://localhost:27017\"
515         })";
516         config_file.close();
517         std::cout << " Sample config.json created\n\n";
518     }
519 }
520
521 // =====
522 // SECTION 5: Best Practices Guide
523 // =====
524
525 void show_best_practices() {
526     std::cout << "\n" << std::string(70, '=') << "\n";
527     std::cout << "==== Configuration Management Best Practices ===\n";
528     std::cout << std::string(70, '=') << "\n\n";
529
530     std::cout << " DESIGN PATTERNS:\n";
531     std::cout << " 1. Singleton pattern for global config access\n";
532     std::cout << " 2. Observer pattern for change notifications\n";
533     std::cout << " 3. Strategy pattern for different config sources\n";
534     std::cout << " 4. Builder pattern for complex configurations\n\n";
535 }
```

```
536     std::cout << " THREAD SAFETY:\n";
537     std::cout << " 1. Use std::mutex for protecting config access\n";
538     std::cout << " 2. std::shared_mutex for read-write locks (C++17)\n";
539     std::cout << " 3. std::atomic for version counters\n";
540     std::cout << " 4. Lock-free structures for high-performance\n\n";
541
542     std::cout << " FILE MONITORING:\n";
543     std::cout << " 1. Check file modification time (fs::last_write_time)\n";
544     std::cout << " 2. Use inotify (Linux) / ReadDirectoryChangesW (Windows)\n";
545     std::cout << " 3. Background thread with periodic checks\n";
546     std::cout << " 4. Debounce rapid changes (wait for settle)\n\n";
547
548     std::cout << " ERROR HANDLING:\n";
549     std::cout << " 1. Validate JSON syntax before parsing\n";
550     std::cout << " 2. Keep previous valid config on parse errors\n";
551     std::cout << " 3. Log all config changes and errors\n";
552     std::cout << " 4. Provide default values for missing keys\n\n";
553
554     std::cout << " PERFORMANCE:\n";
555     std::cout << " 1. Cache frequently accessed values\n";
556     std::cout << " 2. Use unordered_map for O(1) lookups\n";
557     std::cout << " 3. Minimize file I/O (check timestamp first)\n";
558     std::cout << " 4. Parse JSON incrementally for large files\n\n";
559
560     std::cout << " PRODUCTION CONSIDERATIONS:\n";
561     std::cout << " 1. Support multiple config sources (file, env, CLI)\n";
562     std::cout << " 2. Implement config versioning and rollback\n";
563     std::cout << " 3. Add config validation (schemas, constraints)\n";
564     std::cout << " 4. Enable hot-reload without service restart\n";
565     std::cout << " 5. Using nlohmann/json library (header-only, modern C++)\n";
566     std::cout << " 6. Implement config encryption for sensitive data\n";
567     std::cout << " 7. Set appropriate monitoring interval (10 min for\n";
568     std::cout << " production)\n\n";
569 }
570 //=====
571 // MAIN FUNCTION
572 //=====
573
574 int main() {
575     std::cout << "\n";
576     std::cout << "                                     \n";
577     std::cout << " Configuration Loader and Live Monitor\n";
578     std::cout << "                                     \n";
579     std::cout << " Using std::unordered_multimap + File Watching\n";
580     std::cout << "                                     \n";
581 }
```

```
581 // Create sample config if it doesn't exist
582 create_sample_config_if_not_exists();
583
584 // Run demonstrations
585 demonstrate_basic_loading();
586 demonstrate_multimap_features();
587 demonstrate_error_handling();
588
589 // Interactive monitoring demo
590 std::cout << "\n" << std::string(70, '=') << "\n";
591 std::cout << "Would you like to run the live monitoring demo? (y/n): ";
592 char choice;
593 std::cin >> choice;
594
595 if (choice == 'y' || choice == 'Y') {
596     demonstrate_live_monitoring();
597 } else {
598     std::cout << "\nSkipping live monitoring demo.\n";
599     std::cout << "To test manually:\n";
600     std::cout << " 1. Run this program\n";
601     std::cout << " 2. In another terminal: echo '{\"test\": \"value\"}' >
602         config.json\n";
603     std::cout << " 3. Watch the program detect changes!\n";
604 }
605
606 show_best_practices();
607
608 std::cout << "\n" << std::string(70, '=') << "\n";
609 std::cout << "All demonstrations completed!\n";
610 std::cout << "\nKEY FEATURES DEMONSTRATED:\n";
611 std::cout << "    std::unordered_multimap for config storage\n";
612 std::cout << "    JSON parsing and validation\n";
613 std::cout << "    File modification detection\n";
614 std::cout << "    Automatic config reloading\n";
615 std::cout << "    Thread-safe access with std::mutex\n";
616 std::cout << "    Background monitoring thread\n";
617 std::cout << "    Error handling and recovery\n";
618 std::cout << std::string(70, '=') << "\n\n";
619
620 return 0;
}
```

## 11 Source Code: Cpp11Examples.cpp

File: src/Cpp11Examples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <memory>
5 #include <thread>
6 #include <chrono>
7 #include <tuple>
8 #include <array>
9 #include <unordered_map>
10 #include <functional>
11 #include <algorithm>
12 #include <numeric>
13 #include <type_traits>
14
15 // =====
16 // C++11 COMPREHENSIVE EXAMPLES
17 // =====
18
19 // =====
20 // 1. AUTO TYPE DEDUCTION
21 // =====
22 void example_auto() {
23     std::cout << "\n== 1. AUTO TYPE DEDUCTION ==" << std::endl;
24
25     auto a = 42;           // int
26     auto b = 3.14;         // double
27     auto c = "hello";      // const char*
28     auto d = std::string("world"); // std::string
29
30     std::vector<int> vec = {1, 2, 3};
31     auto it = vec.begin(); // std::vector<int>::iterator
32
33     std::cout << "auto deduced types: int, double, const char*, std::string"
34         << std::endl;
35 }
36
37 // =====
38 // 2. NULLPTR
39 // =====
40 void foo(int) { std::cout << "Called foo(int)" << std::endl; }
41 void foo(int*) { std::cout << "Called foo(int*)" << std::endl; }
42
43 void example_nullptr() {
44     std::cout << "\n== 2. NULLPTR ==" << std::endl;
45
46     int* ptr = nullptr;
47     foo(nullptr); // Calls foo(int*)
48     std::cout << "nullptr is " << (ptr == nullptr ? "null" : "not null") <<
49         std::endl;
```

```
48 }
49
50 // =====
51 // 3. RANGE-BASED FOR LOOPS
52 // =====
53 void example_range_for() {
54     std::cout << "\n== 3. RANGE-BASED FOR LOOPS ==" << std::endl;
55
56     std::vector<int> vec = {1, 2, 3, 4, 5};
57     std::cout << "Elements: ";
58     for (const auto& elem : vec) {
59         std::cout << elem << " ";
60     }
61     std::cout << std::endl;
62 }
63
64 // =====
65 // 4. LAMBDA EXPRESSIONS
66 // =====
67 void example_lambdas() {
68     std::cout << "\n== 4. LAMBDA EXPRESSIONS ==" << std::endl;
69
70     int x = 10;
71     auto add_x = [x](int y) { return x + y; };
72     std::cout << "Lambda add_x(5) = " << add_x(5) << std::endl;
73
74     auto square = [] (int n) { return n * n; };
75     std::cout << "Lambda square(7) = " << square(7) << std::endl;
76
77     std::vector<int> vec = {1, 2, 3, 4, 5};
78     int sum = 0;
79     std::for_each(vec.begin(), vec.end(), [&sum](int n) { sum += n; });
80     std::cout << "Sum using lambda: " << sum << std::endl;
81 }
82
83 // =====
84 // 5. RVALUE REFERENCES AND MOVE SEMANTICS
85 // =====
86 class MoveableResource {
87 private:
88     int* data;
89     size_t size;
90 public:
91     MoveableResource(size_t s) : size(s) {
92         data = new int[size];
93         std::cout << "Allocated resource of size " << size << std::endl;
94     }
95
96     ~MoveableResource() {
97         delete[] data;
98     }
99
100    // Move constructor
101    MoveableResource(MoveableResource&& other) noexcept
```

```
102     : data(other.data), size(other.size) {
103     other.data = nullptr;
104     other.size = 0;
105     std::cout << "Moved resource" << std::endl;
106 }
107
108 // Move assignment
109 MoveableResource& operator=(MoveableResource&& other) noexcept {
110     if (this != &other) {
111         delete [] data;
112         data = other.data;
113         size = other.size;
114         other.data = nullptr;
115         other.size = 0;
116     }
117     return *this;
118 }
119 };
120
121 void example_move_semantics() {
122     std::cout << "\n== 5. MOVE SEMANTICS ==" << std::endl;
123
124     MoveableResource res1(100);
125     MoveableResource res2 = std::move(res1); // Move constructor
126 }
127
128 // =====
129 // 6. VARIADIC TEMPLATES
130 // =====
131 template <typename... Args>
132 void print_args(Args... args) {
133     (void)std::initializer_list<int>{
134         (std::cout << args << " ", 0)...
135     };
136     std::cout << std::endl;
137 }
138
139 template <typename... T>
140 struct arity {
141     static constexpr int value = sizeof...(T);
142 };
143
144 void example_variadic_templates() {
145     std::cout << "\n== 6. VARIADIC TEMPLATES ==" << std::endl;
146
147     print_args(1, "hello", 3.14, "world");
148     std::cout << "arity<int, double, char>::value = "
149             << arity<int, double, char>::value << std::endl;
150 }
151
152 // =====
153 // 7. STRONGLY-TYPED ENUMS
154 // =====
155 enum class Color { Red, Green, Blue };
```

```
156 enum class Traffic { Red, Yellow, Green }; // No conflict!
157
158 void example_strongly_typedEnums() {
159     std::cout << "\n== 7. STRONGLY-TYPED ENUMS ==" << std::endl;
160
161     Color c = Color::Red;
162     Traffic t = Traffic::Red;
163
164     // Color and Traffic::Red don't conflict
165     std::cout << "Strongly-typed enums prevent naming conflicts" << std::endl;
166 }
167
168 // =====
169 // 8. INITIALIZER LISTS
170 // =====
171 int sum_list(const std::initializer_list<int>& list) {
172     int total = 0;
173     for (auto elem : list) {
174         total += elem;
175     }
176     return total;
177 }
178
179 void example_initializer_lists() {
180     std::cout << "\n== 8. INITIALIZER LISTS ==" << std::endl;
181
182     std::vector<int> vec = {1, 2, 3, 4, 5};
183     std::cout << "Vector initialized with {1, 2, 3, 4, 5}" << std::endl;
184     std::cout << "sum_list({10, 20, 30}) = " << sum_list({10, 20, 30}) << std
185         ::endl;
186 }
187
188 // =====
189 // 9. DECLTYPE
190 // =====
191 template <typename X, typename Y>
192 auto multiply(X x, Y y) -> decltype(x * y) {
193     return x * y;
194 }
195
196 void example_decltype() {
197     std::cout << "\n== 9. DECLTYPE ==" << std::endl;
198
199     int a = 5;
200     decltype(a) b = 10; // b is int
201
202     std::cout << "multiply(3, 4) = " << multiply(3, 4) << std::endl;
203     std::cout << "multiply(2.5, 4) = " << multiply(2.5, 4) << std::endl;
204 }
205
206 // =====
207 // 10. TYPE ALIASES
208 // =====
209 template <typename T>
```

```
209 using Vec = std::vector<T>;
210
211 using String = std::string;
212
213 void example_type_aliases() {
214     std::cout << "\n==> 10. TYPE ALIASES ==>" << std::endl;
215
216     Vec<int> numbers = {1, 2, 3};
217     String text = "Hello";
218
219     std::cout << "Vec<int> and String aliases work" << std::endl;
220 }
221
222 // =====
223 // 11. CONSTEXPR
224 // =====
225 constexpr int factorial(int n) {
226     return n <= 1 ? 1 : n * factorial(n - 1);
227 }
228
229 constexpr int square(int n) {
230     return n * n;
231 }
232
233 void example_constexpr() {
234     std::cout << "\n==> 11. CONSTEXPR ==>" << std::endl;
235
236     constexpr int fact5 = factorial(5);
237     constexpr int sq7 = square(7);
238
239     std::cout << "constexpr factorial(5) = " << fact5 << std::endl;
240     std::cout << "constexpr square(7) = " << sq7 << std::endl;
241 }
242
243 // =====
244 // 12. STATIC ASSERTIONS
245 // =====
246 void example_static_assert() {
247     std::cout << "\n==> 12. STATIC ASSERTIONS ==>" << std::endl;
248
249     static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
250     static_assert(sizeof(char) == 1, "char must be 1 byte");
251
252     std::cout << "Static assertions passed" << std::endl;
253 }
254
255 // =====
256 // 13. DELEGATING CONSTRUCTORS
257 // =====
258 class Rectangle {
259 private:
260     int width, height;
261 public:
262     Rectangle() : Rectangle(0, 0) {} // Delegate to another constructor
```

```
263     Rectangle(int w) : Rectangle(w, w) {} // Delegate
264     Rectangle(int w, int h) : width(w), height(h) {
265         std::cout << "Rectangle(" << width << ", " << height << ")" << std::endl;
266     }
267 };
268
269 void example_delegating_constructors() {
270     std::cout << "\n== 13. DELEGATING CONSTRUCTORS ==" << std::endl;
271
272     Rectangle r1;
273     Rectangle r2(10);
274     Rectangle r3(5, 8);
275 }
276
277 // =====
278 // 14. DEFAULT AND DELETE
279 // =====
280 class NonCopyable {
281 public:
282     NonCopyable() = default;
283     NonCopyable(const NonCopyable&) = delete;
284     NonCopyable& operator=(const NonCopyable&) = delete;
285 };
286
287 void example_default_delete() {
288     std::cout << "\n== 14. DEFAULT AND DELETE ==" << std::endl;
289
290     NonCopyable obj1;
291     // NonCopyable obj2 = obj1; // Error: copy constructor deleted
292
293     std::cout << "Non-copyable class works" << std::endl;
294 }
295
296 // =====
297 // 15. OVERRIDE AND FINAL
298 // =====
299 class Base {
300 public:
301     virtual void foo() { std::cout << "Base::foo()" << std::endl; }
302     virtual ~Base() = default;
303 };
304
305 class Derived final : public Base {
306 public:
307     void foo() override { std::cout << "Derived::foo()" << std::endl; }
308 };
309
310 void example_override_final() {
311     std::cout << "\n== 15. OVERRIDE AND FINAL ==" << std::endl;
312
313     std::unique_ptr<Base> ptr(new Derived());
314     ptr->foo();
315 }
```

```
316
317 // =====
318 // 16. SMART POINTERS
319 // =====
320 void example_smart_pointers() {
321     std::cout << "\n==== 16. SMART POINTERS ===" << std::endl;
322
323     // unique_ptr
324     std::unique_ptr<int> uptr(new int(42));
325     std::cout << "*uptr = " << *uptr << std::endl;
326
327     // shared_ptr
328     std::shared_ptr<int> sptr1 = std::make_shared<int>(100);
329     std::shared_ptr<int> sptr2 = sptr1;
330     std::cout << "shared_ptr count: " << sptr1.use_count() << std::endl;
331
332     // weak_ptr
333     std::weak_ptr<int> wptr = sptr1;
334     std::cout << "weak_ptr valid: " << !wptr.expired() << std::endl;
335 }
336
337 // =====
338 // 17. TUPLES
339 // =====
340 void example_tuples() {
341     std::cout << "\n==== 17. TUPLES ===" << std::endl;
342
343     std::tuple<int, std::string, double> tup{42, "hello", 3.14};
344
345     std::cout << "Tuple element 0: " << std::get<0>(tup) << std::endl;
346     std::cout << "Tuple element 1: " << std::get<1>(tup) << std::endl;
347     std::cout << "Tuple element 2: " << std::get<2>(tup) << std::endl;
348
349     int a;
350     std::string b;
351     double c;
352     std::tie(a, b, c) = tup;
353     std::cout << "Using tie: " << a << ", " << b << ", " << c << std::endl;
354 }
355
356 // =====
357 // 18. STD::ARRAY
358 // =====
359 void example_array() {
360     std::cout << "\n==== 18. STD::ARRAY ===" << std::endl;
361
362     std::array<int, 5> arr = {1, 2, 3, 4, 5};
363
364     std::cout << "Array size: " << arr.size() << std::endl;
365     std::cout << "Array elements: ";
366     for (const auto& elem : arr) {
367         std::cout << elem << " ";
368     }
369     std::cout << std::endl;
```

```
370 }
371
372 // =====
373 // 19. UNORDERED CONTAINERS
374 // =====
375 void example_unordered_containers() {
376     std::cout << "\n== 19. UNORDERED CONTAINERS ==" << std::endl;
377
378     std::unordered_map<std::string, int> map;
379     map["one"] = 1;
380     map["two"] = 2;
381     map["three"] = 3;
382
383     std::cout << "Unordered map: ";
384     for (const auto& pair : map) {
385         std::cout << pair.first << "=" << pair.second << " ";
386     }
387     std::cout << std::endl;
388 }
389
390 // =====
391 // 20. STD::THREAD
392 // =====
393 void thread_function(int n) {
394     std::cout << "Thread " << n << " running" << std::endl;
395 }
396
397 void example_threads() {
398     std::cout << "\n== 20. STD::THREAD ==" << std::endl;
399
400     std::thread t1(thread_function, 1);
401     std::thread t2(thread_function, 2);
402
403     t1.join();
404     t2.join();
405
406     std::cout << "Threads completed" << std::endl;
407 }
408
409 // =====
410 // 21. CHRONO
411 // =====
412 void example_chrono() {
413     std::cout << "\n== 21. STD::CHRONO ==" << std::endl;
414
415     auto start = std::chrono::high_resolution_clock::now();
416
417     // Do some work
418     std::this_thread::sleep_for(std::chrono::milliseconds(100));
419
420     auto end = std::chrono::high_resolution_clock::now();
421     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end
422         - start);
```

```
423     std::cout << "Duration: " << duration.count() << " ms" << std::endl;
424 }
425
426 // =====
427 // 22. RAW STRING LITERALS
428 // =====
429 void example_raw_strings() {
430     std::cout << "\n==== 22. RAW STRING LITERALS ===" << std::endl;
431
432     std::string normal = "Line 1\\nLine 2\\nLine 3";
433     std::string raw = R"(Line 1
434 Line 2
435 Line 3)";
436
437     std::cout << "Raw string with newlines preserved" << std::endl;
438 }
439
440 // =====
441 // 23. ATTRIBUTES
442 // =====
443 [[noreturn]] void terminate_program() {
444     std::exit(0);
445 }
446
447 void example_attributes() {
448     std::cout << "\n==== 23. ATTRIBUTES ===" << std::endl;
449
450     std::cout << "[[noreturn]] attribute available" << std::endl;
451 }
452
453 // =====
454 // MAIN FUNCTION
455 // =====
456 int main() {
457     std::cout << "\n=====" << std::endl;
458     std::cout << "    C++11 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
459     std::cout << "=====" << std::endl;
460
461     example_auto();
462     example_nullptr();
463     example_range_for();
464     example_lambdas();
465     example_move_semantics();
466     example_variadic_templates();
467     example_strongly_typedEnums();
468     example_initializer_lists();
469     example_decltype();
470     example_type_aliases();
471     example_constexpr();
472     example_static_assert();
473     example_delegating_constructors();
474     example_default_delete();
```

```
475     example_override_final();
476     example_smart_pointers();
477     example_tuples();
478     example_array();
479     example_unordered_containers();
480     example_threads();
481     example_chrono();
482     example_raw_strings();
483     example_attributes();
484
485     std::cout << "\n======" << std::
486         endl;
486     std::cout << "      ALL C++11 EXAMPLES COMPLETED" << std::endl;
487     std::cout << "=====\\n" << std::
488         endl;
489
490     return 0;
490 }
```

## 12 Source Code: Cpp14Examples.cpp

File: src/Cpp14Examples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <memory>
5 #include <algorithm>
6 #include <utility>
7
8 // =====
9 // C++14 LANGUAGE FEATURES
10 // =====
11
12 // =====
13 // 1. BINARY LITERALS
14 // =====
15 void example_binary_literals() {
16     std::cout << "\n==== 1. BINARY LITERALS ===" << std::endl;
17
18     int a = 0b110;           // Binary literal
19     int b = 0b1111'1111;    // Binary with digit separator
20     int c = 0b1010'0101'1100; // Larger binary number
21
22     std::cout << "0b110 = " << a << std::endl;
23     std::cout << "0b1111'1111 = " << b << std::endl;
24     std::cout << "0b1010'0101'1100 = " << c << std::endl;
25 }
26
27 // =====
28 // 2. GENERIC LAMBDA EXPRESSIONS
29 // =====
30 void example_generic_lambdas() {
31     std::cout << "\n==== 2. GENERIC LAMBDA EXPRESSIONS ===" << std::endl;
32
33     auto identity = [] (auto x) { return x; };
34
35     int three = identity(3);
36     std::string foo = identity(std::string("foo"));
37     double pi = identity(3.14);
38
39     std::cout << "Identity(3) = " << three << std::endl;
40     std::cout << "Identity(\"foo\") = " << foo << std::endl;
41     std::cout << "Identity(3.14) = " << pi << std::endl;
42
43     // Generic lambda with multiple parameters
44     auto add = [] (auto a, auto b) { return a + b; };
45     std::cout << "add(10, 20) = " << add(10, 20) << std::endl;
46     std::cout << "add(1.5, 2.5) = " << add(1.5, 2.5) << std::endl;
47 }
48
49 // =====
```

```
50 // 3. LAMBDA CAPTURE INITIALIZERS
51 // =====
52 void example_lambda_capture_initializers() {
53     std::cout << "\n==== 3. LAMBDA CAPTURE INITIALIZERS ===" << std::endl;
54
55     // Initialize capture with expression
56     int factory_value = 2;
57     auto f = [x = factory_value * 10] { return x; };
58     std::cout << "Lambda with initialized capture: " << f() << std::endl;
59
60     // Generator with mutable capture
61     auto generator = [x = 0]() mutable {
62         return x++;
63     };
64     std::cout << "Generator call 1: " << generator() << std::endl;
65     std::cout << "Generator call 2: " << generator() << std::endl;
66     std::cout << "Generator call 3: " << generator() << std::endl;
67
68     // Move-only types in lambda
69     auto ptr = std::make_unique<int>(42);
70     auto task = [p = std::move(ptr)] {
71         std::cout << "Value in lambda: " << *p << std::endl;
72     };
73     task();
74     std::cout << "Original ptr is " << (ptr ? "valid" : "null") << std::endl;
75 }
76
77 // =====
78 // 4. RETURN TYPE DEDUCTION
79 // =====
80 auto multiply(int a, int b) {
81     return a * b; // Return type deduced as int
82 }
83
84 template <typename T>
85 auto add_values(T a, T b) {
86     return a + b; // Return type deduced from expression
87 }
88
89 void example_return_type_deduction() {
90     std::cout << "\n==== 4. RETURN TYPE DEDUCTION ===" << std::endl;
91
92     std::cout << "multiply(5, 6) = " << multiply(5, 6) << std::endl;
93     std::cout << "add_values(10, 20) = " << add_values(10, 20) << std::endl;
94     std::cout << "add_values(1.5, 2.5) = " << add_values(1.5, 2.5) << std::endl;
95
96     // Lambda with auto return type
97     auto square = [] (auto x) { return x * x; };
98     std::cout << "square(7) = " << square(7) << std::endl;
99 }
100
101 // =====
102 // 5. DECLTYPE(AUTO)
```

```
103 // =====
104 template <typename Container>
105 decltype(auto) get_first(Container& c) {
106     return c[0]; // Returns reference if c[0] is a reference
107 }
108
109 void example decltype auto() {
110     std::cout << "\n== 5. DECLTYPE(AUTO) ==" << std::endl;
111
112     const int x = 0;
113     auto x1 = x; // int
114     decltype(auto) x2 = x; // const int
115
116     std::cout << "auto strips const, decltype(auto) preserves it" << std::endl
117         ;
118
119     std::vector<int> vec = {1, 2, 3};
120     decltype(auto) first = get_first(vec);
121     first = 100; // Modifies vec[0]
122     std::cout << "Modified first element: " << vec[0] << std::endl;
123 }
124 // =====
125 // 6. RELAXED CONSTEXPR FUNCTIONS
126 // =====
127 constexpr int factorial(int n) {
128     // C++14 allows multiple statements in constexpr
129     int result = 1;
130     for (int i = 2; i <= n; ++i) {
131         result *= i;
132     }
133     return result;
134 }
135
136 constexpr int fibonacci(int n) {
137     if (n <= 1) return n;
138     int a = 0, b = 1;
139     for (int i = 2; i <= n; ++i) {
140         int temp = a + b;
141         a = b;
142         b = temp;
143     }
144     return b;
145 }
146
147 void example relaxed constexpr() {
148     std::cout << "\n== 6. RELAXED CONSTEXPR FUNCTIONS ==" << std::endl;
149
150     constexpr int fact5 = factorial(5);
151     constexpr int fib10 = fibonacci(10);
152
153     std::cout << "factorial(5) = " << fact5 << std::endl;
154     std::cout << "fibonacci(10) = " << fib10 << std::endl;
155 }
```

```
156
157 // =====
158 // 7. VARIABLE TEMPLATES
159 // =====
160 template <typename T>
161 constexpr T pi = T(3.1415926535897932385);
162
163 template <typename T>
164 constexpr T e = T(2.7182818284590452354);
165
166 void example_variable_templates() {
167     std::cout << "\n== 7. VARIABLE TEMPLATES ==" << std::endl;
168
169     std::cout << "pi<float> = " << pi<float> << std::endl;
170     std::cout << "pi<double> = " << pi<double> << std::endl;
171     std::cout << "e<float> = " << e<float> << std::endl;
172     std::cout << "e<double> = " << e<double> << std::endl;
173 }
174
175 // =====
176 // 8. [[DEPRECATED]] ATTRIBUTE
177 // =====
178 [[deprecated("Use new_function() instead")]]
179 void old_function() {
180     std::cout << "This is deprecated!" << std::endl;
181 }
182
183 void new_function() {
184     std::cout << "This is the new function!" << std::endl;
185 }
186
187 void example_deprecated_attribute() {
188     std::cout << "\n== 8. [[DEPRECATED]] ATTRIBUTE ==" << std::endl;
189
190     // old_function(); // Would generate compiler warning
191     new_function();
192     std::cout << "Using new function instead of deprecated one" << std::endl;
193 }
194
195 // =====
196 // C++14 LIBRARY FEATURES
197 // =====
198
199 // =====
200 // 9. USER-DEFINED LITERALS FOR STANDARD LIBRARY
201 // =====
202 void example_user_defined_literals() {
203     std::cout << "\n== 9. USER-DEFINED LITERALS ==" << std::endl;
204
205     using namespace std::string_literals;
206
207     auto str = "Hello"s; // std::string
208
209     std::cout << "String literal type: std::string" << std::endl;
```

```
210     std::cout << "User-defined string literals work in C++14" << std::endl;
211 }
212
213 // =====
214 // 10. STD::MAKE_UNIQUE
215 // =====
216 void example_make_unique() {
217     std::cout << "\n== 10. STD::MAKE_UNIQUE ==" << std::endl;
218
219     auto ptr1 = std::make_unique<int>(42);
220     auto ptr2 = std::make_unique<std::string>("Hello");
221     auto ptr3 = std::make_unique<std::vector<int>>(5, 10);
222
223     std::cout << "*ptr1 = " << *ptr1 << std::endl;
224     std::cout << "*ptr2 = " << *ptr2 << std::endl;
225     std::cout << "ptr3 size = " << ptr3->size() << std::endl;
226
227     // Array version
228     auto arr = std::make_unique<int []>(5);
229     for (int i = 0; i < 5; ++i) {
230         arr[i] = i * 10;
231     }
232     std::cout << "Array elements: ";
233     for (int i = 0; i < 5; ++i) {
234         std::cout << arr[i] << " ";
235     }
236     std::cout << std::endl;
237 }
238
239 // =====
240 // 11. COMPILE-TIME INTEGER SEQUENCES
241 // =====
242 template <typename T, T... Ints>
243 void print_sequence(std::integer_sequence<T, Ints...>) {
244     std::cout << "Sequence: ";
245     int dummy[] = {0, (std::cout << Ints << " ", 0)...};
246     (void)dummy;
247     std::cout << std::endl;
248 }
249
250 void example_integer_sequences() {
251     std::cout << "\n== 11. COMPILE-TIME INTEGER SEQUENCES ==" << std::endl;
252
253     using Seq = std::integer_sequence<int, 0, 1, 2, 3, 4>;
254     using IdxSeq = std::make_index_sequence<5>;
255
256     std::cout << "Integer sequence example (needs C++17 fold for printing)" <<
257             std::endl;
258     std::cout << "std::make_index_sequence<5> creates indices 0,1,2,3,4" <<
259             std::endl;
260 }
261
262 // =====
263 // 12. DIGIT SEPARATORS
```

```
262 // =====
263 void example_digit_separators() {
264     std::cout << "\n==== 12. DIGIT SEPARATORS ===" << std::endl;
265
266     int decimal = 1'000'000;
267     int hex = 0xDEAD'BEEF;
268     int binary = 0b1010'1010'1010;
269     double floating = 3.141'592'653'589;
270
271     std::cout << "1'000'000 = " << decimal << std::endl;
272     std::cout << "0xDEAD'BEEF = " << hex << std::endl;
273     std::cout << "0b1010'1010'1010 = " << binary << std::endl;
274     std::cout << "3.141'592'653'589 = " << floating << std::endl;
275 }
276
277 // =====
278 // MAIN FUNCTION
279 // =====
280 int main() {
281     std::cout << "\n=====" << std::endl;
282     std::cout << "      C++14 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
283     std::cout << "=====" << std::endl;
284
285     // Language Features
286     example_binary_literals();
287     example_generic_lambdas();
288     example_lambda_capture_initializers();
289     example_return_type_deduction();
290     example_decltype_auto();
291     example_relaxed_constexpr();
292     example_variable_templates();
293     example_DEPRECATED_attribute();
294
295     // Library Features
296     example_user_defined_literals();
297     example_make_unique();
298     example_integer_sequences();
299     example_digit_separators();
300
301     std::cout << "\n=====" << std::endl;
302     std::cout << "      ALL C++14 EXAMPLES COMPLETED" << std::endl;
303     std::cout << "=====\n" << std::endl;
304
305     return 0;
306 }
```

## 13 Source Code: Cpp17Concurrency.cpp

File: src/Cpp17Concurrency.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <future>
3 #include <thread>
4 #include <chrono>
5 #include <vector>
6 #include <numeric>
7 #include <algorithm>
8 #include <string>
9 #include <random>
10 #include <mutex>
11 #include <queue>
12 #include <condition_variable>
13 #include <atomic>
14 #include <type_traits>
15 #include <variant>
16 #include <optional>
17 #include <any>
18 #include <tuple>
19 #include <filesystem>
20 #include <sstream>
21 #include <fstream>
22 #include <map>
23 #include <array>
24 #include <functional>
25
26 using namespace std;
27 using namespace std::chrono;
28 using namespace std::chrono_literals;
29
30 // =====
31 // 1. BASIC FUTURE/PROMISE
32 // =====
33 void example_basic_future_promise() {
34     cout << "\n== 1. BASIC FUTURE/PROMISE ==" << endl;
35
36     promise<int> promise_obj;
37     future<int> future_obj = promise_obj.get_future();
38
39     thread t([&promise_obj]() {
40         cout << " Thread: Doing some computation..." << endl;
41         this_thread::sleep_for(1s);
42
43         int result = 42;
44         cout << " Thread: Setting promise value: " << result << endl;
45         promise_obj.set_value(result);
46     });
47
48     cout << "Main: Waiting for result..." << endl;
49     int result = future_obj.get();
```

```
50     cout << "Main: Got result: " << result << endl;
51
52     t.join();
53 }
54
55 // =====
56 // 2. FUTURE WITH EXCEPTION
57 // =====
58 void example_future_exception() {
59     cout << "\n== 2. FUTURE WITH EXCEPTION ==" << endl;
60
61     promise<int> promise_obj;
62     future<int> future_obj = promise_obj.get_future();
63
64     thread t([&promise_obj]() {
65         try {
66             cout << "  Thread: Simulating error..." << endl;
67             throw runtime_error("Something went wrong!");
68         }
69         catch (...) {
70             promise_obj.set_exception(current_exception());
71         }
72     });
73
74     try {
75         cout << "Main: Waiting for result..." << endl;
76         int result = future_obj.get();
77         cout << "Main: Got result: " << result << endl;
78     }
79     catch (const exception& e) {
80         cout << "Main: Caught exception: " << e.what() << endl;
81     }
82
83     t.join();
84 }
85
86 // =====
87 // 3. ASYNC (HIGH-LEVEL FUTURE)
88 // =====
89 void example_async() {
90     cout << "\n== 3. ASYNC (HIGH-LEVEL FUTURE) ==" << endl;
91
92     auto future1 = async(launch::async, []() {
93         cout << "  Async task 1 running..." << endl;
94         this_thread::sleep_for(500ms);
95         return 42;
96     });
97
98     auto future2 = async(launch::async, []() {
99         cout << "  Async task 2 running..." << endl;
100        this_thread::sleep_for(300ms);
101        return 100;
102    });
103 }
```

```
104     auto future3 = async(launch::deferred, []() {
105         cout << "  Deferred task running..." << endl;
106         return 200;
107     });
108
109     cout << "Main: Doing other work..." << endl;
110     this_thread::sleep_for(100ms);
111
112     cout << "Result 1: " << future1.get() << endl;
113     cout << "Result 2: " << future2.get() << endl;
114     cout << "Result 3 (deferred): " << future3.get() << endl;
115 }
116
117 // =====
118 // 4. SHARED FUTURE (MULTIPLE WAITERS)
119 // =====
120 void example_shared_future() {
121     cout << "\n==== 4. SHARED FUTURE (MULTIPLE WAITERS) ===" << endl;
122
123     promise<int> promise_obj;
124     future<int> future_obj = promise_obj.get_future();
125     shared_future<int> shared_future_obj = future_obj.share();
126
127     vector<thread> threads;
128     for (int i = 0; i < 3; i++) {
129         threads.emplace_back([i, shared_future_obj]() {
130             cout << "  Thread " << i << ": Waiting for result..." << endl;
131             int result = shared_future_obj.get();
132             cout << "  Thread " << i << ": Got result: " << result << endl;
133         });
134     }
135
136     thread producer([&promise_obj]() {
137         this_thread::sleep_for(1s);
138         cout << "  Producer: Setting value to 999" << endl;
139         promise_obj.set_value(999);
140     });
141
142     for (auto& t : threads) t.join();
143     producer.join();
144 }
145
146 // =====
147 // 5. FUTURE WITH TIMEOUT
148 // =====
149 void example_future_timeout() {
150     cout << "\n==== 5. FUTURE WITH TIMEOUT ===" << endl;
151
152     promise<int> promise_obj;
153     future<int> future_obj = promise_obj.get_future();
154
155     thread slow_worker([&promise_obj]() {
156         cout << "  Slow worker: Starting (3 seconds)" << endl;
157         this_thread::sleep_for(3s);
158     });
159
160     cout << "  Future: Waiting for result..." << endl;
161     int result = future_obj.get();
162     cout << "  Future: Got result: " << result << endl;
163
164     cout << "  Slow worker: Joining..." << endl;
165     slow_worker.join();
166     cout << "  Slow worker: Done" << endl;
167
168     cout << "  Future: Final result: " << result << endl;
169 }
```

```
158     promise_obj.set_value(42);
159     cout << " Slow worker: Finished" << endl;
160 };
161
162 cout << "Main: Waiting with timeout (2 seconds)..." << endl;
163
164 future_status status = future_obj.wait_for(2s);
165
166 if (status == future_status::ready) {
167     cout << "Main: Got result: " << future_obj.get() << endl;
168 }
169 else if (status == future_status::timeout) {
170     cout << "Main: Timeout! Task not ready yet" << endl;
171 }
172
173 this_thread::sleep_for(2s);
174 if (slow_worker.joinable()) {
175     if (future_obj.wait_for(0s) == future_status::ready) {
176         cout << "Main: Finally got result: " << future_obj.get() << endl;
177     }
178     slow_worker.join();
179 }
180
181 // =====
182 // 6. PACKAGED TASK
183 // =====
184
185 void example_packaged_task() {
186     cout << "\n== 6. PACKAGED TASK ==" << endl;
187
188     packaged_task<int(int, int)> task([](int a, int b) {
189         cout << " Task executing with " << a << " and " << b << endl;
190         this_thread::sleep_for(500ms);
191         return a + b;
192     });
193
194     future<int> result = task.get_future();
195
196     thread t(move(task), 10, 20);
197
198     cout << "Main: Waiting for packaged task result..." << endl;
199     cout << "Result: " << result.get() << endl;
200
201     t.join();
202 }
203
204 // =====
205 // 7. ASYNC PARALLEL COMPUTATION
206 // =====
207
208 void example_parallel_computation() {
209     cout << "\n== 7. ASYNC PARALLEL COMPUTATION ==" << endl;
210
211     auto compute_chunk = [](int start, int end) {
212         long long sum = 0;
```

```

212     for (int i = start; i <= end; i++) {
213         sum += i;
214     }
215     cout << "  Chunk " << start << "-" << end << ":" << sum << endl;
216     return sum;
217 };
218
219 const int N = 1000000;
220 const int num_threads = 4;
221 const int chunk_size = N / num_threads;
222
223 vector<future<long long>> futures;
224
225 auto start_time = high_resolution_clock::now();
226
227 for (int i = 0; i < num_threads; i++) {
228     int chunk_start = i * chunk_size + 1;
229     int chunk_end = (i == num_threads - 1) ? N : (i + 1) * chunk_size;
230
231     futures.push_back(async(launch::async, compute_chunk,
232                             chunk_start, chunk_end));
233 }
234
235 long long total = 0;
236 for (auto& f : futures) {
237     total += f.get();
238 }
239
240 auto end_time = high_resolution_clock::now();
241 auto duration = duration_cast<milliseconds>(end_time - start_time);
242
243 cout << "Total sum 1.." << N << ":" << total << endl;
244 cout << "Parallel time: " << duration.count() << "ms" << endl;
245
246 // Sequential comparison
247 start_time = high_resolution_clock::now();
248 long long seq_total = 0;
249 for (int i = 1; i <= N; i++) {
250     seq_total += i;
251 }
252 end_time = high_resolution_clock::now();
253 duration = duration_cast<milliseconds>(end_time - start_time);
254
255 cout << "Sequential sum: " << seq_total << endl;
256 cout << "Sequential time: " << duration.count() << "ms" << endl;
257 }
258
259 // =====
260 // 8. FUTURE CONTINUATIONS (MANUAL IMPLEMENTATION)
261 // =====
262 void example_future_continuations() {
263     cout << "\n== 8. FUTURE CONTINUATIONS (MANUAL) ==" << endl;
264
265     // Using nested async for continuations

```

```
266     auto future1 = async(launch::async, []() {
267         cout << "  Step 1: Computing 42..." << endl;
268         this_thread::sleep_for(300ms);
269         return 42;
270     });
271
272     // Continuation 1
273     auto future2 = async(launch::async, [future1 = move(future1)]() mutable {
274         int x = future1.get(); // Wait for first result
275         cout << "  Step 2: Doubling " << x << "..." << endl;
276         this_thread::sleep_for(300ms);
277         return x * 2;
278     });
279
280     // Continuation 2
281     auto future3 = async(launch::async, [future2 = move(future2)]() mutable {
282         int x = future2.get(); // Wait for second result
283         cout << "  Step 3: Adding 100 to " << x << "..." << endl;
284         this_thread::sleep_for(300ms);
285         return x + 100;
286     });
287
288     cout << "Main: Waiting for pipeline..." << endl;
289     int result = future3.get();
290     cout << "Final result: " << result << endl;
291 }
292
293 // =====
294 // 9. WHEN_ALL IMPLEMENTATION (SIMPLIFIED)
295 // =====
296 void example_when_all() {
297     cout << "\n== 9. WHEN_ALL (WAIT FOR ALL FUTURES) ==" << endl;
298
299     auto task1 = async(launch::async, []() {
300         this_thread::sleep_for(800ms);
301         cout << "  Task 1 complete" << endl;
302         return string("Result 1");
303     });
304
305     auto task2 = async(launch::async, []() {
306         this_thread::sleep_for(500ms);
307         cout << "  Task 2 complete" << endl;
308         return 42;
309     });
310
311     auto task3 = async(launch::async, []() {
312         this_thread::sleep_for(300ms);
313         cout << "  Task 3 complete" << endl;
314         return 3.14;
315     });
316
317     cout << "Main: Waiting for all tasks..." << endl;
318
319     // Manual when_all - just get all results
```

```
320     string result1 = task1.get();
321     int result2 = task2.get();
322     double result3 = task3.get();
323
324     cout << "All tasks complete!" << endl;
325     cout << "Results: " << result1 << ", " << result2 << ", " << result3 <<
326         endl;
327 }
328 // =====
329 // 10. WHEN_ANY IMPLEMENTATION (SIMPLIFIED)
330 // =====
331 void example_when_any() {
332     cout << "\n== 10. WHEN_ANY (FIRST COMPLETED) ==" << endl;
333
334     vector<future<string>> tasks;
335
336     tasks.push_back(async(launch::async, []() {
337         this_thread::sleep_for(1200ms);
338         return string("Task 1 (slow)");
339     }));
340
341     tasks.push_back(async(launch::async, []() {
342         this_thread::sleep_for(700ms);
343         return string("Task 2 (medium)");
344     }));
345
346     tasks.push_back(async(launch::async, []() {
347         this_thread::sleep_for(300ms);
348         return string("Task 3 (fast)");
349     }));
350
351     cout << "Main: Waiting for first task to complete..." << endl;
352
353     // Simple when_any using wait_for
354     while (true) {
355         for (size_t i = 0; i < tasks.size(); i++) {
356             if (tasks[i].wait_for(chrono::milliseconds(10)) == future_status::
357                 ready) {
358                 cout << "Task " << i << " completed first!" << endl;
359                 cout << "Result: " << tasks[i].get() << endl;
360
361                 // Get remaining results
362                 for (size_t j = 0; j < tasks.size(); j++) {
363                     if (j != i) {
364                         cout << "Task " << j << ":" << tasks[j].get() << endl
365                         ;
366                     }
367                 }
368             }
369             this_thread::sleep_for(100ms);
370         }
371     }
372 }
```

```
371 }
372
373 // =====
374 // 11. THREAD POOL WITH FUTURES
375 // =====
376 class ThreadPool {
377 private:
378     vector<thread> workers;
379     queue<function<void()>> tasks;
380     mutex queue_mutex;
381     condition_variable condition;
382     atomic<bool> stop;
383
384 public:
385     ThreadPool(size_t num_threads) : stop(false) {
386         for (size_t i = 0; i < num_threads; ++i) {
387             workers.emplace_back([this] {
388                 while (true) {
389                     function<void()> task;
390                     {
391                         unique_lock<mutex> lock(queue_mutex);
392                         condition.wait(lock, [this] {
393                             return stop || !tasks.empty();
394                         });
395
396                         if (stop && tasks.empty()) return;
397
398                         task = move(tasks.front());
399                         tasks.pop();
400                     }
401                     task();
402                 }
403             });
404         }
405     }
406
407     template<class F, class... Args>
408     auto enqueue(F&& f, Args&&... args) -> future<invoke_result_t<F, Args...>>
409     {
410         using return_type = invoke_result_t<F, Args...>;
411
412         auto task = make_shared<packaged_task<return_type()>>(
413             bind(forward<F>(f), forward<Args>(args)...));
414
415         future<return_type> result = task->get_future();
416         {
417             lock_guard<mutex> lock(queue_mutex);
418             if (stop) {
419                 throw runtime_error("enqueue on stopped ThreadPool");
420             }
421             tasks.emplace([task]() { (*task)(); });
422         }
423         condition.notify_one();
424     }
425 }
```

```
424     return result;
425 }
426
427 ~ThreadPool() {
428 {
429     lock_guard<mutex> lock(queue_mutex);
430     stop = true;
431 }
432 condition.notify_all();
433 for (thread& worker : workers) {
434     if (worker.joinable()) worker.join();
435 }
436 }
437 };
438
439 void example_thread_pool() {
440 cout << "\n== 11. THREAD POOL WITH FUTURES ==" << endl;
441
442 ThreadPool pool(4);
443 vector<future<int>> results;
444
445 cout << "Submitting 8 tasks to thread pool..." << endl;
446
447 for (int i = 0; i < 8; i++) {
448     results.emplace_back(pool.enqueue([i]() -> int {
449         cout << "  Task " << i << " executing" << endl;
450         this_thread::sleep_for(chrono::milliseconds(100 * (i % 3 + 1)));
451         return i * i;
452     }));
453 }
454
455 cout << "Collecting results..." << endl;
456 for (size_t i = 0; i < results.size(); i++) {
457     cout << "Result " << i << ":" << results[i].get() << endl;
458 }
459
460 cout << "All tasks completed!" << endl;
461 }
462
463 // =====
464 // 12. FUTURE WITH CANCELLATION
465 // =====
466 class CancellableFuture {
467 private:
468     promise<int> promise_obj;
469     future<int> future_obj;
470     atomic<bool> cancelled;
471
472 public:
473     CancellableFuture() : cancelled(false) {
474         future_obj = promise_obj.get_future();
475     }
476
477     void cancel() {
```

```
478     cancelled = true;
479     promise_obj.set_exception(make_exception_ptr(runtime_error("Cancelled"
480                               )));
480 }
481
482     bool is_cancelled() const { return cancelled; }
483
484     future<int>& get_future() { return future_obj; }
485
486     void execute_async(function<int()> task) {
487         thread([this, task](){
488             if (cancelled) return;
489
490             try {
491                 int result = task();
492                 if (!cancelled) {
493                     promise_obj.set_value(result);
494                 }
495             }
496             catch (...) {
497                 if (!cancelled) {
498                     promise_obj.set_exception(current_exception());
499                 }
500             }
501         }).detach();
502     }
503 };
504
505 void example_cancelable_future() {
506     cout << "\n== 12. CANCELABLE FUTURE ==" << endl;
507
508     CancellableFuture cf;
509
510     cf.execute_async([](){
511         cout << "  Long task starting (5 seconds)..." << endl;
512         for (int i = 0; i < 5; i++) {
513             this_thread::sleep_for(1s);
514             cout << "  Task progress: " << (i + 1) << "/5" << endl;
515         }
516         return 42;
517     });
518
519     this_thread::sleep_for(2500ms);
520     cout << "Main: Cancelling task after 2.5 seconds..." << endl;
521     cf.cancel();
522
523     try {
524         cout << "Main: Waiting for result..." << endl;
525         int result = cf.get_future().get();
526         cout << "Main: Got result: " << result << endl;
527     }
528     catch (const exception& e) {
529         cout << "Main: Caught: " << e.what() << endl;
530     }
531 }
```

```
531     this_thread::sleep_for(1s);
532 }
533
534 // =====
535 // 13. STD::VARIANT (C++17)
536 // =====
537 void example_variant() {
538     cout << "\n== 13. STD::VARIANT (C++17) ==" << endl;
539
540     variant<int, string, double> value;
541
542     value = 42;
543     cout << "Holds int: " << get<int>(value) << endl;
544
545     value = "Hello World";
546     cout << "Holds string: " << get<string>(value) << endl;
547
548     value = 3.14159;
549     cout << "Holds double: " << get<double>(value) << endl;
550
551     // Visit pattern
552     auto visitor = [] (auto&& arg) {
553         using T = decay_t<decltype(arg)>;
554         if constexpr (is_same_v<T, int>) {
555             cout << "Visited int: " << arg << endl;
556         }
557         else if constexpr (is_same_v<T, string>) {
558             cout << "Visited string: " << arg << endl;
559         }
560         else if constexpr (is_same_v<T, double>) {
561             cout << "Visited double: " << arg << endl;
562         }
563     };
564
565     visit(visitor, value);
566
567     value = "Test";
568     if (holds_alternative<string>(value)) {
569         cout << "Value holds a string" << endl;
570     }
571 }
572
573 // =====
574 // 14. STD::OPTIONAL (C++17)
575 // =====
576 optional<int> divide(int a, int b) {
577     if (b == 0) {
578         return nullopt;
579     }
580     return a / b;
581 }
582
583 void example_optional() {
```

```
585     cout << "\n==== 14. STD::OPTIONAL (C++17) ===" << endl;
586
587     auto result1 = divide(10, 2);
588     if (result1) {
589         cout << "10 / 2 = " << *result1 << endl;
590     }
591
592     auto result2 = divide(10, 0);
593     if (!result2) {
594         cout << "10 / 0 = No result (division by zero)" << endl;
595     }
596
597     cout << "Result or default: " << result2.value_or(999) << endl;
598
599     optional<string> name = "Alice";
600     cout << "Name: " << name.value() << endl;
601
602     optional<int> empty_opt;
603     cout << "Empty optional has value: " << boolalpha << empty_opt.has_value()
604         << endl;
605
606 // =====
607 // 15. STD::ANY (C++17)
608 // =====
609 void example_any() {
610     cout << "\n==== 15. STD::ANY (C++17) ===" << endl;
611
612     any value;
613
614     value = 42;
615     cout << "Any holds int: " << any_cast<int>(value) << endl;
616
617     value = string("Hello");
618     cout << "Any holds string: " << any_cast<string>(value) << endl;
619
620     value = 3.14;
621     cout << "Any holds double: " << any_cast<double>(value) << endl;
622
623     if (value.type() == typeid(double)) {
624         cout << "Value is a double" << endl;
625     }
626
627     try {
628         cout << any_cast<int>(value) << endl;
629     }
630     catch (const bad_any_cast& e) {
631         cout << "Caught exception: " << e.what() << endl;
632     }
633 }
634
635 // =====
636 // 16. FILESYSTEM (C++17)
637 // =====
```

```
638 void example_filesystem() {
639     cout << "\n==== 16. FILESYSTEM (C++17) ===" << endl;
640
641     namespace fs = std::filesystem;
642
643     cout << "Current path: " << fs::current_path() << endl;
644
645     fs::create_directory("test_dir");
646     cout << "Created directory: test_dir" << endl;
647
648     {
649         ofstream file("test_dir/test_file.txt");
650         file << "Hello Filesystem!" << endl;
651     }
652
653     if (fs::exists("test_dir/test_file.txt")) {
654         cout << "File exists" << endl;
655         cout << "File size: " << fs::file_size("test_dir/test_file.txt") << " bytes" << endl;
656     }
657
658     cout << "\nDirectory contents:" << endl;
659     for (const auto& entry : fs::directory_iterator("test_dir")) {
660         cout << " " << entry.path().filename()
661             << " (" << fs::file_size(entry) << " bytes)" << endl;
662     }
663
664     fs::remove_all("test_dir");
665     cout << "Cleaned up test directory" << endl;
666 }
667
668 // =====
669 // 17. STRUCTURED BINDINGS (C++17)
670 // =====
671 void example_structured_bindings() {
672     cout << "\n==== 17. STRUCTURED BINDINGS (C++17) ===" << endl;
673
674     // With tuple
675     tuple<int, string, double> tup = {42, "Alice", 3.14};
676     auto [id, name, score] = tup;
677     cout << "Tuple unpacked: " << id << ", " << name << ", " << score << endl;
678
679     // With array
680     array<int, 3> arr = {1, 2, 3};
681     auto [a, b, c] = arr;
682     cout << "Array unpacked: " << a << ", " << b << ", " << c << endl;
683
684     // With struct
685     struct Person {
686         string name;
687         int age;
688         string city;
689     };
690 }
```

```
691     Person p = {"Bob", 30, "New York"};
692     auto [person_name, age, city] = p;
693     cout << "Person: " << person_name << ", " << age << ", " << city << endl;
694
695     // With map
696     map<string, int> scores = {"Alice", 95}, {"Bob", 87}, {"Charlie", 92};
697     for (const auto& [student, score] : scores) {
698         cout << student << ": " << score << endl;
699     }
700 }
701
702 // =====
703 // 18. CONSTEXPR IF (C++17)
704 // =====
705 template<typename T>
706 auto process_value(T value) {
707     if constexpr (is_integral_v<T>) {
708         cout << "Processing integer: " << value * 2 << endl;
709         return value * 2;
710     }
711     else if constexpr (is_floating_point_v<T>) {
712         cout << "Processing float: " << value * 3.14 << endl;
713         return value * 3.14;
714     }
715     else if constexpr (is_same_v<T, string>) {
716         cout << "Processing string: " << value + " processed" << endl;
717         return value + " processed";
718     }
719     else {
720         cout << "Unknown type" << endl;
721         return value;
722     }
723 }
724
725 void example_constexpr_if() {
726     cout << "\n==== 18. CONSTEXPR IF (C++17) ===" << endl;
727
728     process_value(42);
729     process_value(3.14);
730     process_value(string("Hello"));
731 }
732
733 // =====
734 // 19. COMPREHENSIVE EXAMPLE: ASYNC FILE PROCESSOR
735 // =====
736 void example_comprehensive_async_processor() {
737     cout << "\n==== 19. COMPREHENSIVE EXAMPLE: ASYNC FILE PROCESSOR ===" <<
738         endl;
739
739     class AsyncFileProcessor {
740     private:
741         ThreadPool pool{4};
742
743     public:
```

```
744     future<vector<string>> process_files(const vector<string>& filenames)
745     {
746         vector<future<string>> futures;
747
748         for (const auto& filename : filenames) {
749             futures.push_back(pool.enqueue([filename]() -> string {
750                 this_thread::sleep_for(chrono::milliseconds(100 + rand() %
751                     400));
752
753                 if (filename.find("error") != string::npos) {
754                     throw runtime_error("Error processing " + filename);
755                 }
756
757                 return "Processed: " + filename + " (size: "
758                     + to_string(rand() % 1000) + " bytes)";
759             }));
760         }
761
762         return async(launch::async, [futures = move(futures)]() mutable {
763             vector<string> results;
764             for (auto& f : futures) {
765                 try {
766                     results.push_back(f.get());
767                 }
768                 catch (const exception& e) {
769                     results.push_back("ERROR: " + string(e.what()));
770                 }
771             }
772             return results;
773         });
774     }
775
776     AsyncFileProcessor processor;
777
778     vector<string> files = {
779         "document1.txt",
780         "image1.png",
781         "error_file.txt",
782         "data.csv",
783         "config.json"
784     };
785
786     cout << "Processing " << files.size() << " files asynchronously..." <<
787         endl;
788
789     auto start = high_resolution_clock::now();
790     auto result_future = processor.process_files(files);
791
792     cout << "Main thread doing other work..." << endl;
793     this_thread::sleep_for(200ms);
794
795     auto results = result_future.get();
796     auto end = high_resolution_clock::now();
```

```
795     cout << "\nProcessing complete in "
796     << duration_cast<milliseconds>(end - start).count() << "ms" << endl;
797     cout << "\nResults:" << endl;
798     for (const auto& result : results) {
799         cout << "    " << result << endl;
800     }
801 }
802 }
803
804 // =====
805 // 20. ADDITIONAL EXAMPLE: FUTURE WITH MULTIPLE CONTINUATIONS
806 // =====
807 void example_future_chaining() {
808     cout << "\n== 20. FUTURE CHAINING ==" << endl;
809
810     auto future1 = async(launch::async, []() {
811         cout << "    Step 1: Starting with 5" << endl;
812         this_thread::sleep_for(200ms);
813         return 5 + 10; // 15
814     });
815
816     auto future2 = async(launch::async, [f1 = move(future1)]() mutable {
817         int val = f1.get();
818         cout << "    Step 2: Processing " << val << endl;
819         this_thread::sleep_for(200ms);
820         return val * 2; // 30
821     });
822
823     auto future3 = async(launch::async, [f2 = move(future2)]() mutable {
824         int val = f2.get();
825         cout << "    Step 3: Finalizing " << val << endl;
826         this_thread::sleep_for(200ms);
827         return val - 5; // 25
828     });
829
830     int result = future3.get();
831     cout << "Final result: " << result << endl;
832 }
833
834 // =====
835 // MAIN FUNCTION
836 // =====
837 int main() {
838     cout << "
839     ====="
840     << endl;
841     cout << "COMPLETE C++17 FEATURES EXAMPLES" << endl;
842     cout << "
843     ====="
844     << endl;
845
846     srand(time(nullptr));
847
848     try {
```

```
845     example_basic_future_promise();
846     example_future_exception();
847     example_async();
848     example_shared_future();
849     example_future_timeout();
850     example_packaged_task();
851     example_parallel_computation();
852     example_future_continuations();
853     example_when_all();
854     example_when_any();
855     example_thread_pool();
856     example_cancellable_future();
857     example_variant();
858     example_optional();
859     example_any();
860     example_filesystem();
861     example_structured_bindings();
862     example_constexpr_if();
863     example_comprehensive_async_processor();
864     example_future_chaining();
865
866     cout << "\n"
867     =====
868     " << endl;
869     cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!" << endl;
870     cout << "
871     =====
872     " << endl;
873 }
874
875     return 0;
876 }
```

## 14 Source Code: Cpp17Examples.cpp

File: src/Cpp17Examples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <map>
5 #include <tuple>
6 #include <optional>
7 #include <variant>
8 #include <any>
9 #include <string_view>
10 #include <filesystem>
11 #include <algorithm>
12 #include <numeric>
13 #include <memory>
14 #include <functional>
15
16 // =====
17 // C++17 COMPREHENSIVE EXAMPLES
18 // =====
19
20 // =====
21 // 1. TEMPLATE ARGUMENT DEDUCTION FOR CLASS TEMPLATES
22 // =====
23 void example_template_argument_deduction() {
24     std::cout << "\n== 1. TEMPLATE ARGUMENT DEDUCTION ==" << std::endl;
25
26     std::pair p1{1, 2.0};           // std::pair<int, double>
27     std::tuple t1{1, "hello", 3.14}; // std::tuple<int, const char*, double>
28     std::vector v1{1, 2, 3, 4};     // std::vector<int>
29
30     std::cout << "std::pair deduced: (" << p1.first << ", " << p1.second << ")"
31     " << std::endl;
32     std::cout << "std::vector deduced with " << v1.size() << " elements" <<
33     std::endl;
34 }
35
36 // =====
37 // 2. STRUCTURED BINDINGS
38 // =====
39 void example_structured_bindings() {
40     std::pair<int, std::string> pair{1, "hello"};
41     auto [num, str] = pair;
42     std::cout << "Pair: " << num << ", " << str << std::endl;
43
44     std::tuple<int, double, std::string> tup{42, 3.14, "world"};
45     auto [i, d, s] = tup;
46     std::cout << "Tuple: " << i << ", " << d << ", " << s << std::endl;
47 }
```

```
48     std::map<int, std::string> m{{1, "one"}, {2, "two"}};
49     for (const auto& [key, value] : m) {
50         std::cout << "Map: " << key << " -> " << value << std::endl;
51     }
52 }
53
54 // =====
55 // 3. FOLDING EXPRESSIONS
56 // =====
57 template <typename... Args>
58 auto sum_fold(Args... args) {
59     return (... + args); // Unary left fold
60 }
61
62 template <typename... Args>
63 bool all_true(Args... args) {
64     return (... && args); // Unary left fold
65 }
66
67 template <typename... Args>
68 void print_all(Args... args) {
69     ((std::cout << args << " "), ...); // Binary left fold
70     std::cout << std::endl;
71 }
72
73 void example_folding_expressions() {
74     std::cout << "\n== 3. FOLDING EXPRESSIONS ==" << std::endl;
75
76     std::cout << "sum_fold(1, 2, 3, 4, 5) = " << sum_fold(1, 2, 3, 4, 5) <<
77         std::endl;
78     std::cout << "all_true(true, true, true) = " << std::boolalpha << all_true(
79         true, true, true) << std::endl;
80     std::cout << "all_true(true, false, true) = " << all_true(true, false,
81         true) << std::endl;
82     std::cout << "print_all: ";
83     print_all(1, "hello", 3.14, "world");
84 }
85
86 // =====
87 // 4. CONSTEXPR IF
88 // =====
89 template <typename T>
90 auto get_value(T t) {
91     if constexpr (std::is_pointer_v<T>) {
92         return *t; // Dereference pointer
93     } else {
94         return t; // Return value directly
95     }
96 }
97
98 void example_constexpr_if() {
99     std::cout << "\n== 4. CONSTEXPR IF ==" << std::endl;
100
101     int x = 42;
```

```
99     int* ptr = &x;  
100  
101    std::cout << "get_value(10) = " << get_value(10) << std::endl;  
102    std::cout << "get_value(ptr) = " << get_value(ptr) << std::endl;  
103 }  
104  
105 // =====  
106 // 5. CONSTEXPR LAMBDA  
107 // =====  
108 void example_constexpr_lambda() {  
109     std::cout << "\n== 5. CONSTEXPR LAMBDA ==" << std::endl;  
110  
111     constexpr auto square = [](int n) { return n * n; };  
112     constexpr int result = square(5);  
113  
114     std::cout << "Compile-time square(5) = " << result << std::endl;  
115     static_assert(square(4) == 16, "Square function test");  
116 }  
117  
118 // =====  
119 // 6. INLINE VARIABLES  
120 // =====  
121 inline int global_counter = 0;  
122  
123 struct Config {  
124     inline static const std::string app_name = "MyApp";  
125     inline static const int version = 1;  
126 };  
127  
128 void example_inline_variables() {  
129     std::cout << "\n== 6. INLINE VARIABLES ==" << std::endl;  
130  
131     global_counter++;  
132     std::cout << "Global counter: " << global_counter << std::endl;  
133     std::cout << "App name: " << Config::app_name << std::endl;  
134     std::cout << "Version: " << Config::version << std::endl;  
135 }  
136  
137 // =====  
138 // 7. NESTED NAMESPACES  
139 // =====  
140 namespace A::B::C {  
141     void nested_function() {  
142         std::cout << "Inside A::B::C namespace" << std::endl;  
143     }  
144 }  
145  
146 void example_nested_namespaces() {  
147     std::cout << "\n== 7. NESTED NAMESPACES ==" << std::endl;  
148     A::B::C::nested_function();  
149 }  
150  
151 // =====  
152 // 8. SELECTION STATEMENTS WITH INITIALIZER
```

```
153 // =====
154 void example_selection_with_initializer() {
155     std::cout << "\n== 8. SELECTION WITH INITIALIZER ==" << std::endl;
156
157     std::map<int, std::string> m{{1, "one"}, {2, "two"}};
158
159     if (auto it = m.find(1); it != m.end()) {
160         std::cout << "Found: " << it->second << std::endl;
161     }
162
163     switch (auto value = 42; value) {
164         case 42:
165             std::cout << "Value is 42" << std::endl;
166             break;
167         default:
168             std::cout << "Value is not 42" << std::endl;
169     }
170 }
171 // =====
172 // 9. [[FALLTHROUGH]], [[NODISCARD]], [[MAYBE_UNUSED]] ATTRIBUTES
173 // =====
174 [[nodiscard]] int important_function() {
175     return 42;
176 }
177
178 void example_attributes([[maybe_unused]] int unused_param) {
179     std::cout << "\n== 9. ATTRIBUTES ==" << std::endl;
180
181     int result = important_function(); // Must use result or compiler warns
182     std::cout << "Result: " << result << std::endl;
183
184     int value = 2;
185     switch (value) {
186         case 1:
187             std::cout << "Case 1" << std::endl;
188             [[fallthrough]];
189         case 2:
190             std::cout << "Case 2 (maybe from fallthrough)" << std::endl;
191             break;
192     }
193 }
194
195 // =====
196 // 10. STD::OPTIONAL
197 // =====
198 std::optional<int> try_parse_int(const std::string& str) {
199     try {
200         return std::stoi(str);
201     } catch (...) {
202         return std::nullopt;
203     }
204 }
205 }
```

```
207 void example_optional() {
208     std::cout << "\n==== 10. STD::OPTIONAL ===" << std::endl;
209
210     std::optional<int> opt1 = 42;
211     std::optional<int> opt2 = std::nullopt;
212
213     if (opt1) {
214         std::cout << "opt1 has value: " << *opt1 << std::endl;
215     }
216
217     std::cout << "opt2 value or default: " << opt2.value_or(-1) << std::endl;
218
219     auto result1 = try_parse_int("123");
220     auto result2 = try_parse_int("abc");
221
222     std::cout << "Parse '123': " << (result1 ? std::to_string(*result1) : "failed") << std::endl;
223     std::cout << "Parse 'abc': " << (result2 ? std::to_string(*result2) : "failed") << std::endl;
224 }
225
226 // =====
227 // 11. STD::VARIANT
228 // =====
229 void example_variant() {
230     std::cout << "\n==== 11. STD::VARIANT ===" << std::endl;
231
232     std::variant<int, double, std::string> var;
233
234     var = 42;
235     std::cout << "Variant holds int: " << std::get<int>(var) << std::endl;
236
237     var = 3.14;
238     std::cout << "Variant holds double: " << std::get<double>(var) << std::endl;
239
240     var = "hello";
241     std::cout << "Variant holds string: " << std::get<std::string>(var) << std::endl;
242
243     std::visit([](auto&& arg) {
244         std::cout << "Visiting variant: " << arg << std::endl;
245     }, var);
246 }
247
248 // =====
249 // 12. STD::ANY
250 // =====
251 void example_any() {
252     std::cout << "\n==== 12. STD::ANY ===" << std::endl;
253
254     std::any a = 42;
255     std::cout << "Any holds int: " << std::any_cast<int>(a) << std::endl;
256 }
```

```
257     a = std::string("hello");
258     std::cout << "Any holds string: " << std::any_cast<std::string>(a) << std
259         ::endl;
260
261     if (a.type() == typeid(std::string)) {
262         std::cout << "Confirmed: any contains string" << std::endl;
263     }
264
265 // =====
266 // 13. STD::STRING_VIEW
267 // =====
268 void print_string_view(std::string_view sv) {
269     std::cout << "String view: " << sv << " (length: " << sv.length() << ")"
270         << std::endl;
271 }
272
273 void example_string_view() {
274     std::cout << "\n== 13. STD::STRING_VIEW ==" << std::endl;
275
276     std::string str = "Hello, World!";
277     std::string_view sv = str;
278
279     print_string_view(sv);
280     print_string_view("Literal string");
281     print_string_view(sv.substr(0, 5));
282 }
283
284 // =====
285 // 14. STD::FILESYSTEM
286 // =====
287 void example_filesystem() {
288     std::cout << "\n== 14. STD::FILESYSTEM ==" << std::endl;
289
290     namespace fs = std::filesystem;
291
292     fs::path p = fs::current_path();
293     std::cout << "Current path: " << p << std::endl;
294
295     fs::path example_path = "example.txt";
296     std::cout << "Filename: " << example_path.filename() << std::endl;
297     std::cout << "Extension: " << example_path.extension() << std::endl;
298 }
299
300 // =====
301 // 15. STD::CLAMP
302 // =====
303 void example_clamp() {
304     std::cout << "\n== 15. STD::CLAMP ==" << std::endl;
305
306     int value = 50;
307     int clamped = std::clamp(value, 10, 40);
308     std::cout << "clamp(50, 10, 40) = " << clamped << std::endl;
```

```
309     int value2 = 5;
310     int clamped2 = std::clamp(value2, 10, 40);
311     std::cout << "clamp(5, 10, 40) = " << clamped2 << std::endl;
312 }
313
314 // =====
315 // 16. GCD AND LCM
316 // =====
317 void example_gcd_lcm() {
318     std::cout << "\n==== 16. GCD AND LCM ===" << std::endl;
319
320     int a = 12, b = 18;
321     std::cout << "gcd(" << a << ", " << b << ") = " << std::gcd(a, b) << std::endl;
322     std::cout << "lcm(" << a << ", " << b << ") = " << std::lcm(a, b) << std::endl;
323 }
324
325 // =====
326 // 17. LAMBDA CAPTURE THIS BY VALUE
327 // =====
328 struct MyObject {
329     int value = 123;
330
331     auto get_value_copy() {
332         return [*this] { return value; };
333     }
334
335     auto get_value_ref() {
336         return [this] { return value; };
337     }
338 };
339
340 void example_lambda_capture_this() {
341     std::cout << "\n==== 17. LAMBDA CAPTURE THIS BY VALUE ===" << std::endl;
342
343     MyObject obj;
344     auto copy_lambda = obj.get_value_copy();
345     auto ref_lambda = obj.get_value_ref();
346
347     obj.value = 456;
348
349     std::cout << "Copy lambda returns: " << copy_lambda() << " (original value
350         )" << std::endl;
351     std::cout << "Ref lambda returns: " << ref_lambda() << " (modified value)"
352         << std::endl;
353 }
354
355 // =====
356 // 18. STD::INVOKE
357 // =====
358 int add_func(int a, int b) {
359     return a + b;
360 }
```

```
359
360 struct Adder {
361     int operator()(int a, int b) const {
362         return a + b;
363     }
364 };
365
366 void example_invoke() {
367     std::cout << "\n== 18. STD::INVOKE ==" << std::endl;
368
369     std::cout << "invoke(add_func, 3, 4) = " << std::invoke(add_func, 3, 4) <<
370         std::endl;
371
372     Adder adder;
373     std::cout << "invoke(Adder, 5, 6) = " << std::invoke(adder, 5, 6) << std::
374         endl;
375
376     auto lambda = [](int a, int b) { return a * b; };
377     std::cout << "invoke(lambda, 7, 8) = " << std::invoke(lambda, 7, 8) << std::
378         endl;
379 }
380
381 // =====
382 // 19. STD::APPLY
383 // =====
384 int multiply(int a, int b, int c) {
385     return a * b * c;
386 }
387
388 void example_apply() {
389     std::cout << "\n== 19. STD::APPLY ==" << std::endl;
390
391     std::tuple<int, int, int> args{2, 3, 4};
392     int result = std::apply(multiply, args);
393
394     std::cout << "apply(multiply, {2, 3, 4}) = " << result << std::endl;
395 }
396
397 // =====
398 // 20. STD::MAKE_FROM_TUPLE
399 // =====
400 struct Point {
401     int x, y, z;
402     Point(int x, int y, int z) : x(x), y(y), z(z) {}
403 };
404
405 void example_make_from_tuple() {
406     std::cout << "\n== 20. STD::MAKE_FROM_TUPLE ==" << std::endl;
407
408     std::tuple<int, int, int> coords{10, 20, 30};
409     auto point = std::make_from_tuple<Point>(coords);
410
411     std::cout << "Point created from tuple: (" << point.x << ", " << point.y
412         << ", " << point.z << ")" << std::endl;
```

```
409 }
410
411 // =====
412 // MAIN FUNCTION
413 // =====
414 int main() {
415     std::cout << "\n=====\n" << std::endl;
416     std::cout << "      C++17 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
417     std::cout << "=====\n" << std::endl;
418
419     example_template_argument_deduction();
420     example_structured_bindings();
421     example_folding_expressions();
422     example_constexpr_if();
423     example_constexpr_lambda();
424     example_inline_variables();
425     example_nested_namespaces();
426     example_selection_with_initializer();
427     example_attributes(42);
428     example_optional();
429     example_variant();
430     example_any();
431     example_string_view();
432     example_filesystem();
433     example_clamp();
434     example_gcd_lcm();
435     example_lambda_capture_this();
436     example_invoke();
437     example_apply();
438     example_make_from_tuple();
439
440     std::cout << "\n=====\n" << std::endl;
441     std::cout << "      ALL C++17 EXAMPLES COMPLETED" << std::endl;
442     std::cout << "=====\n" << std::endl;
443
444     return 0;
445 }
```

## 15 Source Code: Cpp20Examples.cpp

File: src/Cpp20Examples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <span>
5 #include <ranges>
6 #include <algorithm>
7 #include <concepts>
8 #include <compare>
9 #include <numbers>
10 #include <bit>
11 #include <bitset>
12 #include <array>
13 #include <set>
14 #include <numeric>
15
16 // =====
17 // C++20 COMPREHENSIVE EXAMPLES
18 // =====
19
20 // =====
21 // 1. CONCEPTS
22 // =====
23 template <typename T>
24 concept Integral = std::is_integral_v<T>;
25
26 template <typename T>
27 concept SignedIntegral = Integral<T> && std::is_signed_v<T>;
28
29 template <Integral T>
30 T add(T a, T b) {
31     return a + b;
32 }
33
34 template <SignedIntegral T>
35 T negate(T value) {
36     return -value;
37 }
38
39 void example_concepts() {
40     std::cout << "\n== 1. CONCEPTS ==" << std::endl;
41
42     std::cout << "add(10, 20) = " << add(10, 20) << std::endl;
43     std::cout << "add(5U, 3U) = " << add(5U, 3U) << std::endl;
44     std::cout << "negate(42) = " << negate(42) << std::endl;
45
46     // add(3.14, 2.5); // Error: double doesn't satisfy Integral
47 }
48
49 // =====
```

```
50 // 2. THREE-WAY COMPARISON (SPACESHIP OPERATOR)
51 // =====
52 struct Point {
53     int x, y;
54     auto operator<=(const Point&) const = default;
55 };
56
57 void example_three_way_comparison() {
58     std::cout << "\n== 2. THREE-WAY COMPARISON =="
59     Point p1{1, 2}, p2{1, 3}, p3{1, 2};
60
61     std::cout << "p1 == p3: " << std::boolalpha << (p1 == p3) << std::endl;
62     std::cout << "p1 != p2: " << (p1 != p2) << std::endl;
63     std::cout << "p1 < p2: " << (p1 < p2) << std::endl;
64
65     int a = 10, b = 20;
66     auto result = a <= b;
67     if (result < 0) {
68         std::cout << "a < b" << std::endl;
69     }
70 }
71
72 // =====
73 // 3. DESIGNATED INITIALIZERS
74 // =====
75 struct Config {
76     int width;
77     int height;
78     std::string title;
79 };
80
81 void example_designated_initializers() {
82     std::cout << "\n== 3. DESIGNATED INITIALIZERS =="
83     Config cfg{
84         .width = 1920,
85         .height = 1080,
86         .title = "My Window"
87     };
88
89     std::cout << "Config: " << cfg.width << "x" << cfg.height
90         << " - " << cfg.title << std::endl;
91 }
92
93 // =====
94 // 4. TEMPLATE SYNTAX FOR LAMBDA
95 // =====
96 void example_template_lambdas() {
97     std::cout << "\n== 4. TEMPLATE SYNTAX FOR LAMBDA =="
98     auto generic_add = []<typename T>(T a, T b) {
99         return a + b;
100    };
101 }
```

```
104     std::cout << "generic_add(10, 20) = " << generic_add(10, 20) << std::endl;
105     std::cout << "generic_add(1.5, 2.5) = " << generic_add(1.5, 2.5) << std::endl;
106
107     auto print_type = []<typename T>(T value) {
108         if constexpr (std::is_integral_v<T>) {
109             std::cout << "Integer: " << value << std::endl;
110         } else {
111             std::cout << "Non-integer: " << value << std::endl;
112         }
113     };
114
115     print_type(42);
116     print_type(3.14);
117 }
118
119 // =====
120 // 5. RANGE-BASED FOR LOOP WITH INITIALIZER
121 // =====
122 void example_range_for_initializer() {
123     std::cout << "\n==== 5. RANGE-BASED FOR WITH INITIALIZER ===" << std::endl;
124
125     for (std::vector<int> vec = {1, 2, 3, 4, 5}; auto& elem : vec) {
126         std::cout << elem << " ";
127     }
128     std::cout << std::endl;
129 }
130
131
132 // =====
133 // 6. [[LIKELY]] AND [[UNLIKELY]] ATTRIBUTES
134 // =====
135 int predict_branch(int value) {
136     if (value > 0) [[likely]] {
137         return value * 2;
138     } else [[unlikely]] {
139         return value * 10;
140     }
141 }
142
143 void example_likely_unlikely() {
144     std::cout << "\n==== 6. [[LIKELY]] AND [[UNLIKELY]] ===" << std::endl;
145
146     std::cout << "predict_branch(5) = " << predict_branch(5) << std::endl;
147     std::cout << "predict_branch(-2) = " << predict_branch(-2) << std::endl;
148 }
149
150 // =====
151 // 7. CONSTEXPR VIRTUAL FUNCTIONS
152 // =====
153 struct Base {
154     virtual constexpr int get_value() const {
155         return 10;
156     }
157 }
```

```
157 };
```

```
158
159 struct Derived : Base {
160     constexpr int get_value() const override {
161         return 20;
162     }
163 };
164
165 void example_constexpr_virtual() {
166     std::cout << "\n== 7. CONSTEXPR VIRTUAL FUNCTIONS ==" << std::endl;
167
168     constexpr Derived d;
169     constexpr int value = d.get_value();
170
171     std::cout << "Constexpr virtual function result: " << value << std::endl;
172 }
173
174 // =====
175 // 8. EXPLICIT(BOOL)
176 // =====
177 struct MyInt {
178     int value;
179     explicit(sizeof(int) > 4) MyInt(int v) : value(v) {}
180 };
181
182 void example_explicit_bool() {
183     std::cout << "\n== 8. EXPLICIT(BOOL) ==" << std::endl;
184
185     MyInt mi{42}; // Always requires explicit construction based on condition
186     std::cout << "MyInt created with value: " << mi.value << std::endl;
187 }
188
189 // =====
190 // 9. IMMEDIATE FUNCTIONS (CONSTEVAL)
191 // =====
192 constexpr int square(int n) {
193     return n * n;
194 }
195
196 void example_immediate_functions() {
197     std::cout << "\n== 9. IMMEDIATE FUNCTIONS ==" << std::endl;
198
199     constexpr int result = square(5); // Must be evaluated at compile time
200     std::cout << "Consteval square(5) = " << result << std::endl;
201 }
202
203 // =====
204 // 10. USING ENUM
205 // =====
206 enum class Color { Red, Green, Blue };
```

```
207
208 void example_using_enum() {
209     std::cout << "\n== 10. USING ENUM ==" << std::endl;
210 }
```

```
211  using enum Color;
212  Color c = Red; // No need for Color::Red
213
214  switch (c) {
215      using enum Color; // Can use in switch scope
216      case Red:
217          std::cout << "Color is Red" << std::endl;
218          break;
219      case Green:
220          std::cout << "Color is Green" << std::endl;
221          break;
222      case Blue:
223          std::cout << "Color is Blue" << std::endl;
224          break;
225  }
226 }
227
228 // =====
229 // 11. CHAR8_T
230 // =====
231 void example_char8_t() {
232     std::cout << "\n== 11. CHAR8_T ==" << std::endl;
233
234     char8_t utf8_char = u8'A';
235     const char8_t* utf8_str = u8"Hello UTF-8";
236
237     std::cout << "UTF-8 character type created" << std::endl;
238     std::cout << "UTF-8 string type created" << std::endl;
239 }
240
241 // =====
242 // 12. CONSTINIT
243 // =====
244 constinit int global_value = 42;
245
246 void example_constinit() {
247     std::cout << "\n== 12. CONSTINIT ==" << std::endl;
248
249     std::cout << "Constinit global value: " << global_value << std::endl;
250     global_value = 100; // Can be modified at runtime
251     std::cout << "Modified value: " << global_value << std::endl;
252 }
253
254 // =====
255 // 13. STD::SPAN
256 // =====
257 void process_span(std::span<int> s) {
258     for (auto& elem : s) {
259         elem *= 2;
260     }
261 }
262
263 void example_span() {
264     std::cout << "\n== 13. STD::SPAN ==" << std::endl;
```

```
265     std::vector<int> vec = {1, 2, 3, 4, 5};  
266     std::span<int> s{vec};  
267  
268     std::cout << "Before: ";  
269     for (auto v : vec) std::cout << v << " ";  
270     std::cout << std::endl;  
271  
272     process_span(s);  
273  
274  
275     std::cout << "After: ";  
276     for (auto v : vec) std::cout << v << " ";  
277     std::cout << std::endl;  
278 }  
279  
280 // =====  
281 // 14. RANGES  
282 // =====  
283 void example_ranges() {  
284     std::cout << "\n== 14. RANGES ==" << std::endl;  
285  
286     std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
287  
288     auto even = vec | std::views::filter([](int n) { return n % 2 == 0; });  
289     auto squared = even | std::views::transform([](int n) { return n * n; });  
290  
291     std::cout << "Even numbers squared: ";  
292     for (auto v : squared) {  
293         std::cout << v << " ";  
294     }  
295     std::cout << std::endl;  
296 }  
297  
298 // =====  
299 // 15. BIT OPERATIONS  
300 // =====  
301 void example_bit_operations() {  
302     std::cout << "\n== 15. BIT OPERATIONS ==" << std::endl;  
303  
304     unsigned int value = 0b10110100;  
305  
306     std::cout << "popcount(0b10110100) = " << std::popcount(value) << std::endl;  
307     std::cout << "has_single_bit(8) = " << std::boolalpha << std::has_single_bit(8u) << std::endl;  
308     std::cout << "bit_width(7) = " << std::bit_width(7u) << std::endl;  
309     std::cout << "rotl(0b10110100, 2) = " << std::rotl(value, 2) << std::endl;  
310 }  
311  
312 // =====  
313 // 16. MATH CONSTANTS  
314 // =====  
315 void example_math_constants() {  
316     std::cout << "\n== 16. MATH CONSTANTS ==" << std::endl;
```

```
317     std::cout << "pi = " << std::numbers::pi << std::endl;
318     std::cout << "e = " << std::numbers::e << std::endl;
319     std::cout << "sqrt2 = " << std::numbers::sqrt2 << std::endl;
320     std::cout << "ln2 = " << std::numbers::ln2 << std::endl;
321 }
322 }
323
324 // =====
325 // 17. STD::IS_CONSTANT_EVALUATED
326 // =====
327 constexpr int compute_value() {
328     if (std::is_constant_evaluated()) {
329         return 42; // Compile-time
330     } else {
331         return 100; // Runtime
332     }
333 }
334
335 void example_is_constant_evaluated() {
336     std::cout << "\n==== 17. STD::IS_CONSTANT_EVALUATED ===" << std::endl;
337
338     constexpr int compile_time = compute_value();
339     int runtime = compute_value();
340
341     std::cout << "Compile-time: " << compile_time << std::endl;
342     std::cout << "Runtime: " << runtime << std::endl;
343 }
344
345 // =====
346 // 18. STARTS_WITH / ENDS_WITH
347 // =====
348 void example_starts_ends_with() {
349     std::cout << "\n==== 18. STARTS_WITH / ENDS_WITH ===" << std::endl;
350
351     std::string str = "Hello, World!";
352
353     std::cout << "starts_with('Hello'): " << std::boolalpha
354             << str.starts_with("Hello") << std::endl;
355     std::cout << "ends_with('World!'): " << str.ends_with("World!") << std::endl;
356 }
357
358 // =====
359 // 19. ASSOCIATIVE CONTAINER .CONTAINS()
360 // =====
361 void example_contains() {
362     std::cout << "\n==== 19. ASSOCIATIVE CONTAINER .CONTAINS() ===" << std::endl;
363
364     std::vector<int> vec = {1, 2, 3, 4, 5};
365     std::set<int> s{vec.begin(), vec.end()};
366
367     std::cout << "Set contains 3: " << std::boolalpha << s.contains(3) << std::endl;
```

```
368     std::cout << "Set contains 10: " << s.contains(10) << std::endl;
369 }
370
371 // =====
372 // 20. STD::MIDPOINT
373 // =====
374 void example_midpoint() {
375     std::cout << "\n== 20. STD::MIDPOINT ==" << std::endl;
376
377     std::cout << "midpoint(10, 20) = " << std::midpoint(10, 20) << std::endl;
378     std::cout << "midpoint(1.0, 5.0) = " << std::midpoint(1.0, 5.0) << std::endl;
379 }
380
381 // =====
382 // 21. STD::TO_ARRAY
383 // =====
384 void example_to_array() {
385     std::cout << "\n== 21. STD::TO_ARRAY ==" << std::endl;
386
387     auto arr = std::to_array({1, 2, 3, 4, 5});
388
389     std::cout << "Array from initializer list: ";
390     for (auto v : arr) {
391         std::cout << v << " ";
392     }
393     std::cout << std::endl;
394 }
395
396 // =====
397 // 22. LAMBDA CAPTURE OF PARAMETER PACK
398 // =====
399 template <typename... Args>
400 auto make_lambda(Args... args) {
401     return [...args = std::move(args)] {
402         ((std::cout << args << " "), ...);
403         std::cout << std::endl;
404     };
405 }
406
407 void example_lambda_capture_pack() {
408     std::cout << "\n== 22. LAMBDA CAPTURE OF PARAMETER PACK ==" << std::endl
409     ;
410
411     auto lambda = make_lambda(1, 2, 3, "hello", 4.5);
412     std::cout << "Lambda with captured pack: ";
413     lambda();
414 }
415
416 // =====
417 // 23. CLASS TYPES IN NON-TYPE TEMPLATE PARAMETERS
418 // =====
419 struct CompileTimeValue {
    int value;
```

```
420     constexpr CompileTimeValue(int v) : value(v) {}
421 };
422
423 template <CompileTimeValue val>
424 void print_compile_time() {
425     std::cout << "Compile-time value: " << val.value << std::endl;
426 }
427
428 void example_class_non_type_template() {
429     std::cout << "\n==== 23. CLASS TYPES IN NON-TYPE TEMPLATE ===" << std::endl
430         ;
431
432     print_compile_time<CompileTimeValue{42}>();
433 }
434
435 // =====
436 // MAIN FUNCTION
437 int main() {
438     std::cout << "\n====" << std::endl;
439     std::cout << "      C++20 FEATURES COMPREHENSIVE EXAMPLES" << std::endl;
440     std::cout << "====" << std::endl;
441
442     example_concepts();
443     example_three_way_comparison();
444     example_designated_initializers();
445     example_template_lambdas();
446     example_range_for_initializer();
447     example_likely_unlikely();
448     example_constexpr_virtual();
449     example_explicit_bool();
450     example_immediate_functions();
451     example_using_enum();
452     example_char8_t();
453     example_constinit();
454     example_span();
455     example_ranges();
456     example_bit_operations();
457     example_math_constants();
458     example_is_constant_evaluated();
459     example_starts_ends_with();
460     example_contains();
461     example_midpoint();
462     example_to_array();
463     example_lambda_capture_pack();
464     example_class_non_type_template();
465
466     std::cout << "\n====" << std::endl;
467     std::cout << "      ALL C++20 EXAMPLES COMPLETED" << std::endl;
468     std::cout << "====\n" << std::endl;
```

```
469
470     return 0;
471 }
```

## 16 Source Code: Cpp23Examples.cpp

File: src/Cpp23Examples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <optional>
5 #include <utility> // for std::to_underlying, std::unreachable
6 #include <cmath>
7 #include <cstdint>
8 #include <array>
9 #include <bit>
10
11 // =====
12 // C++23 FEATURES FOR EMBEDDED SYSTEMS, FIRMWARE, AND RTOS
13 // =====
14 // Focused on features useful for:
15 // - Firmware development
16 // - Real-Time Operating Systems (RTOS)
17 // - Embedded Linux platforms
18 // - Hardware interfacing
19 // - Resource-constrained environments
20 // =====
21
22 // =====
23 // 1. std::expected<T, E> - ERROR HANDLING WITHOUT EXCEPTIONS
24 // =====
25 // CRITICAL for embedded: No exception overhead!
26 // Perfect for firmware where exceptions are disabled
27
28 #if __cplusplus >= 202302L && __has_include(<expected>)
29 #include <expected>
30
31 enum class SensorError {
32     NotConnected,
33     ReadTimeout,
34     InvalidData,
35     CalibrationFailed
36 };
37
38 const char* to_string(SensorError error) {
39     switch (error) {
40         case SensorError::NotConnected: return "Sensor not connected";
41         case SensorError::ReadTimeout: return "Read timeout";
42         case SensorError::InvalidData: return "Invalid data";
43         case SensorError::CalibrationFailed: return "Calibration failed";
44     }
45     return "Unknown error";
46 }
47
48 // Return either temperature value or error - no exceptions!
49 std::expected<float, SensorError> read_temperature_sensor() {
```

```
50 // Simulate sensor read
51 static int call_count = 0;
52 call_count++;
53
54 if (call_count == 1) {
55     return std::unexpected(SensorError::NotConnected);
56 }
57 if (call_count == 2) {
58     return std::unexpected(SensorError::ReadTimeout);
59 }
60
61 return 23.5f; // Success
62 }
63
64 // Chaining operations with monadic interface
65 std::expected<float, SensorError> read_and_convert_to_fahrenheit() {
66     return read_temperature_sensor()
67         .and_then([](float celsius) -> std::expected<float, SensorError> {
68             if (celsius < -273.15f) {
69                 return std::unexpected(SensorError::InvalidData);
70             }
71             return celsius * 9.0f / 5.0f + 32.0f;
72         });
73 }
74
75 void demonstrate_expected() {
76     std::cout << "\n==== 1. std::expected - EXCEPTION-FREE ERROR HANDLING ==="
77         << std::endl;
78     std::cout << "Perfect for firmware with -fno-exceptions" << std::endl;
79
80     // First call - error
81     auto result1 = read_temperature_sensor();
82     if (result1) {
83         std::cout << "\n Temperature: " << *result1 << "°C" << std::endl;
84     } else {
85         std::cout << "\n Error: " << to_string(result1.error()) << std::endl;
86     }
87
88     // Second call - error
89     auto result2 = read_temperature_sensor();
90     if (result2) {
91         std::cout << " Temperature: " << *result2 << "°C" << std::endl;
92     } else {
93         std::cout << " Error: " << to_string(result2.error()) << std::endl;
94     }
95
96     // Third call - success
97     auto result3 = read_temperature_sensor();
98     if (result3) {
99         std::cout << " Temperature: " << *result3 << "°C" << std::endl;
100    } else {
101        std::cout << " Error: " << to_string(result3.error()) << std::endl;
102    }
103 }
```

```
103     std::cout << "\n Benefits for Embedded:" << std::endl;
104     std::cout << " • Zero exception overhead" << std::endl;
105     std::cout << " • Works with -fno-exceptions" << std::endl;
106     std::cout << " • Explicit error handling" << std::endl;
107     std::cout << " • Deterministic performance" << std::endl;
108 }
109
110 #else
111 void demonstrate_expected() {
112     std::cout << "\n== 1. std::expected ==" << std::endl;
113     std::cout << " std::expected requires C++23 compiler support" << std::endl;
114     std::cout << "Alternative: Use std::variant<T, Error> in C++17" << std::endl;
115 }
116 #endif
117
118 // =====
119 // 2. std::byteswap - ENDIANNES CONVERSION FOR HARDWARE
120 // =====
121 // Essential for: Network protocols, file formats, hardware registers
122
123 #if __cplusplus >= 202302L && __has_include(<bit>)
124
125 void demonstrate_byteswap() {
126     std::cout << "\n== 2. std::byteswap - HARDWARE BYTE ORDER ==" << std::endl;
127     std::cout << "Critical for network protocols and hardware registers" << std::endl;
128
129     // Reading from network (big-endian) to host (possibly little-endian)
130     uint32_t network_value = 0x12345678;
131     std::cout << "\nNetwork value (big-endian): 0x" << std::hex << network_value << std::dec << std::endl;
132
133     uint32_t host_value = std::byteswap(network_value);
134     std::cout << "After byteswap: 0x" << std::hex << host_value << std::dec << std::endl;
135
136     // 16-bit register value swap
137     uint16_t reg16 = 0xABCD;
138     std::cout << "\n16-bit register: 0x" << std::hex << reg16 << std::dec << std::endl;
139     uint16_t swapped16 = std::byteswap(reg16);
140     std::cout << "Swapped: 0x" << std::hex << swapped16 << std::dec << std::endl;
141
142     // 64-bit timestamp swap
143     uint64_t timestamp = 0x0123456789ABCDEF;
144     std::cout << "\n64-bit timestamp: 0x" << std::hex << timestamp << std::dec << std::endl;
145     uint64_t swapped64 = std::byteswap(timestamp);
146     std::cout << "Swapped: 0x" << std::hex << swapped64 << std::dec << std::endl;
```

```
147     std::cout << "\n Embedded Use Cases:" << std::endl;
148     std::cout << " • Network protocol implementation (TCP/IP)" << std::endl;
149     std::cout << " • File format parsing (headers, metadata)" << std::endl;
150     std::cout << " • Cross-platform binary data exchange" << std::endl;
151     std::cout << " • Hardware register manipulation" << std::endl;
152 }
153
154
155 #else
156 void demonstrate_byteswap() {
157     std::cout << "\n== 2. std::byteswap ===" << std::endl;
158     std::cout << " std::byteswap requires C++23" << std::endl;
159     std::cout << "Alternative: Use __builtin_bswap32() or manual bit shifts"
160         << std::endl;
161 }
162 #endif
163
164 // =====
165 // 3. std::to_underlying - ENUM TO INTEGER (HARDWARE REGISTERS)
166 // =====
167
168 enum class GPIOPin : uint8_t {
169     PIN_0 = 0,
170     PIN_1 = 1,
171     PIN_2 = 2,
172     PIN_3 = 3,
173     PIN_LED = 13,
174     PIN_BUTTON = 7
175 };
176
177 enum class MemoryRegister : uint32_t {
178     STATUS_REG = 0x40000000,
179     CONTROL_REG = 0x40000004,
180     DATA_REG = 0x40000008,
181     CONFIG_REG = 0x4000000C
182 };
183
184 void demonstrate_to_underlying() {
185     std::cout << "\n== 3. std::to_underlying - HARDWARE REGISTER ACCESS ==="
186         << std::endl;
187     std::cout << "Convert enum class to underlying integer type" << std::endl;
188
189     // GPIO pin access
190     GPIOPin led_pin = GPIOPin::PIN_LED;
191     auto pin_number = std::to_underlying(led_pin);
192     std::cout << "\nLED Pin number: " << static_cast<int>(pin_number) << std::endl;
193
194     // Memory-mapped register access
195     MemoryRegister status = MemoryRegister::STATUS_REG;
196     // Example: volatile uint32_t* status_ptr = reinterpret_cast<volatile
197     //           uint32_t*>(std::to_underlying(status));
198     std::cout << "Status register address: 0x" << std::hex << std::
199         to_underlying(status) << std::dec << std::endl;
```

```

196
197 // Array indexing with enum
198 std::array<const char*, 4> reg_names = {"STATUS", "CONTROL", "DATA", "
199     CONFIG"};
200 MemoryRegister reg = MemoryRegister::CONTROL_REG;
201 size_t index = (std::to_underlying(reg) - std::to_underlying(
202     MemoryRegister::STATUS_REG)) / 4;
203 std::cout << "Register name: " << reg_names[index] << std::endl;
204
205 std::cout << "\n Before C++23:" << std::endl;
206 std::cout << "    static_cast<std::underlying_type_t<GPIOPin>>(led_pin)" <<
207     std::endl;
208 std::cout << "\n C++23:" << std::endl;
209 std::cout << "    std::to_underlying(led_pin) // Much cleaner!" << std::
210     endl;
211 }
212
213 // =====
214 // 4. constexpr FOR <cmath> - COMPILE-TIME CALCULATIONS
215 // =====
216
217 // Lookup table generation at compile time
218 constexpr std::array<float, 256> generate_sine_lookup_table() {
219     std::array<float, 256> table{};
220     for (size_t i = 0; i < 256; ++i) {
221         // C++23: std::sin is now constexpr!
222         table[i] = std::sin(2.0 * 3.14159265358979323846 * i / 256.0);
223     }
224     return table;
225 }
226
227 // PWM duty cycle calculation at compile time
228 constexpr uint16_t calculate_pwm_value(float duty_percent) {
229     return static_cast<uint16_t>(duty_percent * 65535.0f / 100.0f);
230 }
231
232 void demonstrate_constexpr_cmath() {
233     std::cout << "\n==== 4. constexpr <cmath> - COMPILE-TIME CALCULATIONS ==="
234         << std::endl;
235     std::cout << "Generate lookup tables at compile time" << std::endl;
236
237     // Sine lookup table generated at compile time
238     constexpr auto sine_table = generate_sine_lookup_table();
239     std::cout << "\nSine lookup table (256 entries) generated at compile time"
240         << std::endl;
241     std::cout << "Sample values:" << std::endl;
242     std::cout << "    sin(0°)      " << sine_table[0] << std::endl;
243     std::cout << "    sin(90°)     " << sine_table[64] << std::endl;
244     std::cout << "    sin(180°)    " << sine_table[128] << std::endl;
245     std::cout << "    sin(270°)    " << sine_table[192] << std::endl;
246
247     // PWM values calculated at compile time
248     constexpr uint16_t pwm_25 = calculate_pwm_value(25.0f);
249     constexpr uint16_t pwm_50 = calculate_pwm_value(50.0f);

```

```

244 constexpr uint16_t pwm_75 = calculate_pwm_value(75.0f);
245
246 std::cout << "\nPWM duty cycle values (calculated at compile time):" <<
247     std::endl;
248 std::cout << " 25%: " << pwm_25 << std::endl;
249 std::cout << " 50%: " << pwm_50 << std::endl;
250 std::cout << " 75%: " << pwm_75 << std::endl;
251
252 std::cout << "\n Embedded Benefits:" << std::endl;
253 std::cout << " • Zero runtime cost for lookup tables" << std::endl;
254 std::cout << " • Stored in ROM/Flash, not RAM" << std::endl;
255 std::cout << " • Fast table lookups vs real-time calculations" << std::endl;
256 std::cout << " • Deterministic execution time" << std::endl;
257 }
258
259 // =====
260 // 5. if consteval - COMPILE-TIME VS RUNTIME PATHS
261 // =====
262
263 constexpr uint32_t crc32_compute(const char* data, size_t len) {
264     if (consteval) {
265         // Compile-time: Simple algorithm
266         uint32_t crc = 0xFFFFFFFF;
267         for (size_t i = 0; i < len; ++i) {
268             crc ^= static_cast<uint32_t>(data[i]);
269             for (int j = 0; j < 8; ++j) {
270                 crc = (crc >> 1) ^ (0xEDB88320 & -(crc & 1));
271             }
272         }
273         return ~crc;
274     } else {
275         // Runtime: Could use hardware CRC or optimized table lookup
276         std::cout << "[Using runtime CRC algorithm]" << std::endl;
277         uint32_t crc = 0xFFFFFFFF;
278         for (size_t i = 0; i < len; ++i) {
279             crc ^= static_cast<uint32_t>(data[i]);
280             for (int j = 0; j < 8; ++j) {
281                 crc = (crc >> 1) ^ (0xEDB88320 & -(crc & 1));
282             }
283         }
284         return ~crc;
285     }
286 }
287
288 void demonstrate_if_consteval() {
289     std::cout << "\n== 5. if consteval - COMPILE-TIME VS RUNTIME ==" << std::endl;
290     std::cout << "Different code paths for compile-time and runtime" << std::endl;
291
292     // Compile-time CRC calculation
293     constexpr const char* firmware_id = "FIRMWARE_V1.2.3";
294     constexpr uint32_t compile_time_crc = crc32_compute(firmware_id, 15);

```

```

294     std::cout << "\nFirmware ID: " << firmware_id << std::endl;
295     std::cout << "CRC32 (compile-time): 0x" << std::hex << compile_time_crc <<
296         std::dec << std::endl;
297
298     // Runtime CRC calculation
299     const char* runtime_data = "RUNTIME_DATA_PACKET";
300     uint32_t runtime_crc = crc32_compute(runtime_data, 19);
301     std::cout << "\nRuntime data: " << runtime_data << std::endl;
302     std::cout << "CRC32 (runtime): 0x" << std::hex << runtime_crc << std::dec
303         << std::endl;
304
305     std::cout << "\n Embedded Applications:" << std::endl;
306     std::cout << " • Compile-time: Simple, portable algorithm" << std::endl;
307     std::cout << " • Runtime: Hardware CRC accelerator" << std::endl;
308     std::cout << " • Firmware verification checksums" << std::endl;
309     std::cout << " • Protocol integrity checks" << std::endl;
310 }
311
312 // =====
313 // 6. std::unreachable - OPTIMIZATION HINTS FOR COMPILER
314 // =====
315
316 enum class DeviceState : uint8_t {
317     IDLE = 0,
318     RUNNING = 1,
319     STOPPED = 2,
320     ERROR = 3
321 };
322
323 [[nodiscard]] constexpr const char* device_state_name(DeviceState state) {
324     switch (state) {
325         case DeviceState::IDLE:    return "IDLE";
326         case DeviceState::RUNNING: return "RUNNING";
327         case DeviceState::STOPPED: return "STOPPED";
328         case DeviceState::ERROR:   return "ERROR";
329     }
330     std::unreachable(); // Tell compiler all cases covered
331 }
332
333 uint32_t process_command(uint8_t cmd) {
334     // Note: cmd >= 0 always true for unsigned type, kept for documentation
335     if (cmd <= 3) {
336         return cmd * 100; // Valid command processing
337     }
338
339     // Invalid command - should never happen with validated input
340     std::unreachable(); // Optimization hint
341 }
342
343 void demonstrate_unreachable() {
344     std::cout << "\n== 6. std::unreachable - COMPILER OPTIMIZATION ==" <<
345         std::endl;
346     std::cout << "Tell compiler certain code paths are impossible" << std::endl;

```

```

344
345     std::cout << "\nDevice states:" << std::endl;
346     std::cout << "    " << device_state_name(DeviceState::IDLE) << std::endl;
347     std::cout << "    " << device_state_name(DeviceState::RUNNING) << std::endl;
348     std::cout << "    " << device_state_name(DeviceState::STOPPED) << std::endl;
349     std::cout << "    " << device_state_name(DeviceState::ERROR) << std::endl;
350
351     std::cout << "\nCommand processing:" << std::endl;
352     std::cout << "    Command 0: " << process_command(0) << std::endl;
353     std::cout << "    Command 2: " << process_command(2) << std::endl;
354
355     std::cout << "\n Optimization Benefits:" << std::endl;
356     std::cout << "    • Eliminates dead code paths" << std::endl;
357     std::cout << "    • Better branch prediction" << std::endl;
358     std::cout << "    • Smaller code size" << std::endl;
359     std::cout << "    • Faster execution" << std::endl;
360
361     std::cout << "\n    WARNING: Reaching std::unreachable() is UB!" << std::endl;
362 }
363
364 // =====
365 // 7. SIZE_T LITERAL SUFFIX (uz/z) - TYPE-SAFE SIZES
366 // =====
367
368 void demonstrate_size_literals() {
369     std::cout << "\n== 7. SIZE_T LITERAL SUFFIX - TYPE SAFETY ==" << std::endl;
370     std::cout << "Avoid unsigned vs signed comparison warnings" << std::endl;
371
372     std::vector<uint32_t> buffer = {1, 2, 3, 4, 5};
373
374     // Old way - warning: comparison between signed and unsigned
375     // for (int i = 0; i < buffer.size(); ++i) { } // Warning!
376
377     // C++23 way - type-safe
378     constexpr size_t BUFFER_SIZE = 256uz; // uz suffix = size_t
379     std::array<uint8_t, BUFFER_SIZE> uart_buffer{};
380
381     std::cout << "\nBuffer allocations:" << std::endl;
382     std::cout << "    UART buffer: " << BUFFER_SIZE << " bytes" << std::endl;
383     std::cout << "    Vector size: " << buffer.size() << " elements" << std::endl;
384     std::cout << "    Array capacity: " << uart_buffer.size() << " bytes" << std::endl;
385
386     // No warning with uz suffix
387     if (buffer.size() > 3uz) {
388         std::cout << "    Buffer has more than 3 elements" << std::endl;
389     }
390
391     std::cout << "\n Before C++23:" << std::endl;
392     std::cout << "    if (buffer.size() > 3)      // May warn" << std::endl;
393     std::cout << "    if (buffer.size() > 3u)     // Wrong type (unsigned)" <<

```

```
        std::endl;
394     std::cout << "    if (buffer.size() > 3UL)    // Platform dependent" << std
395         ::endl;
396     std::cout << "\n C++23:" << std::endl;
397     std::cout << "    if (buffer.size() > 3uz)    // Correct type!" << std::endl
398         ;
399 // =====
400 // 8. MULTIDIMENSIONAL SUBSCRIPT OPERATOR
401 // =====
402
403 template<typename T, size_t Rows, size_t Cols>
404 class Matrix {
405 private:
406     std::array<T, Rows * Cols> data;
407
408 public:
409     // C++23: Multi-dimensional subscript operator
410     // Note: Requires GCC 14+, Clang 17+, or MSVC 2022 17.8+
411     #if defined(__cpp_multidimensional_subscript) &&
412         __cpp_multidimensional_subscript >= 202110L
413     constexpr T& operator[](size_t row, size_t col) {
414         return data[row * Cols + col];
415     }
416
417     constexpr const T& operator[](size_t row, size_t col) const {
418         return data[row * Cols + col];
419     }
420     #endif
421
422     // Fallback: Traditional at() function for older compilers
423     constexpr T& at(size_t row, size_t col) {
424         return data[row * Cols + col];
425     }
426
427     constexpr const T& at(size_t row, size_t col) const {
428         return data[row * Cols + col];
429     }
430
431     constexpr void fill(T value) {
432         data.fill(value);
433     }
434 };
435
436 void demonstrate_multidim_subscript() {
437     std::cout << "\n== 8. MULTIDIMENSIONAL SUBSCRIPT OPERATOR ==" << std::
438         endl;
439     #if defined(__cpp_multidimensional_subscript) &&
440         __cpp_multidimensional_subscript >= 202110L
441     std::cout << " Multi-dimensional operator[] supported!" << std::endl;
442     std::cout << "Direct matrix[row, col] syntax" << std::endl;
443 }
```

```
442 Matrix<uint16_t, 3, 3> sensor_data;
443
444 // C++23: Clean syntax for 2D array access
445 sensor_data[0, 0] = 100;
446 sensor_data[0, 1] = 150;
447 sensor_data[0, 2] = 200;
448 sensor_data[1, 0] = 250;
449 sensor_data[1, 1] = 300;
450 sensor_data[1, 2] = 350;
451 sensor_data[2, 0] = 400;
452 sensor_data[2, 1] = 450;
453 sensor_data[2, 2] = 500;
454
455 std::cout << "\n3x3 Sensor data matrix:" << std::endl;
456 for (size_t row = 0; row < 3; ++row) {
457     std::cout << " ";
458     for (size_t col = 0; col < 3; ++col) {
459         std::cout << sensor_data[row, col] << "\t";
460     }
461     std::cout << std::endl;
462 }
463
464 std::cout << "\n C++23 syntax:" << std::endl;
465 std::cout << "    matrix[row, col]      // Direct syntax!" << std::endl;
466
467 #else
468
469 std::cout << "    Multi-dimensional operator[] requires GCC 14+, Clang 17+,
470         or MSVC 17.8+" << std::endl;
471 std::cout << "Using fallback at(row, col) method for demonstration" << std
472         ::endl;
473
474 Matrix<uint16_t, 3, 3> sensor_data;
475
476 // Fallback: Use at() method
477 sensor_data.at(0, 0) = 100;
478 sensor_data.at(0, 1) = 150;
479 sensor_data.at(0, 2) = 200;
480 sensor_data.at(1, 0) = 250;
481 sensor_data.at(1, 1) = 300;
482 sensor_data.at(1, 2) = 350;
483 sensor_data.at(2, 0) = 400;
484 sensor_data.at(2, 1) = 450;
485 sensor_data.at(2, 2) = 500;
486
487 std::cout << "\n3x3 Sensor data matrix (using at() method):" << std::endl;
488 for (size_t row = 0; row < 3; ++row) {
489     std::cout << " ";
490     for (size_t col = 0; col < 3; ++col) {
491         std::cout << sensor_data.at(row, col) << "\t";
492     }
493     std::cout << std::endl;
494 }
```

```
494     std::cout << "\n Before C++23:" << std::endl;
495     std::cout << "    matrix[row][col]      // Traditional double subscript" <<
496             std::endl;
496     std::cout << "    matrix.at(row, col)  // Member function" << std::endl;
497     std::cout << "\n C++23 (when compiler supports):" << std::endl;
498     std::cout << "    matrix[row, col]      // Direct multi-dimensional syntax!
498             " << std::endl;
499
500 #endif
501 }
502
503 // =====
504 // 9. PRACTICAL EMBEDDED EXAMPLE: FIRMWARE STATUS SYSTEM
505 // =====
506
507 #if __cplusplus >= 202302L && __has_include(<expected>)
508
509 enum class FirmwareError {
510     ConfigInvalid,
511     MemoryInsufficient,
512     HardwareNotResponding,
513     ChecksumMismatch
514 };
515
516 const char* firmware_error_name(FirmwareError err) {
517     switch (err) {
518         case FirmwareError::ConfigInvalid: return "Configuration invalid";
519         case FirmwareError::MemoryInsufficient: return "Insufficient memory";
520         case FirmwareError::HardwareNotResponding: return "Hardware not
520             responding";
521         case FirmwareError::ChecksumMismatch: return "Checksum mismatch";
522     }
523     std::unreachable();
524 }
525
526 class FirmwareStatus {
527 public:
528     std::expected<bool, FirmwareError> initialize() {
529         std::cout << " [1] Checking configuration..." << std::endl;
530         // Simulate config check
531
532         std::cout << " [2] Allocating memory..." << std::endl;
533         // Simulate memory allocation
534
535         std::cout << " [3] Initializing hardware..." << std::endl;
536         // Simulate hardware init
537
538         std::cout << " [4] Verifying checksums..." << std::endl;
539         constexpr uint32_t expected_crc = 0x12345678;
540         constexpr uint32_t actual_crc = 0x12345678;
541
542         if (expected_crc != actual_crc) {
543             return std::unexpected(FirmwareError::ChecksumMismatch);
544         }
545     }
546 }
```

```
545         return true;
546     }
547
548     std::expected<uint32_t, FirmwareError> get_hardware_id() {
549         // Simulate hardware ID read
550         return 0xABCD1234;
551     }
552 }
553
554
555 void demonstrate_practical_firmware() {
556     std::cout << "\n== 9. PRACTICAL EXAMPLE: FIRMWARE STATUS ==" << std::endl;
557     std::cout << "C++23 features in embedded firmware" << std::endl;
558
559     FirmwareStatus firmware;
560
561     std::cout << "\nInitializing firmware..." << std::endl;
562     auto init_result = firmware.initialize();
563
564     if (init_result) {
565         std::cout << " Firmware initialized successfully" << std::endl;
566
567         auto hw_id = firmware.get_hardware_id();
568         if (hw_id) {
569             std::cout << " Hardware ID: 0x" << std::hex << *hw_id << std::dec
570                         << std::endl;
571         }
572     } else {
573         std::cout << " Initialization failed: "
574                         << firmware_error_name(init_result.error()) << std::endl;
575     }
576
577     std::cout << "\n This firmware code:" << std::endl;
578     std::cout << " • Uses std::expected (no exceptions)" << std::endl;
579     std::cout << " • Works with -fno-exceptions" << std::endl;
580     std::cout << " • Deterministic error handling" << std::endl;
581     std::cout << " • Suitable for safety-critical systems" << std::endl;
582 }
583
584 #else
585 void demonstrate_practical_firmware() {
586     std::cout << "\n== 9. PRACTICAL EXAMPLE: FIRMWARE STATUS ==" << std::endl;
587     std::cout << " Requires C++23 std::expected" << std::endl;
588 }
589 #endif
590
591 // =====
592 // MAIN FUNCTION
593 // =====
594
595 int main() {
596     std::cout << "\n
```

```
      =====" <<
596     std::endl;
597     std::cout << "  C++23 FEATURES FOR EMBEDDED SYSTEMS & FIRMWARE" << std::
598     endl;
599     std::cout << "
600     =====" <<
601     std::endl;
602     std::cout << "Focused on: Firmware, RTOS, Embedded Linux" << std::endl;
603
604     demonstrate_expected();
605     demonstrate_byteswap();
606     demonstrate_to_underlying();
607     demonstrate_constexpr_cmath();
608     demonstrate_if_consteval();
609     demonstrate_unreachable();
610     demonstrate_size_literals();
611     demonstrate_multidim_subscript();
612     demonstrate_practical_firmware();
613
614     std::cout << "\n
615     =====" <<
616     std::endl;
617     std::cout << "  C++23 FEATURES SUMMARY FOR EMBEDDED" << std::endl;
618     std::cout << "
619     =====" <<
620     std::endl;
621
622     std::cout << "\n1  std::expected<T, E>" << std::endl;
623     std::cout << "  •  No exception overhead" << std::endl;
624     std::cout << "  •  Works with -fno-exceptions" << std::endl;
625     std::cout << "  •  Explicit error handling" << std::endl;
626     std::cout << "  •  Perfect for RTOS and firmware" << std::endl;
627     std::cout << "  Use case: Sensor reads, hardware init, protocol parsing"
628           << std::endl;
629
630     std::cout << "\n2  std::byteswap" << std::endl;
631     std::cout << "  •  Endianness conversion" << std::endl;
632     std::cout << "  •  Network protocol implementation" << std::endl;
633     std::cout << "  •  Binary file format parsing" << std::endl;
634     std::cout << "  •  Hardware register access" << std::endl;
635     std::cout << "  Use case: TCP/IP stack, file systems, cross-platform data
636           " << std::endl;
637
638     std::cout << "\n3  std::to_underlying" << std::endl;
639     std::cout << "  •  Cleaner enum to integer conversion" << std::endl;
640     std::cout << "  •  Memory-mapped register access" << std::endl;
641     std::cout << "  •  GPIO pin manipulation" << std::endl;
642     std::cout << "  •  Hardware abstraction layers" << std::endl;
643     std::cout << "  Use case: Register maps, pin assignments, state machines"
644           << std::endl;
645
646     std::cout << "\n4  constexpr <cmath>" << std::endl;
647     std::cout << "  •  Compile-time lookup table generation" << std::endl;
```

```
638     std::cout << " • Stored in ROM, not RAM" << std::endl;
639     std::cout << " • Zero runtime cost" << std::endl;
640     std::cout << " • Fast deterministic execution" << std::endl;
641     std::cout << " Use case: Sine/cosine tables, PWM values, calibration
642           data" << std::endl;
643
644     std::cout << "\n5 if consteval" << std::endl;
645     std::cout << " • Different paths for compile-time vs runtime" << std::
646           endl;
647     std::cout << " • Compile-time: simple algorithm" << std::endl;
648     std::cout << " • Runtime: hardware accelerator" << std::endl;
649     std::cout << " Use case: CRC calculation, crypto, compression" << std::
650           endl;
651
652     std::cout << "\n6 std::unreachable" << std::endl;
653     std::cout << " • Compiler optimization hints" << std::endl;
654     std::cout << " • Smaller code size" << std::endl;
655     std::cout << " • Better branch prediction" << std::endl;
656     std::cout << " Use case: State machines, validated inputs, switch
657           statements" << std::endl;
658
659     std::cout << "\n7 Size literal suffix (uz)" << std::endl;
660     std::cout << " • Type-safe size comparisons" << std::endl;
661     std::cout << " • No signed/unsigned warnings" << std::endl;
662     std::cout << " • Cleaner buffer management code" << std::endl;
663     std::cout << " Use case: Buffer sizes, array indexing, memory allocation
664           " << std::endl;
665
666     std::cout << "\n8 Multidimensional operator[]" << std::endl;
667     std::cout << " • Direct matrix[row, col] syntax" << std::endl;
668     std::cout << " • Cleaner multi-dimensional arrays" << std::endl;
669     std::cout << " • Natural matrix/tensor access" << std::endl;
670     std::cout << " Use case: Image buffers, sensor arrays, DSP data" << std
671           ::endl;
672
673
674     std::cout << "\n COMPILER SUPPORT (January 2026):" << std::endl;
675     std::cout << "\n  GCC 12+:" << std::endl;
676     std::cout << " • std::expected (use GCC 13+)" << std::endl;
677     std::cout << " • std::byteswap " << std::endl;
678     std::cout << " • std::to_underlying " << std::endl;
679     std::cout << " • if consteval " << std::endl;
680
681
682     std::cout << "\n  Clang 16+:" << std::endl;
683     std::cout << " • std::expected " << std::endl;
684     std::cout << " • Most C++23 features " << std::endl;
685
686
687     std::cout << "\n  MSVC 2022 (17.6+):" << std::endl;
688     std::cout << " • Excellent C++23 support " << std::endl;
689
690
691     std::cout << "\n  ARM Compiler 6:" << std::endl;
692     std::cout << " • Based on Clang, good C++23 support " << std::endl;
693
694
695     std::cout << "\n RECOMMENDED EMBEDDED COMPILER FLAGS:" << std::endl;
696     std::cout << " -std=c++23 # Enable C++23" << std::endl;
```

```
686     std::cout << " -fno-exceptions" # Disable exceptions" << std::endl
687     ;
688     std::cout << " -fno-rtti" # Disable RTTI" << std::endl;
689     std::cout << " -Os or -O2" # Optimize for size/speed" << std::
690     endl;
691     std::cout << " -fno-rtti" # Link-time optimization" << std::
692     endl;
693     std::cout << " -ffunction-sections" # Dead code elimination" << std::
694     endl;
695
696     std::cout << "\n MIGRATION STRATEGY:" << std::endl;
697     std::cout << " 1. Start with std::to_underlying (easy win)" << std::endl;
698     std::cout << " 2. Add constexpr lookup tables (ROM savings)" << std::endl
699     ;
700     std::cout << " 3. Replace error codes with std::expected (safer)" << std
701     ::endl;
702     std::cout << " 4. Use std::byteswap for protocols (cleaner)" << std::endl
703     ;
704     std::cout << " 5. Add std::unreachable for optimizations" << std::endl;
705
706     std::cout << "\n
707     ======\n" << std::endl;
708
709     return 0;
710 }
```

## 17 Source Code: CppWrappingCLibrary.cpp

File: src/CppWrappingCLibrary.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // WRAPPING C LIBRARIES IN MODERN C++
3 // =====
4 // This example demonstrates best practices for wrapping C libraries
5 // (TCP/UDP sockets) in modern C++ with:
6 // - RAII for automatic resource management
7 // - noexcept for exception safety
8 // - [[nodiscard]] for preventing ignored errors
9 // - extern "C" for C library linkage
10 // - Strong types and error handling
11 //
12 // TOPICS COVERED:
13 // 1. C library integration (POSIX sockets)
14 // 2. RAII wrappers for C resources
15 // 3. noexcept specifications (safe and unsafe uses)
16 // 4. [[nodiscard]] attribute
17 // 5. std::optional and std::expected for error handling
18 // 6. Strong typing over raw C types
19 // =====
20
21 #include <iostream>
22 #include <string>
23 #include <string_view>
24 #include <optional>
25 #include <system_error>
26 #include <cstring>
27 #include <vector>
28 #include <memory>
29 #include <chrono>
30
31 // Platform-specific socket headers
32 #ifdef _WIN32
33     #include <winsock2.h>
34     #include <ws2tcpip.h>
35     #pragma comment(lib, "ws2_32.lib")
36     using socket_t = SOCKET;
37     constexpr socket_t INVALID_SOCKET_VALUE = INVALID_SOCKET;
38     #define CLOSE_SOCKET closesocket
39 #else
40     #include <sys/socket.h>
41     #include <netinet/in.h>
42     #include <arpa/inet.h>
43     #include <unistd.h>
44     #include <fcntl.h>
45     #include <netdb.h>
46     using socket_t = int;
47     constexpr socket_t INVALID_SOCKET_VALUE = -1;
48     #define CLOSE_SOCKET close
49 #endif
```

```
50 // =====
51 // SECTION 1: C LIBRARY FUNCTIONS (extern "C" linkage)
52 // =====
53 // These are C functions - they use C calling conventions and linkage
54 // extern "C" prevents C++ name mangling
55
56 extern "C" {
57     // Example: Custom C utility function we might have in a C library
58     int c_validate_port(int port) {
59         return (port > 0 && port <= 65535) ? 1 : 0;
60     }
61
62     // Note: socket(), bind(), listen(), etc. are already extern "C"
63     // from system headers, so we don't redeclare them
64 }
65
66 // =====
67 // SECTION 2: ERROR HANDLING TYPES
68 // =====
69
70 // Modern C++ error type using std::optional
71 enum class SocketError {
72     Success,
73     InvalidSocket,
74     BindFailed,
75     ListenFailed,
76     ConnectFailed,
77     SendFailed,
78     ReceiveFailed,
79     AcceptFailed,
80     SocketOptionFailed,
81     AddressResolutionFailed,
82     TimeoutExpired,
83     WouldBlock
84 };
85
86
87 // Convert error code to string
88 [[nodiscard]] constexpr std::string_view error_to_string(SocketError error)
89     noexcept {
90     switch (error) {
91         case SocketError::Success: return "Success";
92         case SocketError::InvalidSocket: return "Invalid socket";
93         case SocketError::BindFailed: return "Bind failed";
94         case SocketError::ListenFailed: return "Listen failed";
95         case SocketError::ConnectFailed: return "Connect failed";
96         case SocketError::SendFailed: return "Send failed";
97         case SocketError::ReceiveFailed: return "Receive failed";
98         case SocketError::AcceptFailed: return "Accept failed";
99         case SocketError::SocketOptionFailed: return "Socket option failed";
100        case SocketError::AddressResolutionFailed: return "Address resolution
101            failed";
102        case SocketError::TimeoutExpired: return "Timeout expired";
103        case SocketError::WouldBlock: return "Operation would block";
```

```
102     default: return "Unknown error";
103 }
104 }
105
106 // Result type for operations that can fail
107 template<typename T>
108 using Result = std::optional<T>;
109
110 // =====
111 // SECTION 3: RAII SOCKET WRAPPER
112 // =====
113 // Modern C++ wrapper that automatically manages socket lifecycle
114
115 class Socket {
116 private:
117     socket_t fd_;
118     bool is_valid_;
119
120 public:
121     // Constructor - creates socket
122     // noexcept(false) because it may throw
123     Socket(int domain = AF_INET, int type = SOCK_STREAM, int protocol = 0)
124         : fd_(::socket(domain, type, protocol)),
125           is_valid_(fd_ != INVALID_SOCKET_VALUE) {
126
127         if (!is_valid_) {
128             std::cerr << "Socket creation failed" << std::endl;
129         } else {
130             std::cout << " Socket created (fd=" << fd_ << ")" << std::endl;
131         }
132     }
133
134     // Tag type for from_fd constructor
135     struct from_fd_t {};
136     static constexpr from_fd_t from_fd{};
137
138     // Construct from existing socket (for accept())
139     Socket(from_fd_t, socket_t fd) noexcept
140         : fd_(fd), is_valid_(fd != INVALID_SOCKET_VALUE) {}
141
142     // Destructor - RAII automatically closes socket
143     // noexcept because destructors should not throw
144     ~Socket() noexcept {
145         if (is_valid_ && fd_ != INVALID_SOCKET_VALUE) {
146             std::cout << " Socket closing (fd=" << fd_ << ")" << std::endl;
147             CLOSE_SOCKET(fd_);
148         }
149     }
150
151     // Delete copy operations - sockets are unique resources
152     Socket(const Socket&) = delete;
153     Socket& operator=(const Socket&) = delete;
154
155     // Move operations - transfer ownership
```

```
156     Socket(Socket&& other) noexcept
157     : fd_(other.fd_), is_valid_(other.is_valid_) {
158     other.fd_ = INVALID_SOCKET_VALUE;
159     other.is_valid_ = false;
160 }
161
162     Socket& operator=(Socket&& other) noexcept {
163     if (this != &other) {
164         if (is_valid_) {
165             CLOSE_SOCKET(fd_);
166         }
167         fd_ = other.fd_;
168         is_valid_ = other.is_valid_;
169         other.fd_ = INVALID_SOCKET_VALUE;
170         other.is_valid_ = false;
171     }
172     return *this;
173 }
174
175 // Check if socket is valid
176 // [[nodiscard]] - result should not be ignored!
177 [[nodiscard]] bool is_valid() const noexcept {
178     return is_valid_;
179 }
180
181 // Get raw socket descriptor
182 // [[nodiscard]] - caller needs this value!
183 [[nodiscard]] socket_t get() const noexcept {
184     return fd_;
185 }
186
187 // Bind to address
188 // [[nodiscard]] - error must be checked!
189 [[nodiscard]] SocketError bind(const std::string& address, uint16_t port)
190     noexcept {
191     if (!is_valid_) {
192         return SocketError::InvalidSocket;
193     }
194
195     sockaddr_in addr{};
196     addr.sin_family = AF_INET;
197     addr.sin_port = htons(port);
198
199     if (address.empty() || address == "0.0.0.0") {
200         addr.sin_addr.s_addr = INADDR_ANY;
201     } else {
202         addr.sin_addr.s_addr = inet_addr(address.c_str());
203     }
204
205     if (::bind(fd_, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) < 0)
206     {
207         return SocketError::BindFailed;
208     }
209 }
```

```
208     std::cout << " Socket bound to " << address << ":" << port << std::endl;
209     return SocketError::Success;
210 }
211
212 // Listen for connections
213 // [[nodiscard]] - error must be checked!
214 [[nodiscard]] SocketError listen(int backlog = 10) noexcept {
215     if (!is_valid_) {
216         return SocketError::InvalidSocket;
217     }
218
219     if (::listen(fd_, backlog) < 0) {
220         return SocketError::ListenFailed;
221     }
222
223     std::cout << " Socket listening (backlog=" << backlog << ")" << std::endl;
224     return SocketError::Success;
225 }
226
227 // Accept connection - returns new Socket
228 // [[nodiscard]] - must handle the result!
229 [[nodiscard]] Result<Socket> accept() noexcept {
230     if (!is_valid_) {
231         return std::nullopt;
232     }
233
234     sockaddr_in client_addr{};
235     socklen_t addr_len = sizeof(client_addr);
236
237     socket_t client_fd = ::accept(fd_, reinterpret_cast<sockaddr*>(&client_addr), &addr_len);
238
239     if (client_fd == INVALID_SOCKET_VALUE) {
240         return std::nullopt;
241     }
242
243     char client_ip[INET_ADDRSTRLEN];
244     inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, INET_ADDRSTRLEN);
245     std::cout << " Connection accepted from " << client_ip
246             << ":" << ntohs(client_addr.sin_port) << std::endl;
247
248     return Socket(Socket::from_fd, client_fd);
249 }
250
251 // Connect to remote address
252 // [[nodiscard]] - error must be checked!
253 [[nodiscard]] SocketError connect(const std::string& address, uint16_t port) noexcept {
254     if (!is_valid_) {
255         return SocketError::InvalidSocket;
256     }
257 }
```

```
258     sockaddr_in addr{};
259     addr.sin_family = AF_INET;
260     addr.sin_port = htons(port);
261     addr.sin_addr.s_addr = inet_addr(address.c_str());
262
263     if (::connect(fd_, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) <
264         0) {
265         return SocketError::ConnectFailed;
266     }
267
268     std::cout << " Connected to " << address << ":" << port << std::endl;
269     return SocketError::Success;
270 }
271
272 // Send data
273 // [[nodiscard]] - must check if send succeeded!
274 [[nodiscard]] Result<size_t> send(const std::string& data) noexcept {
275     if (!is_valid_) {
276         return std::nullopt;
277     }
278
279     ssize_t sent = ::send(fd_, data.c_str(), data.length(), 0);
280
281     if (sent < 0) {
282         return std::nullopt;
283     }
284
285     std::cout << " Sent " << sent << " bytes" << std::endl;
286     return static_cast<size_t>(sent);
287 }
288
289 // Receive data
290 // [[nodiscard]] - must handle received data!
291 [[nodiscard]] Result<std::string> receive(size_t max_length = 4096)
292     noexcept {
293     if (!is_valid_) {
294         return std::nullopt;
295     }
296
297     std::vector<char> buffer(max_length);
298     ssize_t received = ::recv(fd_, buffer.data(), buffer.size(), 0);
299
300     if (received <= 0) {
301         return std::nullopt;
302     }
303
304     std::cout << " Received " << received << " bytes" << std::endl;
305     return std::string(buffer.data(), received);
306 }
307
308 // Set socket option
309 // [[nodiscard]] - error must be checked!
310 [[nodiscard]] SocketError set_reuse_address(bool enable) noexcept {
311     if (!is_valid_) {
```

```
310         return SocketError::InvalidSocket;
311     }
312
313     int opt = enable ? 1 : 0;
314     if (setsockopt(fd_, SOL_SOCKET, SO_REUSEADDR,
315                     reinterpret_cast<const char*>(&opt), sizeof(opt)) < 0) {
316         return SocketError::SocketOptionFailed;
317     }
318
319     std::cout << " SO_REUSEADDR set to " << (enable ? "true" : "false")
320     << std::endl;
321     return SocketError::Success;
322 }
323
324 // =====
325 // SECTION 4: HIGH-LEVEL TCP SERVER CLASS
326 // =====
327
328 class TcpServer {
329 private:
330     Socket listen_socket_;
331     uint16_t port_;
332     bool is_running_;
333
334 public:
335     // Constructor
336     // noexcept(false) - may throw if initialization fails critically
337     explicit TcpServer(uint16_t port)
338         : listen_socket_(AF_INET, SOCK_STREAM, 0),
339           port_(port),
340           is_running_(false) {
341
342     if (!listen_socket_.is_valid()) {
343         throw std::runtime_error("Failed to create listen socket");
344     }
345 }
346
347     // Start server
348     // [[nodiscard]] - must check if server started successfully!
349     [[nodiscard]] SocketError start() noexcept {
350         // Enable address reuse
351         auto err = listen_socket_.set_reuse_address(true);
352         if (err != SocketError::Success) {
353             return err;
354         }
355
356         // Bind to port
357         err = listen_socket_.bind("0.0.0.0", port_);
358         if (err != SocketError::Success) {
359             return err;
360         }
361
362         // Start listening
```

```
363     err = listen_socket_.listen();
364     if (err != SocketError::Success) {
365         return err;
366     }
367
368     is_running_ = true;
369     std::cout << " TCP Server started on port " << port_ << std::endl;
370     return SocketError::Success;
371 }
372
373 // Accept one client connection
374 // [[nodiscard]] - must handle the client socket!
375 [[nodiscard]] Result<Socket> accept_client() noexcept {
376     if (!is_running_) {
377         return std::nullopt;
378     }
379     return listen_socket_.accept();
380 }
381
382 // Check if server is running
383 [[nodiscard]] bool is_running() const noexcept {
384     return is_running_;
385 }
386
387 // Stop server
388 void stop() noexcept {
389     is_running_ = false;
390     std::cout << " TCP Server stopped" << std::endl;
391 }
392 };
393
394 // =====
395 // SECTION 5: HIGH-LEVEL TCP CLIENT CLASS
396 // =====
397
398 class TcpClient {
399 private:
400     Socket socket_;
401     bool is_connected_;
402
403 public:
404     // Constructor
405     TcpClient()
406         : socket_(AF_INET, SOCK_STREAM, 0),
407           is_connected_(false) {}
408
409     // Connect to server
410     // [[nodiscard]] - must check connection status!
411     [[nodiscard]] SocketError connect(const std::string& host, uint16_t port)
412         noexcept {
413         if (!socket_.is_valid()) {
414             return SocketError::InvalidSocket;
415         }
416     }
417 }
```

```
416     auto err = socket_.connect(host, port);
417     if (err == SocketError::Success) {
418         is_connected_ = true;
419     }
420     return err;
421 }
422
423 // Send message
424 // [[nodiscard]] - must check if send succeeded!
425 [[nodiscard]] Result<size_t> send(const std::string& message) noexcept {
426     if (!is_connected_) {
427         return std::nullopt;
428     }
429     return socket_.send(message);
430 }
431
432 // Receive message
433 // [[nodiscard]] - must handle received data!
434 [[nodiscard]] Result<std::string> receive() noexcept {
435     if (!is_connected_) {
436         return std::nullopt;
437     }
438     return socket_.receive();
439 }
440
441 // Check if connected
442 [[nodiscard]] bool is_connected() const noexcept {
443     return is_connected_;
444 }
445 };
446
447 // =====
448 // SECTION 6: DEMONSTRATION OF noexcept USAGE
449 // =====
450
451 void demonstrate_noexcept() {
452     std::cout << "\n== NOEXCEPT USAGE ==" << std::endl;
453
454     std::cout << "\n SAFE noexcept usage:" << std::endl;
455     std::cout << " • Getters that don't throw: is_valid() noexcept" << std::endl;
456     std::cout << " • Destructors: ~Socket() noexcept" << std::endl;
457     std::cout << " • Move operations: Socket(Socket&&) noexcept" << std::endl;
458     std::cout << " • Simple checks: error_to_string() noexcept" << std::endl;
459     ;
460
461     std::cout << "\n Operations marked noexcept (but handle errors
462     internally):" << std::endl;
463     std::cout << " • bind() noexcept - returns error code instead of
464     throwing" << std::endl;
465     std::cout << " • send() noexcept - returns std::optional for errors" <<
466     std::endl;
467     std::cout << " • receive() noexcept - returns std::optional" << std::endl;
```

```
        endl;

464     std::cout << "\n NOT noexcept (may throw):" << std::endl;
465     std::cout << " • Constructor if critical initialization fails" << std::
466         endl;
467     std::cout << " • std::string operations (may throw std::bad_alloc)" <<
468         std::endl;

469     std::cout << "\n KEY PRINCIPLE:" << std::endl;
470     std::cout << "    Mark noexcept when you guarantee no exceptions will
471         escape" << std::endl;
472     std::cout << "    Use error codes/std::optional for expected failures" <<
473         std::endl;
474 }

475 // =====
476 // SECTION 7: DEMONSTRATION OF [[nodiscard]]
477 // =====

478 void demonstrate_nodiscard() {
479     std::cout << "\n== [[nodiscard]] ATTRIBUTE ==" << std::endl;
480
481     std::cout << "\n [[nodiscard]] forces checking return values:" << std::
482         endl;
483
484     Socket sock(AF_INET, SOCK_STREAM, 0);
485
486     // GOOD: Checking the return value
487     if (auto err = sock.bind("0.0.0.0", 8080); err != SocketError::Success) {
488         std::cout << "    Bind error checked: " << error_to_string(err) << std
489             ::endl;
490     }
491
492     // BAD: Would produce compiler warning (if we uncommented)
493     // sock.bind("0.0.0.0", 8080); // Warning: ignoring return value!
494
495     std::cout << "\n BENEFITS:" << std::endl;
496     std::cout << "    Prevents forgetting to check errors" << std::endl;
497     std::cout << "    Compiler warns about ignored return values" << std::
498         endl;
499     std::cout << "    Makes APIs safer to use" << std::endl;
500     std::cout << "    Self-documenting - shows value is important" << std::
501         endl;
502 }

503 // =====
504 // SECTION 8: RAII BENEFITS DEMONSTRATION
505 // =====

506 void demonstrate_raii() {
507     std::cout << "\n== RAI AUTOMATIC RESOURCE MANAGEMENT ==" << std::endl;
508
509     std::cout << "\nCreating socket in scope:" << std::endl;
510 }
```

```
509     Socket sock(AF_INET, SOCK_STREAM, 0);
510     auto err = sock.bind("0.0.0.0", 9999);
511     if (err == SocketError::Success) {
512         std::cout << "    Socket is active (fd=" << sock.get() << ")" <<
513                     std::endl;
514     }
515     // No need to manually close - destructor handles it!
516 }
517 std::cout << "    Socket automatically closed when out of scope!" << std
518     ::endl;
519
520 std::cout << "\n RAII GUARANTEES:" << std::endl;
521 std::cout << "    Resource acquired in constructor" << std::endl;
522 std::cout << "    Resource released in destructor" << std::endl;
523 std::cout << "    Exception-safe - cleanup always happens" << std::endl;
524 std::cout << "    No manual cleanup needed" << std::endl;
525 std::cout << "    No resource leaks possible" << std::endl;
526 }
527
528 // =====
529 // SECTION 9: EXTERN "C" USAGE
530 // =====
531
532 void demonstrate_extern_c() {
533     std::cout << "\n==== EXTERN \"C\" LINKAGE ===" << std::endl;
534
535     std::cout << "\n Purpose: Interface with C libraries" << std::endl;
536     std::cout << " • Prevents C++ name mangling" << std::endl;
537     std::cout << " • Allows C code to call C++ functions" << std::endl;
538     std::cout << " • Enables linking with C libraries" << std::endl;
539
540     // Call our C function
541     int port = 8080;
542     int is_valid = c_validate_port(port);
543     std::cout << "\n Called C function: c_validate_port(" << port << ") = "
544                 << (is_valid ? "valid" : "invalid") << std::endl;
545
546     std::cout << "\n USAGE PATTERNS:" << std::endl;
547     std::cout << "    extern \"C\" { ... }      - Wrap C declarations" << std
548                     ::endl;
549     std::cout << "    #ifdef __cplusplus      - Conditional compilation" << std
550                     ::endl;
551     std::cout << "    System headers (socket.h) already use extern \"C\""
552                     << std::endl;
553 }
554
555 // =====
556 // SECTION 10: PRACTICAL EXAMPLES
557 // =====
558
559 void example_echo_server() {
560     std::cout << "\n==== EXAMPLE: SIMPLE ECHO SERVER ===" << std::endl;
561     std::cout << "Simulating echo server workflow...\n" << std::endl;
562 }
```

```
558     try {
559         TcpServer server(12345);
560
561         // Start server
562         if (auto err = server.start(); err != SocketError::Success) {
563             std::cout << "Failed to start server: " << error_to_string(err) <<
564             std::endl;
565             return;
566         }
567
568         std::cout << "Echo server would accept connections and echo messages
569             ..." << std::endl;
570         std::cout << "(Actual network operations skipped for demo)" << std::
571             endl;
572
573         server.stop();
574
575     } catch (const std::exception& e) {
576         std::cout << "Exception: " << e.what() << std::endl;
577     }
578
579 // =====
580 // MAIN FUNCTION
581 // =====
582
583 int main() {
584     std::cout << "\n";
585     std::cout << "                                         \n";
586     std::cout << "             WRAPPING C LIBRARIES IN MODERN C++ \n";
587     std::cout << "                                         \n";
588     std::cout << "     Demonstrates: RAI, noexcept, [[nodiscard]], extern \"C\" \n";
589     std::cout << "     Example: TCP/UDP Socket Wrapper \n";
590     std::cout << "                                         \n";
591
592 #ifdef _WIN32
593     // Initialize Winsock on Windows
594     WSADATA wsaData;
595     if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
596         std::cerr << "WSAStartup failed" << std::endl;
597         return 1;
598     }
599     std::cout << "\n Winsock initialized (Windows)" << std::endl;
600 #else
601     std::cout << "\n Using POSIX sockets (Linux/Unix)" << std::endl;
602 #endif
603 }
```

```
604 demonstrate_raii();
605 demonstrate_noexcept();
606 demonstrate_nodiscard();
607 demonstrate_extern_c();
608 example_echo_server();

609
610 std::cout << "\n" << std::string(70, '=') << std::endl;
611 std::cout << "BEST PRACTICES SUMMARY:\n";
612 std::cout << std::string(70, '=') << std::endl;
613
614 std::cout << "\n1. RAI (Resource Acquisition Is Initialization):" << std
615     ::endl;
616 std::cout << "    Acquire resources in constructor" << std::endl;
617 std::cout << "    Release resources in destructor" << std::endl;
618 std::cout << "    Make classes non-copyable for unique resources" << std
619     ::endl;
620 std::cout << "    Implement move semantics for transfer" << std::endl;
621
622 std::cout << "\n2. noexcept Usage:" << std::endl;
623 std::cout << "    Mark destructors noexcept (implicit in C++11+)" << std
624     ::endl;
625 std::cout << "    Mark move operations noexcept (enables optimizations)" <<
626     std::endl;
627 std::cout << "    Mark operations that handle errors via return codes" <<
628     std::endl;
629 std::cout << "    DON'T mark operations that may allocate memory" << std
630     ::endl;
631
632 std::cout << "\n3. [[nodiscard]] Usage:" << std::endl;
633 std::cout << "    Use for error codes that must be checked" << std::endl;
634 std::cout << "    Use for expensive operations (no wasted work)" << std:
635     endl;
636 std::cout << "    Use for functions where ignoring result is likely a bug
637     " << std::endl;
638 std::cout << "    Makes APIs self-documenting and safer" << std::endl;
639
640 std::cout << "\n4. extern \"C\" Linkage:" << std::endl;
641 std::cout << "    Wrap C library includes in extern \"C\" blocks" << std
642     ::endl;
643 std::cout << "    Use #ifdef __cplusplus for C/C++ compatibility" << std
644     ::endl;
645 std::cout << "    Only C-compatible functions can be extern \"C\"" << std
646     ::endl;
647 std::cout << "    No overloading, no classes, no templates" << std::endl;
648
649 std::cout << "\n5. C/C++ Interop Patterns:" << std::endl;
650 std::cout << "    Create thin C++ wrapper classes (RAII)" << std::endl;
651 std::cout << "    Hide C types behind strong C++ types" << std::endl;
652 std::cout << "    Use std::optional for nullable results" << std::endl;
653 std::cout << "    Use error codes/exceptions instead of C error globals"
654     << std::endl;
655 std::cout << "    Provide modern C++ interfaces (string_view, span, etc.)"
656     << std::endl;
```

```
645     std::cout << "\n6. Modern C++ Features for C Wrappers:" << std::endl;
646     std::cout << "    std::optional<T> for operations that may fail" << std::
647         endl;
647     std::cout << "    std::string_view for non-owning string parameters" <<
648         std::endl;
648     std::cout << "    std::span<T> for array views (C++20)" << std::endl;
649     std::cout << "    std::unique_ptr with custom deleter" << std::endl;
650     std::cout << "    enum class for type-safe error codes" << std::endl;
651
652     std::cout << "\n All socket resources properly cleaned up by RAII!\n" <<
652         std::endl;
653
654 #ifdef _WIN32
655     WSACleanup();
656     std::cout << " Winsock cleaned up\n" << std::endl;
657 #endif
658
659     return 0;
660 }
```

## 18 Source Code: CreatingCApiFromCpp.cpp

File: src/CreatingCApiFromCpp.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // CREATING A C API FROM C++ CODE
3 // =====
4 // This example demonstrates how to expose C++ functionality through
5 // a C-compatible API. This is the REVERSE direction from the previous
6 // example - we're wrapping C++ in C, not C in C++.
7 //
8 // USE CASES:
9 // - Creating libraries usable from C code
10 // - Plugin systems with C interfaces
11 // - FFI (Foreign Function Interface) for Python, Rust, etc.
12 // - Legacy code integration
13 // - Stable ABI across compiler versions
14 //
15 // TOPICS COVERED:
16 // 1. Opaque pointers (pimpl idiom for C API)
17 // 2. extern "C" for C linkage
18 // 3. Exception handling across C boundary
19 // 4. C++ features hidden behind C API
20 // 5. Memory management strategies
21 // 6. Header guards and C/C++ compatibility
22 // =====
23
24 #include <iostream>
25 #include <string>
26 #include <vector>
27 #include <memory>
28 #include <cstring>
29 #include <stdexcept>
30 #include <algorithm>
31 #include <cstdint>
32
33 // =====
34 // SECTION 1: C++ IMPLEMENTATION (INTERNAL)
35 // =====
36 // This is the actual C++ code with modern features that we'll expose
37 // through a C API
38
39 namespace image_processing {
40
41 class Image {
42 private:
43     size_t width_;
44     size_t height_;
45     std::vector<uint8_t> pixels_;
46
47 public:
48     Image(size_t width, size_t height)
49         : width_(width), height_(height), pixels_(width * height, 0) {
```

```
50     std::cout << " C++ Image created (" << width_ << "x" << height_ << ")"
51     " << std::endl;
52 }
53 ~Image() {
54     std::cout << " C++ Image destroyed" << std::endl;
55 }
56
57 // C++ features: const correctness, exceptions
58 size_t width() const noexcept { return width_; }
59 size_t height() const noexcept { return height_; }
60
61 uint8_t& at(size_t x, size_t y) {
62     if (x >= width_ || y >= height_) {
63         throw std::out_of_range("Pixel coordinates out of range");
64     }
65     return pixels_[y * width_ + x];
66 }
67
68 const uint8_t& at(size_t x, size_t y) const {
69     if (x >= width_ || y >= height_) {
70         throw std::out_of_range("Pixel coordinates out of range");
71     }
72     return pixels_[y * width_ + x];
73 }
74
75 // Modern C++ algorithm
76 void invert() {
77     std::transform(pixels_.begin(), pixels_.end(), pixels_.begin(),
78                   [] (uint8_t pixel) { return 255 - pixel; });
79     std::cout << " Image inverted (C++ algorithm)" << std::endl;
80 }
81
82 void fill(uint8_t value) {
83     std::fill(pixels_.begin(), pixels_.end(), value);
84     std::cout << " Image filled with value " << static_cast<int>(value)
85     << std::endl;
86 }
87
88 void apply_threshold(uint8_t threshold) {
89     for (auto& pixel : pixels_) {
90         pixel = (pixel >= threshold) ? 255 : 0;
91     }
92     std::cout << " Threshold applied at " << static_cast<int>(threshold)
93     << std::endl;
94 }
95
96 // Get raw data
97 const uint8_t* data() const noexcept { return pixels_.data(); }
98 size_t size() const noexcept { return pixels_.size(); }
99
100 } // namespace image_processing
```

```
101 // =====
102 // SECTION 2: C API HEADER (WHAT C CODE SEES)
103 // =====
104 // This is what would go in a .h file that C programs include
105
106 #ifdef __cplusplus
107 extern "C" {
108 #endif
109
110 // Opaque handle - C code sees this as incomplete type
111 // This hides the C++ implementation details
112 typedef struct ImageHandle* ImageHandle_t;
113
114 // Error codes (C-compatible enum)
115 typedef enum {
116     IMAGE_SUCCESS = 0,
117     IMAGE_ERROR_INVALID_HANDLE = -1,
118     IMAGE_ERROR_INVALID_DIMENSIONS = -2,
119     IMAGE_ERROR_OUT_OF_RANGE = -3,
120     IMAGE_ERROR_OUT_OF_MEMORY = -4,
121     IMAGE_ERROR_UNKNOWN = -99
122 } ImageError;
123
124 // C API functions
125 // Note: All return error codes, use output parameters for data
126
127 /**
128 * Create a new image
129 * @param width Image width in pixels
130 * @param height Image height in pixels
131 * @param out_handle Output parameter for image handle
132 * @return Error code
133 */
134 ImageError image_create(size_t width, size_t height, ImageHandle_t* out_handle
135 );
136 /**
137 * Destroy an image and free resources
138 * @param handle Image handle
139 * @return Error code
140 */
141 ImageError image_destroy(ImageHandle_t handle);
142
143 /**
144 * Get image dimensions
145 * @param handle Image handle
146 * @param out_width Output parameter for width
147 * @param out_height Output parameter for height
148 * @return Error code
149 */
150 ImageError image_get_dimensions(ImageHandle_t handle, size_t* out_width,
151                                 size_t* out_height);
152 /**
153 */
```

```
153 * Set pixel value
154 * @param handle Image handle
155 * @param x X coordinate
156 * @param y Y coordinate
157 * @param value Pixel value (0-255)
158 * @return Error code
159 */
160 ImageError image_set_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t
161 value);
162 /**
163 * Get pixel value
164 * @param handle Image handle
165 * @param x X coordinate
166 * @param y Y coordinate
167 * @param out_value Output parameter for pixel value
168 * @return Error code
169 */
170 ImageError image_get_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t*
171 out_value);
172 /**
173 * Invert all pixel values
174 * @param handle Image handle
175 * @return Error code
176 */
177 ImageError image_invert(ImageHandle_t handle);
178 /**
179 * Fill image with a value
180 * @param handle Image handle
181 * @param value Fill value (0-255)
182 * @return Error code
183 */
184 ImageError image_fill(ImageHandle_t handle, uint8_t value);
185 /**
186 * Apply threshold to image
187 * @param handle Image handle
188 * @param threshold Threshold value (0-255)
189 * @return Error code
190 */
191 ImageError image_apply_threshold(ImageHandle_t handle, uint8_t threshold);
192 /**
193 * Get error message for error code
194 * @param error Error code
195 * @return Human-readable error message (static string)
196 */
197 const char* image_error_string(ImageError error);
198
199 #ifdef __cplusplus
200 }
201 #endif
```

```
205 // =====
206 // SECTION 3: C API IMPLEMENTATION (BRIDGE LAYER)
207 // =====
208 // This bridges between C and C++ code
209
210 // Helper function to catch C++ exceptions and convert to error codes
211 // NOTE: Must be OUTSIDE extern "C" block since it's a template
212 template<typename Func>
213 ImageError safe_call(Func&& func) noexcept {
214     try {
215         func();
216         return IMAGE_SUCCESS;
217     } catch (const std::out_of_range& e) {
218         std::cerr << "Exception caught: " << e.what() << std::endl;
219         return IMAGE_ERROR_OUT_OF_RANGE;
220     } catch (const std::bad_alloc& e) {
221         std::cerr << "Exception caught: " << e.what() << std::endl;
222         return IMAGE_ERROR_OUT_OF_MEMORY;
223     } catch (const std::exception& e) {
224         std::cerr << "Exception caught: " << e.what() << std::endl;
225         return IMAGE_ERROR_UNKNOWN;
226     } catch (...) {
227         std::cerr << "Unknown exception caught" << std::endl;
228         return IMAGE_ERROR_UNKNOWN;
229     }
230 }
231
232 // The actual implementation uses extern "C"
233 extern "C" {
234
235     ImageError image_create(size_t width, size_t height, ImageHandle_t* out_handle
236     ) {
237         if (!out_handle) {
238             return IMAGE_ERROR_INVALID_HANDLE;
239         }
240
241         if (width == 0 || height == 0) {
242             return IMAGE_ERROR_INVALID_DIMENSIONS;
243         }
244
245         return safe_call([&]() {
246             // Create C++ object and cast to opaque handle
247             auto* img = new image_processing::Image(width, height);
248             *out_handle = reinterpret_cast<ImageHandle_t>(img);
249         });
250     }
251
252     ImageError image_destroy(ImageHandle_t handle) {
253         if (!handle) {
254             return IMAGE_ERROR_INVALID_HANDLE;
255         }
256
257         return safe_call([&]() {
```

```
258     // Cast back to C++ object and delete
259     auto* img = reinterpret_cast<image_processing::Image*>(handle);
260     delete img;
261 };
262 }
263
264 ImageError image_get_dimensions(ImageHandle_t handle, size_t* out_width,
265     size_t* out_height) {
266     if (!handle || !out_width || !out_height) {
267         return IMAGE_ERROR_INVALID_HANDLE;
268     }
269
270     return safe_call([&]() {
271         auto* img = reinterpret_cast<image_processing::Image*>(handle);
272         *out_width = img->width();
273         *out_height = img->height();
274     });
275 }
276
277 ImageError image_set_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t
278     value) {
279     if (!handle) {
280         return IMAGE_ERROR_INVALID_HANDLE;
281     }
282
283     return safe_call([&]() {
284         auto* img = reinterpret_cast<image_processing::Image*>(handle);
285         img->at(x, y) = value;
286     });
287 }
288
289 ImageError image_get_pixel(ImageHandle_t handle, size_t x, size_t y, uint8_t*
290     out_value) {
291     if (!handle || !out_value) {
292         return IMAGE_ERROR_INVALID_HANDLE;
293     }
294
295     return safe_call([&]() {
296         auto* img = reinterpret_cast<image_processing::Image*>(handle);
297         *out_value = img->at(x, y);
298     });
299 }
300
301 ImageError image_invert(ImageHandle_t handle) {
302     if (!handle) {
303         return IMAGE_ERROR_INVALID_HANDLE;
304     }
305
306     return safe_call([&]() {
307         auto* img = reinterpret_cast<image_processing::Image*>(handle);
308         img->invert();
309     });
310 }
```

```
309 ImageError image_fill(ImageHandle_t handle, uint8_t value) {
310     if (!handle) {
311         return IMAGE_ERROR_INVALID_HANDLE;
312     }
313
314     return safe_call([&]() {
315         auto* img = reinterpret_cast<image_processing::Image*>(handle);
316         img->fill(value);
317     });
318 }
319
320 ImageError image_apply_threshold(ImageHandle_t handle, uint8_t threshold) {
321     if (!handle) {
322         return IMAGE_ERROR_INVALID_HANDLE;
323     }
324
325     return safe_call([&]() {
326         auto* img = reinterpret_cast<image_processing::Image*>(handle);
327         img->apply_threshold(threshold);
328     });
329 }
330
331 const char* image_error_string(ImageError error) {
332     switch (error) {
333         case IMAGE_SUCCESS: return "Success";
334         case IMAGE_ERROR_INVALID_HANDLE: return "Invalid handle";
335         case IMAGE_ERROR_INVALID_DIMENSIONS: return "Invalid dimensions";
336         case IMAGE_ERROR_OUT_OF_RANGE: return "Coordinates out of range";
337         case IMAGE_ERROR_OUT_OF_MEMORY: return "Out of memory";
338         case IMAGE_ERROR_UNKNOWN: return "Unknown error";
339         default: return "Invalid error code";
340     }
341 }
342
343 } // extern "C"
344
345 // =====
346 // SECTION 4: DEMONSTRATION - USING THE C API
347 // =====
348
349 void demonstrate_c_api_usage() {
350     std::cout << "\n== USING C API (SIMULATING C CODE) ==" << std::endl;
351     std::cout << "Note: This C++ code simulates how C code would use the API\n"
352         " << std::endl;
353
354     // In real C code, these would be the only includes needed:
355     // #include "image_api.h"
356
357     ImageHandle_t image = nullptr;
358     ImageError err;
359
360     // Create image
361     std::cout << "1. Creating 10x10 image:" << std::endl;
362     err = image_create(10, 10, &image);
```

```
362     if (err != IMAGE_SUCCESS) {
363         std::cout << "    Error: " << image_error_string(err) << std::endl;
364         return;
365     }
366     std::cout << "    Image created successfully" << std::endl;
367
368     // Get dimensions
369     size_t width, height;
370     err = image_get_dimensions(image, &width, &height);
371     if (err == IMAGE_SUCCESS) {
372         std::cout << "\n2. Image dimensions: " << width << "x" << height <<
373             std::endl;
374     }
375
376     // Fill with value
377     std::cout << "\n3. Filling image with value 128:" << std::endl;
378     err = image_fill(image, 128);
379     if (err != IMAGE_SUCCESS) {
380         std::cout << "    Error: " << image_error_string(err) << std::endl;
381     }
382
383     // Set some pixels
384     std::cout << "\n4. Setting individual pixels:" << std::endl;
385     image_set_pixel(image, 0, 0, 255);
386     image_set_pixel(image, 1, 1, 200);
387     image_set_pixel(image, 2, 2, 150);
388     std::cout << "    Pixels set" << std::endl;
389
390     // Get pixel values
391     std::cout << "\n5. Reading pixel values:" << std::endl;
392     uint8_t value;
393     for (int i = 0; i < 3; i++) {
394         err = image_get_pixel(image, i, i, &value);
395         if (err == IMAGE_SUCCESS) {
396             std::cout << "    Pixel(" << i << "," << i << ") = "
397                 << static_cast<int>(value) << std::endl;
398         }
399     }
400
401     // Invert
402     std::cout << "\n6. Inverting image:" << std::endl;
403     err = image_invert(image);
404     if (err != IMAGE_SUCCESS) {
405         std::cout << "    Error: " << image_error_string(err) << std::endl;
406     }
407
408     // Read inverted values
409     std::cout << "\n7. Reading inverted pixel values:" << std::endl;
410     for (int i = 0; i < 3; i++) {
411         err = image_get_pixel(image, i, i, &value);
412         if (err == IMAGE_SUCCESS) {
413             std::cout << "    Pixel(" << i << "," << i << ") = "
414                 << static_cast<int>(value) << std::endl;
415     }
```

```
415     }
416
417     // Apply threshold
418     std::cout << "\n8. Applying threshold at 100:" << std::endl;
419     err = image_apply_threshold(image, 100);
420
421     // Error handling
422     std::cout << "\n9. Testing error handling (out of range access):" << std::endl;
423     err = image_get_pixel(image, 100, 100, &value);
424     if (err != IMAGE_SUCCESS) {
425         std::cout << "    Error caught: " << image_error_string(err) << std::endl;
426     }
427
428     // Cleanup
429     std::cout << "\n10. Destroying image:" << std::endl;
430     err = image_destroy(image);
431     if (err == IMAGE_SUCCESS) {
432         std::cout << "    Image destroyed successfully" << std::endl;
433     }
434 }
435
436 // =====
437 // SECTION 5: BEST PRACTICES EXPLANATION
438 // =====
439
440 void explain_best_practices() {
441     std::cout << "\n" << std::string(70, '=') << std::endl;
442     std::cout << "BEST PRACTICES FOR C API FROM C++ CODE:\n";
443     std::cout << std::string(70, '=') << std::endl;
444
445     std::cout << "\n1. OPAQUE HANDLES (PIMPL IDIOM FOR C):" << std::endl;
446     std::cout << "    Hide C++ implementation details" << std::endl;
447     std::cout << "    Use typedef struct Name* Handle_t;" << std::endl;
448     std::cout << "    Cast to/from C++ objects internally" << std::endl;
449     std::cout << "    Stable ABI - C++ changes don't affect C API" << std::endl;
450
451     std::cout << "\n2. EXCEPTION HANDLING:" << std::endl;
452     std::cout << "    NEVER let C++ exceptions cross C boundary!" << std::endl;
453     std::cout << "    Catch ALL exceptions in extern \"C\" functions" << std::endl;
454     std::cout << "    Convert exceptions to error codes" << std::endl;
455     std::cout << "    Use try-catch wrappers consistently" << std::endl;
456
457     std::cout << "\n3. ERROR HANDLING:" << std::endl;
458     std::cout << "    Return error codes (int or enum)" << std::endl;
459     std::cout << "    Use output parameters for data" << std::endl;
460     std::cout << "    Provide error_to_string() function" << std::endl;
461     std::cout << "    Check for NULL pointers before dereferencing" << std::endl;
462 }
```

```

463     std::cout << "\n4. MEMORY MANAGEMENT:" << std::endl;
464     std::cout << "    C code creates/destroys via API functions" << std::endl
465     ;
466     std::cout << "    NEVER expose C++ new/delete directly" << std::endl;
467     std::cout << "    Provide create() and destroy() functions" << std::endl;
468     std::cout << "    Match create/destroy across DLL boundaries" << std::endl;
469
470     std::cout << "\n5. HEADER GUARDS AND COMPATIBILITY:" << std::endl;
471     std::cout << "    Use #ifdef __cplusplus for extern \"C\" blocks" << std::endl;
472     std::cout << "    Include guards in all headers" << std::endl;
473     std::cout << "    Use C-compatible types (no bool, use int)" << std::endl;
474     std::cout << "    No function overloading in C API" << std::endl;
475
476     std::cout << "\n6. C++ FEATURES TO AVOID IN C API:" << std::endl;
477     std::cout << "    Classes (use opaque handles instead)" << std::endl;
478     std::cout << "    Templates (not C-compatible)" << std::endl;
479     std::cout << "    Function overloading (C doesn't support it)" << std::endl;
480     std::cout << "    Default arguments (C doesn't support them)" << std::endl;
481     std::cout << "    References (use pointers)" << std::endl;
482     std::cout << "    bool type (use int, 0 for false, 1 for true)" << std::endl;
483
484     std::cout << "\n7. DOCUMENTATION:" << std::endl;
485     std::cout << "    Document ownership semantics clearly" << std::endl;
486     std::cout << "    Specify thread-safety guarantees" << std::endl;
487     std::cout << "    List all possible error codes" << std::endl;
488     std::cout << "    Provide usage examples" << std::endl;
489
490     std::cout << "\n8. TESTING:" << std::endl;
491     std::cout << "    Test C API from actual C code" << std::endl;
492     std::cout << "    Verify exception safety" << std::endl;
493     std::cout << "    Test error conditions" << std::endl;
494     std::cout << "    Check for memory leaks" << std::endl;
495 }
496
497 // =====
498 // SECTION 6: COMPARISON WITH PREVIOUS EXAMPLE
499 // =====
500 void compare_approaches() {
501     std::cout << "\n" << std::string(70, '=') << std::endl;
502     std::cout << "COMPARISON: C IN C++ vs C++ IN C:\n";
503     std::cout << std::string(70, '=') << std::endl;
504
505     std::cout << "\nPREVIOUS EXAMPLE (Wrapping C in C++):" << std::endl;
506     std::cout << "    Purpose: Use C libraries from C++ code" << std::endl;
507     std::cout << "    Technique: RAII wrappers, smart pointers" << std::endl;
508     std::cout << "    Example: Socket library → C++ Socket class" << std::endl;
509     std::cout << "    Benefits: Modern C++ safety, automatic cleanup" << std::endl;

```

```
        endl;

510     std::cout << "\nTHIS EXAMPLE (Exposing C++ as C):" << std::endl;
511     std::cout << "  Purpose: Make C++ code usable from C" << std::endl;
512     std::cout << "  Technique: Opaque handles, extern \"C\" " << std::endl;
513     std::cout << "  Example: C++ Image class → C image_* functions" << std::endl;
514     std::cout << "  Benefits: C compatibility, stable ABI, FFI-ready" << std::endl;
515
516     std::cout << "\nWHEN TO USE EACH APPROACH:" << std::endl;
517     std::cout << "\nWrapping C in C++ (Previous):" << std::endl;
518     std::cout << "  • You have C libraries to use" << std::endl;
519     std::cout << "  • You want modern C++ features" << std::endl;
520     std::cout << "  • You want automatic resource management" << std::endl;
521     std::cout << "  • Internal project use" << std::endl;
522
523     std::cout << "\nExposing C++ as C (This):" << std::endl;
524     std::cout << "  • You want to create a C-compatible library" << std::endl;
525     std::cout << "  • You need stable ABI across versions" << std::endl;
526     std::cout << "  • You want FFI for other languages" << std::endl;
527     std::cout << "  • You need to support legacy C code" << std::endl;
528     std::cout << "  • Plugin systems with C interfaces" << std::endl;
529
530 }

531 // =====
532 // MAIN FUNCTION
533 // =====
534
535
536 int main() {
537     std::cout << "\n";
538     std::cout << "                                \n";
539     std::cout << "          CREATING A C API FROM C++ CODE
540                         \n";
541     std::cout << "                                \n";
542     std::cout << "  Demonstrates: Opaque Handles, extern \"C\", Exception
543           Safety          \n";
544     std::cout << "  Example: Image Processing Library with C Interface
545           \n";
546     std::cout << "                                \n";
547
548     demonstrate_c_api_usage();
549     explain_best_practices();
550     compare_approaches();
551
552     std::cout << "\n" << std::string(70, '=') << std::endl;
553     std::cout << "SUMMARY:\n";
554     std::cout << std::string(70, '=') << std::endl;
555
556     std::cout << "\n KEY TECHNIQUES DEMONSTRATED:" << std::endl;
557     std::cout << "  1. Opaque handles for hiding C++ implementation" << std::endl;
558     std::cout << "  2. extern \"C\" for C-compatible linkage" << std::endl;
```

```
556     std::cout << " 3. Exception safety across C/C++ boundary" << std::endl;
557     std::cout << " 4. Error code-based error handling" << std::endl;
558     std::cout << " 5. Output parameters instead of return values" << std::
559     endl;
560     std::cout << " 6. Proper memory management (create/destroy)" << std::
561     endl;
562
563     std::cout << "\n REAL-WORLD APPLICATIONS:" << std::endl;
564     std::cout << " • Game engine C APIs (Unity, Unreal plugins)" << std::
565     endl;
566     std::cout << " • Database drivers (SQLite, PostgreSQL)" << std::endl;
567     std::cout << " • Graphics libraries (Vulkan, OpenGL wrappers)" << std::
568     endl;
569     std::cout << " • Compression libraries (zlib, bzip2 style)" << std::endl
570     ;
571     std::cout << " • Python/Ruby/Lua bindings via C API" << std::endl;
572
573     std::cout << "\n BENEFITS:" << std::endl;
574     std::cout << " • C++ power with C compatibility" << std::endl;
575     std::cout << " • Stable ABI (no name mangling issues)" << std::endl;
576     std::cout << " • Can be used from ANY language with C FFI" << std::endl;
577     std::cout << " • Hide implementation details completely" << std::endl;
578     std::cout << " • Version changes don't break binary compatibility" <<
579     std::endl;
580
581     std::cout << "\n All resources properly managed!\n" << std::endl;
582
583     return 0;
584 }
```

## 19 Source Code: DependencyInjection.cpp

File: src/DependencyInjection.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <memory>
3 #include <string>
4 #include <vector>
5 #include <functional>
6
7 // =====
8 // DEPENDENCY INJECTION IN C++ - NO PRIVATE INHERITANCE NEEDED
9 // =====
10
11 // =====
12 // 1. TRADITIONAL DI - PUBLIC INHERITANCE FOR INTERFACES
13 // =====
14
15 // Abstract interface (use PUBLIC inheritance)
16 class ILogger {
17 public:
18     virtual ~ILogger() = default;
19     virtual void log(const std::string& message) = 0;
20 };
21
22 // Concrete implementations
23 class ConsoleLogger : public ILogger { // PUBLIC inheritance
24 public:
25     void log(const std::string& message) override {
26         std::cout << "[CONSOLE] " << message << std::endl;
27     }
28 };
29
30 class FileLogger : public ILogger { // PUBLIC inheritance
31 private:
32     std::string filename;
33
34 public:
35     FileLogger(const std::string& file) : filename(file) {}
36
37     void log(const std::string& message) override {
38         std::cout << "[FILE:" << filename << "] " << message << std::endl;
39     }
40 };
41
42 // Service that depends on ILogger - uses COMPOSITION, not inheritance
43 class UserService {
44 private:
45     std::shared_ptr<ILogger> logger; // Has-a relationship via composition
46
47 public:
48     // Constructor injection
49     UserService(std::shared_ptr<ILogger> log) : logger(std::move(log)) {}
```

```
50
51     void createUser(const std::string& username) {
52         logger->log("Creating user: " + username);
53         // User creation logic...
54         logger->log("User created successfully: " + username);
55     }
56
57     void deleteUser(const std::string& username) {
58         logger->log("Deleting user: " + username);
59         // User deletion logic...
60         logger->log("User deleted: " + username);
61     }
62 };
63
64 void example_traditional_di() {
65     std::cout << "\n==== 1. TRADITIONAL DI - PUBLIC INHERITANCE + COMPOSITION
66     ===" << std::endl;
67     std::cout << "Pattern: Interface (public) + Composition (has-a)\n" << std
68         ::endl;
69
70     // Inject ConsoleLogger
71     auto consoleLogger = std::make_shared<ConsoleLogger>();
72     UserService service1(consoleLogger);
73     service1.createUser("Alice");
74
75     std::cout << std::endl;
76
77     // Inject FileLogger - same service, different dependency
78     auto fileLogger = std::make_shared<FileLogger>("users.log");
79     UserService service2(fileLogger);
80     service2.createUser("Bob");
81
82     std::cout << "\n DEPENDENCY INJECTION USES:" << std::endl;
83     std::cout << " • PUBLIC inheritance for interfaces (ILogger)" << std::
84         endl;
85     std::cout << " • COMPOSITION for dependencies (has-a ILogger)" << std::
86         endl;
87     std::cout << " • NO private inheritance needed!" << std::endl;
88 }
89
90 // =====
91 // 2. CONSTRUCTOR INJECTION (MOST COMMON)
92 // =====
93
94 class IDatabase {
95 public:
96     virtual ~IDatabase() = default;
97     virtual void save(const std::string& data) = 0;
98     virtual std::string load(const std::string& key) = 0;
99 };
100
101 class MockDatabase : public IDatabase {
102 public:
103     void save(const std::string& data) override {
```

```
100     std::cout << "  [MockDB] Saving: " << data << std::endl;
101 }
102
103 std::string load(const std::string& key) override {
104     return "mock_data_for_" + key;
105 }
106 };
107
108 class PostgresDatabase : public IDatabase {
109 public:
110     void save(const std::string& data) override {
111         std::cout << "  [PostgreSQL] Saving to database: " << data << std::endl;
112     }
113
114     std::string load(const std::string& key) override {
115         return "postgres_data_for_" + key;
116     }
117 };
118
119 // Repository with multiple dependencies injected via constructor
120 class OrderRepository {
121 private:
122     std::shared_ptr<IDatabase> database; // Composition
123     std::shared_ptr<ILogger> logger; // Composition
124
125 public:
126     // Constructor injection of multiple dependencies
127     OrderRepository(std::shared_ptr<IDatabase> db, std::shared_ptr<ILogger>
128                     log)
129         : database(std::move(db)), logger(std::move(log)) {}
130
131     void saveOrder(const std::string& orderId) {
132         logger->log("Saving order: " + orderId);
133         database->save("Order:" + orderId);
134         logger->log("Order saved successfully");
135     }
136
137     void loadOrder(const std::string& orderId) {
138         logger->log("Loading order: " + orderId);
139         auto data = database->load(orderId);
140         std::cout << "  Loaded: " << data << std::endl;
141         logger->log("Order loaded successfully");
142     }
143
144 void example_constructor_injection() {
145     std::cout << "\n==== 2. CONSTRUCTOR INJECTION (MULTIPLE DEPENDENCIES) ==="
146     << std::endl;
147     std::cout << "Pattern: Inject all dependencies via constructor\n" << std::endl;
148
149     auto logger = std::make_shared<ConsoleLogger>();
150     auto database = std::make_shared<PostgresDatabase>();
```

```
150
151     OrderRepository repo(database, logger);
152     repo.saveOrder("ORD-12345");
153
154     std::cout << std::endl;
155     repo.loadOrder("ORD-12345");
156
157     std::cout << "\n BENEFITS:" << std::endl;
158     std::cout << " • Dependencies are explicit and immutable" << std::endl;
159     std::cout << " • Easy to test (inject mocks)" << std::endl;
160     std::cout << " • No private inheritance - uses composition" << std::endl
161     ;
162 }
163
164 // =====
165 // 3. INTERFACE INJECTION (STRATEGY PATTERN)
166 // =====
167
168 class IEmailSender {
169 public:
170     virtual ~IEmailSender() = default;
171     virtual void sendEmail(const std::string& to, const std::string& message)
172         = 0;
173 };
174
175 class SmtpEmailSender : public IEmailSender {
176 public:
177     void sendEmail(const std::string& to, const std::string& message) override
178     {
179         std::cout << "[SMTP] Sending email to " << to << ":" << message <<
180             std::endl;
181     }
182 };
183
184 class MockEmailSender : public IEmailSender {
185 public:
186     void sendEmail(const std::string& to, const std::string& message) override
187     {
188         std::cout << "[MOCK] Would send email to " << to << ":" << message
189             << std::endl;
190     }
191 };
192
193 class NotificationService {
194 private:
195     std::shared_ptr<IEmailSender> emailSender; // Composition
196
197 public:
198     NotificationService(std::shared_ptr<IEmailSender> sender)
199         : emailSender(std::move(sender)) {}
200
201     // Setter injection (optional, less common)
202     void setEmailSender(std::shared_ptr<IEmailSender> sender) {
203         emailSender = std::move(sender);
204     }
205 }
```

```
198     }
199
200     void notifyUser(const std::string& email, const std::string& notification)
201     {
202         std::cout << "Sending notification..." << std::endl;
203         emailSender->sendEmail(email, notification);
204     }
205 };
206
207 void example_interface_injection() {
208     std::cout << "\n==== 3. INTERFACE INJECTION (STRATEGY PATTERN) ===" << std
209         ::endl;
210     std::cout << "Pattern: Inject behavior through interfaces\n" << std::endl;
211
212     auto smtpSender = std::make_shared<SmtpEmailSender>();
213     NotificationService service(smtpSender);
214
215     service.notifyUser("user@example.com", "Your order has shipped!");
216
217     std::cout << "\nSwitching to mock sender (setter injection):" << std::endl
218         ;
219     auto mockSender = std::make_shared<MockEmailSender>();
220     service.setEmailSender(mockSender);
221     service.notifyUser("user@example.com", "Your order is delivered!");
222
223     std::cout << "\n KEY POINTS:" << std::endl;
224     std::cout << " • Strategy pattern uses PUBLIC inheritance" << std::endl;
225     std::cout << " • Can switch implementations at runtime" << std::endl;
226     std::cout << " • No private inheritance involved" << std::endl;
227 }
228
229 // =====
230 // 4. TEMPLATE-BASED DI (COMPILE-TIME INJECTION)
231 // =====
232
233 // No interface needed - duck typing via templates
234 class ConsoleOutput {
235 public:
236     void write(const std::string& msg) {
237         std::cout << "[Console] " << msg << std::endl;
238     }
239 };
240
241 class FileOutput {
242 public:
243     void write(const std::string& msg) {
244         std::cout << "[File] " << msg << std::endl;
245     }
246 };
247
248 // Template-based dependency injection (no inheritance at all!)
249 template<typename TOutput>
250 class MessageProcessor {
251 private:
```

```
249 TOutput output; // Composition with template type
250
251 public:
252     MessageProcessor(TOutput out) : output(std::move(out)) {}
253
254     void process(const std::string& message) {
255         std::cout << "Processing message: " << message << std::endl;
256         output.write("Processed: " + message);
257     }
258 };
259
260 void example_template_di() {
261     std::cout << "\n==== 4. TEMPLATE-BASED DI (COMPILE-TIME) ===" << std::endl;
262     std::cout << "Pattern: No inheritance, no interfaces - pure templates\n"
263             << std::endl;
264
265     ConsoleOutput console;
266     MessageProcessor<ConsoleOutput> processor1(console);
267     processor1.process("Hello World");
268
269     std::cout << std::endl;
270
271     FileOutput file;
272     MessageProcessor<FileOutput> processor2(file);
273     processor2.process("Template DI");
274
275     std::cout << "\n ADVANTAGES:" << std::endl;
276     std::cout << " • Zero runtime overhead (compile-time binding)" << std::endl;
277     std::cout << " • No virtual functions needed" << std::endl;
278     std::cout << " • No inheritance (public OR private) needed!" << std::endl;
279     std::cout << " • Duck typing - if it has write(), it works" << std::endl
280             ;
281 }
282
283 // =====
284 // 5. DI CONTAINER / SERVICE LOCATOR PATTERN
285 // =====
286
287 class DIContainer {
288 private:
289     std::shared_ptr<ILogger> logger;
290     std::shared_ptr<IDatabase> database;
291     std::shared_ptr<IEmailSender> emailSender;
292
293 public:
294     // Register dependencies
295     void registerLogger(std::shared_ptr<ILogger> log) {
296         logger = std::move(log);
297     }
298
299     void registerDatabase(std::shared_ptr<IDatabase> db) {
300         database = std::move(db);
301     }
302 }
```

```
299     }
300
301     void registerEmailSender(std::shared_ptr<IEmailSender> sender) {
302         emailSender = std::move(sender);
303     }
304
305     // Resolve dependencies
306     std::shared_ptr<ILogger> getLogger() { return logger; }
307     std::shared_ptr<IDatabase> getDatabase() { return database; }
308     std::shared_ptr<IEmailSender> getEmailSender() { return emailSender; }
309
310     // Factory method to create services with auto-injected dependencies
311     std::shared_ptr<OrderRepository> createOrderRepository() {
312         return std::make_shared<OrderRepository>(database, logger);
313     }
314 };
315
316 void example_di_container() {
317     std::cout << "\n== 5. DI CONTAINER / SERVICE LOCATOR ==" << std::endl;
318     std::cout << "Pattern: Central registry for dependency management\n" <<
319         std::endl;
320
321     DIContainer container;
322
323     // Setup (composition root)
324     container.registerLogger(std::make_shared<ConsoleLogger>());
325     container.registerDatabase(std::make_shared<MockDatabase>());
326     container.registerEmailSender(std::make_shared<MockEmailSender>());
327
328     // Resolve dependencies
329     auto repo = container.createOrderRepository();
330     repo->saveOrder("ORD-99999");
331
332     auto emailSender = container.getEmailSender();
333     emailSender->sendEmail("admin@example.com", "System started");
334
335     std::cout << "\n DI CONTAINER USES:" << std::endl;
336     std::cout << " • Stores dependencies as members (composition)" << std::endl;
337     std::cout << " • Public inheritance for interfaces only" << std::endl;
338     std::cout << " • No private inheritance needed" << std::endl;
339 }
340
341 // =====
342 // 6. FUNCTIONAL DI (C++11/14 STYLE)
343 // =====
344
345 class PaymentService {
346     private:
347         std::function<void(const std::string&)> logFunc;
348         std::function<bool(double)> validateFunc;
349
350     public:
351         // Inject functions instead of objects
```

```

351     PaymentService(
352         std::function<void(const std::string&)> logger,
353         std::function<bool(double)> validator)
354         : logFunc(std::move(logger))
355         , validateFunc(std::move(validator)) {}
356
357     void processPayment(double amount) {
358         logFunc("Processing payment of $" + std::to_string(amount));
359
360         if (validateFunc(amount)) {
361             logFunc("Payment validated and processed");
362         } else {
363             logFunc("Payment validation failed");
364         }
365     }
366 };
367
368 void example_functional_di() {
369     std::cout << "\n==== 6. FUNCTIONAL DI (INJECT FUNCTIONS) ===" << std::endl;
370     std::cout << "Pattern: Inject std::function instead of interfaces\n" <<
371         std::endl;
372
373     // Define behaviors as lambdas
374     auto logger = [] (const std::string& msg) {
375         std::cout << "  [Payment Log] " << msg << std::endl;
376     };
377
378     auto validator = [] (double amount) {
379         return amount > 0 && amount < 10000; // Simple validation
380     };
381
382     PaymentService service(logger, validator);
383     service.processPayment(150.50);
384
385     std::cout << std::endl;
386     service.processPayment(-50.0); // Invalid
387
388     std::cout << "\n FUNCTIONAL DI:" << std::endl;
389     std::cout << " • No interfaces needed" << std::endl;
390     std::cout << " • No inheritance at all (public or private)" << std::endl
391         ;
392     std::cout << " • Lightweight and flexible" << std::endl;
393     std::cout << " • Uses std::function and lambdas" << std::endl;
394 }
395
396 // =====
397 // 7. WHY NOT PRIVATE INHERITANCE FOR DI?
398 // =====
399
400 // WRONG: Using private inheritance for DI (anti-pattern)
401 class WrongServiceWithPrivateInheritance : private ILogger {
402     private:
403         // This makes ILogger methods private - defeats the purpose of DI!

```

```

403 | public:
404 |     void doSomething() {
405 |         log("This is confusing!"); // Can only use internally
406 |     }
407 |
408 |     // Cannot inject different logger implementations
409 |     // Cannot swap loggers at runtime
410 |     // Tightly coupled to ILogger
411 | };
412 |
413 | // CORRECT: Using composition for DI
414 | class CorrectServiceWithComposition {
415 | private:
416 |     std::shared_ptr<ILogger> logger; // Injected dependency
417 |
418 | public:
419 |     CorrectServiceWithComposition(std::shared_ptr<ILogger> log)
420 |         : logger(std::move(log)) {}
421 |
422 |     void doSomething() {
423 |         logger->log("This is the right way!");
424 |     }
425 |
426 |     // Can inject any ILogger implementation
427 |     // Can swap loggers at runtime
428 |     // Loosely coupled
429 | };
430 |
431 | void example_why_not_private() {
432 |     std::cout << "\n==== 7. WHY NOT PRIVATE INHERITANCE FOR DI? ===" << std::endl;
433 |
434 |     std::cout << "\n PRIVATE INHERITANCE FOR DI IS WRONG BECAUSE:" << std::endl;
435 |     std::cout << " • Cannot inject different implementations" << std::endl;
436 |     std::cout << " • Tightly coupled to specific base class" << std::endl;
437 |     std::cout << " • Cannot swap dependencies at runtime" << std::endl;
438 |     std::cout << " • Violates Dependency Inversion Principle" << std::endl;
439 |     std::cout << " • Makes testing difficult (can't inject mocks)" << std::endl;
440 |
441 |     std::cout << "\n DI REQUIRES LOOSE COUPLING:" << std::endl;
442 |     std::cout << " • Use interfaces (abstract base classes)" << std::endl;
443 |     std::cout << " • Use PUBLIC inheritance for interfaces" << std::endl;
444 |     std::cout << " • Use COMPOSITION to hold dependencies" << std::endl;
445 |     std::cout << " • Inject dependencies via constructor/setter" << std::endl;
446 | }
447 |
448 | // =====
449 | // MAIN FUNCTION
450 | // =====
451 |
452 | int main() {

```

```
453     std::cout << "\n"
454     =====" <<
455     std::endl;
456     std::cout << "  DEPENDENCY INJECTION IN C++ - NO PRIVATE INHERITANCE!" <<
457     std::endl;
458     std::cout << "
459     =====" <<
460     std::endl;
461
462     example_traditional_di();
463     example_constructor_injection();
464     example_interface_injection();
465     example_template_di();
466     example_di_container();
467     example_functional_di();
468     example_why_not_private();
469
470     std::cout << "\n"
471     =====" <<
472     std::endl;
473     std::cout << "  DEPENDENCY INJECTION SUMMARY" << std::endl;
474     std::cout << "
475     =====" <<
476     std::endl;
477
478     std::cout << "\n DI PATTERNS IN C++ (NO PRIVATE INHERITANCE):" << std::endl;
479
480     std::cout << "n1. INTERFACE-BASED DI (MOST COMMON):" << std::endl;
481     std::cout << "  •  Define abstract interfaces (ILogger, IDatabase, etc.)"
482     << std::endl;
483     std::cout << "  •  Implementations use PUBLIC inheritance" << std::endl;
484     std::cout << "  •  Services use COMPOSITION (has-a relationship)" << std::endl;
485     std::cout << "  •  Inject via constructor or setter" << std::endl;
486
487     std::cout << "n2. TEMPLATE-BASED DI:" << std::endl;
488     std::cout << "  •  No inheritance at all!" << std::endl;
489     std::cout << "  •  Compile-time polymorphism via templates" << std::endl;
490     std::cout << "  •  Duck typing - if it quacks like a duck..." << std::endl;
491     ;
492     std::cout << "  •  Zero runtime overhead" << std::endl;
493
494     std::cout << "n3. FUNCTIONAL DI:" << std::endl;
495     std::cout << "  •  Inject std::function instead of interfaces" << std::endl;
496     std::cout << "  •  No inheritance needed" << std::endl;
497     std::cout << "  •  Lightweight and flexible" << std::endl;
498     std::cout << "  •  Great for simple dependencies" << std::endl;
499
500     std::cout << "n4. DI CONTAINER:" << std::endl;
501     std::cout << "  •  Central registry for dependencies" << std::endl;
502     std::cout << "  •  Auto-wiring of complex object graphs" << std::endl;
503     std::cout << "  •  Uses composition internally" << std::endl;
504     std::cout << "  •  Similar to Spring (Java) or Autofac (.NET)" << std::endl;
```

```
        endl;

493    std::cout << "\n NEVER USE PRIVATE INHERITANCE FOR DI:" << std::endl;
494    std::cout << " • Private inheritance = tight coupling" << std::endl;
495    std::cout << " • Cannot inject different implementations" << std::endl;
496    std::cout << " • Cannot swap at runtime" << std::endl;
497    std::cout << " • Defeats the entire purpose of DI!" << std::endl;
498
499
500    std::cout << "\n CORRECT DI APPROACH:" << std::endl;
501    std::cout << " • PUBLIC inheritance for interfaces (polymorphism)" <<
502        std::endl;
503    std::cout << " • COMPOSITION for dependencies (has-a)" << std::endl;
504    std::cout << " • Constructor/setter injection" << std::endl;
505    std::cout << " • Inversion of Control (IoC)" << std::endl;
506
507    std::cout << "\n KEY PRINCIPLE:" << std::endl;
508    std::cout << "     \"Depend on abstractions, not concretions\"" << std::endl
509        ;
510    std::cout << "     - Dependency Inversion Principle (SOLID)" << std::endl;
511
512    std::cout << "\n WHEN TO USE EACH:" << std::endl;
513    std::cout << " • Interface-based DI: Runtime polymorphism, complex
514        systems" << std::endl;
515    std::cout << " • Template-based DI: Performance-critical code, simple
516        deps" << std::endl;
517    std::cout << " • Functional DI: Simple behaviors, lambdas" << std::endl;
518    std::cout << " • DI Container: Large applications with many dependencies
519        " << std::endl;
520    std::cout << " • Private inheritance: NEVER for DI!" << std::endl;
521
522    std::cout << "\n
523        =====\n" << std::endl;
524
525    return 0;
526}
```

## 20 Source Code: DiamondProblem.cpp

File: src/DiamondProblem.cpp

Repository: [View on GitHub](#)

```
1 // DiamondProblem.cpp
2 // Demonstrates the Diamond Problem (Diamond of Death) in C++ and its solution
3 //
4 // The Diamond Problem occurs when a class inherits from two classes that both
5 // inherit from the same base class, creating a diamond-shaped inheritance
6 // hierarchy.
7 //
8 // PROBLEM:
9 // Without virtual inheritance, the derived class contains TWO copies of the
10 // base class, leading to ambiguity and wasted memory.
11 //
12 // SOLUTION:
13 // Use 'virtual' keyword in inheritance to ensure only ONE copy of the base
14 // class exists in the final derived class.
15 //
16 //      Base
17 //      /   \
18 //      A     B
19 //      \   /
20 //      Derived
21 //
22 // KEY CONCEPTS:
23 // 1. Multiple Inheritance creates duplicate base class instances
24 // 2. Virtual Inheritance solves the duplication
25 // 3. Constructor initialization order with virtual bases
26 // 4. When to use and when to avoid multiple inheritance
27
28 #include <iostream>
29 #include <string>
30 #include <memory>
31 //
32 // SECTION 1: THE PROBLEM - Diamond Without Virtual Inheritance
33 //
34
35 namespace diamond_problem {
36
37 class Device {
38 protected:
39     std::string name_;
40     int id_;
41
42 public:
43     Device(const std::string& name, int id)
44         : name_(name), id_(id) {
```

```
45     std::cout << "      [Device] Constructed: " << name_ << " (ID: " << id_
46             << ")\\n";
47 }
48 void displayInfo() const {
49     std::cout << "      Device: " << name_ << ", ID: " << id_ << "\\n";
50 }
51 int getId() const { return id_; }
52 };
53
54
55 class InputDevice : public Device {
56 protected:
57     std::string inputType_;
58
59 public:
60     InputDevice(const std::string& name, int id, const std::string& inputType)
61         : Device(name, id), inputType_(inputType) {
62         std::cout << "      [InputDevice] Constructed: " << inputType_ << "\\n";
63     }
64
65     void showInput() const {
66         std::cout << "      Input Type: " << inputType_ << "\\n";
67     }
68 };
69
70 class OutputDevice : public Device {
71 protected:
72     std::string outputType_;
73
74 public:
75     OutputDevice(const std::string& name, int id, const std::string&
76                 outputType)
77         : Device(name, id), outputType_(outputType) {
78         std::cout << "      [OutputDevice] Constructed: " << outputType_ << "\\n"
79             ;
80     }
81
82     void showOutput() const {
83         std::cout << "      Output Type: " << outputType_ << "\\n";
84     }
85 }
86
87 // PROBLEM: IODevice inherits Device TWICE (through InputDevice and
88 //           OutputDevice)
89 class IODevice : public InputDevice, public OutputDevice {
90 public:
91     IODevice(const std::string& name, int id)
92         : InputDevice(name, id, "Keyboard"),
93             OutputDevice(name, id, "Display") {
94         std::cout << "      [IODevice] Constructed\\n";
95     }
96
97     void showInfo() const {
```

```
95 // AMBIGUITY: Which Device::displayInfo() to call?
96 // AMBIGUITY: Which Device::getId() to call?
97 // Compiler error without explicit qualification
98
99 // Must specify which base class path to use:
100 std::cout << "\n  Input Device Info:\n";
101 InputDevice::displayInfo(); // Calls Device from InputDevice path
102 showInput();
103
104 std::cout << "\n  Output Device Info:\n";
105 OutputDevice::displayInfo(); // Calls Device from OutputDevice path
106 showOutput();
107
108 std::cout << "\n      Problem: Two copies of Device exist!\n";
109 std::cout << "      InputDevice::getId() = " << InputDevice::getId()
110     << "\n";
111 std::cout << "      OutputDevice::getId() = " << OutputDevice::getId()
112     << "\n";
113 }
114 };
115
116 void demonstrate() {
117 std::cout << "\n" << std::string(70, '=') << "\n";
118 std::cout << "==== SECTION 1: Diamond Problem (Without Virtual Inheritance)
119     ===\n";
120 std::cout << std::string(70, '=') << "\n\n";
121
122 std::cout << "Creating IODevice...\n";
123 std::cout << "Notice: Device constructor called TWICE!\n\n";
124
125 IODevice device("MyIODevice", 42);
126
127 std::cout << "\nShowing device information:\n";
128 device.showInfo();
129
130 std::cout << "Memory Analysis:\n";
131 std::cout << "  sizeof(Device) = " << sizeof(Device) << " bytes\n";
132 std::cout << "  sizeof(InputDevice) = " << sizeof(InputDevice) << " bytes\n";
133 std::cout << "  sizeof(OutputDevice) = " << sizeof(OutputDevice) << "
134     bytes\n";
135 std::cout << "  sizeof(IODevice) = " << sizeof(IODevice) << " bytes\n";
136 std::cout << "  IODevice contains TWO Device instances (memory waste)\n
137     ";
138 }
139 }
140
141 // namespace diamond_problem
142
143 // =====
144
145 // SECTION 2: THE SOLUTION - Virtual Inheritance
146 // =====
```

```
140
141 namespace virtual_inheritance_solution {
142
143     class Device {
144         protected:
145             std::string name_;
146             int id_;
147
148         public:
149             Device(const std::string& name, int id)
150                 : name_(name), id_(id) {
151                     std::cout << "    [Device] Constructed: " << name_ << " (ID: " << id_
152                     << ")\n";
153             }
154
155             virtual ~Device() = default;
156
157             void displayInfo() const {
158                 std::cout << "    Device: " << name_ << ", ID: " << id_ << "\n";
159             }
160
161             int getId() const { return id_; }
162             std::string getName() const { return name_; }
163     };
164
165     // SOLUTION: Use 'virtual' keyword in inheritance
166     class InputDevice : virtual public Device {
167         protected:
168             std::string inputType_;
169
170         public:
171             InputDevice(const std::string& name, int id, const std::string& inputType)
172                 : Device(name, id), inputType_(inputType) {
173                     std::cout << "    [InputDevice] Constructed: " << inputType_ << "\n";
174             }
175
176             void showInput() const {
177                 std::cout << "    Input Type: " << inputType_ << "\n";
178             }
179     };
180
181     // SOLUTION: Use 'virtual' keyword in inheritance
182     class OutputDevice : virtual public Device {
183         protected:
184             std::string outputType_;
185
186         public:
187             OutputDevice(const std::string& name, int id, const std::string&
188                 outputType)
189                 : Device(name, id), outputType_(outputType) {
200                     std::cout << "    [OutputDevice] Constructed: " << outputType_ << "\n"
201                     ;
202             }
203     };
204 }
```

```
190
191     void showOutput() const {
192         std::cout << "    Output Type: " << outputType_ << "\n";
193     }
194 };
195
196 // Now IODevice has only ONE copy of Device
197 class IODevice : public InputDevice, public OutputDevice {
198 public:
199     // IMPORTANT: With virtual inheritance, the most derived class
200     // is responsible for initializing the virtual base class
201     IODevice(const std::string& name, int id)
202         : Device(name, id), // Must initialize Device directly!
203             InputDevice(name, id, "Keyboard"),
204             OutputDevice(name, id, "Display") {
205         std::cout << "    [IODevice] Constructed\n";
206     }
207
208     void showInfo() const {
209         // No ambiguity now - only one Device exists
210         std::cout << "\n    Device Info:\n";
211         displayInfo(); // Unambiguous!
212
213         std::cout << "\n    Input/Output Details:\n";
214         showInput();
215         showOutput();
216
217         std::cout << "\n    Solution: Only ONE copy of Device exists!\n";
218         std::cout << "    getId() = " << getId() << " (no ambiguity)\n";
219     }
220 };
221
222 void demonstrate() {
223     std::cout << "\n" << std::string(70, '=') << "\n";
224     std::cout << "==== SECTION 2: Solution - Virtual Inheritance ===\n";
225     std::cout << std::string(70, '=') << "\n\n";
226
227     std::cout << "Creating IODevice with virtual inheritance...\n";
228     std::cout << "Notice: Device constructor called ONCE!\n\n";
229
230     IODevice device("MyIODevice", 42);
231
232     std::cout << "\nShowing device information:\n";
233     device.showInfo();
234
235     std::cout << "\nMemory Analysis:\n";
236     std::cout << "    sizeof(Device) = " << sizeof(Device) << " bytes\n";
237     std::cout << "    sizeof(InputDevice) = " << sizeof(InputDevice) << " bytes\n";
238     std::cout << "    sizeof(OutputDevice) = " << sizeof(OutputDevice) << " bytes\n";
239     std::cout << "    sizeof(IODevice) = " << sizeof(IODevice) << " bytes\n";
240     std::cout << "    IODevice contains only ONE Device instance (no
duplication)\n";
```

```
241 }
242
243 } // namespace virtual_inheritance_solution
244
245 // =====
246 // SECTION 3: Real-World Example - File System
247 // =====
248
249 namespace filesystem_example {
250
251 class File {
252 protected:
253     std::string path_;
254     size_t size_;
255
256 public:
257     File(const std::string& path, size_t size)
258         : path_(path), size_(size) {
259         std::cout << "    [File] Created: " << path_ << " (" << size_ << "
260             bytes)\n";
261     }
262
263     virtual ~File() = default;
264
265     std::string getPath() const { return path_; }
266     size_t getSize() const { return size_; }
267
268     void displayInfo() const {
269         std::cout << "    File: " << path_ << " (" << size_ << " bytes)\n";
270     }
271 };
272
273 // Readable file
274 class ReadableFile : virtual public File {
275 public:
276     ReadableFile(const std::string& path, size_t size)
277         : File(path, size) {
278         std::cout << "    [ReadableFile] Readable permissions added\n";
279     }
280
281     void read() const {
282         std::cout << "    Reading from: " << path_ << "\n";
283     }
284 };
285
286 // Writable file
287 class WritableFile : virtual public File {
288 public:
289     WritableFile(const std::string& path, size_t size)
290         : File(path, size) {
```

```
290         std::cout << "      [WritableFile] Writable permissions added\n";
291     }
292
293     void write(const std::string& data) {
294         std::cout << "      Writing to: " << path_ << " - Data: " << data << "
295             \"\n";
296         size_ += data.length();
297     }
298
299 // Read-Write file - inherits from both
300 class ReadWriteFile : public ReadableFile, public WritableFile {
301 public:
302     ReadWriteFile(const std::string& path, size_t size)
303         : File(path, size), // Initialize virtual base
304           ReadableFile(path, size),
305           WritableFile(path, size) {
306         std::cout << "      [ReadWriteFile] Read-Write file created\n";
307     }
308
309     void showInfo() const {
310         displayInfo(); // No ambiguity
311         std::cout << "      Permissions: Read + Write\n";
312     }
313 };
314
315 void demonstrate() {
316     std::cout << "\n" << std::string(70, '=') << "\n";
317     std::cout << "==== SECTION 3: Real-World Example - File System ===\n";
318     std::cout << std::string(70, '=') << "\n\n";
319
320     std::cout << "Creating ReadWriteFile with virtual inheritance:\n\n";
321     ReadWriteFile file("document.txt", 1024);
322
323     std::cout << "\nUsing the file:\n";
324     file.showInfo();
325     file.read();
326     file.write("Hello, World!");
327
328     std::cout << "\nFinal state:\n";
329     file.displayInfo();
330 }
331
332 } // namespace filesystem_example
333
334 // =====
335 // SECTION 4: Constructor Initialization Order
336 // =====
337
338 namespace initialization_order {
```

```
339
340 class Base {
341 public:
342     Base(int value) {
343         std::cout << "[Base] Constructed with value: " << value << "\n";
344     }
345 }
346
347 class A : virtual public Base {
348 public:
349     A(int value) : Base(value) {
350         std::cout << "[A] Constructed\n";
351     }
352 }
353
354 class B : virtual public Base {
355 public:
356     B(int value) : Base(value) {
357         std::cout << "[B] Constructed\n";
358     }
359 }
360
361 class Derived : public A, public B {
362 public:
363     // CRITICAL: Derived class must initialize virtual base class directly!
364     // The Base(value) calls in A and B are IGNORED
365     Derived(int value)
366         : Base(value), // This is the one that matters!
367             A(value),
368             B(value) {
369         std::cout << "[Derived] Constructed\n";
370     }
371 }
372
373 void demonstrate() {
374     std::cout << "\n" << std::string(70, '=') << "\n";
375     std::cout << "==== SECTION 4: Constructor Initialization Order ====\n";
376     std::cout << std::string(70, '=') << "\n\n";
377
378     std::cout << "Important: Virtual base classes are initialized first,\n";
379     std::cout << "                    by the MOST DERIVED class.\n\n";
380
381     std::cout << "Creating Derived object:\n";
382     Derived d(42);
383
384     std::cout << "\nInitialization order:\n";
385     std::cout << " 1. Base (initialized by Derived, not A or B)\n";
386     std::cout << " 2. A\n";
387     std::cout << " 3. B\n";
388     std::cout << " 4. Derived\n";
389 }
390
391 } // namespace initialization_order
```

```
393 // =====
394 // SECTION 5: When to Avoid Multiple Inheritance
395 // =====
396
397 namespace alternative_designs {
398
399 // Instead of multiple inheritance, use composition
400 class Device {
401 protected:
402     std::string name_;
403     int id_;
404
405 public:
406     Device(const std::string& name, int id)
407         : name_(name), id_(id) {}
408
409     std::string getName() const { return name_; }
410     int getId() const { return id_; }
411 };
412
413 class InputCapability {
414     std::string inputType_;
415
416 public:
417     explicit InputCapability(const std::string& type) : inputType_(type) {}
418
419     void handleInput() const {
420         std::cout << "    Handling input: " << inputType_ << "\n";
421     }
422 };
423
424 class OutputCapability {
425     std::string outputType_;
426
427 public:
428     explicit OutputCapability(const std::string& type) : outputType_(type) {}
429
430     void handleOutput() const {
431         std::cout << "    Handling output: " << outputType_ << "\n";
432     }
433 };
434
435 // Composition-based design - cleaner and more flexible
436 class IODevice {
437     Device device_;
438     InputCapability input_;
439     OutputCapability output_;
440
441 public:
442     IODevice(const std::string& name, int id)
```

```
443     : device_(name, id),
444     input_("Keyboard"),
445     output_("Display") {
446     std::cout << "    [IODevice] Constructed using composition\n";
447 }
448
449 void showInfo() const {
450     std::cout << "    Device: " << device_.getName()
451             << " (ID: " << device_.getId() << ")\n";
452     input_.handleInput();
453     output_.handleOutput();
454 }
455 };
456
457 void demonstrate() {
458     std::cout << "\n" << std::string(70, '=') << "\n";
459     std::cout << "==== SECTION 5: Alternative - Composition Over Inheritance
460             ===\n";
461     std::cout << std::string(70, '=') << "\n\n";
462
463     std::cout << "Creating IODevice with composition:\n";
464     IODevice device("MyIODevice", 42);
465
466     std::cout << "\nUsing the device:\n";
467     device.showInfo();
468
469     std::cout << "\nBenefits:\n";
470     std::cout << "    No diamond problem\n";
471     std::cout << "    More flexible\n";
472     std::cout << "    Easier to understand\n";
473     std::cout << "    Simpler construction\n";
474 }
475 } // namespace alternative_designs
476
477 // =====
478 // SECTION 6: Best Practices
479 // =====
480
481 void show_best_practices() {
482     std::cout << "\n" << std::string(70, '=') << "\n";
483     std::cout << "==== Best Practices for Diamond Problem ===\n";
484     std::cout << std::string(70, '=') << "\n\n";
485
486     std::cout << "1. UNDERSTAND THE PROBLEM\n";
487     std::cout << "    •    Multiple inheritance creates duplicate base class
488             instances\n";
489     std::cout << "    •    Causes ambiguity and memory waste\n";
490     std::cout << "    •    Creates the \"diamond\" inheritance shape\n\n";
```

```
491     std::cout << "2. USE VIRTUAL INHERITANCE WHEN NEEDED\n";
492     std::cout << " • Add 'virtual' keyword: class Derived : virtual public
493         Base\n";
494     std::cout << " • Ensures only ONE base class instance exists\n";
495     std::cout << " • Most derived class initializes the virtual base\n\n";
496
497     std::cout << "3. INITIALIZATION ORDER RULES\n";
498     std::cout << " • Virtual base classes initialized FIRST\n";
499     std::cout << " • Most derived class is responsible for initialization\n"
500         ;
501     std::cout << " • Base class constructors in intermediate classes are
502         ignored\n\n";
503
504     std::cout << "4. PERFORMANCE CONSIDERATIONS\n";
505     std::cout << " • Virtual inheritance has small runtime overhead (vtable
506         pointer)\n";
507     std::cout << " • Slightly larger object size\n";
508     std::cout << " • Usually worth it to avoid duplication\n\n";
509
510     std::cout << "5. PREFER COMPOSITION OVER MULTIPLE INHERITANCE\n";
511     std::cout << " • Composition is often clearer and more flexible\n";
512     std::cout << " • No diamond problem at all\n";
513     std::cout << " • Easier to test and maintain\n\n";
514
515     std::cout << "6. WHEN TO USE MULTIPLE INHERITANCE\n";
516     std::cout << "     Inheriting multiple pure interfaces (abstract classes)\n
517         ";
518     std::cout << "     Mixins (adding orthogonal functionality)\n";
519     std::cout << "     Inheriting implementation from multiple classes\n\n";
520
521     std::cout << "7. INTERFACE SEGREGATION\n";
522     std::cout << " • Use multiple pure abstract classes (interfaces)\n";
523     std::cout << " • No diamond problem with pure interfaces\n";
524     std::cout << " • Clear contracts without implementation conflicts\n\n";
525
526     std::cout << "REMEMBER:\n";
527     std::cout << "     \"Prefer composition over inheritance\" - Gang of Four\n";
528     std::cout << "     \"Virtual inheritance solves the diamond problem\" -
529         Bjarne Stroustrup\n";
530 }
531
532 // MAIN FUNCTION
533 // =====
534
535 int main() {
536     std::cout << "\n";
537     std::cout << "                                         \n";
538     std::cout << "                                         Diamond Problem (Diamond of Death) in C++\n";
539 }
```

```
534     std::cout << "           Solution: Virtual Inheritance
535                     \n";
536     std::cout << "           \n";
537     // Demonstrate the problem
538     diamond_problem::demonstrate();
539
540     // Show the solution
541     virtual_inheritance_solution::demonstrate();
542
543     // Real-world example
544     filesystem_example::demonstrate();
545
546     // Initialization order
547     initialization_order::demonstrate();
548
549     // Alternative designs
550     alternative_designs::demonstrate();
551
552     // Best practices
553     show_best_practices();
554
555     std::cout << "\n" << std::string(70, '=') << "\n";
556     std::cout << "All demonstrations completed!\n";
557     std::cout << std::string(70, '=') << "\n\n";
558
559     return 0;
560 }
```

## 21 Source Code: EigenSensorFusion.cpp

File: src/EigenSensorFusion.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // EIGEN LIBRARY: SENSOR FUSION AND PARTICLE FILTER
3 // =====
4 // This example demonstrates advanced robotics and embedded systems
5 // applications using the Eigen library for linear algebra and tensors.
6 //
7 // TOPICS COVERED:
8 // 1. Eigen Tensors - Multi-dimensional arrays for sensor data
9 // 2. Kalman Filter - Optimal sensor fusion (IMU + GPS)
10 // 3. Complementary Filter - Lightweight sensor fusion (Accel + Gyro)
11 // 4. Extended Kalman Filter (EKF) - Nonlinear sensor fusion
12 // 5. Particle Filter - Monte Carlo localization and tracking
13 // 6. Sensor Fusion Pipeline - Real-time data processing
14 //
15 // INSTALL EIGEN:
16 // =====
17 // Ubuntu/Debian: sudo apt-get install libeigen3-dev
18 // macOS: brew install eigen
19 // Windows: vcpkg install eigen3
20 // Manual: Download from https://eigen.tuxfamily.org/
21 //
22 // BUILD INSTRUCTIONS:
23 // =====
24 // g++ -std=c++17 -O3 -I/usr/include/eigen3 EigenSensorFusion.cpp -o
25 // EigenSensorFusion
26 //
27 // For CMake:
28 // find_package(Eigen3 REQUIRED)
29 // target_link_libraries(EigenSensorFusion Eigen3::Eigen)
30 //
31 // APPLICATIONS:
32 // =====
33 // - Drone attitude estimation (IMU fusion)
34 // - Robot localization (particle filter)
35 // - Autonomous vehicles (multi-sensor fusion)
36 // - Indoor navigation (dead reckoning + WiFi)
37 // - Wearable devices (activity recognition)
38 // =====
39 #include <iostream>
40 #include <vector>
41 #include <random>
42 #include <cmath>
43 #include <iomanip>
44 #include <Eigen/Dense>
45 #include <Eigen/Core>
46
47 using namespace Eigen;
48
```

```
49 // =====
50 // 1. SENSOR DATA STRUCTURES
51 // =====
52
53 struct IMUData {
54     Vector3d accel;        // Accelerometer (m/s2)
55     Vector3d gyro;         // Gyroscope (rad/s)
56     double timestamp;
57
58     IMUData(const Vector3d& a, const Vector3d& g, double t)
59         : accel(a), gyro(g), timestamp(t) {}
60 };
61
62 struct GPSData {
63     Vector2d position;    // Latitude, Longitude (or x, y in meters)
64     double accuracy;      // GPS accuracy (meters)
65     double timestamp;
66
67     GPSData(const Vector2d& pos, double acc, double t)
68         : position(pos), accuracy(acc), timestamp(t) {}
69 };
70
71 struct State {
72     Vector3d position;    // x, y, z
73     Vector3d velocity;    // vx, vy, vz
74     Vector3d orientation; // roll, pitch, yaw (Euler angles)
75
76     State() : position(Vector3d::Zero()),
77                velocity(Vector3d::Zero()),
78                orientation(Vector3d::Zero()) {}
79 };
80
81 // =====
82 // 2. KALMAN FILTER FOR SENSOR FUSION
83 // =====
84
85 class KalmanFilter {
86 private:
87     // State: [x, y, vx, vy] - position and velocity in 2D
88     VectorXd x;           // State vector (4x1)
89     MatrixXd P;           // State covariance (4x4)
90     MatrixXd F;           // State transition (4x4)
91     MatrixXd H;           // Measurement matrix (2x4)
92     MatrixXd Q;           // Process noise (4x4)
93     MatrixXd R;           // Measurement noise (2x2)
94
95 public:
96     KalmanFilter() {
97         // Initialize state: [x, y, vx, vy]
98         x = VectorXd::Zero(4);
99
100        // Initialize covariance
101        P = MatrixXd::Identity(4, 4) * 1000.0;
102    }
```

```
103     // State transition matrix (constant velocity model)
104     F = MatrixXd::Identity(4, 4);
105     // F will be updated with dt in predict()
106
107     // Measurement matrix (we measure position only)
108     H = MatrixXd::Zero(2, 4);
109     H(0, 0) = 1.0; // Measure x
110     H(1, 1) = 1.0; // Measure y
111
112     // Process noise (model uncertainty)
113     Q = MatrixXd::Identity(4, 4) * 0.1;
114
115     // Measurement noise (GPS uncertainty)
116     R = MatrixXd::Identity(2, 2) * 5.0; // 5m GPS accuracy
117 }
118
119 // Prediction step (using IMU or motion model)
120 void predict(double dt) {
121     // Update state transition matrix with dt
122     F(0, 2) = dt; // x = x + vx * dt
123     F(1, 3) = dt; // y = y + vy * dt
124
125     // Predict state
126     x = F * x;
127
128     // Predict covariance
129     P = F * P * F.transpose() + Q;
130 }
131
132 // Update step (using GPS measurement)
133 void update(const Vector2d& measurement, double measurement_noise) {
134     // Update measurement noise with actual GPS accuracy
135     R = MatrixXd::Identity(2, 2) * (measurement_noise * measurement_noise)
136         ;
137
138     // Innovation (measurement residual)
139     VectorXd z = VectorXd(2);
140     z << measurement(0), measurement(1);
141     VectorXd y = z - H * x;
142
143     // Innovation covariance
144     MatrixXd S = H * P * H.transpose() + R;
145
146     // Kalman gain
147     MatrixXd K = P * H.transpose() * S.inverse();
148
149     // Update state
150     x = x + K * y;
151
152     // Update covariance
153     MatrixXd I = MatrixXd::Identity(4, 4);
154     P = (I - K * H) * P;
155 }
```

```

156     Vector2d getPosition() const {
157         return Vector2d(x(0), x(1));
158     }
159
160     Vector2d getVelocity() const {
161         return Vector2d(x(2), x(3));
162     }
163
164     VectorXd getState() const { return x; }
165     MatrixXd getCovariance() const { return P; }
166 };
167
168 // =====
169 // 3. COMPLEMENTARY FILTER (Lightweight Sensor Fusion)
170 // =====
171
172 class ComplementaryFilter {
173 private:
174     Vector3d orientation; // Roll, Pitch, Yaw
175     double alpha; // Filter coefficient (0-1)
176
177 public:
178     ComplementaryFilter(double filter_alpha = 0.98)
179         : orientation(Vector3d::Zero()), alpha(filter_alpha) {}
180
181     // Fuse accelerometer and gyroscope data
182     void update(const Vector3d& accel, const Vector3d& gyro, double dt) {
183         // Calculate angles from accelerometer (for roll and pitch only)
184         double accel_roll = std::atan2(accel.y(), accel.z());
185         double accel_pitch = std::atan2(-accel.x(),
186                                         std::sqrt(accel.y()*accel.y() + accel.z()
187                                         ()*accel.z()));
188
188         // Integrate gyroscope (rate) to get angle
189         Vector3d gyro_angle = orientation + gyro * dt;
190
191         // Complementary filter: trust gyro for short-term, accel for long-
192         // term
192         orientation(0) = alpha * gyro_angle(0) + (1.0 - alpha) * accel_roll;
193         // Roll
193         orientation(1) = alpha * gyro_angle(1) + (1.0 - alpha) * accel_pitch;
194         // Pitch
194         orientation(2) = gyro_angle(2); // Yaw (no accel reference, use gyro
195         // only)
195     }
196
197     Vector3d getOrientation() const { return orientation; }
198
199     // Convert to degrees for readability
200     Vector3d getOrientationDegrees() const {
201         return orientation * 180.0 / M_PI;
202     }
203 };
204

```

```
205 // =====
206 // 4. PARTICLE FILTER (Monte Carlo Localization)
207 // =====
208
209 struct Particle {
210     Vector2d position; // x, y
211     double weight; // Importance weight
212
213     Particle() : position(Vector2d::Zero()), weight(1.0) {}
214     Particle(const Vector2d& pos, double w) : position(pos), weight(w) {}
215 };
216
217 class ParticleFilter {
218 private:
219     std::vector<Particle> particles;
220     int num_particles;
221     std::mt19937 rng;
222
223 public:
224     ParticleFilter(int n_particles = 1000)
225         : num_particles(n_particles), rng(std::random_device{}()) {
226
227         // Initialize particles with random positions
228         std::normal_distribution<double> dist(0.0, 10.0);
229         particles.reserve(num_particles);
230
231         for (int i = 0; i < num_particles; ++i) {
232             Vector2d pos(dist(rng), dist(rng));
233             particles.emplace_back(pos, 1.0 / num_particles);
234         }
235     }
236
237     // Prediction step: move particles based on motion model
238     void predict(const Vector2d& control_input, double dt, double motion_noise)
239     {
240         std::normal_distribution<double> noise(0.0, motion_noise);
241
242         for (auto& particle : particles) {
243             // Move particle according to motion model + noise
244             particle.position += control_input * dt;
245             particle.position(0) += noise(rng);
246             particle.position(1) += noise(rng);
247         }
248     }
249
250     // Update step: weight particles based on measurement likelihood
251     void update(const Vector2d& measurement, double measurement_noise) {
252         double weight_sum = 0.0;
253
254         for (auto& particle : particles) {
255             // Calculate distance from particle to measurement
256             double distance = (particle.position - measurement).norm();
257
258             // Gaussian likelihood (closer = higher weight)
259         }
260     }
261 }
```

```
258     double likelihood = std::exp(-0.5 * (distance * distance) /
259                                     (measurement_noise * measurement_noise
260                                      ));
261
262     particle.weight *= likelihood;
263     weight_sum += particle.weight;
264 }
265
266 // Normalize weights
267 if (weight_sum > 0.0) {
268     for (auto& particle : particles) {
269         particle.weight /= weight_sum;
270     }
271 }
272
273 // Resample particles based on weights (importance resampling)
274 void resample() {
275     std::vector<Particle> new_particles;
276     new_particles.reserve(num_particles);
277
278     // Cumulative sum of weights
279     std::vector<double> cumsum(num_particles);
280     cumsum[0] = particles[0].weight;
281     for (int i = 1; i < num_particles; ++i) {
282         cumsum[i] = cumsum[i-1] + particles[i].weight;
283     }
284
285     // Systematic resampling
286     std::uniform_real_distribution<double> uniform(0.0, 1.0 /
287                                                 num_particles);
288     double r = uniform(rng);
289
290     int idx = 0;
291     for (int i = 0; i < num_particles; ++i) {
292         double u = r + (double)i / num_particles;
293
294         while (idx < num_particles - 1 && u > cumsum[idx]) {
295             idx++;
296         }
297
298         new_particles.emplace_back(particles[idx].position,
299                                   1.0 / num_particles);
300     }
301
302     particles = std::move(new_particles);
303 }
304
305 // Estimate position (weighted average)
306 Vector2d getEstimate() const {
307     Vector2d estimate = Vector2d::Zero();
308
309     for (const auto& particle : particles) {
310         estimate += particle.position * particle.weight;
311     }
312 }
```

```
310     }
311
312     return estimate;
313 }
314
315 // Get effective number of particles (measure of degeneracy)
316 double getEffectiveParticles() const {
317     double weight_sum_sq = 0.0;
318     for (const auto& particle : particles) {
319         weight_sum_sq += particle.weight * particle.weight;
320     }
321     return 1.0 / weight_sum_sq;
322 }
323
324 const std::vector<Particle>& getParticles() const { return particles; }
325 };
326
327 // =====
328 // 5. EXTENDED KALMAN FILTER (EKF) FOR NONLINEAR SYSTEMS
329 // =====
330
331 class ExtendedKalmanFilter {
332 private:
333     VectorXd x;      // State: [x, y, theta, v]
334     MatrixXd P;      // Covariance
335     MatrixXd Q;      // Process noise
336     MatrixXd R;      // Measurement noise
337
338 public:
339     ExtendedKalmanFilter() {
340         // State: [x, y, theta, v] - position, heading, velocity
341         x = VectorXd::Zero(4);
342         P = MatrixXd::Identity(4, 4) * 100.0;
343         Q = MatrixXd::Identity(4, 4) * 0.1;
344         R = MatrixXd::Identity(2, 2) * 5.0;
345     }
346
347     // Predict with nonlinear motion model
348     void predict(double v, double omega, double dt) {
349         // Nonlinear motion model for differential drive robot
350         double theta = x(2);
351
352         // Predict state
353         x(0) += v * std::cos(theta) * dt; // x
354         x(1) += v * std::sin(theta) * dt; // y
355         x(2) += omega * dt; // theta
356         x(3) = v; // velocity
357
358         // Jacobian of motion model
359         MatrixXd F = MatrixXd::Identity(4, 4);
360         F(0, 2) = -v * std::sin(theta) * dt;
361         F(1, 2) = v * std::cos(theta) * dt;
362
363         // Predict covariance
```

```
364     P = F * P * F.transpose() + Q;
365 }
366
367 // Update with GPS measurement [x, y]
368 void update(const Vector2d& measurement) {
369     // Measurement model (linear: measure x, y directly)
370     MatrixXd H = MatrixXd::Zero(2, 4);
371     H(0, 0) = 1.0;
372     H(1, 1) = 1.0;
373
374     // Innovation
375     VectorXd z(2);
376     z << measurement(0), measurement(1);
377     VectorXd y = z - H * x;
378
379     // Innovation covariance
380     MatrixXd S = H * P * H.transpose() + R;
381
382     // Kalman gain
383     MatrixXd K = P * H.transpose() * S.inverse();
384
385     // Update
386     x = x + K * y;
387     P = (MatrixXd::Identity(4, 4) - K * H) * P;
388 }
389
390 Vector2d getPosition() const { return Vector2d(x(0), x(1)); }
391 double getHeading() const { return x(2); }
392 double getVelocity() const { return x(3); }
393 };
394
395 // =====
396 // 6. SENSOR FUSION PIPELINE (Multi-Sensor Integration)
397 // =====
398
399 class SensorFusionPipeline {
400 private:
401     KalmanFilter kf;
402     ComplementaryFilter cf;
403     ParticleFilter pf;
404
405     std::vector<IMUData> imu_buffer;
406     std::vector<GPSData> gps_buffer;
407
408     Vector2d last_position;
409     double last_timestamp;
410
411 public:
412     SensorFusionPipeline()
413         : pf(500), last_position(Vector2d::Zero()), last_timestamp(0.0) {}
414
415     void addIMUData(const IMUData& imu) {
416         imu_buffer.push_back(imu);
```

```
418     // Update complementary filter for orientation
419     if (!imu_buffer.empty()) {
420         double dt = 0.01; // Assume 100 Hz IMU
421         if (imu_buffer.size() > 1) {
422             dt = imu.timestamp - imu_buffer[imu_buffer.size() - 2].timestamp
423                 ;
424         }
425     }
426 }
427
428 void addGPSData(const GPSData& gps) {
429     gps_buffer.push_back(gps);
430
431     // Update Kalman filter
432     if (last_timestamp > 0.0) {
433         double dt = gps.timestamp - last_timestamp;
434         kf.predict(dt);
435     }
436     kf.update(gps.position, gps.accuracy);
437
438     // Update particle filter
439     Vector2d velocity = kf.getVelocity();
440     double dt = gps.timestamp - last_timestamp;
441     if (dt > 0.0) {
442         pf.predict(velocity, dt, 1.0);
443         pf.update(gps.position, gps.accuracy);
444
445         // Resample if particles degenerate
446         if (pf.getEffectiveParticles() < 100) {
447             pf.resample();
448         }
449     }
450
451     last_position = gps.position;
452     last_timestamp = gps.timestamp;
453 }
454
455 Vector2d getKalmanPosition() const { return kf.getPosition(); }
456 Vector2d getParticlePosition() const { return pf.getEstimate(); }
457 Vector3d getOrientation() const { return cf.getOrientationDegrees(); }
458
459 void printStatus() const {
460     std::cout << "\nSensor Fusion Status:\n";
461     std::cout << "-----\n";
462
463     auto kf_pos = kf.getPosition();
464     std::cout << "Kalman Filter: (" << std::fixed << std::setprecision
465         (2)
466             << kf_pos(0) << ", " << kf_pos(1) << ")\n";
467
468     auto pf_pos = pf.getEstimate();
469     std::cout << "Particle Filter: (" << std::fixed << std::setprecision
470         (2)
```

```
469         << pf_pos(0) << ", " << pf_pos(1) << ")\\n";
470
471     auto orient = cf.getOrientationDegrees();
472     std::cout << "Orientation: Roll=" << std::fixed << std::
473             setprecision(1)
474             << orient(0) << "° Pitch=" << orient(1)
475             << "° Yaw=" << orient(2) << "°\\n";
476
477     std::cout << "Effective Particles: " << (int)pf.getEffectiveParticles
478             () << "\\n";
479 }
480
481 // =====
482 // DEMONSTRATION FUNCTIONS
483 // =====
484 void demonstrate_kalman_filter() {
485     std::cout << "=====\\n"
486             ;
487     std::cout << "1. KALMAN FILTER - GPS/IMU FUSION\\n";
488     std::cout << "=====\\n"
489             ;
490     std::cout << "Scenario: Robot moving in a circle, fusing GPS measurements\\
491             n\\n";
492
493     KalmanFilter kf;
494     std::mt19937 rng(42);
495     std::normal_distribution<double> gps_noise(0.0, 3.0); // 3m GPS noise
496
497     // Simulate circular motion
498     double dt = 0.1; // 10 Hz
499     double radius = 50.0;
500     double angular_vel = 0.1; // rad/s
501
502     std::cout << std::fixed << std::setprecision(2);
503     std::cout << "Time True Position GPS Measurement Kalman Estimate
504             Error\\n";
505     std::cout << "---- ----- ----- -----\\n";
506
507     for (int i = 0; i < 20; ++i) {
508         double t = i * dt;
509
510         // True position (circular motion)
511         Vector2d true_pos(radius * std::cos(angular_vel * t),
512                           radius * std::sin(angular_vel * t));
513
514         // Noisy GPS measurement
515         Vector2d gps_measurement = true_pos + Vector2d(gps_noise(rng),
516                                                       gps_noise(rng));
517
518         // Kalman filter predict and update
519         kf.predict(dt);
```

```
515     kf.update(gps_measurement, 3.0);
516
517     Vector2d kf_estimate = kf.getPosition();
518     double error = (kf_estimate - true_pos).norm();
519
520     std::cout << std::setw(4) << t << " "
521             << "(" << std::setw(6) << true_pos(0) << ", "
522             << std::setw(6) << true_pos(1) << ")"
523             << "(" << std::setw(6) << gps_measurement(0) << ", "
524             << std::setw(6) << gps_measurement(1) << ")"
525             << "(" << std::setw(6) << kf_estimate(0) << ", "
526             << std::setw(6) << kf_estimate(1) << ")"
527             << std::setw(5) << error << "m\n";
528 }
529
530 std::cout << "\n Kalman filter smooths noisy GPS and predicts motion!\n";
531 }
532
533 void demonstrate_complementary_filter() {
534     std::cout << "\n=====\\n";
535     std::cout << "2. COMPLEMENTARY FILTER - ACCELEROMETER/GYROSCOPE FUSION\\n";
536     std::cout << "=====\\n";
537     std::cout << "Scenario: IMU measuring tilt angles (roll, pitch)\\n\\n";
538
539     ComplementaryFilter cf(0.98);
540     std::mt19937 rng(42);
541     std::normal_distribution<double> accel_noise(0.0, 0.5);
542     std::normal_distribution<double> gyro_noise(0.0, 0.01);
543
544     double dt = 0.01; // 100 Hz IMU
545     double true_roll = 0.0;
546     double true_pitch = 0.0;
547
548     std::cout << std::fixed << std::setprecision(2);
549     std::cout << "Time    True Angles        Accel Angles        Filtered Angles\\n";
550     std::cout << "----  -----  -----  -----\\n";
551
552     for (int i = 0; i < 15; ++i) {
553         double t = i * 0.1;
554
555         // Simulate sinusoidal motion
556         true_roll = 30.0 * std::sin(2 * M_PI * 0.5 * t) * M_PI / 180.0; // ±30°
557         true_pitch = 20.0 * std::cos(2 * M_PI * 0.3 * t) * M_PI / 180.0; // ±20°
558
559         // Simulate accelerometer (measures gravity vector)
560         double g = 9.81;
561         Vector3d true_accel(-g * std::sin(true_pitch),
562                             g * std::sin(true_roll) * std::cos(true_pitch),
```

```

563                     g * std::cos(true_roll) * std::cos(true_pitch));
564
565     Vector3d accel = true_accel + Vector3d(accel_noise(rng),
566                                              accel_noise(rng),
567                                              accel_noise(rng));
568
569     // Simulate gyroscope (measures angular rates)
570     double roll_rate = 30.0 * 2 * M_PI * 0.5 * std::cos(2 * M_PI * 0.5 * t
571     ) * M_PI / 180.0;
572     double pitch_rate = -20.0 * 2 * M_PI * 0.3 * std::sin(2 * M_PI * 0.3 * t
573     ) * M_PI / 180.0;
574
575     Vector3d gyro(roll_rate + gyro_noise(rng),
576                   pitch_rate + gyro_noise(rng),
577                   gyro_noise(rng));
578
579     // Update filter 10 times (simulating 100 Hz between prints)
580     for (int j = 0; j < 10; ++j) {
581         cf.update(accel, gyro, dt);
582     }
583
584     // Calculate angles from accelerometer only (for comparison)
585     double accel_roll = std::atan2(accel.y(), accel.z()) * 180.0 / M_PI;
586     double accel_pitch = std::atan2(-accel.x(),
587                                     std::sqrt(accel.y()*accel.y() + accel.z()
588                                     ()*accel.z())))
589     * 180.0 / M_PI;
590
591     auto filtered = cf.getOrientationDegrees();
592
593     std::cout << std::setw(4) << t << " "
594     << "R=" << std::setw(6) << (true_roll * 180.0 / M_PI)
595     << "° P=" << std::setw(6) << (true_pitch * 180.0 / M_PI) <<
596     "° "
597     << "R=" << std::setw(6) << accel_roll
598     << "° P=" << std::setw(6) << accel_pitch << "° "
599     << "R=" << std::setw(6) << filtered(0)
600     << "° P=" << std::setw(6) << filtered(1) << "°\n";
601 }
602
603 void demonstrate_particle_filter() {
604     std::cout << "\n=====\\n";
605     std::cout << "3. PARTICLE FILTER - ROBOT LOCALIZATION\\n";
606     std::cout << "=====\\n";
607     ;
608     std::cout << "Scenario: Robot with odometry and GPS, 500 particles\\n\\n";
609     ParticleFilter pf(500);
610     std::mt19937 rng(42);

```

```
610     std::normal_distribution<double> gps_noise(0.0, 5.0);
611
612     // Robot starts at origin, moves in square path
613     Vector2d true_pos(0.0, 0.0);
614     Vector2d velocity(2.0, 0.0); // 2 m/s
615     double dt = 1.0;
616
617     std::cout << std::fixed << std::setprecision(2);
618     std::cout << "Time      True Position      GPS Measurement      Particle
619             Estimate      Error      N_eff\n";
620     std::cout << "----      -----      -----      -----";
621     std::cout << "-----\n";
622
623     for (int i = 0; i < 20; ++i) {
624         // Move robot (square path)
625         if (i % 5 == 0 && i > 0) {
626             // Turn 90 degrees
627             double temp = velocity(0);
628             velocity(0) = -velocity(1);
629             velocity(1) = temp;
630         }
631
632         true_pos += velocity * dt;
633
634         // Particle filter prediction
635         pf.predict(velocity, dt, 0.5);
636
637         // GPS measurement (every 2 seconds)
638         if (i % 2 == 0) {
639             Vector2d gps = true_pos + Vector2d(gps_noise(rng), gps_noise(rng));
640             ;
641             pf.update(gps, 5.0);
642
643             // Resample if needed
644             if (pf.getEffectiveParticles() < 100) {
645                 pf.resample();
646             }
647
648             std::cout << std::setw(4) << (i * dt) << "      "
649             << "(" << std::setw(6) << true_pos(0) << ", "
650             << std::setw(6) << true_pos(1) << ")      "
651             << "(" << std::setw(6) << gps(0) << ", "
652             << std::setw(6) << gps(1) << ")      ";
653
654         } else {
655             std::cout << std::setw(4) << (i * dt) << "      "
656             << "(" << std::setw(6) << true_pos(0) << ", "
657             << std::setw(6) << true_pos(1) << ")      "
658             << "      No GPS      ";
659
660         }
661
662         Vector2d estimate = pf.getEstimate();
663         double error = (estimate - true_pos).norm();
664
665         std::cout << "(" << std::setw(6) << estimate(0) << ", "
666             << std::setw(6) << estimate(1) << ")\n";
667     }
668 }
```

```
661         << std::setw(6) << estimate(1) << ")      "
662         << std::setw(5) << error << "m   "
663         << std::setw(5) << (int)pf.getEffectiveParticles() << "\n";
664     }
665
666     std::cout << "\n Particle filter handles multimodal distributions and
667     nonlinearity!\n";
668 }
669
670 void demonstrate_sensor_fusion_pipeline() {
671     std::cout << "\n=====\
672     =====\n";
673     std::cout << "4. COMPLETE SENSOR FUSION PIPELINE\n";
674     std::cout << "=====\
675     =====\n";
676     std::cout << "Scenario: Drone with IMU (100 Hz) and GPS (1 Hz)\n\n";
677
678 SensorFusionPipeline pipeline;
679 std::mt19937 rng(42);
680 std::normal_distribution<double> accel_noise(0.0, 0.3);
681 std::normal_distribution<double> gyro_noise(0.0, 0.01);
682 std::normal_distribution<double> gps_noise(0.0, 3.0);
683
684 double t = 0.0;
685 double dt_imu = 0.01;    // 100 Hz
686 double dt_gps = 1.0;    // 1 Hz
687
688 Vector2d position(0.0, 0.0);
689 Vector2d velocity(5.0, 3.0); // 5 m/s east, 3 m/s north
690
691 std::cout << "Fusing IMU (100 Hz) with GPS (1 Hz)... \n\n";
692
693 for (int i = 0; i < 10; ++i) {
694     // Simulate 100 IMU samples between GPS updates
695     for (int j = 0; j < 100; ++j) {
696         t += dt_imu;
697
698         // Simulate IMU data
699         Vector3d accel(accel_noise(rng), accel_noise(rng),
700                         9.81 + accel_noise(rng));
701         Vector3d gyro(gyro_noise(rng), gyro_noise(rng),
702                         0.1 + gyro_noise(rng)); // 0.1 rad/s yaw rate
703
704         IMUData imu(accel, gyro, t);
705         pipeline.addIMUData(imu);
706     }
707
708     // GPS update (1 Hz)
709     position += velocity * dt_gps;
710     Vector2d gps_pos = position + Vector2d(gps_noise(rng), gps_noise(rng))
711     ;
712     GPSData gps(gps_pos, 3.0, t);
713     pipeline.addGPSData(gps);
714 }
```

```
711     // Print status
712     std::cout << "t = " << std::fixed << std::setprecision(1)
713         << t << "s:\n";
714     pipeline.printStatus();
715     std::cout << "\n";
716 }
717
718 std::cout << " Complete sensor fusion combines all algorithms!\n";
719 }
720
721 // =====
722 // MAIN FUNCTION
723 // =====
724
725 int main() {
726     std::cout << "\n";
727     std::cout << "
728         =====\n"
729         ;
730     std::cout << "EIGEN LIBRARY: SENSOR FUSION AND PARTICLE FILTER\n";
731     std::cout << "
732         =====\n"
733         ;
734     std::cout << "Advanced robotics and embedded systems examples\n";
735     std::cout << "
736         =====\n"
737         ;
738
739     try {
740         demonstrate_kalman_filter();
741         demonstrate_complementary_filter();
742         demonstrate_particle_filter();
743         demonstrate_sensor_fusion_pipeline();
744
745         std::cout << "\n
746             =====\n";
747         std::cout << "SUMMARY\n";
748         std::cout << "
749             =====\n";
750         std::cout << "\n";
751         std::cout << "Algorithms Demonstrated:\n";
752         std::cout << " 1. Kalman Filter      - Optimal linear sensor
753             fusion\n";
754         std::cout << " 2. Complementary Filter - Lightweight IMU fusion\n";
755         std::cout << " 3. Particle Filter     - Nonlinear localization\n";
756         std::cout << " 4. Complete Pipeline    - Multi-sensor integration\n";
757         ;
758         std::cout << "\n";
759         std::cout << "Applications:\n";
760         std::cout << "  • Drone attitude estimation (IMU)\n";
761         std::cout << "  • Robot localization (GPS + odometry)\n";
762         std::cout << "  • Autonomous vehicles (multi-sensor fusion)\n";
763 }
```

```

753     std::cout << " • Indoor navigation (WiFi + dead reckoning)\n";
754     std::cout << " • Wearable devices (activity recognition)\n";
755     std::cout << "\n";
756     std::cout << "Eigen Library Benefits:\n";
757     std::cout << "     Fast matrix operations (vectorized)\n";
758     std::cout << "     Header-only (easy integration)\n";
759     std::cout << "     Industry standard (used in ROS, OpenCV)\n";
760     std::cout << "     Compile-time optimization\n";
761     std::cout << "     Suitable for embedded systems\n";
762     std::cout << "\n";
763     std::cout << "
764     =====\n";
765     std::cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!\n";
766     std::cout << "
767     =====\n";
768 } catch (const std::exception& e) {
769     std::cerr << "Error: " << e.what() << std::endl;
770     return 1;
771 }
772
773 }
774
775 /*
776 EXPECTED OUTPUT:
777 =====
778 =====
779 1. KALMAN FILTER - GPS/IMU FUSION
780 =====
781 Scenario: Robot moving in a circle, fusing GPS measurements
782
783
784 Time      True Position      GPS Measurement      Kalman Estimate      Error
785 -----  -----
786 0.0      ( 50.00,    0.00)      ( 50.42,   -1.83)      ( 50.21,   -0.91)      1.06m
787 0.1      ( 49.95,    4.99)      ( 52.45,    7.12)      ( 51.33,    3.11)      2.41m
788 0.2      ( 49.80,    9.98)      ( 48.73,   12.89)      ( 50.03,   11.44)      2.02m
789 ...
790
791 Kalman filter smooths noisy GPS and predicts motion!
792
793 =====
794 2. COMPLEMENTARY FILTER - ACCELEROMETER/GYROSCOPE FUSION
795 =====
796 Scenario: IMU measuring tilt angles (roll, pitch)
797
798 Time      True Angles      Accel Angles      Filtered Angles
799 -----  -----
800 0.0      R= 0.00° P= 20.00°      R= -0.53° P= 20.41°      R= -0.05° P= 20.04°
801 0.1      R= 14.63° P= 19.40°      R= 14.98° P= 19.78°      R= 14.68° P= 19.45°
802 ...

```

```
803
804     Complementary filter combines fast gyro and stable accel!
805
806 =====
807 3. PARTICLE FILTER - ROBOT LOCALIZATION
808 =====
809 Scenario: Robot with odometry and GPS, 500 particles
810
811 Time    True Position        GPS Measurement      Particle Estimate    Error    N_eff
812 ----  -----  -----  -----  -----  -----  -----
813 0.0    ( 2.00,    0.00)    ( 4.42,   -3.57)    ( 3.21,   -1.79)    2.49m    500
814 2.0    ( 4.00,    0.00)    ( 5.73,   -1.08)    ( 4.86,   -0.54)    1.23m    487
815 ...
816
817     Particle filter handles multimodal distributions and nonlinearity!
818
819 =====
820 4. COMPLETE SENSOR FUSION PIPELINE
821 =====
822 Scenario: Drone with IMU (100 Hz) and GPS (1 Hz)
823
824 Fusing IMU (100 Hz) with GPS (1 Hz)...
825
826 t = 1.0s:
827
828 Sensor Fusion Status:
829 -----
830 Kalman Filter: (5.00, 3.00)
831 Particle Filter: (5.00, 3.00)
832 Orientation: Roll=0.0° Pitch=0.0° Yaw=5.7°
833 Effective Particles: 500
834 ...
835
836 Complete sensor fusion combines all algorithms!
837
838 APPLICATIONS IN EMBEDDED SYSTEMS:
839 =====
840
841 1. DRONE FLIGHT CONTROLLER
842     - Complementary filter for attitude (roll, pitch, yaw)
843     - Kalman filter for position/velocity estimation
844     - Runs at 500 Hz on STM32F4 (168 MHz ARM Cortex-M4)
845
846 2. AUTONOMOUS ROBOT
847     - Particle filter for localization
848     - EKF for sensor fusion (LIDAR + wheel encoders)
849     - Real-time operation on Raspberry Pi
850
851 3. WEARABLE FITNESS TRACKER
852     - Complementary filter for step detection
853     - Lightweight sensor fusion (accelerometer + gyroscope)
854     - Low power consumption (<1 mW)
855
856 4. INDOOR POSITIONING SYSTEM
```

```
857     - Particle filter with WiFi RSSI measurements
858     - Dead reckoning with IMU between updates
859     - Accuracy: 2-5 meters
860
861 PERFORMANCE CHARACTERISTICS:
862 =====
863
864 Algorithm           Computational Cost      Memory Usage      Accuracy
865 -----           -----      -----
866 Kalman Filter      O(n2) per update      O(n2)          Optimal (linear)
867 Complementary Filter O(1) per sample      O(1)            Good (simple)
868 Particle Filter     O(N) per update      O(N)            Excellent (
869           nonlinear)
870 EKF                O(n2) per update      O(n2)          Good (linearized)
871 Where: n = state dimension, N = number of particles
872
873 EIGEN LIBRARY ADVANTAGES:
874 =====
875
876 1. PERFORMANCE
877     - Vectorization (SSE, AVX)
878     - Loop unrolling
879     - Cache optimization
880     - Compile-time size optimization
881
882 2. EASE OF USE
883     - Intuitive matrix syntax
884     - Automatic memory management
885     - Expression templates
886
887 3. PORTABILITY
888     - Header-only (no linking)
889     - Cross-platform
890     - ARM NEON support for embedded
891
892 4. INDUSTRY ADOPTION
893     - ROS (Robot Operating System)
894     - OpenCV (Computer Vision)
895     - TensorFlow (Machine Learning)
896     - Many robotics companies
897
898 BUILD INSTRUCTIONS:
899 =====
900
901 # Install Eigen
902 sudo apt-get install libeigen3-dev  # Ubuntu/Debian
903 brew install eigen                  # macOS
904 vcpkg install eigen3               # Windows
905
906 # Compile
907 g++ -std=c++17 -O3 -march=native -I/usr/include/eigen3 \
908   EigenSensorFusion.cpp -o EigenSensorFusion
909
```

```
910 # For embedded (ARM Cortex-M4 with NEON)
911 arm-none-eabi-g++ -std=c++17 -O3 -mfpu=neon -mthumb -mcpu=cortex-m4 \
912     -I/path/to/eigen EigenSensorFusion.cpp
913
914 # CMake
915 find_package(Eigen3 REQUIRED)
916 target_link_libraries(EigenSensorFusion Eigen3::Eigen)
917
918 FURTHER READING:
919 =====
920
921 1. Kalman Filter:
922     - "Kalman and Bayesian Filters in Python" by Roger Labbe
923     - http://doi.org/10.1109/PROC.1976.10155
924
925 2. Particle Filter:
926     - "Probabilistic Robotics" by Thrun, Burgard, Fox
927     - Monte Carlo Localization (MCL)
928
929 3. Sensor Fusion:
930     - "State Estimation for Robotics" by Timothy Barfoot
931     - "Optimal State Estimation" by Dan Simon
932
933 4. Eigen Library:
934     - https://eigen.tuxfamily.org/
935     - Eigen documentation and tutorials
936 */
```

## 22 Source Code: EmbeddedSystemsAvoid.cpp

File: src/EmbeddedSystemsAvoid.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
4 #include <memory>
5 #include <cstdint>
6 #include <array>
7 #include <cstring>
8
9 // =====
10 // WHY AVOID THESE IN EMBEDDED SYSTEMS - DETAILED EXPLANATIONS
11 // =====
12 // Target System Specification (Typical Microcontroller):
13 // - MCU: ARM Cortex-M4 @ 80MHz
14 // - Flash: 256KB (code storage)
15 // - RAM: 64KB (data, stack, heap)
16 // - Stack: 4KB (limited call depth)
17 // - Heap: 8KB (if enabled, often disabled)
18 // - No MMU (no virtual memory, no memory protection)
19 // - Real-time constraints: 1ms task deadlines
20 // =====
21
22 // =====
23 // 1. PROBLEM: new/delete AND HEAP ALLOCATION
24 // =====
25
26 void demonstrate_heap_fragmentation() {
27     std::cout << "\n== 1. PROBLEM: new/delete (HEAP FRAGMENTATION) ==" <<
28         std::endl;
29
30     std::cout << "\n TARGET SYSTEM CONSTRAINTS:" << std::endl;
31     std::cout << " • Heap size: 8KB (8192 bytes)" << std::endl;
32     std::cout << " • No MMU, no memory defragmentation" << std::endl;
33     std::cout << " • Fragmentation = permanent until reboot" << std::endl;
34
35     std::cout << "\n BAD CODE (Using new/delete):" << std::endl;
36     std::cout << R"(
37         // Allocate various sizes over time
38         void* ptr1 = new char[100];    // 100 bytes
39         void* ptr2 = new char[200];    // 200 bytes
40         void* ptr3 = new char[100];    // 100 bytes
41
42         delete[] ptr2;    // Free middle block
43
44         // Now heap looks like: [100 used] [200 FREE] [100 used]
45         // Try to allocate 250 bytes - FAILS!
46         // Even though 200 bytes are free, they're not contiguous
47         void* ptr4 = new char[250];    // Out of memory!
48     )" << std::endl;
```

```
49     std::cout << " PROBLEMS:" << std::endl;
50     std::cout << " 1. Fragmentation: Free memory becomes unusable" << std::endl;
51     std::cout << " 2. Non-deterministic: Allocation time varies (0.1μs to 10
52         ms)" << std::endl;
53     std::cout << " 3. Leaks: One forgotten delete = permanent memory loss" <<
54         std::endl;
55     std::cout << " 4. Overhead: 8-16 bytes per allocation for bookkeeping" <<
56         std::endl;
57     std::cout << " 5. System failure: malloc() returns nullptr, hard to
58         recover" << std::endl;
59
60
61
62     std::cout << "\n SOLUTION: Memory Pools" << std::endl;
63     std::cout << " • Pre-allocated fixed-size blocks" << std::endl;
64     std::cout << " • O(1) allocation time: always <1μs" << std::endl;
65     std::cout << " • No fragmentation" << std::endl;
66     std::cout << " • Deterministic behavior" << std::endl;
67
68
69 // =====
70 // 2. PROBLEM: std::vector (DYNAMIC REALLOCATION)
71 // =====
72
73 void demonstrate_vector_problems() {
74     std::cout << "\n==== 2. PROBLEM: std::vector (DYNAMIC REALLOCATION) ===" <<
75         std::endl;
76
77     std::cout << "\n SCENARIO: Sensor data collection" << std::endl;
78     std::cout << " • Need to store 100 sensor readings" << std::endl;
79     std::cout << " • Each reading: 4 bytes (float)" << std::endl;
80     std::cout << " • Expected memory: 400 bytes" << std::endl;
81
82     std::cout << "\n BAD CODE (Using std::vector):" << std::endl;
83     std::cout << R"(
84         std::vector<float> sensor_data;
85
86         for (int i = 0; i < 100; ++i) {
87             sensor_data.push_back(read_sensor()); // Unpredictable!
88         }
89     )" << std::endl;
90
91     std::cout << " WHAT ACTUALLY HAPPENS:" << std::endl;
92     std::cout << " Iteration 1: Allocate 4 bytes (capacity=1)" << std::endl;
93     std::cout << " Iteration 2: Reallocate to 8 bytes, copy 1 element" << std::endl;
94     std::cout << " Iteration 3: Reallocate to 16 bytes, copy 2 elements" <<
95         std::endl;
96     std::cout << " Iteration 4: Reallocate to 32 bytes, copy 4 elements" <<
```

```
        std::endl;
94     std::cout << "    Iteration 9: Reallocate to 64 bytes, copy 8 elements" <<
95         std::endl;
96     std::cout << "    ... (continues with exponential growth)" << std::endl;
97
98     std::cout << "\n PROBLEMS:" << std::endl;
99     std::cout << "    1. Multiple allocations: ~7 allocations for 100 elements"
100        << std::endl;
101    std::cout << "    2. Memory copies: ~100 element copies total" << std::endl;
102    std::cout << "    3. Peak memory: Needs 2x during reallocation" << std::endl;
103        ;
104    std::cout << "    (800 bytes instead of 400 bytes!)" << std::endl;
105    std::cout << "    4. Heap fragmentation: Old blocks left behind" << std::
106        endl;
107    std::cout << "    5. Timing: Each push_back() takes different time" << std::
108        endl;
109    std::cout << "        - Normal: 0.1µs" << std::endl;
110    std::cout << "        - Reallocation: 50µs (500x slower!)" << std::endl;
111
112    std::cout << "\n SOLUTION: std::array or fixed buffer" << std::endl;
113    std::cout << R"(
114        std::array<float, 100> sensor_data; // Fixed size, stack allocated
115        size_t count = 0;
116
117        for (int i = 0; i < 100; ++i) {
118            sensor_data[count++] = read_sensor(); // Constant time!
119        }
120    )" << std::endl;
121
122    std::cout << "\n MEMORY COMPARISON:" << std::endl;
123    std::cout << "    std::vector: 400 bytes + 24 bytes overhead + heap
124        fragmentation" << std::endl;
125    std::cout << "    std::array: 400 bytes (stack) + 0 bytes overhead" << std
126        ::endl;
127 }
128
129 // =====
130 // 3. PROBLEM: std::string (DYNAMIC ALLOCATION)
131 // =====
132
133 void demonstrate_string_problems() {
134     std::cout << "\n==== 3. PROBLEM: std::string (DYNAMIC ALLOCATION) ===" <<
135         std::endl;
136
137     std::cout << "\n SCENARIO: UART message building" << std::endl;
138     std::cout << "    • Need message: \"TEMP:25.5C\r\n\" " << std::endl;
139     std::cout << "    • Length: 13 characters" << std::endl;
140
141     std::cout << "\n BAD CODE (Using std::string):" << std::endl;
142     std::cout << R"(
143         std::string msg = "TEMP:";           // Allocation 1
144         msg += std::to_string(25.5);        // Allocation 2 + reallocation
145         msg += "C\r\n";                  // Possible reallocation 3
146     )"
```

```
139     send_uart(msg.c_str());
140     )" << std::endl;
141
142     std::cout << "  PROBLEMS:" << std::endl;
143     std::cout << "    1. Multiple allocations: 2-3 heap allocations" << std::endl;
144     std::cout << "    2. SSO ambiguity: Small String Optimization" << std::endl;
145     std::cout << "      - Strings <16 chars: stored inline (good)" << std::endl;
146     std::cout << "      - Strings >=16 chars: heap allocation (bad)" << std::endl;
147     std::cout << "      - Behavior changes at runtime!" << std::endl;
148     std::cout << "    3. Hidden cost: std::to_string() always allocates" << std::endl;
149     std::cout << "    4. Exception throwing: Can throw std::bad_alloc" << std::endl;
150
151     std::cout << "\n MEMORY COST:" << std::endl;
152     std::cout << "  • std::string object: 24-32 bytes" << std::endl;
153     std::cout << "  • String data: 13 bytes" << std::endl;
154     std::cout << "  • Heap overhead: 8-16 bytes" << std::endl;
155     std::cout << "  • Total: ~45-60 bytes for 13-char string!" << std::endl;
156
157     std::cout << "\n SOLUTION: Fixed-size buffer" << std::endl;
158     std::cout << R"(
159     char msg[32];
160     snprintf(msg, sizeof(msg), "TEMP:%.1fC\r\n", 25.5f);
161     send_uart(msg);
162
163     // Total stack usage: 32 bytes, zero heap allocations
164     )" << std::endl;
165
166     std::cout << "\n TIMING COMPARISON:" << std::endl;
167     std::cout << "  std::string version: 15-50µs (unpredictable)" << std::endl;
168     std::cout << "  snprintf version:      2-5µs (deterministic)" << std::endl;
169 }
170
171 // =====
172 // 4. PROBLEM: VIRTUAL FUNCTIONS (VTABLE OVERHEAD)
173 // =====
174
175 class BaseSensor {
176 public:
177     virtual ~BaseSensor() = default;
178     virtual float read() = 0;
179     virtual const char* name() = 0;
180 };
181
182 class TempSensor : public BaseSensor {
183 public:
184     float read() override { return 25.5f; }
185     const char* name() override { return "Temperature"; }
186 };
```

```
187
188 void demonstrate_virtual_function_overhead() {
189     std::cout << "\n==== 4. PROBLEM: VIRTUAL FUNCTIONS (VTABLE) ===" << std::endl;
190
191     std::cout << "\n MEMORY OVERHEAD PER OBJECT:" << std::endl;
192     std::cout << " • vtable pointer (vptr): 4 bytes (32-bit) or 8 bytes (64-bit)" << std::endl;
193     std::cout << " • vtable itself: 8 bytes per virtual function" << std::endl;
194     std::cout << " • Small objects become bloated!" << std::endl;
195
196     std::cout << "\nExample object sizes:" << std::endl;
197     std::cout << "     sizeof(BaseSensor) = " << sizeof(BaseSensor) << " bytes" << std::endl;
198     std::cout << "     sizeof(TempSensor) = " << sizeof(TempSensor) << " bytes" << std::endl;
199
200     std::cout << "\n PROBLEMS:" << std::endl;
201     std::cout << " 1. RAM overhead: Every object has vptr (4-8 bytes)" << std::endl;
202     std::cout << "      - With 100 sensors: 400-800 bytes wasted!" << std::endl;
203     std::cout << " 2. Flash overhead: vtable stored in Flash" << std::endl;
204     std::cout << "      - 3 virtual functions = 24 bytes Flash per class" << std::endl;
205     std::cout << " 3. Indirection: Function call requires:" << std::endl;
206     std::cout << "      a. Load vptr from object (1 memory access)" << std::endl;
207     std::cout << "      b. Load function pointer from vtable (2nd memory access)" << std::endl;
208     std::cout << "      c. Indirect jump (breaks CPU pipeline)" << std::endl;
209     std::cout << " 4. Not inlineable: Compiler can't optimize" << std::endl;
210     std::cout << " 5. Cache misses: Vtable and code in different locations" << std::endl;
211
212     std::cout << "\n PERFORMANCE IMPACT:" << std::endl;
213     std::cout << "  Direct call: 1-2 CPU cycles" << std::endl;
214     std::cout << "  Virtual call: 10-20 CPU cycles (10x slower!)" << std::endl;
215     std::cout << " • At 80MHz: Virtual call = 125-250ns overhead" << std::endl;
216     std::cout << " • In 1ms task: Can fit 4000 direct calls or 400 virtual calls" << std::endl;
217
218     std::cout << "\n SOLUTION: Static polymorphism (CRTP or templates)" << std::endl;
219     std::cout << R"(  

220     template<typename Derived>  

221     class SensorBase {  

222         float read() { return static_cast<Derived*>(this)->read_impl(); }  

223     };  

224
225 // Zero overhead, fully inlineable, no vtable!
```

```
226     )" << std::endl;
227 }
228
229 // =====
230 // 5. PROBLEM: RTTI (Runtime Type Information)
231 // =====
232
233 void demonstrate_rtti_overhead() {
234     std::cout << "\n==== 5. PROBLEM: RTTI (typeid, dynamic_cast) ===" << std::endl;
235
236     std::cout << "\n OVERHEAD:" << std::endl;
237     std::cout << " • Type info in Flash: 20-50 bytes per polymorphic class"
238         << std::endl;
239     std::cout << " • Name strings: Variable length (stored in Flash)" << std::endl;
240
241     std::cout << " • Dynamic_cast: String comparison overhead" << std::endl;
242
243     std::cout << "\n BAD CODE:" << std::endl;
244     std::cout << R"(

245         BaseSensor* sensor = get_sensor();

246         // dynamic_cast requires RTTI
247         if (TempSensor* temp = dynamic_cast<TempSensor*>(sensor)) {
248             // Process temperature sensor
249         }
250     )" << std::endl;

251     std::cout << "\n PROBLEMS:" << std::endl;
252     std::cout << " 1. Flash overhead: RTTI adds 5-10KB to binary size" << std::endl;
253         << std::endl;
254     std::cout << "      - On 256KB Flash: 2-4% wasted!" << std::endl;
255     std::cout << " 2. RAM overhead: Type info structures in RAM" << std::endl;
256         << std::endl;
257     std::cout << " 3. CPU overhead: String comparisons for casts" << std::endl;
258         << std::endl;
259     std::cout << " 4. Complexity: Name mangling, RTTI tables" << std::endl;
260
261     std::cout << "\n TIMING:" << std::endl;
262     std::cout << " static_cast<>: 0 cycles (compile-time)" << std::endl;
263     std::cout << " dynamic_cast<>: 100-500 cycles (runtime check!)" << std::endl;
264         << std::endl;
265
266     std::cout << "\n SOLUTION: std::variant or manual type tags" << std::endl;
267         ;
268     std::cout << R"(

269         enum class SensorType { Temperature, Humidity };

270
271         struct Sensor {
272             SensorType type;
273             float value;
274         };
275
276         // Check type with simple enum comparison (1 cycle)
```

```
272     if (sensor.type == SensorType::Temperature) { ... }
273     )" << std::endl;
274
275     std::cout << "\n GCC FLAG:" << std::endl;
276     std::cout << " -fno-rtti (disables RTTI, saves 5-10KB)" << std::endl;
277 }
278
279 // =====
280 // 6. PROBLEM: EXCEPTIONS
281 // =====
282
283 void demonstrate_exception_overhead() {
284     std::cout << "\n==== 6. PROBLEM: EXCEPTIONS ===" << std::endl;
285
286     std::cout << "\n OVERHEAD:" << std::endl;
287     std::cout << " • Exception tables in Flash: 10-30KB" << std::endl;
288     std::cout << " • Unwinding code: 5-10KB" << std::endl;
289     std::cout << " • Per-function overhead: Extra instructions for cleanup"
290             << std::endl;
291     std::cout << " • Total: 15-40KB on 256KB Flash (6-15%!)" << std::endl;
292
293     std::cout << "\n BAD CODE:" << std::endl;
294     std::cout << R"(

295     try {
296         sensor_data.push_back(value); // May throw std::bad_alloc
297         config.parse(json_string); // May throw parse_error
298         send_network(data); // May throw network_error
299     } catch (const std::exception& e) {
300         log_error(e.what());
301     }
302 )" << std::endl;
303
304     std::cout << "\n PROBLEMS:" << std::endl;
305     std::cout << " 1. Code bloat: Exception tables for every function" << std::endl;
306     std::cout << " 2. Unpredictable: Stack unwinding takes unknown time" << std::endl;
307     std::cout << "     - Normal: 0ns (no exception)" << std::endl;
308     std::cout << "     - Exception: 10-1000µs (milliseconds!)" << std::endl;
309     std::cout << " 3. RAM for stack: Need extra stack for unwinding" << std::endl;
310     std::cout << " 4. Real-time unsafe: Can't use in ISRs" << std::endl;
311     std::cout << " 5. Hidden control flow: Function can exit anywhere" << std::endl;
312
313     std::cout << "\n REAL-TIME VIOLATION:" << std::endl;
314     std::cout << " Task deadline: 1ms" << std::endl;
315     std::cout << " Exception thrown: 500µs to unwind" << std::endl;
316     std::cout << " Result: DEADLINE MISSED! System fails!" << std::endl;
317
318     std::cout << "\n SOLUTION: Error codes or std::optional" << std::endl;
319     std::cout << R"(

320     enum class Error { None, OutOfMemory, ParseError, NetworkError };
```

```
321     Error result = sensor_data.add(value);
322     if (result != Error::None) {
323         handle_error(result); // Explicit, predictable
324     }
325
326     // Or use std::optional (C++17)
327     std::optional<float> maybe_value = read_sensor();
328     if (maybe_value) {
329         process(*maybe_value);
330     }
331     )" << std::endl;
332
333     std::cout << "\n GCC FLAG:" << std::endl;
334     std::cout << " -fno-exceptions (disables exceptions, saves 15-40KB)" <<
335         std::endl;
336
337 // =====
338 // 7. PROBLEM: iostream (CODE BLOAT)
339 // =====
340
341 void demonstrate_iostream_bloat() {
342     std::cout << "\n==== 7. PROBLEM: iostream (CODE BLOAT) ===" << std::endl;
343
344     std::cout << "\n BINARY SIZE IMPACT:" << std::endl;
345
346     std::cout << "\n BAD CODE:" << std::endl;
347     std::cout << R"(

348 #include <iostream>

349

350     int main() {
351         std::cout << "Temperature: " << 25.5 << "C" << std::endl;
352         return 0;
353     }
354 )" << std::endl;
355
356     std::cout << "\n BINARY SIZE:" << std::endl;
357     std::cout << " Without iostream: 2-5KB" << std::endl;
358     std::cout << " With iostream: 60-150KB (!)" << std::endl;
359     std::cout << " Increase: 30-60x larger!" << std::endl;
360
361     std::cout << "\n WHAT'S INCLUDED:" << std::endl;
362     std::cout << " • Locale support: 20-40KB" << std::endl;
363     std::cout << " • Formatting code: 15-30KB" << std::endl;
364     std::cout << " • Stream state management: 10-20KB" << std::endl;
365     std::cout << " • Virtual functions (std::ios_base): vtables" << std::endl;
366         ;
367     std::cout << " • Static initialization: Global constructors" << std::endl;
368         ;
369
370     std::cout << "\n SOLUTION: printf/snprintf" << std::endl;
371     std::cout << R"(

#include <stdio.h>
```

```
372 int main() {
373     printf("Temperature: %.1fC\n", 25.5);
374     return 0;
375 }
376
377 // Binary size: 5-10KB (10x smaller!)
378 )" << std::endl;
379
380 std::cout << "\n COMPARISON (ARM Cortex-M4):" << std::endl;
381 std::cout << "    iostream: 80KB Flash + 2KB RAM" << std::endl;
382 std::cout << "    printf:    8KB Flash + 512 bytes RAM" << std::endl;
383 std::cout << "    sprintf:  6KB Flash + 0 bytes heap" << std::endl;
384
385 std::cout << "\n ON 256KB FLASH:" << std::endl;
386 std::cout << " • iostream uses 31% of Flash!" << std::endl;
387 std::cout << " • printf uses 3% of Flash" << std::endl;
388 }
389
390 // =====
391 // 8. PROBLEM: RECURSION (STACK OVERFLOW)
392 // =====
393
394 uint32_t factorial_recursive(uint32_t n) {
395     if (n <= 1) return 1;
396     return n * factorial_recursive(n - 1); // Stack grows
397 }
398
399 uint32_t factorial_iterative(uint32_t n) {
400     uint32_t result = 1;
401     for (uint32_t i = 2; i <= n; ++i) {
402         result *= i;
403     }
404     return result; // Constant stack
405 }
406
407 void demonstrate_recursion_danger() {
408     std::cout << "\n== 8. PROBLEM: RECURSION (STACK OVERFLOW) ==" << std::endl;
409
410     std::cout << "\n STACK CONSTRAINTS:" << std::endl;
411     std::cout << " • Total stack: 4KB (4096 bytes)" << std::endl;
412     std::cout << " • Stack frame: ~32 bytes per function call" << std::endl;
413     std::cout << " • Maximum depth: ~128 calls" << std::endl;
414     std::cout << " • NO PROTECTION: Stack overflow = silent corruption!" << std::endl;
415
416     std::cout << "\n BAD CODE (Recursive factorial):" << std::endl;
417     std::cout << R"(
418     uint32_t factorial(uint32_t n) {
419         if (n <= 1) return 1;
420         return n * factorial(n - 1); // Each call uses stack
421     }
422
423 // factorial(10) = 10 stack frames = 320 bytes OK
```

```
424 // factorial(100) = 100 stack frames = 3200 bytes Close!
425 // factorial(150) = 150 stack frames = 4800 bytes OVERFLOW!
426 )" << std::endl;
427
428 std::cout << "\n WHAT HAPPENS ON OVERFLOW:" << std::endl;
429 std::cout << " 1. Stack grows downward into heap" << std::endl;
430 std::cout << " 2. Corrupts heap data structures" << std::endl;
431 std::cout << " 3. Corrupts global variables" << std::endl;
432 std::cout << " 4. System crashes (hard fault)" << std::endl;
433 std::cout << " 5. NO WARNING! Silent death!" << std::endl;
434
435 std::cout << "\n REAL EXAMPLE: JSON Parser" << std::endl;
436 std::cout << R"(

437 void parse_json(JsonNode* node) {
438     if (node->type == Object) {
439         for (auto& child : node->children) {
440             parse_json(child); // Recursion!
441         }
442     }
443 }

444 // Deeply nested JSON (100 levels) = STACK OVERFLOW!
445 )" << std::endl;
446
447
448 std::cout << "\n SOLUTION: Iterative with explicit stack" << std::endl;
449 std::cout << R"(

450 uint32_t factorial(uint32_t n) {
451     uint32_t result = 1;
452     for (uint32_t i = 2; i <= n; ++i) {
453         result *= i;
454     }
455     return result; // Constant 8 bytes stack
456 }
457

458 )" << std::endl;
459
460 std::cout << "\n STACK USAGE COMPARISON:" << std::endl;
461 std::cout << " Recursive: 32 bytes x depth (unbounded)" << std::endl;
462 std::cout << " Iterative: 8-16 bytes (constant)" << std::endl;
463
464 std::cout << "\nTest with n=10:" << std::endl;
465 std::cout << " factorial_recursive(10) = " << factorial_recursive(10) <<
466     std::endl;
467 std::cout << " factorial_iterative(10) = " << factorial_iterative(10) <<
468     std::endl;
469 }

470 // =====
471 // MAIN FUNCTION
472 // =====
473 int main() {
474     std::cout << "\n
475     =====" <<
476     std::endl;
```

```

474     std::cout << "    WHY AVOID THESE IN EMBEDDED SYSTEMS" << std::endl;
475     std::cout << "
476         =====" <<
477         std::endl;
478
479     std::cout << "\n TARGET SYSTEM SPECIFICATION:" << std::endl;
480     std::cout << "    MCU: ARM Cortex-M4F @ 80MHz" << std::endl;
481     std::cout << "    Flash: 256KB (code + constants)" << std::endl;
482     std::cout << "    RAM: 64KB (data + stack + heap)" << std::endl;
483     std::cout << "        - Stack: 4KB (main task)" << std::endl;
484     std::cout << "        - Heap: 8KB (if enabled)" << std::endl;
485     std::cout << "        - .bss/.data: ~52KB available" << std::endl;
486     std::cout << "    No MMU (no virtual memory)" << std::endl;
487     std::cout << "    No OS (bare metal or lightweight RTOS)" << std::endl;
488     std::cout << "    Real-time: 1ms task deadlines" << std::endl;
489     std::cout << "    Power: Battery powered (sleep modes important)" << std::endl;
490
491     demonstrate_heap_fragmentation();
492     demonstrate_vector_problems();
493     demonstrate_string_problems();
494     demonstrate_virtual_function_overhead();
495     demonstrate_rtti_overhead();
496     demonstrate_exception_overhead();
497     demonstrate_iostream_bloat();
498     demonstrate_recursion_danger();
499
500     std::cout << "\n
501         =====" <<
502         std::endl;
503     std::cout << "    SUMMARY: RESOURCE IMPACT" << std::endl;
504     std::cout << "
505         =====" <<
506         std::endl;
507
508     std::cout << "\n FLASH USAGE (256KB total):" << std::endl;
509     std::cout << "    " << std::endl;
510     std::cout << "        iostream:      80KB (31%)" << std::endl;
511     std::cout << "        Exceptions:   30KB (12%)" << std::endl;
512     std::cout << "        RTTI:          8KB (3%)" << std::endl;
513     std::cout << "        Virtual tables: 5KB (2%)" << std::endl;
514     std::cout << "        TOTAL WASTE:    123KB (48%)" << std::endl;
515     std::cout << "        " << std::endl;
516     std::cout << "        Your code:     133KB (52%)" << std::endl;
517     std::cout << "        " << std::endl;
518
519     std::cout << "\n RAM USAGE (64KB total):" << std::endl;
520     std::cout << "    " << std::endl;
521     std::cout << "        Heap:          8KB (12.5%)" << std::endl;
522     std::cout << "        Stack:         4KB (6.25%)" << std::endl;
523     std::cout << "        iostream:     2KB (3.12%)" << std::endl;
524     std::cout << "        vtable ptrs:  1KB (1.56%)" << std::endl;
525     std::cout << "        " << std::endl;
526     std::cout << "        Your data:    49KB (76.6%)" << std::endl;

```

```
521     std::cout << "                                     " << std::endl;
522
523     std::cout << "\n REAL-TIME IMPACT:" << std::endl;
524     std::cout << "  Task deadline: 1ms = 80,000 CPU cycles" << std::endl;
525     std::cout << "\n  Direct function call: 2 cycles (40,000 calls per ms)"
526           << std::endl;
527     std::cout << "  Virtual call:           20 cycles (4,000 calls per ms)" <<
528           std::endl;
529     std::cout << "  std::vector::push:      50 cycles (1,600 calls per ms)" <<
530           std::endl;
531     std::cout << "  std::vector realloc:   4,000 cycles (20 calls per ms)" <<
532           std::endl;
533     std::cout << "  Exception throw:      40,000 cycles (2 per ms = DEADLINE
534           MISS!)" << std::endl;
535
536     std::cout << "\n BEST PRACTICES FOR EMBEDDED:" << std::endl;
537     std::cout << "\n    USE:" << std::endl;
538     std::cout << "    • std::array (fixed-size, stack)" << std::endl;
539     std::cout << "    • Memory pools (controlled allocation)" << std::endl;
540     std::cout << "    • Static polymorphism (CRTP, templates)" << std::endl;
541     std::cout << "    • Error codes, std::optional (no exceptions)" << std::
542           endl;
543     std::cout << "    • printf/snprintf (smaller than iostream)" << std::endl;
544     std::cout << "    • Iterative algorithms (constant stack)" << std::endl;
545     std::cout << "    • constexpr (compile-time computation)" << std::endl;
546     std::cout << "    • std::variant (type-safe unions)" << std::endl;
547
548     std::cout << "\n    AVOID:" << std::endl;
549     std::cout << "    • new/delete, malloc/free" << std::endl;
550     std::cout << "    • std::vector, std::string, std::map" << std::endl;
551     std::cout << "    • Virtual functions (use sparingly)" << std::endl;
552     std::cout << "    • RTTI (typeid, dynamic_cast)" << std::endl;
553     std::cout << "    • Exceptions (try/catch/throw)" << std::endl;
554     std::cout << "    • iostream (cout, cin, etc.)" << std::endl;
555     std::cout << "    • Recursion (unbounded depth)" << std::endl;
556
557     std::cout << "\n COMPILER FLAGS FOR EMBEDDED:" << std::endl;
558     std::cout << "    -fno-exceptions      (saves 15-40KB Flash)" << std::endl;
559     std::cout << "    -fno-rtti            (saves 5-10KB Flash)" << std::endl;
560     std::cout << "    -fno-threadsafe-statics (saves 2-5KB Flash)" << std::endl;
561     std::cout << "    -ffunction-sections  (enables dead code removal)" << std::
562           endl;
563     std::cout << "    -fdata-sections     (enables dead data removal)" << std::
564           endl;
565     std::cout << "    -Wl,--gc-sections   (linker removes unused code)" << std
566           ::endl;
567     std::cout << "    -Os                 (optimize for size)" << std::endl;
568
569     std::cout << "\n MEMORY SAVINGS SUMMARY:" << std::endl;
570     std::cout << "  Disabling features above: ~130KB Flash savings!" << std::
571           endl;
572     std::cout << "  From 256KB Flash → 126KB available (49% → 98% usable!)" <<
573           std::endl;
```

```
564     std::cout << "\n" =====\\n" <<  
565     std::endl;  
566  
567     return 0;  
}
```

## 23 Source Code: EmbeddedSystemsProgramming.cpp

File: src/EmbeddedSystemsProgramming.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <array>
3 #include <cstdint>
4 #include <cstring>
5 #include <type_traits>
6 #include <utility>
7 #include <optional>
8 #include <cassert>
9
10 // =====
11 // EMBEDDED SYSTEMS PROGRAMMING IN MODERN C++
12 // =====
13 // This file demonstrates embedded C++ patterns and best practices:
14 // - No dynamic allocation (no new/delete)
15 // - Fixed-size buffers
16 // - Compile-time computation
17 // - RAII for hardware resources
18 // - Static polymorphism (no vtables)
19 // - Memory pools
20 // - Interrupt-safe techniques
21 // =====
22
23 // =====
24 // 1. MEMORY POOL (NO HEAP ALLOCATION)
25 // =====
26
27 template<typename T, size_t PoolSize>
28 class MemoryPool {
29 private:
30     alignas(T) uint8_t storage[PoolSize * sizeof(T)];
31     bool used[PoolSize] = {};
32
33 public:
34     template<typename... Args>
35     T* allocate(Args&&... args) {
36         for (size_t i = 0; i < PoolSize; ++i) {
37             if (!used[i]) {
38                 used[i] = true;
39                 T* ptr = reinterpret_cast<T*>(&storage[i * sizeof(T)]);
40                 new (ptr) T(std::forward<Args>(args)...); // Placement new
41                 return ptr;
42             }
43         }
44         return nullptr; // Pool exhausted
45     }
46
47     void deallocate(T* ptr) {
48         if (!ptr) return;
49     }
50 }
```

```
50     size_t index = (reinterpret_cast<uint8_t*>(ptr) - storage) / sizeof(T)
51     ;
52     if (index < PoolSize && used[index]) {
53         ptr->~T();
54         used[index] = false;
55     }
56 }
57
58 size_t available() const {
59     size_t count = 0;
60     for (size_t i = 0; i < PoolSize; ++i) {
61         if (!used[i]) ++count;
62     }
63     return count;
64 }
65
66 size_t capacity() const { return PoolSize; }
67 };
68
69 void example_memory_pool() {
70     std::cout << "\n==== 1. MEMORY POOL (NO HEAP) ====" << std::endl;
71
72     struct Sensor {
73         int id;
74         float value;
75         Sensor(int i, float v) : id(i), value(v) {
76             std::cout << "      Sensor " << id << " created" << std::endl;
77         }
78         ~Sensor() {
79             std::cout << "      Sensor " << id << " destroyed" << std::endl;
80         }
81     };
82
83     MemoryPool<Sensor, 5> pool;
84     std::cout << "  Pool capacity: " << pool.capacity() << std::endl;
85
86     std::cout << "\nAllocating sensors:" << std::endl;
87     Sensor* s1 = pool.allocate(1, 25.5f);
88     Sensor* s2 = pool.allocate(2, 30.2f);
89     Sensor* s3 = pool.allocate(3, 28.7f);
90
91     std::cout << "  Available slots: " << pool.available() << std::endl;
92
93     std::cout << "\nDeallocating sensor 2:" << std::endl;
94     pool.deallocate(s2);
95     std::cout << "  Available slots: " << pool.available() << std::endl;
96
97     std::cout << "\nAllocating new sensor:" << std::endl;
98     Sensor* s4 = pool.allocate(4, 27.1f);
99
100    std::cout << "\nCleanup:" << std::endl;
101    pool.deallocate(s1);
102    pool.deallocate(s3);
103    pool.deallocate(s4);
```

```
103     std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
104     std::cout << " • No heap fragmentation" << std::endl;
105     std::cout << " • Deterministic allocation time" << std::endl;
106     std::cout << " • Fixed memory footprint" << std::endl;
107     std::cout << " • Suitable for real-time systems" << std::endl;
108 }
109
110 // =====
111 // 2. CIRCULAR BUFFER (RING BUFFER)
112 // =====
113
114
115 template<typename T, size_t Size>
116 class CircularBuffer {
117 private:
118     std::array<T, Size> buffer;
119     size_t head = 0;
120     size_t tail = 0;
121     size_t count = 0;
122
123 public:
124     bool push(const T& item) {
125         if (is_full()) return false;
126
127         buffer[head] = item;
128         head = (head + 1) % Size;
129         ++count;
130         return true;
131     }
132
133     bool pop(T& item) {
134         if (is_empty()) return false;
135
136         item = buffer[tail];
137         tail = (tail + 1) % Size;
138         --count;
139         return true;
140     }
141
142     bool is_empty() const { return count == 0; }
143     bool is_full() const { return count == Size; }
144     size_t size() const { return count; }
145     size_t capacity() const { return Size; }
146 };
147
148 void example_circular_buffer() {
149     std::cout << "\n== 2. CIRCULAR BUFFER (INTERRUPT-SAFE) ==" << std::endl;
150
151     CircularBuffer<uint8_t, 8> uart_rx_buffer;
152
153     std::cout << "\nSimulating UART receive (ISR):" << std::endl;
154     for (uint8_t i = 0; i < 5; ++i) {
155         if (uart_rx_buffer.push(0x30 + i)) {
156             std::cout << " [ISR] Received: 0x" << std::hex
```

```
157             << (int)(0x30 + i) << std::dec << std::endl;
158     }
159 }
160
161 std::cout << "\nBuffer status: " << uart_rx_buffer.size()
162     << "/" << uart_rx_buffer.capacity() << std::endl;
163
164 std::cout << "\nMain loop processing:" << std::endl;
165 uint8_t data;
166 while (uart_rx_buffer.pop(data)) {
167     std::cout << " [Main] Processing: 0x" << std::hex
168         << (int)data << std::dec << std::endl;
169 }
170
171 std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
172 std::cout << " • Lock-free for single producer/consumer" << std::endl;
173 std::cout << " • Constant-time operations" << std::endl;
174 std::cout << " • ISR-safe data transfer" << std::endl;
175 std::cout << " • No dynamic allocation" << std::endl;
176 }
177
178 // =====
179 // 3. HARDWARE REGISTER ABSTRACTION (RAII)
180 // =====
181
182 // Simulate hardware registers
183 struct HardwareRegisters {
184     static inline uint32_t GPIO_CONFIG = 0;
185     static inline uint32_t GPIO_OUTPUT = 0;
186     static inline uint32_t ADC_CONTROL = 0;
187 };
188
189 enum class PinMode : uint8_t {
190     Input = 0,
191     Output = 1,
192     Analog = 2
193 };
194
195 class GpioPin {
196 private:
197     uint8_t pin_number;
198     PinMode original_mode;
199
200     static constexpr uint32_t get_pin_mask(uint8_t pin) {
201         return 1u << pin;
202     }
203
204 public:
205     GpioPin(uint8_t pin, PinMode mode)
206         : pin_number(pin)
207         , original_mode(static_cast<PinMode>(
208             (HardwareRegisters::GPIO_CONFIG >> (pin * 2)) & 0x3)) {
209
210         // Configure pin mode
```

```
211     uint32_t mask = 0x3u << (pin * 2);
212     HardwareRegisters::GPIO_CONFIG &= ~mask;
213     HardwareRegisters::GPIO_CONFIG |= (static_cast<uint32_t>(mode) << (pin
214         * 2));
215
215     std::cout << "    GPIO Pin " << (int)pin_number
216         << " configured as " << (mode == PinMode::Output ? "Output"
217             : "Input")
218             << std::endl;
219
220 }
221
222 ~GpioPin() {
223     // Restore original mode (RAII cleanup)
224     uint32_t mask = 0x3u << (pin_number * 2);
225     HardwareRegisters::GPIO_CONFIG &= ~mask;
226     HardwareRegisters::GPIO_CONFIG |= (static_cast<uint32_t>(original_mode
227         ) << (pin_number * 2));
228
229     std::cout << "    GPIO Pin " << (int)pin_number
230         << " restored to original mode" << std::endl;
231 }
232
233 // Prevent copying (hardware resources are unique)
234 GpioPin(const GpioPin&) = delete;
235 GpioPin& operator=(const GpioPin&) = delete;
236
237 // Allow moving
238 GpioPin(GpioPin&& other) noexcept
239     : pin_number(other.pin_number)
240     , original_mode(other.original_mode) {
241     other.pin_number = 0xFF; // Mark as moved
242 }
243
244 void write(bool value) {
245     if (value) {
246         HardwareRegisters::GPIO_OUTPUT |= get_pin_mask(pin_number);
247     } else {
248         HardwareRegisters::GPIO_OUTPUT &= ~get_pin_mask(pin_number);
249     }
250 }
251
252 };
253
254 void example_hardware_raii() {
255     std::cout << "\n==== 3. HARDWARE REGISTER ABSTRACTION (RAII) ===" << std::
256         endl;
257
258     std::cout << "\nConfiguring LED pin:" << std::endl;
259     {
260         GpioPin led(5, PinMode::Output);
```

```
260     std::cout << "  Setting LED ON" << std::endl;
261     led.write(true);
262
263     std::cout << "  LED state: " << (led.read() ? "ON" : "OFF") << std::endl;
264
265     // RAI: Pin automatically restored when going out of scope
266 }
267 std::cout << "  (Pin automatically restored)\n" << std::endl;
268
269 std::cout << "  EMBEDDED BENEFITS:" << std::endl;
270 std::cout << "  •  Automatic resource cleanup" << std::endl;
271 std::cout << "  •  Exception-safe (if exceptions enabled)" << std::endl;
272 std::cout << "  •  Can't forget to restore state" << std::endl;
273 std::cout << "  •  Move semantics for ownership transfer" << std::endl;
274 }
275
276 // =====
277 // 4. STATIC POLYMORPHISM (NO VTABLES)
278 // =====
279
280 // CRTP (Curiously Recurring Template Pattern)
281 template<typename Derived>
282 class SensorBase {
283 public:
284     float read() {
285         return static_cast<Derived*>(this)->read_impl();
286     }
287
288     const char* name() {
289         return static_cast<Derived*>(this)->name_impl();
290     }
291 };
292
293 class TemperatureSensor : public SensorBase<TemperatureSensor> {
294 public:
295     float read_impl() { return 25.5f; }
296     const char* name_impl() { return "Temperature"; }
297 };
298
299 class HumiditySensor : public SensorBase<HumiditySensor> {
300 public:
301     float read_impl() { return 65.2f; }
302     const char* name_impl() { return "Humidity"; }
303 };
304
305 template<typename Sensor>
306 void process_sensor(Sensor& sensor) {
307     std::cout << "  Reading " << sensor.name() << " sensor: "
308             << sensor.read() << std::endl;
309 }
310
311 void example_static_polymorphism() {
```

```

313     std::cout << "\n==== 4. STATIC POLYMORPHISM (NO VTABLES) ===" << std::endl;
314
315     TemperatureSensor temp;
316     HumiditySensor humid;
317
318     std::cout << "\nReading sensors (compile-time dispatch):" << std::endl;
319     process_sensor(temp);
320     process_sensor(humid);
321
322     std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
323     std::cout << " • No vtable overhead (saves RAM)" << std::endl;
324     std::cout << " • No runtime indirection (faster)" << std::endl;
325     std::cout << " • Fully inlineable" << std::endl;
326     std::cout << " • Smaller code size" << std::endl;
327 }
328
329 // =====
330 // 5. COMPILE-TIME COMPUTATION (constexpr)
331 // =====
332
333 constexpr uint32_t calculate_baud_rate_register(uint32_t system_clock,
334                                                 uint32_t baud_rate) {
335     return (system_clock / (16 * baud_rate)) - 1;
336 }
337
338 constexpr uint32_t calculate_pwm_period(uint32_t timer_clock, uint32_t
339                                         frequency) {
340     return timer_clock / frequency;
341 }
342
343 template<size_t N>
344 constexpr uint32_t fibonacci() {
345     if constexpr (N <= 1) {
346         return N;
347     } else {
348         return fibonacci<N-1>() + fibonacci<N-2>();
349     }
350 }
351
352 void example_compile_time() {
353     std::cout << "\n==== 5. COMPILE-TIME COMPUTATION (constexpr) ===" << std::endl;
354
355     // Computed at compile time!
356     constexpr uint32_t BAUD_115200 = calculate_baud_rate_register(16000000,
357                                                               115200);
358     constexpr uint32_t PWM_1KHZ = calculate_pwm_period(1000000, 1000);
359     constexpr uint32_t FIB_10 = fibonacci<10>();
360
361     std::cout << "\nCompile-time computed values:" << std::endl;
362     std::cout << "    UART baud rate register: 0x" << std::hex << BAUD_115200 <<
363             std::dec << std::endl;
364     std::cout << "    PWM period (1kHz): " << PWM_1KHZ << std::endl;
365     std::cout << "    Fibonacci(10): " << FIB_10 << std::endl;

```

```
362     std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
363     std::cout << " • Zero runtime overhead" << std::endl;
364     std::cout << " • Compile-time error checking" << std::endl;
365     std::cout << " • Values in ROM, not computed" << std::endl;
366     std::cout << " • No initialization code needed" << std::endl;
367 }
368
369
370 // =====
371 // 6. FIXED-SIZE STRING (NO STD::STRING)
372 // =====
373
374 template<size_t MaxSize>
375 class FixedString {
376     private:
377         char data[MaxSize + 1] = {}; // +1 for null terminator
378         size_t length = 0;
379
380     public:
381         FixedString() = default;
382
383         FixedString(const char* str) {
384             if (str) {
385                 length = 0;
386                 while (str[length] && length < MaxSize) {
387                     data[length] = str[length];
388                     ++length;
389                 }
390                 data[length] = '\0';
391             }
392         }
393
394         bool append(char c) {
395             if (length < MaxSize) {
396                 data[length++] = c;
397                 data[length] = '\0';
398                 return true;
399             }
400             return false;
401         }
402
403         bool append(const char* str) {
404             if (!str) return false;
405
406             while (*str && length < MaxSize) {
407                 data[length++] = *str++;
408             }
409             data[length] = '\0';
410             return *str == '\0';  // True if entire string was appended
411         }
412
413         void clear() {
414             length = 0;
415             data[0] = '\0';
416         }
417 }
```

```
416     }
417
418     const char* c_str() const { return data; }
419     size_t size() const { return length; }
420     size_t capacity() const { return MaxSize; }
421     bool is_full() const { return length >= MaxSize; }
422 };
423
424 void example_fixed_string() {
425     std::cout << "\n==== 6. FIXED-SIZE STRING (NO HEAP) ===" << std::endl;
426
427     FixedString<32> msg("Sensor: ");
428
429     std::cout << "\nBuilding message:" << std::endl;
430     std::cout << "  Initial: \" " << msg.c_str() << " \" " << std::endl;
431
432     msg.append("Temp=");
433     msg.append("25.5");
434     msg.append("C");
435
436     std::cout << "  Final: \" " << msg.c_str() << " \" " << std::endl;
437     std::cout << "  Size: " << msg.size() << "/" << msg.capacity() << std::endl;
438
439     std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
440     std::cout << "  •  No heap allocation" << std::endl;
441     std::cout << "  •  Fixed memory footprint" << std::endl;
442     std::cout << "  •  Predictable behavior" << std::endl;
443     std::cout << "  •  Stack-based storage" << std::endl;
444 }
445
446 // =====
447 // 7. STATE MACHINE (EMBEDDED PATTERN)
448 // =====
449
450 enum class State {
451     Idle,
452     Reading,
453     Processing,
454     Sending,
455     Error
456 };
457
458 enum class Event {
459     StartRead,
460     DataReady,
461     ProcessComplete,
462     SendComplete,
463     ErrorOccurred,
464     Reset
465 };
466
467 class StateMachine {
468     private:
```

```
469     State current_state = State::Idle;
470
471 public:
472     void process_event(Event event) {
473         std::cout << " [" << state_name(current_state) << "] -> Event: "
474             << event_name(event);
475
476         State new_state = current_state;
477
478         switch (current_state) {
479             case State::Idle:
480                 if (event == Event::StartRead) {
481                     new_state = State::Reading;
482                 }
483                 break;
484
485             case State::Reading:
486                 if (event == Event::DataReady) {
487                     new_state = State::Processing;
488                 } else if (event == Event::ErrorOccurred) {
489                     new_state = State::Error;
490                 }
491                 break;
492
493             case State::Processing:
494                 if (event == Event::ProcessComplete) {
495                     new_state = State::Sending;
496                 } else if (event == Event::ErrorOccurred) {
497                     new_state = State::Error;
498                 }
499                 break;
500
501             case State::Sending:
502                 if (event == Event::SendComplete) {
503                     new_state = State::Idle;
504                 } else if (event == Event::ErrorOccurred) {
505                     new_state = State::Error;
506                 }
507                 break;
508
509             case State::Error:
510                 if (event == Event::Reset) {
511                     new_state = State::Idle;
512                 }
513                 break;
514         }
515
516         if (new_state != current_state) {
517             current_state = new_state;
518             std::cout << " -> [" << state_name(current_state) << "] " << std::
519                 endl;
520         } else {
521             std::cout << " (no transition)" << std::endl;
522         }
523     }
```

```
522     }
523
524     State get_state() const { return current_state; }
525
526 private:
527     static const char* state_name(State s) {
528         switch (s) {
529             case State::Idle: return "Idle";
530             case State::Reading: return "Reading";
531             case State::Processing: return "Processing";
532             case State::Sending: return "Sending";
533             case State::Error: return "Error";
534             default: return "Unknown";
535         }
536     }
537
538     static const char* event_name(Event e) {
539         switch (e) {
540             case Event::StartRead: return "StartRead";
541             case Event::DataReady: return "DataReady";
542             case Event::ProcessComplete: return "ProcessComplete";
543             case Event::SendComplete: return "SendComplete";
544             case Event::ErrorOccurred: return "ErrorOccurred";
545             case Event::Reset: return "Reset";
546             default: return "Unknown";
547         }
548     }
549 };
550
551 void example_state_machine() {
552     std::cout << "\n==== 7. STATE MACHINE (EMBEDDED PATTERN) ===" << std::endl;
553
554     StateMachine sm;
555
556     std::cout << "\nNormal operation sequence:" << std::endl;
557     sm.process_event(Event::StartRead);
558     sm.process_event(Event::DataReady);
559     sm.process_event(Event::ProcessComplete);
560     sm.process_event(Event::SendComplete);
561
562     std::cout << "\nError handling:" << std::endl;
563     sm.process_event(Event::StartRead);
564     sm.process_event(Event::ErrorOccurred);
565     sm.process_event(Event::Reset);
566
567     std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
568     std::cout << " • Clear, predictable behavior" << std::endl;
569     std::cout << " • Easy to validate and test" << std::endl;
570     std::cout << " • Common in embedded protocols" << std::endl;
571     std::cout << " • No dynamic dispatch overhead" << std::endl;
572 }
573
574 // =====
575 // 8. BIT MANIPULATION UTILITIES
```

```
576 // =====
577
578 namespace BitOps {
579     template<typename T>
580     constexpr void set_bit(T& reg, uint8_t bit) {
581         reg |= (static_cast<T>(1) << bit);
582     }
583
584     template<typename T>
585     constexpr void clear_bit(T& reg, uint8_t bit) {
586         reg &= ~(static_cast<T>(1) << bit);
587     }
588
589     template<typename T>
590     constexpr void toggle_bit(T& reg, uint8_t bit) {
591         reg ^= (static_cast<T>(1) << bit);
592     }
593
594     template<typename T>
595     constexpr bool test_bit(T reg, uint8_t bit) {
596         return (reg & (static_cast<T>(1) << bit)) != 0;
597     }
598
599     template<typename T>
600     constexpr void write_bits(T& reg, uint8_t start_bit, uint8_t num_bits, T
601         value) {
602         T mask = ((static_cast<T>(1) << num_bits) - 1) << start_bit;
603         reg = (reg & ~mask) | ((value << start_bit) & mask);
604     }
605
606 void example_bit_manipulation() {
607     std::cout << "\n==== 8. BIT MANIPULATION UTILITIES ===" << std::endl;
608
609     uint32_t control_reg = 0;
610
611     std::cout << "\nManipulating control register:" << std::endl;
612     std::cout << "    Initial value: 0x" << std::hex << control_reg << std::dec
613         << std::endl;
614
615     BitOps::set_bit(control_reg, 0); // Enable bit 0
616     std::cout << "    After set bit 0: 0x" << std::hex << control_reg << std::dec
617         << std::endl;
618
619     BitOps::set_bit(control_reg, 5); // Enable bit 5
620     std::cout << "    After set bit 5: 0x" << std::hex << control_reg << std::dec
621         << std::endl;
622
623     BitOps::write_bits(control_reg, 2, 3, 0b101u); // Write 3 bits starting
624         at bit 2
625     std::cout << "    After write bits 2-4: 0x" << std::hex << control_reg <<
626         std::dec << std::endl;
627
628     std::cout << "\nTesting bits:" << std::endl;
```

```
624     std::cout << "  Bit 0 is " << (BitOps::test_bit(control_reg, 0) ? "SET" :
625         "CLEAR") << std::endl;
626     std::cout << "  Bit 1 is " << (BitOps::test_bit(control_reg, 1) ? "SET" :
627         "CLEAR") << std::endl;
628
629     std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
630     std::cout << " • Type-safe bit operations" << std::endl;
631     std::cout << " • Constexpr for compile-time use" << std::endl;
632     std::cout << " • Clear, readable code" << std::endl;
633     std::cout << " • No magic numbers" << std::endl;
634 }
635
636 // =====
637 // MAIN FUNCTION
638 // =====
639
640 int main() {
641     std::cout << "\n
642         =====" <<
643         std::endl;
644     std::cout << "  EMBEDDED SYSTEMS PROGRAMMING IN MODERN C++" << std::endl;
645     std::cout << "
646         =====" <<
647         std::endl;
648
649     example_memory_pool();
650     example_circular_buffer();
651     example_hardware_raii();
652     example_static_polymorphism();
653     example_compile_time();
654     example_fixed_string();
655     example_state_machine();
656     example_bit_manipulation();
657
658     std::cout << "\n
659         =====" <<
660         std::endl;
661     std::cout << "  EMBEDDED C++ BEST PRACTICES SUMMARY" << std::endl;
662     std::cout << "
663         =====" <<
664         std::endl;
665
666     std::cout << "\n KEY PRINCIPLES FOR EMBEDDED C++:" << std::endl;
667
668     std::cout << "\n1. NO DYNAMIC ALLOCATION" << std::endl;
669     std::cout << " • Use memory pools instead of new/delete" << std::endl;
670     std::cout << " • Fixed-size containers (std::array)" << std::endl;
671     std::cout << " • Stack allocation preferred" << std::endl;
672     std::cout << " • Predictable memory usage" << std::endl;
673
674     std::cout << "\n2. COMPILE-TIME COMPUTATION" << std::endl;
675     std::cout << " • constexpr functions for configuration" << std::endl;
676     std::cout << " • Template metaprogramming" << std::endl;
677     std::cout << " • if constexpr for conditional compilation" << std::endl;
```

```
668     std::cout << " • Zero runtime overhead" << std::endl;
669
670     std::cout << "\n3. STATIC POLYMORPHISM" << std::endl;
671     std::cout << " • CRTP instead of virtual functions" << std::endl;
672     std::cout << " • No vtable overhead" << std::endl;
673     std::cout << " • Fully inlineable" << std::endl;
674     std::cout << " • Saves RAM and Flash" << std::endl;
675
676     std::cout << "\n4. RAI FOR HARDWARE" << std::endl;
677     std::cout << " • Automatic resource management" << std::endl;
678     std::cout << " • Scope-based cleanup" << std::endl;
679     std::cout << " • Move semantics for ownership" << std::endl;
680     std::cout << " • Exception-safe (if enabled)" << std::endl;
681
682     std::cout << "\n5. INTERRUPT-SAFE PATTERNS" << std::endl;
683     std::cout << " • Circular buffers for ISR->Main communication" << std::endl;
684     std::cout << " • Atomic operations" << std::endl;
685     std::cout << " • Lock-free data structures" << std::endl;
686     std::cout << " • Minimal ISR processing" << std::endl;
687
688     std::cout << "\n6. DETERMINISTIC BEHAVIOR" << std::endl;
689     std::cout << " • No recursion (or limited)" << std::endl;
690     std::cout << " • Bounded loops" << std::endl;
691     std::cout << " • Fixed execution time" << std::endl;
692     std::cout << " • Real-time constraints" << std::endl;
693
694     std::cout << "\n MODERN C++ FEATURES FOR EMBEDDED:" << std::endl;
695     std::cout << "     constexpr (C++11/14/17) - Compile-time computation" << std::endl;
696     std::cout << "     std::array (C++11) - Fixed-size arrays" << std::endl;
697     std::cout << "     std::optional (C++17) - Error handling without exceptions" << std::endl;
698     std::cout << "     if constexpr (C++17) - Conditional compilation" << std::endl;
699     std::cout << "     Templates - Static polymorphism" << std::endl;
700     std::cout << "     Move semantics (C++11) - Efficient ownership transfer" << std::endl;
701     std::cout << "     RAI - Automatic resource management" << std::endl;
702     std::cout << "     Type traits - Compile-time type checking" << std::endl;
703
704     std::cout << "\n AVOID IN EMBEDDED:" << std::endl;
705     std::cout << "     new/delete (use memory pools)" << std::endl;
706     std::cout << "     std::vector, std::string (dynamic allocation)" << std::endl;
707     std::cout << "     Virtual functions (unless justified)" << std::endl;
708     std::cout << "     RTTI (enable only if needed)" << std::endl;
709     std::cout << "     Exceptions (often disabled in embedded)" << std::endl;
710     std::cout << "     iostream (large code size)" << std::endl;
711     std::cout << "     Recursion (stack overflow risk)" << std::endl;
712
713     std::cout << "\n EMBEDDED C++ IDIOMS:" << std::endl;
714     std::cout << " • Singleton pattern for hardware peripherals" << std::endl;
```

```
715     std::cout << " • State machines for protocol handling" << std::endl;
716     std::cout << " • Circular buffers for data streaming" << std::endl;
717     std::cout << " • Memory pools for dynamic-like allocation" << std::endl;
718     std::cout << " • CRTP for zero-cost abstraction" << std::endl;
719     std::cout << " • Register classes with bit fields" << std::endl;
720
721     std::cout << "\n TYPICAL MEMORY CONSTRAINTS:" << std::endl;
722     std::cout << " • Flash: 32KB - 2MB (code storage)" << std::endl;
723     std::cout << " • RAM: 4KB - 256KB (runtime data)" << std::endl;
724     std::cout << " • Stack: 1KB - 8KB (limited depth)" << std::endl;
725     std::cout << " • No heap or tiny heap (fragmentation risk)" << std::endl
726     ;
727
728     std::cout << "\n REAL-TIME CONSIDERATIONS:" << std::endl;
729     std::cout << " • Interrupt latency: <10µs typical" << std::endl;
730     std::cout << " • Task scheduling: RTOS or bare-metal" << std::endl;
731     std::cout << " • Timing critical sections" << std::endl;
732     std::cout << " • Watchdog timer management" << std::endl;
733
734     std::cout << "\n RECOMMENDED READING:" << std::endl;
735     std::cout << " • \"Embedded C++ Coding Standard\" by BARR Group" << std::endl;
736     std::cout << " • \"Real-Time C++\" by Christopher Kormanyos" << std::endl;
737     std::cout << " • \"Effective Modern C++\" (embedded-applicable patterns)" << std::endl;
738
739     std::cout << "\n"
740     ======\n" << std::endl;
741
742     return 0;
743 }
```

## 24 Source Code: ErrorHandling.cpp

File: src/ErrorHandling.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <stdexcept>
3 #include <string>
4 #include <memory>
5 #include <vector>
6 #include <optional>
7 #include <variant>
8 #include <fstream>
9 #include <system_error>
10 #include <cassert>
11 #include <type_traits>
12
13 // =====
14 // C++ ERROR HANDLING: COMPREHENSIVE EXAMPLES
15 // =====
16 // Demonstrates:
17 // 1. Throw by value, catch by reference (BEST PRACTICE)
18 // 2. Standard exception hierarchy
19 // 3. Custom exceptions
20 // 4. Exception safety guarantees
21 // 5. Compile-time error detection
22 // 6. Alternative error handling patterns
23 // =====
24
25 // =====
26 // 1. STANDARD EXCEPTIONS - THROW BY VALUE, CATCH BY REFERENCE
27 // =====
28
29 void demonstrate_basic_exception_handling() {
30     std::cout << "\n==== 1. BASIC EXCEPTION HANDLING ===" << std::endl;
31     std::cout << "Rule: THROW BY VALUE, CATCH BY REFERENCE" << std::endl;
32
33     // Example 1: std::runtime_error
34     try {
35         std::cout << "\nThrowing std::runtime_error..." << std::endl;
36         throw std::runtime_error("Something went wrong at runtime!"); // Throw by value
37     }
38     catch (const std::runtime_error& e) { // Catch by const reference
39         std::cout << "Caught runtime_error: " << e.what() << std::endl;
40     }
41
42     // Example 2: std::logic_error
43     try {
44         std::cout << "\nThrowing std::logic_error..." << std::endl;
45         throw std::logic_error("Logic error in the program!"); // Throw by value
46     }
47     catch (const std::logic_error& e) { // Catch by const reference
```

```
48     std::cout << "Caught logic_error: " << e.what() << std::endl;
49 }
50
51 // Example 3: std::invalid_argument
52 try {
53     std::cout << "\nThrowing std::invalid_argument..." << std::endl;
54     throw std::invalid_argument("Invalid argument provided!"); // Throw
55     by value
56 }
57 catch (const std::invalid_argument& e) { // Catch by const reference
58     std::cout << "Caught invalid_argument: " << e.what() << std::endl;
59 }
60
61 // Example 4: std::out_of_range
62 try {
63     std::cout << "\nThrowing std::out_of_range..." << std::endl;
64     throw std::out_of_range("Index out of range!"); // Throw by value
65 }
66 catch (const std::out_of_range& e) { // Catch by const reference
67     std::cout << "Caught out_of_range: " << e.what() << std::endl;
68 }
69
70 // =====
71 // 2. STANDARD EXCEPTION HIERARCHY
72 // =====
73
74 void demonstrate_exception_hierarchy() {
75     std::cout << "\n== 2. EXCEPTION HIERARCHY ==" << std::endl;
76     std::cout << "std::exception (base)" << std::endl;
77     std::cout << "    std::logic_error" << std::endl;
78     std::cout << "        std::invalid_argument" << std::endl;
79     std::cout << "        std::domain_error" << std::endl;
80     std::cout << "        std::length_error" << std::endl;
81     std::cout << "        std::out_of_range" << std::endl;
82     std::cout << "        std::runtime_error" << std::endl;
83     std::cout << "        std::range_error" << std::endl;
84     std::cout << "        std::overflow_error" << std::endl;
85     std::cout << "        std::underflow_error" << std::endl;
86
87     // Catch base class to handle all derived exceptions
88     try {
89         std::cout << "\nThrowing derived exception..." << std::endl;
90         throw std::overflow_error("Integer overflow occurred!");
91     }
92     catch (const std::runtime_error& e) { // Catches std::overflow_error (
93         derived)
94         std::cout << "Caught via base class runtime_error: " << e.what() <<
95         std::endl;
96     }
97
98     // Multiple catch blocks - order matters!
99     try {
100         std::cout << "\nDemonstrating catch order..." << std::endl;
```

```
99     throw std::invalid_argument("Bad argument");
100 }
101 catch (const std::invalid_argument& e) { // Specific exception first
102     std::cout << "Caught specific: invalid_argument: " << e.what() << std
103     ::endl;
104 }
105 catch (const std::logic_error& e) { // Base class second
106     std::cout << "Caught base: logic_error: " << e.what() << std::endl;
107 }
108 catch (const std::exception& e) { // Most general last
109     std::cout << "Caught most general: exception: " << e.what() << std::
110     endl;
111 }
112 // =====
113 // 3. CUSTOM EXCEPTIONS
114 // =====
115
116 // Custom exception class - inherit from std::exception
117 class CameraException : public std::runtime_error {
118 private:
119     int error_code;
120
121 public:
122     explicit CameraException(const std::string& message, int code = 0)
123         : std::runtime_error(message), error_code(code) {}
124
125     int get_error_code() const noexcept { return error_code; }
126 };
127
128 class CameraNotConnectedException : public CameraException {
129 public:
130     explicit CameraNotConnectedException(const std::string& camera_name)
131         : CameraException("Camera not connected: " + camera_name, 100) {}
132 };
133
134 class CameraConfigurationException : public CameraException {
135 public:
136     explicit CameraConfigurationException(const std::string& config_error)
137         : CameraException("Configuration error: " + config_error, 200) {}
138 };
139
140 void demonstrate_custom_exceptions() {
141     std::cout << "\n== 3. CUSTOM EXCEPTIONS ==" << std::endl;
142
143     // Throw custom exception by value
144     try {
145         std::cout << "\nThrowing CameraNotConnectedException..." << std::endl;
146         throw CameraNotConnectedException("USB-CAM-001"); // Throw by value
147     }
148     catch (const CameraNotConnectedException& e) { // Catch by const
149         std::cout << "Caught: " << e.what() << std::endl;
```

```
150     std::cout << "Error code: " << e.get_error_code() << std::endl;
151 }
152
153 // Catch hierarchy with custom exceptions
154 try {
155     std::cout << "\nThrowing CameraConfigurationException..." << std::endl
156     ;
157     throw CameraConfigurationException("Invalid resolution: 9999x9999");
158 }
159 catch (const CameraException& e) { // Catch base custom exception
160     std::cout << "Caught via base CameraException: " << e.what() << std::
161     endl;
162     std::cout << "Error code: " << e.get_error_code() << std::endl;
163 }
164 catch (const std::exception& e) { // Catch any std::exception
165     std::cout << "Caught via std::exception: " << e.what() << std::endl;
166 }
167
168 // =====
169 // 4. EXCEPTION SAFETY GUARANTEES
170 // =====
171
172 class SafeVector {
173 private:
174     std::vector<int> data;
175
176 public:
177     // Basic guarantee: No resources leaked, but object state may change
178     void push_back_basic(int value) {
179         data.push_back(value); // May throw, but vector cleans up
180     }
181
182     // Strong guarantee: Operation succeeds or has no effect (rollback)
183     void push_back_strong(int value) {
184         std::vector<int> temp = data; // Copy current state
185         temp.push_back(value); // Modify copy (if this throws,
186         // original unchanged)
187         data = std::move(temp); // Commit (noexcept move)
188     }
189
190     // No-throw guarantee: Marked noexcept
191     size_t size() const noexcept {
192         return data.size();
193     }
194
195     // No-throw guarantee: Never throws
196     void clear() noexcept {
197         data.clear();
198     }
199
200     int& at(size_t index) {
201         return data.at(index); // May throw std::out_of_range
202     }
```

```
201 };
```

```
202
203 void demonstrate_exception_safety() {
204     std::cout << "\n==== 4. EXCEPTION SAFETY GUARANTEES ===" << std::endl;
205
206     std::cout << "\nException Safety Levels:" << std::endl;
207     std::cout << "1. No-throw guarantee (noexcept): Never throws" << std::endl;
208     ;
209     std::cout << "2. Strong guarantee: Rollback on exception" << std::endl;
210     std::cout << "3. Basic guarantee: No resource leaks" << std::endl;
211     std::cout << "4. No guarantee: May leak resources" << std::endl;
212
213     SafeVector vec;
214     vec.push_back_basic(1);
215     vec.push_back_basic(2);
216
217     try {
218         std::cout << "\nAccessing valid index: vec.at(0)" << std::endl;
219         std::cout << "Value: " << vec.at(0) << std::endl;
220
221         std::cout << "\nAccessing invalid index: vec.at(100)" << std::endl;
222         vec.at(100); // Throws std::out_of_range
223     }
224     catch (const std::out_of_range& e) {
225         std::cout << "Caught: " << e.what() << std::endl;
226         std::cout << "Vector is still valid, size: " << vec.size() << std::endl;
227     }
228 }
229
230 // =====
231 // 5. RAI AND EXCEPTION SAFETY
232 // =====
233
234 class FileHandle {
235     private:
236         std::string filename;
237         FILE* file;
238
239     public:
240         explicit FileHandle(const std::string& fname, const char* mode)
241             : filename(fname), file(nullptr) {
242             file = fopen(fname.c_str(), mode);
243             if (!file) {
244                 throw std::runtime_error("Failed to open file: " + fname);
245             }
246             std::cout << "File opened: " << filename << std::endl;
247         }
248
249         ~FileHandle() {
250             if (file) {
251                 fclose(file);
252                 std::cout << "File closed: " << filename << std::endl;
253             }
254 }
```

```
253     }
254
255     // Delete copy operations
256     FileHandle(const FileHandle&) = delete;
257     FileHandle& operator=(const FileHandle&) = delete;
258
259     // Move operations
260     FileHandle(FileHandle&& other) noexcept
261         : filename(std::move(other.filename)), file(other.file) {
262         other.file = nullptr;
263     }
264
265     FILE* get() const noexcept { return file; }
266 };
267
268 void demonstrate_raii() {
269     std::cout << "\n==== 5. RAI AND EXCEPTION SAFETY ===" << std::endl;
270
271     try {
272         std::cout << "\nOpening file (will throw)..." << std::endl;
273         FileHandle handle("nonexistent_file_xyz.txt", "r");
274         std::cout << "This won't print" << std::endl;
275     }
276     catch (const std::runtime_error& e) {
277         std::cout << "Caught: " << e.what() << std::endl;
278         std::cout << "No resource leak - file was never opened!" << std::endl;
279     }
280
281     std::cout << "\nDemonstrating automatic cleanup with scope:" << std::endl;
282     try {
283         // Create temporary file for demo
284         {
285             FileHandle handle1("temp_test.txt", "w");
286             fprintf(handle1.get(), "Test data\n");
287             // File automatically closed when handle1 goes out of scope
288         }
289         std::cout << "File closed automatically (RAII)" << std::endl;
290
291         // Clean up temp file
292         std::remove("temp_test.txt");
293     }
294     catch (const std::exception& e) {
295         std::cout << "Exception: " << e.what() << std::endl;
296     }
297 }
298
299 // =====
300 // 6. COMPILE-TIME ERROR DETECTION
301 // =====
302
303 // Static assertions - compile-time checks
304 template<typename T>
305 class Buffer {
306     private:
```

```
307     std::vector<T> data;
308
309 public:
310     // Compile-time check: Only allow trivially copyable types
311     static_assert(std::is_trivially_copyable_v<T>,
312                   "Buffer only works with trivially copyable types!");
313
314     Buffer(size_t size) : data(size) {}
315
316     T* get_data() { return data.data(); }
317 };
318
319 // constexpr for compile-time evaluation
320 constexpr int divide_compile_time(int a, int b) {
321     // This will cause compile error if b is 0 at compile time
322     return (b == 0) ? throw std::invalid_argument("Division by zero!") : a / b
323     ;
324 }
325
326 // C++20 Concepts for compile-time type checking
327 template<typename T>
328 concept Numeric = std::is_arithmetic_v<T>;
329
330 template<Numeric T>
331 T safe_divide(T a, T b) {
332     if (b == T(0)) {
333         throw std::invalid_argument("Division by zero!");
334     }
335     return a / b;
336 }
337
338 void demonstrate_compile_time_checks() {
339     std::cout << "\n==== 6. COMPILE-TIME ERROR DETECTION ===" << std::endl;
340
341     std::cout << "\n1. Static Assertions:" << std::endl;
342     Buffer<int> int_buffer(10); // OK: int is trivially copyable
343     std::cout << "    Buffer<int> compiles (trivially copyable)" << std::endl
344         ;
345
346     // This would NOT compile:
347     // Buffer<std::string> string_buffer(10); // ERROR: string not trivially
348     // copyable
349     std::cout << "    Buffer<std::string> would fail to compile!" << std::endl;
350
351     std::cout << "\n2. constexpr Functions:" << std::endl;
352     constexpr int result = divide_compile_time(10, 2); // OK at compile time
353     std::cout << "    10 / 2 = " << result << " (evaluated at compile time)" <<
354         std::endl;
355
356     // This would NOT compile:
357     // constexpr int error = divide_compile_time(10, 0); // Compile error!
358     std::cout << "    10 / 0 would fail at compile time!" << std::endl;
359 }
```

```
356     std::cout << "\n3. Concepts (C++20):" << std::endl;
357     try {
358         int a = safe_divide(10, 2);
359         std::cout << "    10 / 2 = " << a << std::endl;
360
361         double b = safe_divide(10.0, 0.0); // Runtime error
362         std::cout << "    This won't print: " << b << std::endl;
363     }
364     catch (const std::invalid_argument& e) {
365         std::cout << "    Caught: " << e.what() << std::endl;
366     }
367
368     // This would NOT compile:
369     // safe_divide(std::string("10"), std::string("2")); // Concept
370     // constraint violated!
371     std::cout << "    safe_divide with std::string would fail to compile!" <<
372     std::endl;
373 }
374
375 // =====
376 // 7. NOEXCEPT SPECIFICATION
377 // =====
378
379 class NoexceptDemo {
380 public:
381     // Guaranteed not to throw
382     int get_value() const noexcept {
383         return 42;
384     }
385
386     // May throw
387     int divide(int a, int b) {
388         if (b == 0) {
389             throw std::invalid_argument("Division by zero");
390         }
391         return a / b;
392     }
393
394     // Conditionally noexcept based on template parameter
395     template<typename T>
396     void swap(T& a, T& b) noexcept(std::is_nothrow_move_constructible_v<T>) {
397         T temp = std::move(a);
398         a = std::move(b);
399         b = std::move(temp);
400     }
401 };
402
403 void demonstrate_noexcept() {
404     std::cout << "\n==== 7. NOEXCEPT SPECIFICATION ===" << std::endl;
405
406     NoexceptDemo demo;
407
408     std::cout << "\nnoexcept functions:" << std::endl;
409     std::cout << "    get_value() is noexcept: "
```

```
408         << noexcept(demo.get_value()) << std::endl;
409     std::cout << "    divide() is noexcept: "
410             << noexcept(demo.divide(10, 2)) << std::endl;
411
412     std::cout << "\nBenefits of noexcept:" << std::endl;
413     std::cout << "    • Compiler optimizations" << std::endl;
414     std::cout << "    • Move constructors in std::vector" << std::endl;
415     std::cout << "    • Enables certain optimizations in algorithms" << std::
416             endl;
417     std::cout << "    • Self-documenting code" << std::endl;
418
419     std::cout << "\n  If noexcept function throws: std::terminate() is called
420             !" << std::endl;
421 }
422
423 // =====
424 // 8. ALTERNATIVE ERROR HANDLING: std::optional
425 // =====
426
427 std::optional<int> safe_divide_optional(int a, int b) {
428     if (b == 0) {
429         return std::nullopt; // Return empty optional instead of throwing
430     }
431     return a / b;
432 }
433
434 void demonstrate_optional() {
435     std::cout << "\n== 8. ERROR HANDLING WITH std::optional ==" << std::endl
436         ;
437     std::cout << "Alternative to exceptions for expected failures" << std::
438             endl;
439
440     auto result1 = safe_divide_optional(10, 2);
441     if (result1) {
442         std::cout << "\n10 / 2 = " << *result1 << " " << std::endl;
443     } else {
444         std::cout << "\n10 / 2 failed " << std::endl;
445     }
446
447     auto result2 = safe_divide_optional(10, 0);
448     if (result2) {
449         std::cout << "10 / 0 = " << *result2 << " " << std::endl;
450     } else {
451         std::cout << "10 / 0 failed (expected) " << std::endl;
452     }
453
454     std::cout << "\nBenefits:" << std::endl;
455     std::cout << "    • No exception overhead" << std::endl;
456     std::cout << "    • Makes errors explicit in return type" << std::endl;
457     std::cout << "    • Good for expected failures" << std::endl;
458 }
459
460 // =====
461 // 9. ALTERNATIVE ERROR HANDLING: std::variant
462
```

```
458 // =====
459
460 struct Error {
461     int code;
462     std::string message;
463 };
464
465 std::variant<int, Error> safe_divide_variant(int a, int b) {
466     if (b == 0) {
467         return Error{1, "Division by zero"};
468     }
469     return a / b;
470 }
471
472 void demonstrate_variant() {
473     std::cout << "\n== 9. ERROR HANDLING WITH std::variant ==" << std::endl;
474     std::cout << "Return either result or error" << std::endl;
475
476     auto result1 = safe_divide_variant(10, 2);
477     if (std::holds_alternative<int>(result1)) {
478         std::cout << "\n10 / 2 = " << std::get<int>(result1) << " " << std::endl;
479     } else {
480         const auto& err = std::get<Error>(result1);
481         std::cout << "Error " << err.code << ":" << err.message << " " << std::endl;
482     }
483
484     auto result2 = safe_divide_variant(10, 0);
485     if (std::holds_alternative<int>(result2)) {
486         std::cout << "10 / 0 = " << std::get<int>(result2) << " " << std::endl;
487     } else {
488         const auto& err = std::get<Error>(result2);
489         std::cout << "Error " << err.code << ":" << err.message << " " << std::endl;
490     }
491
492     std::cout << "\nBenefits:" << std::endl;
493     std::cout << " • Can return detailed error information" << std::endl;
494     std::cout << " • Type-safe error handling" << std::endl;
495     std::cout << " • No exception overhead" << std::endl;
496 }
497
498 // =====
499 // 10. RETHROWING AND NESTED EXCEPTIONS
500 // =====
501
502 void inner_function() {
503     throw std::runtime_error("Error in inner function");
504 }
505
506 void middle_function() {
507     try {
```

```
508     inner_function();
509 }
510 catch (...) {
511     std::cout << "middle_function: Caught exception, rethrowing..." << std::endl;
512     throw; // Rethrow the same exception
513 }
514 }
515
516 void demonstrate_rethrowing() {
517     std::cout << "\n== 10. RETHROWING EXCEPTIONS ==" << std::endl;
518
519     try {
520         middle_function();
521     }
522     catch (const std::runtime_error& e) {
523         std::cout << "Caught in demonstrate_rethrowing: " << e.what() << std::endl;
524     }
525
526     std::cout << "\nRethrow with throw; (not throw e;)" << std::endl;
527     std::cout << "    throw; - Rethrows original exception (correct)" << std::endl;
528     std::cout << "    throw e; - Creates new exception (slicing!)" << std::endl;
529 }
530
531 // =====
532 // MAIN FUNCTION
533 // =====
534
535 int main() {
536     std::cout << "\n"
537         =====
538         std::endl;
539     std::cout << "  C++ ERROR HANDLING: COMPREHENSIVE GUIDE" << std::endl;
540     std::cout << "
541         =====
542         std::endl;
543
544     demonstrate_basic_exception_handling();
545     demonstrate_exception_hierarchy();
546     demonstrate_custom_exceptions();
547     demonstrate_exception_safety();
548     demonstrate_raii();
549     demonstrate_compile_time_checks();
550     demonstrate_noexcept();
551     demonstrate_optional();
552     demonstrate_variant();
553     demonstrate_rethrowing();
554
555     std::cout << "\n"
556         =====
557         std::endl;
558     std::cout << "  BEST PRACTICES SUMMARY" << std::endl;
```

```
553     std::cout << "  
554         ======" <<  
555         std::endl;  
556  
557     std::cout << "\n EXCEPTION HANDLING RULES:" << std::endl;  
558     std::cout << "n1. THROW BY VALUE, CATCH BY REFERENCE" << std::endl;  
559     std::cout << "    throw std::runtime_error(\"message\");" << std::endl;  
560     std::cout << "    catch (const std::runtime_error& e)" << std::endl;  
561     std::cout << "    throw new std::runtime_error(\"message\"); // NO!" <<  
562         std::endl;  
563     std::cout << "    catch (std::runtime_error e) // Slicing!" << std::endl;  
564         ;  
565  
566     std::cout << "\n2. CATCH ORDER MATTERS" << std::endl;  
567     std::cout << "    • Catch specific exceptions first" << std::endl;  
568     std::cout << "    • Catch base classes last" << std::endl;  
569     std::cout << "    • catch(...) for unknown exceptions" << std::endl;  
570  
571     std::cout << "\n3. USE STANDARD EXCEPTIONS" << std::endl;  
572     std::cout << "    • std::runtime_error - Runtime failures" << std::endl;  
573     std::cout << "    • std::logic_error - Programming errors" << std::endl;  
574     std::cout << "    • std::invalid_argument - Bad function arguments" << std  
575         ::endl;  
576     std::cout << "    • std::out_of_range - Index out of bounds" << std::endl;  
577  
578     std::cout << "\n4. RAI FOR EXCEPTION SAFETY" << std::endl;  
579     std::cout << "    • Use smart pointers (std::unique_ptr, std::shared_ptr)"  
580         << std::endl;  
581     std::cout << "    • RAI wrappers for resources" << std::endl;  
582     std::cout << "    • Destructors never throw" << std::endl;  
583  
584     std::cout << "\n5. NOEXCEPT FOR NON-THROWING FUNCTIONS" << std::endl;  
585     std::cout << "    • Mark functions that never throw" << std::endl;  
586     std::cout << "    • Enables compiler optimizations" << std::endl;  
587     std::cout << "    • Move constructors should be noexcept" << std::endl;  
588  
589     std::cout << "\n6. COMPILE-TIME ERROR DETECTION" << std::endl;  
590     std::cout << "    • static_assert for compile-time checks" << std::endl;  
591     std::cout << "    • constexpr for compile-time evaluation" << std::endl;  
592     std::cout << "    • Concepts (C++20) for type constraints" << std::endl;  
593     std::cout << "    • Template SFINAE for type checking" << std::endl;  
594  
595     std::cout << "\n7. ALTERNATIVES TO EXCEPTIONS" << std::endl;  
596     std::cout << "    • std::optional<T> - For expected failures" << std::endl;  
597     std::cout << "    • std::variant<T, Error> - Return result or error" << std  
598         ::endl;  
599     std::cout << "    • Error codes - For performance-critical code" << std::  
600         endl;  
601     std::cout << "    • std::expected<T, E> (C++23) - Best of both worlds" <<  
602         std::endl;  
603  
604     std::cout << "\n WHEN TO USE WHAT:" << std::endl;  
605     std::cout << "\nExceptions:" << std::endl;  
606     std::cout << "    Unexpected errors (file not found, network failure)" <<
```

```
      std::endl;
598  std::cout << "    Constructor failures" << std::endl;
599  std::cout << "    Deep call stacks (error propagation)" << std::endl;
600  std::cout << "    Performance-critical code" << std::endl;
601  std::cout << "    Expected failures (validation)" << std::endl;
602
603  std::cout << "\nstd::optional:" << std::endl;
604  std::cout << "    Expected failures (search not found)" << std::endl;
605  std::cout << "    Optional return values" << std::endl;
606  std::cout << "    Performance-critical code" << std::endl;
607  std::cout << "    Need detailed error information" << std::endl;
608
609  std::cout << "\nstd::variant<T, Error>:" << std::endl;
610  std::cout << "    Need detailed error information" << std::endl;
611  std::cout << "    Multiple error types" << std::endl;
612  std::cout << "    Performance-critical code" << std::endl;
613  std::cout << "    Simple success/failure cases" << std::endl;
614
615  std::cout << "\nCompile-time checks:" << std::endl;
616  std::cout << "    Type constraints" << std::endl;
617  std::cout << "    API misuse prevention" << std::endl;
618  std::cout << "    Zero runtime cost" << std::endl;
619  std::cout << "    Early error detection" << std::endl;
620
621  std::cout << "\n PERFORMANCE NOTES:" << std::endl;
622  std::cout << "    • Exceptions have zero cost if not thrown (modern
623    compilers)" << std::endl;
624  std::cout << "    • Throwing exception is expensive (~1000x slower than
625    return)" << std::endl;
626  std::cout << "    • noexcept enables optimizations (especially in std::
627    vector)" << std::endl;
628  std::cout << "    • Compile-time checks have zero runtime cost" << std::endl
629  ;
630
631  std::cout << "\n
632  ======\n" << std::endl;
633
634  return 0;
635 }
```

## 25 Source Code: ErrorHandlingStroustrup.cpp

File: src/ErrorHandlingStroustrup.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // ERROR HANDLING STRATEGIES - Following Bjarne Stroustrup's Guidance
3 // Based on "A Tour of C++", 3rd Edition, Chapter 4
4 // =====
5 //
6 // Three approaches to error handling:
7 // 1. RETURN VALUES - When function can't complete the task
8 // 2. EXCEPTIONS - When constructor can't establish invariants
9 // 3. TERMINATION - When continuing would be dangerous
10 //
11 // Key Principles:
12 // - Constructors cannot return values → must throw or terminate
13 // - Partially constructed objects are dangerous
14 // - RAII ensures proper cleanup even with exceptions
15 // - Return values for expected/recoverable errors
16 // - Exceptions for unexpected/exceptional situations
17 // - Terminate for programming errors/invariant violations
18 // =====
19
20 #include <iostream>
21 #include <string>
22 #include <vector>
23 #include <optional>
24 #include <memory>
25 #include <fstream>
26 #include <cstdlib>
27 #include <stdexcept>
28 #include <cassert>
29
30 // =====
31 // 1. RETURN VALUES - For Expected/Recoverable Errors
32 // =====
33
34 // Use case: Function can't complete task, but caller can handle it
35 // Appropriate when: Error is expected and can be recovered
36
37 class ConfigParser {
38 public:
39     // Return std::optional - error is expected (file might not exist)
40     static std::optional<std::string> read_config_value(const std::string& key
41     ) {
42         std::cout << "[1.1] Reading config key: " << key << std::endl;
43
44         // Simulate config lookup
45         if (key == "server_url") {
46             return "http://localhost:8080";
47         }
48         if (key == "timeout") {
49             return "30";
50     }
51 }
```

```
49     }
50
51     // Key not found - this is EXPECTED, not exceptional
52     std::cout << "      Config key '" << key << "' not found (expected
53     scenario)" << std::endl;
54     return std::nullopt; // Caller can provide default or retry
55 }
56
57 // Return int with special value - traditional C-style
58 static int parse_integer(const std::string& str, int default_value = -1) {
59     std::cout << "[1.2] Parsing integer from: " << str << std::endl;
60
61     try {
62         return std::stoi(str);
63     } catch (...) {
64         // Parsing failed - return sentinel value
65         std::cout << "      Parse failed, returning default: " <<
66         default_value << std::endl;
67         return default_value;
68     }
69 }
70
71 // Return bool with output parameter - when you need to return value too
72 static bool try_connect(const std::string& url, int& out_status_code) {
73     std::cout << "[1.3] Attempting connection to: " << url << std::endl;
74
75     // Simulate connection attempt
76     if (url.find("localhost") != std::string::npos) {
77         out_status_code = 200;
78         std::cout << "      Connected successfully, status: " <<
79         out_status_code << std::endl;
80         return true; // Success
81     }
82
83     out_status_code = 404;
84     std::cout << "      Connection failed, status: " << out_status_code <<
85     std::endl;
86     return false; // Expected failure - caller can retry
87 }
88
89 void demonstrate_return_values() {
90     std::cout << "\n== 1. RETURN VALUES - Expected/Recoverable Errors ==" <<
91     std::endl;
92     std::cout << "Use when: Function can't complete, but caller can handle it"
93     << std::endl;
94
95     // Example 1: Optional value
96     auto url = ConfigParser::read_config_value("server_url");
97     if (url) {
98         std::cout << "  Found URL: " << *url << std::endl;
99     } else {
100         std::cout << "  Using default URL" << std::endl;
101     }
102 }
```

```
97 // Missing key - caller provides default
98 auto missing = ConfigParser::read_config_value("missing_key");
99 std::string final_value = missing.value_or("default_value");
100 std::cout << " Using value: " << final_value << std::endl;
101
102 // Example 2: Integer parsing with default
103 int timeout = ConfigParser::parse_integer("abc", 10);
104 std::cout << " Timeout value: " << timeout << " seconds" << std::endl;
105
106 // Example 3: Try pattern
107 int status_code;
108 if (ConfigParser::try_connect("http://invalid.url", status_code)) {
109     std::cout << " Connected" << std::endl;
110 } else {
111     std::cout << " Connection failed with code: " << status_code << " (
112         retry possible)" << std::endl;
113 }
114
115 std::cout << "\n When to use RETURN VALUES:" << std::endl;
116 std::cout << " • Error is EXPECTED (file not found, network timeout)" <<
117     std::endl;
118 std::cout << " • Caller can RECOVER (retry, use default, ask user)" <<
119     std::endl;
120 std::cout << " • Normal flow of control" << std::endl;
121 std::cout << " • Performance-critical code" << std::endl;
122 }
123
124 // =====
125 // 2. EXCEPTIONS - For Constructors and Unexpected Errors
126 // =====
127
128 // Problem: Partially constructed objects are dangerous!
129
130 class FileHandler_BAD {
131     std::string filename;
132     FILE* file_ptr; // Raw pointer - dangerous!
133     bool is_valid; // Flag to track construction success
134
135 public:
136     // BAD: Constructor can't return error, uses flag instead
137     FileHandler_BAD(const std::string& fname)
138         : filename(fname), file_ptr(nullptr), is_valid(false) {
139
140         std::cout << "\n[2.1 BAD] Attempting to open file: " << fname << std::endl;
141
142         file_ptr = fopen(fname.c_str(), "r");
143         if (file_ptr == nullptr) {
144             // PROBLEM: Can't return error from constructor
145             // Object is partially constructed with invalid state!
146             std::cout << " File open failed, but object still created!" << std::endl;
147             is_valid = false; // Caller must check this flag
148 }
```

```
146         return;
147     }
148
149     is_valid = true;
150 }
151
152 // Every member function must check is_valid!
153 bool read_line(std::string& line) {
154     if (!is_valid || !file_ptr) {
155         return false; // Undefined behavior if caller forgets to check!
156     }
157     // ... read logic
158     return true;
159 }
160
161 ~FileHandler_BAD() {
162     if (file_ptr) {
163         fclose(file_ptr);
164     }
165 }
166
167 bool valid() const { return is_valid; }
168 };
169
170 // Solution: Use RAII and throw from constructor
171
172 class FileHandler_GOOD {
173     std::string filename;
174     std::unique_ptr<FILE, decltype(&fclose)> file_ptr; // RAII wrapper
175
176 public:
177     // GOOD: Constructor either succeeds completely or throws
178     FileHandler_GOOD(const std::string& fname)
179         : filename(fname), file_ptr(nullptr, &fclose) {
180
181         std::cout << "\n[2.2 GOOD] Attempting to open file: " << fname << std
182             ::endl;
183
184         FILE* raw_ptr = fopen(fname.c_str(), "r");
185         if (raw_ptr == nullptr) {
186             // THROW: Constructor can't establish invariant
187             std::cout << "           Throwing exception - no partial object!" <<
188                 std::endl;
189             throw std::runtime_error("Failed to open file: " + fname);
190         }
191
192         file_ptr.reset(raw_ptr);
193         std::cout << "           File opened successfully, object fully
194             constructed" << std::endl;
195
196         // Invariant established: file_ptr is valid
197         // No need for is_valid flag - if object exists, it's valid!
198     }
199 }
```

```
197 // No need to check validity - if object exists, it's valid
198 std::string read_line() {
199     // Can safely use file_ptr - invariant guaranteed by constructor
200     char buffer[256];
201     if (fgets(buffer, sizeof(buffer), file_ptr.get())) {
202         return std::string(buffer);
203     }
204     return "";
205 }
206
207 // Destructor automatically called even if constructor throws
208 // unique_ptr ensures file is closed
209 ~FileHandler_GOOD() {
210     std::cout << "      Closing file: " << filename << std::endl;
211     // file_ptr's destructor automatically closes file
212 }
213 };
214
215 // Another example: Vector with invariants
216
217 class BoundedVector_BAD {
218     std::vector<int> data;
219     size_t max_size;
220     bool is_valid;
221
222 public:
223     // BAD: Doesn't enforce invariant in constructor
224     BoundedVector_BAD(size_t max_sz, const std::vector<int>& initial)
225         : max_size(max_sz), is_valid(false) {
226
227         std::cout << "\n[2.3 BAD] Creating bounded vector, max_size="
228             << max_sz << ", initial.size()=" << initial.size() << std::endl;
229
230         if (initial.size() > max_sz) {
231             // PROBLEM: Partial construction!
232             std::cout << "      Initial size exceeds max, but object created
233             anyway!" << std::endl;
234             is_valid = false;
235             return;
236         }
237
238         data = initial;
239         is_valid = true;
240     }
241
242     bool push_back(int value) {
243         if (!is_valid) return false; // Must check everywhere!
244         if (data.size() >= max_size) return false;
245         data.push_back(value);
246         return true;
247     }
248
249     bool valid() const { return is_valid; }
```

```
249 };  
250  
251 class BoundedVector_GOOD {  
252     std::vector<int> data;  
253     size_t max_size;  
254  
255 public:  
256     // GOOD: Enforces invariant, throws if violated  
257     BoundedVector_GOOD(size_t max_sz, const std::vector<int>& initial)  
258         : max_size(max_sz) {  
259  
260         std::cout << "\n[2.4 GOOD] Creating bounded vector, max_size="  
261             << max_sz << ", initial.size()=" << initial.size() << std:::  
262             endl;  
263  
264         if (initial.size() > max_sz) {  
265             // THROW: Invariant cannot be established  
266             std::cout << "           Throwing - initial size exceeds maximum!" <<  
267             std::endl;  
268             throw std::invalid_argument(  
269                 "Initial size " + std::to_string(initial.size()) +  
270                 " exceeds maximum " + std::to_string(max_sz)  
271             );  
272         }  
273  
274         data = initial;  
275         std::cout << "           Bounded vector created, invariant satisfied" <<  
276             std::endl;  
277     }  
278  
279     void push_back(int value) {  
280         // Invariant guaranteed - no need to check is_valid  
281         if (data.size() >= max_size) {  
282             throw std::length_error("Vector at maximum capacity");  
283         }  
284         data.push_back(value);  
285     }  
286  
287     size_t size() const { return data.size(); }  
288 };  
289  
290 // RAII Example: Exception safety with resource management  
291  
292 class ResourceManager {  
293     std::unique_ptr<int[]> buffer;  
294     FILE* log_file;  
295     bool initialized;  
296  
297 public:  
298     // RAII: Resources acquired in constructor  
299     ResourceManager(size_t buffer_size, const std::string& log_path)
```

```
300     : buffer(nullptr), log_file(nullptr), initialized(false) {
301
302     std::cout << "\n[2.5] Constructing ResourceManager..." << std::endl;
303
304     // Step 1: Allocate buffer (RAII with unique_ptr)
305     buffer = std::make_unique<int[]>(buffer_size);
306     std::cout << "           Buffer allocated" << std::endl;
307
308     // Step 2: Open log file
309     log_file = fopen(log_path.c_str(), "a");
310     if (!log_file) {
311         std::cout << "           Log file open failed - throwing..." << std::endl;
312         // buffer automatically deallocated by unique_ptr destructor!
313         throw std::runtime_error("Failed to open log file");
314     }
315     std::cout << "           Log file opened" << std::endl;
316
317     initialized = true;
318     std::cout << "           ResourceManager fully constructed" << std::endl;
319 }
320
321 ~ResourceManager() {
322     std::cout << "           Cleaning up ResourceManager..." << std::endl;
323
324     if (log_file) {
325         fclose(log_file);
326         std::cout << "           Log file closed" << std::endl;
327     }
328
329     // buffer automatically deallocated by unique_ptr
330     std::cout << "           Buffer deallocated" << std::endl;
331 }
332
333 // Disable copy to prevent resource issues
334 ResourceManager(const ResourceManager&) = delete;
335 ResourceManager& operator=(const ResourceManager&) = delete;
336
337 // Enable move for transfer of ownership
338 ResourceManager(ResourceManager&& other) noexcept
339     : buffer(std::move(other.buffer))
340     , log_file(other.log_file)
341     , initialized(other.initialized) {
342     other.log_file = nullptr;
343     other.initialized = false;
344 }
345 };
346
347 void demonstrate_exceptions() {
348     std::cout << "\n== 2. EXCEPTIONS - Constructors and Invariants ==" <<
349     std::endl;
350     std::cout << "Use when: Constructor can't establish object invariants" <<
351     std::endl;
352 }
```

```
351 // Example 2.1: BAD - Partially constructed object
352 std::cout << "\n--- Demonstrating BAD approach ---" << std::endl;
353 {
354     FileHandler_BAD bad_handler("nonexistent_file.txt");
355     if (!bad_handler.valid()) {
356         std::cout << " Object exists but is invalid - DANGEROUS!" << std
357             ::endl;
358         std::cout << " Every function must check valid() flag" << std::
359             endl;
360     }
361 }
362
363 // Example 2.2: GOOD - Exception prevents partial construction
364 std::cout << "\n--- Demonstrating GOOD approach ---" << std::endl;
365 try {
366     FileHandler_GOOD good_handler("nonexistent_file.txt");
367     // This line never executes if file doesn't exist
368     std::cout << " Object constructed, guaranteed valid" << std::endl;
369 } catch (const std::exception& e) {
370     std::cout << " Exception caught: " << e.what() << std::endl;
371     std::cout << " No object exists - no invalid state possible!" << std
372         ::endl;
373 }
374
375 // Example 2.3: BAD - Invariant violation
376 std::cout << "\n--- Bounded Vector: BAD approach ---" << std::endl;
377 {
378     std::vector<int> too_large = {1, 2, 3, 4, 5};
379     BoundedVector_BAD bad_vec(3, too_large); // max_size=3, but initial
380         has 5
381     if (!bad_vec.valid()) {
382         std::cout << " Invalid object exists with violated invariant!" <<
383             std::endl;
384     }
385 }
386
387 // Example 2.4: GOOD - Exception prevents invariant violation
388 std::cout << "\n--- Bounded Vector: GOOD approach ---" << std::endl;
389 try {
390     std::vector<int> too_large = {1, 2, 3, 4, 5};
391     BoundedVector_GOOD good_vec(3, too_large); // Will throw
392 } catch (const std::invalid_argument& e) {
393     std::cout << " Exception caught: " << e.what() << std::endl;
394     std::cout << " Invariant violation prevented!" << std::endl;
395 }
396
397 // Example 2.5: RAII ensures cleanup even with exceptions
398 std::cout << "\n--- RAII with Exception ---" << std::endl;
399 try {
400     ResourceManager mgr(1024, "/invalid/path/log.txt"); // Will fail
401 } catch (const std::exception& e) {
402     std::cout << " Exception caught: " << e.what() << std::endl;
403     std::cout << " Buffer was automatically cleaned up (RAII)!" << std::
404         endl;
```

```

399 }
400
401     std::cout << "\n When to use EXCEPTIONS:" << std::endl;
402     std::cout << " • Constructor can't establish invariants" << std::endl;
403     std::cout << " • Partially constructed object would be dangerous" << std::endl;
404     std::cout << " • Error is UNEXPECTED (programming error, resource exhaustion)" << std::endl;
405     std::cout << " • Can't return error value (constructors, operators)" << std::endl;
406     std::cout << " • RAII ensures cleanup even if exception thrown" << std::endl;
407 }
408
409 // =====
410 // 3. TERMINATION - For Unrecoverable Errors
411 // =====
412
413 // Use case: Programming error or invariant violation where continuing is dangerous
414
415 class CriticalSystem {
416     int* data_ptr;
417     size_t size;
418
419     // Internal invariant check
420     void checkInvariant() const {
421         if (data_ptr == nullptr && size > 0) {
422             // PROGRAMMING ERROR: Inconsistent state!
423             std::cerr << "\n FATAL: Invariant violated - nullptr with size > 0" << std::endl;
424             std::cerr << "    This should never happen - terminating!" << std::endl;
425             std::terminate(); // Cannot continue safely
426         }
427     }
428
429 public:
430     CriticalSystem(size_t sz) : data_ptr(nullptr), size(sz) {
431         if (sz == 0) {
432             std::cerr << "\n FATAL: Zero-size system not allowed" << std::endl;
433             std::terminate(); // Precondition violation
434         }
435
436         data_ptr = new int[sz];
437         std::cout << "[3.1] CriticalSystem constructed with size " << sz << std::endl;
438     }
439
440     void set_value(size_t index, int value) {
441         checkInvariant(); // Verify internal consistency
442
443         if (index >= size) {

```

```
444         // PROGRAMMING ERROR: Out of bounds access
445         std::cerr << "\n FATAL: Index " << index << " out of bounds [0, "
446             << size << ")" << std::endl;
447         std::cerr << "    This indicates a bug - terminating!" << std::endl
448             ;
449         std::abort(); // Cannot recover from logic error
450     }
451
452     data_ptr[index] = value;
453 }
454
455 int get_value(size_t index) const {
456     // Use assert for debug builds (removed in release with NDEBUG)
457     assert(index < size && "Index out of bounds");
458
459     if (index >= size) {
460         std::cerr << "\n FATAL: Index out of bounds in release build" <<
461             std::endl;
462         std::terminate();
463     }
464
465     return data_ptr[index];
466 }
467
468 ~CriticalSystem() {
469     delete[] data_ptr;
470 }
471 // Comparison: When to use each approach
472
473 class SafetyLevel {
474 public:
475     enum class Level {
476         INFO,           // Normal operation
477         WARNING,        // Something unexpected but recoverable
478         ERROR,          // Error that can be handled
479         CRITICAL,       // Error requiring immediate attention
480         FATAL           // Cannot continue
481     };
482
483     // Different error handling based on severity
484     static void handle_error(Level level, const std::string& message) {
485         switch (level) {
486             case Level::INFO:
487                 // Just log - continue normally
488                 std::cout << "\n INFO: " << message << std::endl;
489                 break;
490
491             case Level::WARNING:
492                 // Log and return - caller handles it
493                 std::cout << "\n WARNING: " << message << std::endl;
494                 // Return value or optional
495                 break;
496         }
497     }
498 }
```

```
496
497     case Level::ERROR:
498         // Throw exception - caller can catch and recover
499         std::cout << "\n ERROR: " << message << std::endl;
500         throw std::runtime_error(message);
501
502     case Level::CRITICAL:
503         // Log critical error, attempt emergency cleanup
504         std::cerr << "\n CRITICAL: " << message << std::endl;
505         std::cerr << "    Attempting emergency save..." << std::endl;
506         // ... emergency cleanup ...
507         throw std::runtime_error("Critical: " + message);
508
509     case Level::FATAL:
510         // Cannot continue - terminate immediately
511         std::cerr << "\n FATAL: " << message << std::endl;
512         std::cerr << "    Cannot continue safely - terminating!" << std::endl;
513         std::terminate();
514     }
515 }
516 };
517
518 void demonstrate_termination() {
519     std::cout << "\n--- 3. TERMINATION - Unrecoverable Errors ---" << std::endl;
520     std::cout << "Use when: Continuing would be dangerous or impossible" << std::endl;
521
522     // Example: Normal operation
523     std::cout << "\n--- Normal operation ---" << std::endl;
524     {
525         CriticalSystem sys(10);
526         sys.set_value(0, 42);
527         sys.set_value(5, 100);
528         std::cout << "    Values set successfully" << std::endl;
529     }
530
531     // Example: Safety levels
532     std::cout << "\n--- Different safety levels ---" << std::endl;
533
534     // INFO - just logging
535     SafetyLevel::handle_error(SafetyLevel::Level::INFO, "System started");
536
537     // WARNING - caller can ignore or handle
538     SafetyLevel::handle_error(SafetyLevel::Level::WARNING, "Cache miss (using
539         fallback)");
540
541     // ERROR - throw exception
542     try {
543         SafetyLevel::handle_error(SafetyLevel::Level::ERROR, "Network timeout");
544     } catch (const std::exception& e) {
545         std::cout << "    Caught and recovered from ERROR: " << e.what() << std::endl;
```

```
        ::endl;
545 }
546
547 std::cout << "\n When to use TERMINATION:" << std::endl;
548 std::cout << " • PROGRAMMING ERROR detected (invariant violated)" << std
      ::endl;
549 std::cout << " • Precondition violated (contract broken)" << std::endl;
550 std::cout << " • Continuing would corrupt data or cause undefined
      behavior" << std::endl;
551 std::cout << " • Use assert() in debug builds (removed in release)" <<
      std::endl;
552 std::cout << " • Use std::terminate() or std::abort() for fatal errors"
      << std::endl;
553
554 // Note: The examples below would terminate if uncommented
555 std::cout << "\n The following would terminate (commented out):" << std
      ::endl;
556 std::cout << " // CriticalSystem sys(0); // Zero size - would terminate
      " << std::endl;
557 std::cout << " // sys.set_value(999, 42); // Out of bounds - would
      terminate" << std::endl;
558 }
559
560 // =====
561 // SUMMARY: Decision Guide
562 // =====
563
564 void print_decision_guide() {
565     std::cout << "\n
566         =====
567         std::endl;
568     std::cout << " DECISION GUIDE: Which Error Handling Strategy?" << std::
569         endl;
570     std::cout << "
571         =====
572         std::endl;
573
574     std::cout << "\n 1. RETURN VALUES (std::optional, bool, error codes)" <<
575         std::endl;
576     std::cout << "     Use when:" << std::endl;
577     std::cout << " •     Error is EXPECTED and part of normal operation" <<
578         std::endl;
579     std::cout << " •     Caller can and should handle the error" << std::endl
580         ;
581     std::cout << " •     Checking for availability (file exists, key found)" <<
582         std::endl;
583     std::cout << " •     Performance-critical code" << std::endl;
584     std::cout << " •     Parsing/validation where failure is common" << std::
585         endl;
586
587     std::cout << "\n     Examples:" << std::endl;
588     std::cout << " •     std::optional<T> find_user(int id)" << std::endl;
589     std::cout << " •     bool try_connect(const string& url, int& status)" <<
590         std::endl;
591     std::cout << " •     std::expected<Value, Error> parse_config()" << std::
```

```

        endl;

580    std::cout << "\n\n 2. EXCEPTIONS (throw/catch)" << std::endl;
581    std::cout << "    Use when:" << std::endl;
582    std::cout << " •     Constructor cannot establish invariants" << std::
583        endl;
584    std::cout << " •     Partially constructed object would be dangerous" <<
585        std::endl;
585    std::cout << " •     Error is UNEXPECTED (resource exhaustion, logic
586        error)" << std::endl;
586    std::cout << " •     Cannot return error value (constructors, operators)" "
587        << std::endl;
587    std::cout << " •     Need to skip multiple stack frames to handler" <<
588        std::endl;
588    std::cout << " •     RAIII ensures automatic cleanup" << std::endl;
589    std::cout << "\n    Examples:" << std::endl;
590    std::cout << " •     FileHandle(const string& path) // Constructor" <<
591        std::endl;
591    std::cout << " •     BoundedVector(size_t max, vector<T> initial) // "
592        "Invariant" << std::endl;
592    std::cout << " •     operator[](size_t idx) // Can't return error" <<
593        std::endl;

593    std::cout << "\n\n 3. TERMINATION (std::terminate, std::abort, assert)"
594        << std::endl;
595    std::cout << "    Use when:" << std::endl;
596    std::cout << " •     PROGRAMMING ERROR (bug in code)" << std::endl;
597    std::cout << " •     Invariant violated (internal consistency broken)" <<
598        std::endl;
598    std::cout << " •     Precondition not met (contract violated)" << std::
599        endl;
599    std::cout << " •     Continuing would cause data corruption" << std::endl
600        ;
600    std::cout << " •     Undefined behavior would result" << std::endl;
601    std::cout << "\n    Examples:" << std::endl;
602    std::cout << " •     assert(ptr != nullptr); // Debug check" << std::
603        endl;
603    std::cout << " •     if (invariant_broken()) std::terminate();" << std::
604        endl;
604    std::cout << " •     if (out_of_bounds) std::abort();" << std::endl;

605    std::cout << "\n\n COMPARISON TABLE:" << std::endl;
606    std::cout << "                                         " << std::endl;
607    std::cout << " Aspect           Return Value   Exception   Termination "
608        << std::endl;
609    std::cout << "                                         " << std::endl;
610    std::cout << " Error frequency   Common       Rare        Never* "
611        << std::endl;
611    std::cout << " Recoverability     Recoverable   May recover  Fatal "
612        << std::endl;
612    std::cout << " Caller control     Explicit     Can catch   None "
613        << std::endl;
613    std::cout << " Performance        Fast         Slow        N/A "

```

```

614     std::cout << " Constructor use No Yes Yes (rare)"
615         " << std::endl;
616     std::cout << " RAIII cleanup Manual Automatic No
617         " << std::endl;
618     std::cout << " " << std::endl;
619     std::cout << "* \"Never\" = should never happen in correct program" << std
620         ::endl;
621
622     std::cout << "\n\n BJARNE'S KEY INSIGHTS:" << std::endl;
623     std::cout << "\n 1 \"A constructor establishes the invariant for a
624         class\" " << std::endl;
625     std::cout << " → Must throw if invariant cannot be established" <<
626         std::endl;
627     std::cout << " → No partially constructed objects" << std::endl;
628
629     std::cout << "\n 2 \"Use exceptions for exceptional circumstances\" "
630         << std::endl;
631     std::cout << " → Not for normal control flow" << std::endl;
632     std::cout << " → When caller can't be expected to check every call"
633         << std::endl;
634
635     std::cout << "\n 3 \"Use error codes when errors are expected\" "
636         << std::endl;
637     std::cout << " → File not found, network timeout, parse error" << std
638         ::endl;
639     std::cout << " → Caller should handle explicitly" << std::endl;
640
641     std::cout << "\n 4 \"Terminate when continuing would be dangerous\" "
642         << std::endl;
643     std::cout << " → Programming errors (bugs)" << std::endl;
644     std::cout << " → Invariant violations" << std::endl;
645
646     std::cout << "\n 5 \"RAII is fundamental to exception safety\" "
647         << std
648         ::endl;
649     std::cout << " → Resources acquired in constructor" << std::endl;
650     std::cout << " → Released in destructor" << std::endl;
651     std::cout << " → Cleanup happens even with exceptions" << std::endl;
652
653     std::cout << "\n ====="
654         std::endl;
655
656 // =====
657 // MAIN
658 // =====
659
660 int main() {
661     std::cout << "\n ====="
662         std::endl;
663     std::cout << "  ERROR HANDLING STRATEGIES" << std::endl;
664     std::cout << "  Following Bjarne Stroustrup's Philosophy" << std::endl;
665     std::cout << "

```

```
===== " <<
653     std::endl;
654
655     try {
656         demonstrate_return_values();
657         demonstrate_exceptions();
658         demonstrate_termination();
659         print_decision_guide();
660
661         std::cout << "\n All demonstrations completed successfully!" << std::
662             endl;
663     } catch (const std::exception& e) {
664         std::cerr << "\n Unexpected exception: " << e.what() << std::endl;
665         return 1;
666     }
667
668     return 0;
669 }
```

## 26 Source Code: EventDrivenProgramming\_Inheritance.cpp

File: src/EventDrivenProgramming\_Inheritance.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <memory>
5 #include <algorithm>
6 #include <queue>
7
8 // =====
9 // EVENT-DRIVEN PROGRAMMING: TRADITIONAL INHERITANCE-BASED APPROACH
10 // =====
11 // This file demonstrates traditional event-driven programming using:
12 // - Abstract base classes
13 // - Virtual functions
14 // - Inheritance hierarchies
15 // - Classic OOP patterns
16 // =====
17
18 // =====
19 // 1. OBSERVER PATTERN (INHERITANCE-BASED)
20 // =====
21
22 // Abstract observer interface
23 class IObserver {
24 public:
25     virtual ~IObserver() = default;
26     virtual void on_notify(const std::string& event) = 0;
27 };
28
29 // Concrete observer implementations
30 class ConsoleObserver : public IObserver {
31 private:
32     std::string name;
33
34 public:
35     ConsoleObserver(std::string n) : name(std::move(n)) {}
36
37     void on_notify(const std::string& event) override {
38         std::cout << "      [" << name << "] Received: " << event << std::endl;
39     }
40 };
41
42 class CountingObserver : public IObserver {
43 private:
44     int count = 0;
45
46 public:
47     void on_notify(const std::string& event) override {
48         count++;
49         std::cout << "      [Counter] Event #" << count << ":" << event << std::endl;
50 }
```

```
        ::endl;
50    }
51
52    int get_count() const { return count; }
53};
54
55 class Subject {
56 private:
57     std::vector<IObserver*> observers; // Raw pointers or shared_ptr
58
59 public:
60     void attach(IObserver* observer) {
61         observers.push_back(observer);
62     }
63
64     void detach(IObserver* observer) {
65         observers.erase(
66             std::remove(observers.begin(), observers.end(), observer),
67             observers.end()
68         );
69     }
70
71     void notify(const std::string& event) {
72         std::cout << " [Subject] Notifying " << observers.size() << " "
73             "observers" << std::endl;
74         for (auto* observer : observers) {
75             observer->on_notify(event);
76         }
77     }
78 };
79
80 void example_inheritance_observer() {
81     std::cout << "\n==== 1. OBSERVER PATTERN (INHERITANCE-BASED) ===" << std::endl;
82
83     Subject subject;
84
85     // Need to create concrete observer objects
86     ConsoleObserver obs1("Observer1");
87     ConsoleObserver obs2("Observer2");
88     CountingObserver counter;
89
90     subject.attach(&obs1);
91     subject.attach(&obs2);
92     subject.attach(&counter);
93
94     subject.notify("User logged in");
95     subject.notify("Data updated");
96
97     std::cout << "\n DISADVANTAGES:" << std::endl;
98     std::cout << " • Must define observer classes" << std::endl;
99     std::cout << " • Need inheritance hierarchy" << std::endl;
100    std::cout << " • More boilerplate code" << std::endl;
101    std::cout << " • Lifetime management issues (pointers)" << std::endl;
```

```
101 }
102
103 // =====
104 // 2. EVENT HIERARCHY (INHERITANCE-BASED)
105 // =====
106
107 // Abstract event base class
108 class Event {
109 public:
110     virtual ~Event() = default;
111     virtual std::string get_name() const = 0;
112     virtual void print() const = 0;
113 };
114
115 class MouseClickEvent : public Event {
116 private:
117     int x, y;
118     std::string button;
119
120 public:
121     MouseClickEvent(int x, int y, std::string btn)
122         : x(x), y(y), button(std::move(btn)) {}
123
124     std::string get_name() const override { return "MouseClick"; }
125
126     void print() const override {
127         std::cout << "    Mouse clicked: " << button
128             << " at (" << x << "," << y << ")" << std::endl;
129     }
130
131     int get_x() const { return x; }
132     int get_y() const { return y; }
133     const std::string& get_button() const { return button; }
134 };
135
136 class KeyPressEvent : public Event {
137 private:
138     char key;
139     bool ctrl;
140
141 public:
142     KeyPressEvent(char k, bool c) : key(k), ctrl(c) {}
143
144     std::string get_name() const override { return "KeyPress"; }
145
146     void print() const override {
147         std::cout << "    Key pressed: '" << key << "'"
148             << (ctrl ? " (Ctrl)" : "") << std::endl;
149     }
150
151     char get_key() const { return key; }
152     bool has_ctrl() const { return ctrl; }
153 };
154
```

```
155 // Abstract event handler
156 class IEventHandler {
157 public:
158     virtual ~IEventHandler() = default;
159     virtual void handle_event(const Event& event) = 0;
160 };
161
162 class EventLogger : public IEventHandler {
163 public:
164     void handle_event(const Event& event) override {
165         std::cout << "    [Logger] Handling " << event.get_name() << std::endl
166             ;
167         event.print();
168     }
169 };
170
171 class MouseHandler : public IEventHandler {
172 public:
173     void handle_event(const Event& event) override {
174         // Need to use dynamic_cast to access specific event data
175         if (auto* mouse_event = dynamic_cast<const MouseClickEvent*>(&event))
176         {
177             std::cout << "    [MouseHandler] Processing mouse at "
178                 << mouse_event->get_x() << "," << mouse_event->get_y()
179                 << ")" << std::endl;
180         }
181         // Ignore other event types
182     }
183 };
184
185 class EventDispatcher {
186 private:
187     std::vector<IEventHandler*> handlers;
188
189 public:
190     void subscribe(IEventHandler* handler) {
191         handlers.push_back(handler);
192     }
193
194     void dispatch(const Event& event) {
195         for (auto* handler : handlers) {
196             handler->handle_event(event);
197         }
198     }
199
200     void example_event_hierarchy() {
201         std::cout << "\n==== 2. EVENT HIERARCHY (INHERITANCE-BASED) ===" << std::endl;
202
203         EventDispatcher dispatcher;
204
205         EventLogger logger;
206         MouseHandler mouse_handler;
```

```
205     dispatcher.subscribe(&logger);
206     dispatcher.subscribe(&mouse_handler);
207
208     std::cout << "\nDispatching events:" << std::endl;
209     MouseClickEvent click(100, 200, "left");
210     dispatcher.dispatch(click);
211
212     KeyPressEvent key('A', true);
213     dispatcher.dispatch(key);
214
215     std::cout << "\n DISADVANTAGES:" << std::endl;
216     std::cout << " • Need event class hierarchy" << std::endl;
217     std::cout << " • Must use dynamic_cast (RTTI)" << std::endl;
218     std::cout << " • Pointer semantics required" << std::endl;
219     std::cout << " • More complex type checking" << std::endl;
220 }
221
222 // =====
223 // 3. LISTENER PATTERN (CLASSIC GUI APPROACH)
224 // =====
225
226
227 class IButtonListener {
228 public:
229     virtual ~IButtonListener() = default;
230     virtual void on_button_clicked(const std::string& button_id) = 0;
231 };
232
233 class Button {
234 private:
235     std::string id;
236     std::vector<IButtonListener*> listeners;
237
238 public:
239     Button(std::string button_id) : id(std::move(button_id)) {}
240
241     void add_listener(IButtonListener* listener) {
242         listeners.push_back(listener);
243     }
244
245     void click() {
246         std::cout << " [Button '" << id << "'] Clicked!" << std::endl;
247         for (auto* listener : listeners) {
248             listener->on_button_clicked(id);
249         }
250     }
251 };
252
253 class SaveHandler : public IButtonListener {
254 public:
255     void on_button_clicked(const std::string& button_id) override {
256         std::cout << " [SaveHandler] Saving file..." << std::endl;
257     }
258 };
```

```
259
260 class LogHandler : public IButtonListener {
261 public:
262     void on_button_clicked(const std::string& button_id) override {
263         std::cout << "    [LogHandler] Logging click on " << button_id << std
264             ::endl;
265     }
266 };
267
268 void example_listener_pattern() {
269     std::cout << "\n==== 3. LISTENER PATTERN (CLASSIC GUI) ===" << std::endl;
270
271     Button save_btn("SaveButton");
272
273     SaveHandler save_handler;
274     LogHandler log_handler;
275
276     save_btn.add_listener(&save_handler);
277     save_btn.add_listener(&log_handler);
278
279     save_btn.click();
280
281     std::cout << "\n DISADVANTAGES:" << std::endl;
282     std::cout << " • Need listener interface classes" << std::endl;
283     std::cout << " • Can't define behavior inline" << std::endl;
284     std::cout << " • More classes to maintain" << std::endl;
285     std::cout << " • Harder to share state" << std::endl;
286 }
287
288 // =====
289 // 4. COMMAND PATTERN (INHERITANCE-BASED)
290 // =====
291
292 class ICommand {
293 public:
294     virtual ~ICommand() = default;
295     virtual void execute() = 0;
296     virtual void undo() = 0;
297     virtual std::string get_description() const = 0;
298 };
299
300 class TextCommand : public ICommand {
301 private:
302     std::string& text_ref;
303     std::string new_text;
304     std::string old_text;
305
306 public:
307     TextCommand(std::string& text, std::string new_val)
308         : text_ref(text), new_text(std::move(new_val)), old_text(text) {}
309
310     void execute() override {
311         std::cout << "    Execute: Set text to '" << new_text << "'" << std::
312             endl;
```

```
311     text_ref = new_text;
312 }
313
314 void undo() override {
315     std::cout << "      Undo: Restore text to '" << old_text << "'";
316     endl;
317     text_ref = old_text;
318 }
319
320 std::string get_description() const override {
321     return "Change text to '" + new_text + "'";
322 }
323
324 class CommandManager {
325 private:
326     std::vector<std::unique_ptr< ICommand>> history;
327     size_t current = 0;
328
329 public:
330     void execute(std::unique_ptr< ICommand> cmd) {
331         cmd->execute();
332
333         // Clear redo history
334         history.erase(history.begin() + current, history.end());
335
336         history.push_back(std::move(cmd));
337         current = history.size();
338     }
339
340     void undo() {
341         if (current > 0) {
342             current--;
343             history[current]->undo();
344         }
345     }
346
347     void redo() {
348         if (current < history.size()) {
349             history[current]->execute();
350             current++;
351         }
352     }
353 };
354
355 void example_command_pattern() {
356     std::cout << "\n==== 4. COMMAND PATTERN (INHERITANCE-BASED) ===" << std::endl;
357
358     std::string document = "Original";
359     CommandManager manager;
360
361     std::cout << "\nExecuting commands:" << std::endl;
362     manager.execute(std::make_unique<TextCommand>(document, "Modified 1"));
```

```
363     std::cout << "      Text: " << document << std::endl;
364
365     manager.execute(std::make_unique<TextCommand>(document, "Modified 2"));
366     std::cout << "      Text: " << document << std::endl;
367
368     std::cout << "\nUndoing commands:" << std::endl;
369     manager.undo();
370     std::cout << "      Text: " << document << std::endl;
371
372     manager.undo();
373     std::cout << "      Text: " << document << std::endl;
374
375     std::cout << "\n DISADVANTAGES:" << std::endl;
376     std::cout << " • Need command class for each operation" << std::endl;
377     std::cout << " • More verbose class definitions" << std::endl;
378     std::cout << " • State management in constructors" << std::endl;
379     std::cout << " • Can't define commands inline" << std::endl;
380 }
381
382 // =====
383 // 5. CALLBACK INTERFACE (OLD-SCHOOL)
384 // =====
385
386 class ICallback {
387 public:
388     virtual ~ICallback() = default;
389     virtual void on_success(int result) = 0;
390     virtual void on_error(const std::string& error) = 0;
391 };
392
393 class AsyncOperation {
394 private:
395     ICallback* callback = nullptr;
396
397 public:
398     void set_callback(ICallback* cb) {
399         callback = cb;
400     }
401
402     void execute(bool should_fail = false) {
403         if (callback) {
404             if (should_fail) {
405                 callback->on_error("Operation failed!");
406             } else {
407                 callback->on_success(42);
408             }
409         }
410     }
411 };
412
413 class MyCallback : public ICallback {
414 public:
415     void on_success(int result) override {
416         std::cout << " Success! Result: " << result << std::endl;
```

```
417     }
418
419     void on_error(const std::string& error) override {
420         std::cout << "    Error: " << error << std::endl;
421     }
422 };
423
424 void example_callback_interface() {
425     std::cout << "\n==> 5. CALLBACK INTERFACE (OLD-SCHOOL) ==<" << std::endl;
426
427     AsyncOperation op;
428     MyCallback callback;
429
430     op.set_callback(&callback);
431
432     std::cout << "\n--- Success case ---" << std::endl;
433     op.execute(false);
434
435     std::cout << "\n--- Error case ---" << std::endl;
436     op.execute(true);
437
438     std::cout << "\n DISADVANTAGES:" << std::endl;
439     std::cout << " • Must define callback class" << std::endl;
440     std::cout << " • Can't chain callbacks easily" << std::endl;
441     std::cout << " • Only one callback per operation" << std::endl;
442     std::cout << " • Less readable than fluent API" << std::endl;
443 }
444
445 // =====
446 // MAIN FUNCTION
447 // =====
448
449 int main() {
450     std::cout << "\n
451         ====="
452         std::endl;
453     std::cout << "    EVENT-DRIVEN PROGRAMMING: TRADITIONAL INHERITANCE APPROACH
454         " << std::endl;
455     std::cout << "
456         ====="
457         std::endl;
458
459     example_inheritance_observer();
460     example_event_hierarchy();
461     example_listener_pattern();
462     example_command_pattern();
463     example_callback_interface();
464
465     std::cout << "\n
466         ====="
467         std::endl;
468     std::cout << "    SUMMARY: PROBLEMS WITH INHERITANCE-BASED APPROACH" << std
469         ::endl;
470     std::cout << "
```

```
=====
    std::endl;

463 std::cout << "\n DISADVANTAGES OF INHERITANCE-BASED APPROACH:" << std::endl;
464
465 std::cout << "\n1. BOILERPLATE CODE" << std::endl;
466 std::cout << " • Must define interface classes (IObserver, IListener,
467   etc.)" << std::endl;
468 std::cout << " • Must define concrete implementations" << std::endl;
469 std::cout << " • Each behavior needs a new class" << std::endl;
470 std::cout << " • Lots of virtual functions" << std::endl;
471
472 std::cout << "\n2. INFLEXIBILITY" << std::endl;
473 std::cout << " • Can't define behavior inline" << std::endl;
474 std::cout << " • Hard to share state between callbacks" << std::endl;
475 std::cout << " • Difficult to compose behaviors" << std::endl;
476 std::cout << " • Rigid class hierarchies" << std::endl;
477
478 std::cout << "\n3. LIFETIME MANAGEMENT" << std::endl;
479 std::cout << " • Raw pointers cause dangling reference bugs" << std::endl;
480 std::cout << " • shared_ptr adds complexity and overhead" << std::endl;
481 std::cout << " • Must manage attach/detach carefully" << std::endl;
482 std::cout << " • Who owns the observer objects?" << std::endl;
483
484 std::cout << "\n4. POOR READABILITY" << std::endl;
485 std::cout << " • Behavior scattered across multiple files" << std::endl;
486 std::cout << " • Must jump between class definitions" << std::endl;
487 std::cout << " • Harder to see full picture" << std::endl;
488 std::cout << " • Intent hidden in class names" << std::endl;
489
490 std::cout << "\n5. RTTI AND CASTING" << std::endl;
491 std::cout << " • Need dynamic_cast for event hierarchies" << std::endl;
492 std::cout << " • Runtime type checking (slower)" << std::endl;
493 std::cout << " • Can fail at runtime" << std::endl;
494 std::cout << " • Not as type-safe" << std::endl;
495
496 std::cout << "\n6. MAINTENANCE BURDEN" << std::endl;
497 std::cout << " • More files to maintain" << std::endl;
498 std::cout << " • More classes in codebase" << std::endl;
499 std::cout << " • Harder to refactor" << std::endl;
500 std::cout << " • Changes ripple through hierarchy" << std::endl;
501
502 std::cout << "\n WHEN INHERITANCE IS JUSTIFIED:" << std::endl;
503 std::cout << " • Polymorphic object collections" << std::endl;
504 std::cout << " • Complex state machines" << std::endl;
505 std::cout << " • Need object identity (not just behavior)" << std::endl;
506 std::cout << " • Multiple related virtual methods" << std::endl;
507 std::cout << " • Performance critical (no std::function overhead)" <<
508   std::endl;
509 std::cout << " • Plugin architectures" << std::endl;
510
511 std::cout << "\n MODERN ALTERNATIVE:" << std::endl;
```

```
511     std::cout << "    Use EventDrivenProgramming_Lambdas.cpp for:" << std::endl
512     ;
513     std::cout << "    std::function + lambdas (no inheritance)" << std::endl;
514     std::cout << "    std::variant (no event hierarchy)" << std::endl;
515     std::cout << "    Inline behavior definitions" << std::endl;
516     std::cout << "    Automatic state capture" << std::endl;
517     std::cout << "    Much less boilerplate" << std::endl;
518
519     std::cout << "\n COMPARISON:" << std::endl;
520     std::cout << "    Inheritance approach: ~200 lines for basic observer" <<
521         std::endl;
522     std::cout << "    Lambda approach:           ~50 lines for same functionality"
523         << std::endl;
524     std::cout << "    Code reduction:           75% less code!" << std::endl;
525
526     std::cout << "\n
527     =====\n" <<
528     std::endl;
529
530     return 0;
531 }
```

## 27 Source Code: EventDrivenProgramming\_Lambdas.cpp

File: src/EventDrivenProgramming\_Lambdas.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <functional>
3 #include <string>
4 #include <vector>
5 #include <map>
6 #include <memory>
7 #include <variant>
8 #include <algorithm>
9 #include <queue>
10 #include <typeindex>
11
12 // =====
13 // EVENT-DRIVEN PROGRAMMING: MODERN LAMBDA-BASED APPROACH
14 // =====
15 // This file demonstrates event-driven programming using:
16 // - std::function
17 // - Lambdas with captures
18 // - std::variant
19 // - No inheritance hierarchies needed!
20 // =====
21
22 // =====
23 // 1. OBSERVER PATTERN (LAMBDA-BASED)
24 // =====
25
26 class Subject {
27 private:
28     std::vector<std::function<void(const std::string&)>> observers;
29
30 public:
31     void subscribe(std::function<void(const std::string&)> observer) {
32         observers.push_back(std::move(observer));
33     }
34
35     void notify(const std::string& event) {
36         std::cout << " [Subject] Notifying " << observers.size() << " "
37             observers" << std::endl;
38         for (auto& observer : observers) {
39             observer(event);
40         }
41     }
42 };
43 void example_lambda_observer() {
44     std::cout << "\n==== 1. OBSERVER PATTERN (LAMBDA-BASED) ===" << std::endl;
45
46     Subject subject;
47
48     // No observer class needed! Just pass lambdas
```

```
49     subject.subscribe([](const std::string& event) {
50         std::cout << "    Lambda Observer 1: " << event << std::endl;
51     });
52
53     // Can capture state inline
54     int count = 0;
55     subject.subscribe([&count](const std::string& event) {
56         std::cout << "    Lambda Observer 2 (count=" << ++count << "): " <<
57             event << std::endl;
58     });
59
60     // Can define behavior at subscription time
61     std::string prefix = "CustomPrefix";
62     subject.subscribe([prefix](const std::string& event) {
63         std::cout << "    [" << prefix << "] " << event << std::endl;
64     });
65
66     subject.notify("User logged in");
67     subject.notify("Data updated");
68
69     std::cout << "\n ADVANTAGES:" << std::endl;
70     std::cout << " • No observer interface/base class needed" << std::endl;
71     std::cout << " • Inline behavior definition" << std::endl;
72     std::cout << " • Can capture local state" << std::endl;
73     std::cout << " • Less boilerplate code" << std::endl;
74 }
75
76 // =====
77 // 2. SIGNAL/SLOT PATTERN (MODERN)
78 // =====
79
80 template<typename... Args>
81 class Signal {
82     private:
83         std::vector<std::function<void(Args...)>> slots;
84
85     public:
86         void connect(std::function<void(Args...)> slot) {
87             slots.push_back(std::move(slot));
88         }
89
90         void emit(Args... args) {
91             for (auto& slot : slots) {
92                 slot(args...);
93             }
94         }
95
96         void operator()(Args... args) { emit(args...); }
97     };
98
99     class Button {
100     public:
101         Signal<const std::string&> clicked;
102         Signal<int, int> positionChanged;
```

```
102     std::string id;
103
104     Button(std::string id) : id(std::move(id)) {}
105 };
106
107
108 void example_signal_slot() {
109     std::cout << "\n== 2. SIGNAL/SLOT PATTERN (MODERN) ==" << std::endl;
110
111     Button btn("SaveButton");
112
113     // Connect signals to lambdas (no slot classes needed!)
114     btn.clicked.connect([](const std::string& id) {
115         std::cout << "    Handler 1: Button " << id << " was clicked" << std::endl;
116     });
117
118     btn.clicked.connect([](const std::string& id) {
119         std::cout << "    Handler 2: Logging click on " << id << std::endl;
120     });
121
122     btn.positionChanged.connect([](int x, int y) {
123         std::cout << "    Position handler: Moved to (" << x << ", " << y << ")"
124             << std::endl;
125     });
126
127     std::cout << "\nTriggering signals:" << std::endl;
128     btn.clicked.emit("SaveButton");
129     btn.positionChanged.emit(150, 200);
130
131     std::cout << "\n ADVANTAGES:" << std::endl;
132     std::cout << " • Type-safe (template-based)" << std::endl;
133     std::cout << " • No slot base class needed" << std::endl;
134     std::cout << " • Multiple handlers per signal" << std::endl;
135     std::cout << " • Clean, declarative syntax" << std::endl;
136 }
137
138 // =====
139 // 3. EVENT DISPATCHER WITH std::variant
140 // =====
141
142 struct MouseClick { int x; int y; std::string button; };
143 struct KeyPress { char key; bool ctrl; bool shift; };
144 struct WindowResize { int width; int height; };
145
146 using UIEvent = std::variant<MouseClick, KeyPress, WindowResize>;
147
148 class EventDispatcher {
149 private:
150     std::vector<std::function<void(const UIEvent&)>> handlers;
151
152 public:
153     void subscribe(std::function<void(const UIEvent&)> handler) {
154         handlers.push_back(std::move(handler));
155     }
```

```

154 }
155
156 void dispatch(const UIEvent& event) {
157     for (auto& handler : handlers) {
158         handler(event);
159     }
160 }
161 };
162
163 void example_variant_dispatcher() {
164     std::cout << "\n==== 3. EVENT DISPATCHER WITH std::variant ===" << std::endl;
165
166     EventDispatcher dispatcher;
167
168     // Subscribe with lambda that uses std::visit
169     dispatcher.subscribe([](const UIEvent& event) {
170         std::visit([](const auto& e) {
171             using T = std::decay_t<decltype(e)>;
172
173             if constexpr (std::is_same_v<T, MouseClick>) {
174                 std::cout << "    Mouse clicked: " << e.button
175                     << " at (" << e.x << "," << e.y << ")" << std::endl;
176             } else if constexpr (std::is_same_v<T, KeyPress>) {
177                 std::cout << "    Key pressed: '" << e.key << "'"
178                     << (e.ctrl ? "(Ctrl)" : "") 
179                     << (e.shift ? "(Shift)" : "") << std::endl;
180             } else if constexpr (std::is_same_v<T, WindowResize>) {
181                 std::cout << "    Window resized: " << e.width << "x" << e.
182                     height << std::endl;
183             }
184         }, event);
185     });
186
187     // Another handler that only cares about mouse clicks
188     dispatcher.subscribe([](const UIEvent& event) {
189         if (auto* click = std::get_if<MouseClick>(&event)) {
190             std::cout << "[Logger] Mouse event at (" << click->x << "," <<
191                 click->y << ")" << std::endl;
192         }
193     });
194
195     std::cout << "\nDispatching events:" << std::endl;
196     dispatcher.dispatch(MouseClick{100, 200, "left"});
197     dispatcher.dispatch_KeyPress{'A', true, false});
198     dispatcher.dispatch(WindowResize{1920, 1080});
199
200     std::cout << "\n ADVANTAGES:" << std::endl;
201     std::cout << " • No event hierarchy needed" << std::endl;
202     std::cout << " • Type-safe with std::variant" << std::endl;
203     std::cout << " • Compile-time dispatch with std::visit" << std::endl;
204     std::cout << " • Value semantics (no pointers)" << std::endl;
205 }

```

```
205 // =====
206 // 4. CALLBACK COMPOSITION (FLUENT API)
207 // =====
208
209 class AsyncTask {
210 private:
211     std::function<void(int)> success_callback;
212     std::function<void(const std::string&)> error_callback;
213     std::function<void()> finally_callback;
214
215 public:
216     AsyncTask& on_success(std::function<void(int)> callback) {
217         success_callback = std::move(callback);
218         return *this;
219     }
220
221     AsyncTask& on_error(std::function<void(const std::string&)> callback) {
222         error_callback = std::move(callback);
223         return *this;
224     }
225
226     AsyncTask& finally(std::function<void()> callback) {
227         finally_callback = std::move(callback);
228         return *this;
229     }
230
231     void execute(bool should_fail = false) {
232         try {
233             if (should_fail) {
234                 if (error_callback) error_callback("Task failed!");
235             } else {
236                 if (success_callback) success_callback(42);
237             }
238         } catch (...) {
239             if (error_callback) error_callback("Exception occurred");
240         }
241
242         if (finally_callback) finally_callback();
243     }
244 };
245
246 void example_callback_composition() {
247     std::cout << "\n== 4. CALLBACK COMPOSITION (FLUENT API) ==" << std::endl;
248     ;
249
250     std::cout << "\n--- Success case ---" << std::endl;
251     AsyncTask task1;
252     task1.on_success([](int result) {
253         std::cout << "    Success! Result: " << result << std::endl;
254     })
255     .on_error([](const std::string& error) {
256         std::cout << "    Error: " << error << std::endl;
257     })
258     .finally([]() {
```

```
258         std::cout << "      Cleanup completed" << std::endl;
259     })
260     .execute(false);
261
262     std::cout << "\n--- Error case ---" << std::endl;
263     AsyncTask task2;
264     task2.on_success([](int result) {
265         std::cout << "      Success! Result: " << result << std::endl;
266     })
267     .on_error([](const std::string& error) {
268         std::cout << "      Error: " << error << std::endl;
269     })
270     .finally([]() {
271         std::cout << "      Cleanup completed" << std::endl;
272     })
273     .execute(true);
274
275     std::cout << "\n ADVANTAGES:" << std::endl;
276     std::cout << " • No callback interface classes" << std::endl;
277     std::cout << " • Readable, sequential setup" << std::endl;
278     std::cout << " • Similar to JavaScript Promises" << std::endl;
279     std::cout << " • Fluent API (method chaining)" << std::endl;
280 }
281
282 // =====
283 // 5. REACTIVE OBSERVABLE
284 // =====
285
286 template<typename T>
287 class Observable {
288 private:
289     T value;
290     std::vector<std::function<void(const T&)>> subscribers;
291
292 public:
293     Observable(T initial) : value(std::move(initial)) {}
294
295     void subscribe(std::function<void(const T&)> subscriber) {
296         subscriber(value); // Immediate notification
297         subscribers.push_back(std::move(subscriber));
298     }
299
300     void set(T new_value) {
301         if (value != new_value) {
302             value = std::move(new_value);
303             for (auto& subscriber : subscribers) {
304                 subscriber(value);
305             }
306         }
307     }
308
309     const T& get() const { return value; }
310 };
311
```

```
312 void example_reactive_observable() {
313     std::cout << "\n==== 5. REACTIVE OBSERVABLE ===" << std::endl;
314
315     Observable<int> counter(0);
316
317     // Subscribe with lambdas
318     counter.subscribe([](int value) {
319         std::cout << "      Subscriber 1: Counter = " << value << std::endl;
320     });
321
322     counter.subscribe([](int value) {
323         if (value > 0 && value % 2 == 0) {
324             std::cout << "      Subscriber 2: Even number " << value << std::endl;
325         }
326     });
327
328     std::cout << "\nUpdating counter:" << std::endl;
329     counter.set(1);
330     counter.set(2);
331     counter.set(3);
332     counter.set(4);
333
334     std::cout << "\n ADVANTAGES:" << std::endl;
335     std::cout << " • Automatic propagation" << std::endl;
336     std::cout << " • No observer base class" << std::endl;
337     std::cout << " • Declarative subscriptions" << std::endl;
338     std::cout << " • Great for UI data binding" << std::endl;
339 }
340
341 // =====
342 // 6. EVENT QUEUE WITH LAMBDAS
343 // =====
344
345 class EventQueue {
346 private:
347     std::queue<std::function<void()>> events;
348
349 public:
350     void post(std::function<void()> event) {
351         events.push(std::move(event));
352     }
353
354     void process_all() {
355         while (!events.empty()) {
356             auto event = std::move(events.front());
357             events.pop();
358             event();
359         }
360     }
361
362     size_t size() const { return events.size(); }
363 };
364
```

```
365 void example_event_queue() {
366     std::cout << "\n==== 6. EVENT QUEUE WITH LAMBDAS ===" << std::endl;
367
368     EventQueue queue;
369
370     // Post events as lambdas (no event classes needed!)
371     queue.post([]() {
372         std::cout << "    Event 1: Initialize" << std::endl;
373     });
374
375     queue.post([]() {
376         std::cout << "    Event 2: Load config" << std::endl;
377     });
378
379     int data = 42;
380     queue.post([data]() {
381         std::cout << "    Event 3: Process data: " << data << std::endl;
382     });
383
384     std::cout << "\nProcessing" << queue.size() << " events:" << std::endl;
385     queue.process_all();
386
387     std::cout << "\n ADVANTAGES:" << std::endl;
388     std::cout << " • No event class hierarchy" << std::endl;
389     std::cout << " • Can capture context in closure" << std::endl;
390     std::cout << " • Extremely flexible" << std::endl;
391     std::cout << " • Minimal boilerplate" << std::endl;
392 }
393
394 // =====
395 // 7. COMMAND PATTERN WITH LAMBDAS
396 // =====
397
398 class Command {
399 private:
400     std::function<void()> execute_func;
401     std::function<void()> undo_func;
402     std::string description;
403
404 public:
405     Command(std::function<void()> exec, std::function<void()> undo, std::string desc)
406         : execute_func(std::move(exec))
407         , undo_func(std::move(undo))
408         , description(std::move(desc)) {}
409
410     void execute() {
411         std::cout << "    Execute: " << description << std::endl;
412         if (execute_func) execute_func();
413     }
414
415     void undo() {
416         std::cout << "    Undo: " << description << std::endl;
417         if (undo_func) undo_func();
418 }
```

```
418     }
419 };
420
421 void example_lambda_command() {
422     std::cout << "\n==== 7. COMMAND PATTERN WITH LAMBDAS ===" << std::endl;
423
424     std::string text = "Original";
425     std::vector<Command> history;
426
427     // Create commands with lambdas (no command classes!)
428     history.emplace_back(
429         [&text]() { text = "Modified 1"; },
430         [&text]() { text = "Original"; },
431         "Change to 'Modified 1'"
432     );
433
434     history.emplace_back(
435         [&text]() { text = "Modified 2"; },
436         [&text]() { text = "Modified 1"; },
437         "Change to 'Modified 2'"
438     );
439
440     std::cout << "\nExecuting commands:" << std::endl;
441     for (auto& cmd : history) {
442         cmd.execute();
443         std::cout << "      Text: " << text << std::endl;
444     }
445
446     std::cout << "\nUndoing commands:" << std::endl;
447     for (auto it = history.rbegin(); it != history.rend(); ++it) {
448         it->undo();
449         std::cout << "      Text: " << text << std::endl;
450     }
451
452     std::cout << "\n ADVANTAGES:" << std::endl;
453     std::cout << " • No command class hierarchy" << std::endl;
454     std::cout << " • Define behavior inline" << std::endl;
455     std::cout << " • Capture state automatically" << std::endl;
456     std::cout << " • Very concise" << std::endl;
457 }
458
459 // =====
460 // MAIN FUNCTION
461 // =====
462
463 int main() {
464     std::cout << "\n
465         =====" <<
466         std::endl;
467     std::cout << "  EVENT-DRIVEN PROGRAMMING: MODERN LAMBDA-BASED APPROACH" <<
468         std::endl;
469     std::cout << "
470         =====" <<
471         std::endl;
```

```
467
468     example_lambda_observer();
469     example_signal_slot();
470     example_variant_dispatcher();
471     example_callback_composition();
472     example_reactive_observable();
473     example_event_queue();
474     example_lambda_command();
475
476     std::cout << "\n"
477     ===== " << std::endl;
478     std::cout << "  SUMMARY: WHY LAMBDAS ARE BETTER" << std::endl;
479     std::cout << "  ===== " << std::endl;
480
481     std::cout << "\n  ADVANTAGES OF LAMBDA-BASED APPROACH:" << std::endl;
482     std::cout << "\n1. LESS BOILERPLATE" << std::endl;
483     std::cout << "  •  No need to define observer/listener classes" << std::endl;
484     std::cout << "  •  No inheritance hierarchies" << std::endl;
485     std::cout << "  •  Behavior defined at point of use" << std::endl;
486
487     std::cout << "\n2. MORE FLEXIBLE" << std::endl;
488     std::cout << "  •  Can capture local state [=, [&]" << std::endl;
489     std::cout << "  •  Different behavior per subscription" << std::endl;
490     std::cout << "  •  Easy to compose and chain" << std::endl;
491
492     std::cout << "\n3. BETTER READABILITY" << std::endl;
493     std::cout << "  •  Callback logic near subscription point" << std::endl;
494     std::cout << "  •  No jumping between class definitions" << std::endl;
495     std::cout << "  •  Intent is clear and local" << std::endl;
496
497     std::cout << "\n4. TYPE SAFETY" << std::endl;
498     std::cout << "  •  std::function provides type checking" << std::endl;
499     std::cout << "  •  Template parameters enforce signatures" << std::endl;
500     std::cout << "  •  Compile-time errors for mismatches" << std::endl;
501
502     std::cout << "\n5. MODERN C++ FEATURES" << std::endl;
503     std::cout << "  •  std::function (C++11)" << std::endl;
504     std::cout << "  •  Lambda expressions (C++11)" << std::endl;
505     std::cout << "  •  std::variant (C++17)" << std::endl;
506     std::cout << "  •  if constexpr (C++17)" << std::endl;
507     std::cout << "  •  Move semantics" << std::endl;
508
509     std::cout << "\n  THINGS TO WATCH OUT FOR:" << std::endl;
510     std::cout << "  •  Lambda lifetime: Don't capture [&] if object may be
511       destroyed" << std::endl;
512     std::cout << "  •  std::function has overhead: Consider templates for hot
513       paths" << std::endl;
514     std::cout << "  •  Circular references: Can cause memory leaks" << std::endl;
515     std::cout << "  •  Check if std::function is empty before calling" << std::endl;
```

```
    ::endl;  
513  
514     std::cout << "\n WHEN TO USE INHERITANCE INSTEAD:" << std::endl;  
515     std::cout << " • Need polymorphic object identity" << std::endl;  
516     std::cout << " • Complex state management" << std::endl;  
517     std::cout << " • Multiple virtual methods needed" << std::endl;  
518     std::cout << " • Performance-critical (avoid std::function overhead)" <<  
         std::endl;  
519  
520     std::cout << "\n  
         =====\n" <<  
         std::endl;  
521  
522     return 0;  
523 }
```

## 28 Source Code: ExceptionWithSourceLocation.cpp

File: src/ExceptionWithSourceLocation.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <stdexcept>
3 #include <string>
4 #include <sstream>
5 #include <vector>
6 #include <memory>
7
8 // =====
9 // EXCEPTION WITH SOURCE LOCATION TRACKING
10 // =====
11 // Demonstrates how to capture and display source file name and line
12 // number when exceptions are thrown.
13 // Format: "ModuleName->Line #123: Error message"
14 // =====
15
16 // =====
17 // 1. BASIC EXCEPTION WITH SOURCE LOCATION
18 // =====
19
20 class SourceLocationException : public std::runtime_error {
21 private:
22     std::string source_file;
23     int source_line;
24     std::string formatted_message;
25
26     std::string format_message(const std::string& file, int line, const std::string& msg) {
27         // Extract just the filename without path
28         size_t last_slash = file.find_last_of("/\\");
29         std::string filename = (last_slash != std::string::npos)
30             ? file.substr(last_slash + 1)
31             : file;
32
33         std::ostringstream oss;
34         oss << filename << "->Line #" << line << ":" << msg;
35         return oss.str();
36     }
37
38 public:
39     SourceLocationException(const std::string& file, int line, const std::string& message)
40         : std::runtime_error(format_message(file, line, message)),
41         source_file(file),
42         source_line(line),
43         formatted_message(format_message(file, line, message)) {}
44
45     const std::string& get_source_file() const noexcept { return source_file; }
46     int get_source_line() const noexcept { return source_line; }
```

```
47     const char* what() const noexcept override { return formatted_message.  
48         c_str(); }  
49     };  
50     // Macro to automatically capture __FILE__ and __LINE__  
51     #define THROW_WITH_LOCATION(message) \  
52         throw SourceLocationException(__FILE__, __LINE__, message)  
53     void demonstrate_basic_location() {  
54         std::cout << "\n==== 1. BASIC SOURCE LOCATION TRACKING ===" << std::endl;  
55     }  
56     try {  
57         std::cout << "\nAbout to throw exception with location..." << std:::  
58             endl;  
59         THROW_WITH_LOCATION("Something went wrong!");  
60     }  
61     catch (const SourceLocationException& e) {  
62         std::cout << "Caught exception: " << e.what() << std::endl;  
63         std::cout << "  Source file: " << e.get_source_file() << std::endl;  
64         std::cout << "  Source line: " << e.get_source_line() << std::endl;  
65     }  
66 }  
67 // =====  
68 // 2. ENHANCED EXCEPTION WITH FUNCTION NAME  
69 // =====  
70  
71 class DetailedException : public std::runtime_error {  
72     private:  
73         std::string source_file;  
74         int source_line;  
75         std::string function_name;  
76         std::string error_message;  
77         std::string formatted_message;  
78  
79         std::string format_message() {  
80             size_t last_slash = source_file.find_last_of("/\\");  
81             std::string filename = (last_slash != std::string::npos)  
82                 ? source_file.substr(last_slash + 1)  
83                 : source_file;  
84  
85             std::ostringstream oss;  
86             oss << filename << "->Line #" << source_line;  
87             if (!function_name.empty()) {  
88                 oss << " [" << function_name << "]";  
89             }  
90             oss << ": " << error_message;  
91             return oss.str();  
92         }  
93     }  
94  
95     public:  
96         DetailedException(const std::string& file, int line, const std::string&  
97             function,  
98             const std::string& message)
```

```
98     : std::runtime_error(""),
99     source_file(file),
100    source_line(line),
101    function_name(function),
102    error_message(message) {
103        formatted_message = format_message();
104    }
105
106    const std::string& get_source_file() const noexcept { return source_file;
107    }
108    int get_source_line() const noexcept { return source_line; }
109    const std::string& get_function_name() const noexcept { return
110        function_name; }
111    const char* what() const noexcept override { return formatted_message.
112        c_str(); }
113
114 // Macros for different detail levels
115 #define THROW_DETAILED(message) \
116     throw DetailedException(__FILE__, __LINE__, __FUNCTION__, message)
117
118 #define THROW_DETAILED_CUSTOM(message, function) \
119     throw DetailedException(__FILE__, __LINE__, function, message)
120
121 void some_function_that_fails() {
122     THROW_DETAILED("Failed to process data");
123 }
124
125 void demonstrate_detailed_location() {
126     std::cout << "\n==== 2. DETAILED EXCEPTION WITH FUNCTION NAME ===" << std:::
127         endl;
128
129     try {
130         std::cout << "\nCalling function that will throw..." << std::endl;
131         some_function_that_fails();
132     }
133     catch (const DetailedException& e) {
134         std::cout << "Caught exception: " << e.what() << std::endl;
135         std::cout << "Module: " << e.get_source_file() << std::endl;
136         std::cout << "Line: " << e.get_source_line() << std::endl;
137         std::cout << "Function: " << e.get_function_name() << std::endl;
138     }
139 }
140
141 // =====
142 // 3. EXCEPTION HIERARCHY WITH SOURCE LOCATION
143 // =====
144
145 // Base exception with source location
146 class BaseLocationException : public std::runtime_error {
147 protected:
148     std::string source_file;
149     int source_line;
150     std::string function_name;
```



```
197 class NetworkException : public BaseLocationException {
198 public:
199     NetworkException(const std::string& file, int line, const std::string&
200                     function,
201                     const std::string& message)
202         : BaseLocationException(file, line, function, "NetworkError", message)
203     {}
204 };
205
206 class ValidationException : public BaseLocationException {
207 public:
208     ValidationException(const std::string& file, int line, const std::string&
209                     function,
210                     const std::string& message)
211         : BaseLocationException(file, line, function, "ValidationException",
212                               message)
212     {}
213
214 // Macros for specific exception types
215 #define THROW_CAMERA_ERROR(message) \
216     throw CameraException(__FILE__, __LINE__, __FUNCTION__, message)
217
218 #define THROW_NETWORK_ERROR(message) \
219     throw NetworkException(__FILE__, __LINE__, __FUNCTION__, message)
220
221 #define THROW_VALIDATION_ERROR(message) \
222     throw ValidationException(__FILE__, __LINE__, __FUNCTION__, message)
223
224 void camera_capture() {
225     THROW_CAMERA_ERROR("Failed to capture image - device not responding");
226 }
227
228 void network_send_data() {
229     THROW_NETWORK_ERROR("Connection timeout after 30 seconds");
230 }
231
232 void validate_input(int value) {
233     if (value < 0) {
234         THROW_VALIDATION_ERROR("Value must be non-negative");
235     }
236 }
237
238 void demonstrate_exception_hierarchy() {
239     std::cout << "\n== 3. EXCEPTION HIERARCHY WITH SOURCE LOCATION ==" <<
240         std::endl;
241
242     // Camera exception
243     try {
244         std::cout << "\nTesting camera exception..." << std::endl;
245         camera_capture();
246     }
247     catch (const CameraException& e) {
248         std::cout << "Caught: " << e.what() << std::endl;
249     }
250 }
```

```
246 // Network exception
247 try {
248     std::cout << "\nTesting network exception..." << std::endl;
249     network_send_data();
250 }
251 catch (const NetworkException& e) {
252     std::cout << "Caught: " << e.what() << std::endl;
253 }
254
255 // Validation exception
256 try {
257     std::cout << "\nTesting validation exception..." << std::endl;
258     validate_input(-5);
259 }
260 catch (const ValidationException& e) {
261     std::cout << "Caught: " << e.what() << std::endl;
262 }
263 }
264
265 // =====
266 // 4. STACK TRACE SIMULATION
267 // =====
268
269 class StackTraceException : public std::runtime_error {
270 private:
271     struct StackFrame {
272         std::string file;
273         int line;
274         std::string function;
275
276         std::string format() const {
277             size_t last_slash = file.find_last_of("/\\");
278             std::string filename = (last_slash != std::string::npos)
279                 ? file.substr(last_slash + 1)
280                 : file;
281             return filename + "->Line #" + std::to_string(line) + " in " +
282                 function + "()";
283         }
284     };
285
286     std::vector<StackFrame> stack_trace;
287     std::string error_message;
288
289     std::string format_full_message() const {
290         std::ostringstream oss;
291         oss << "Exception: " << error_message << "\n";
292         oss << "Stack trace:\n";
293         for (size_t i = 0; i < stack_trace.size(); ++i) {
294             oss << " # " << i << " " << stack_trace[i].format() << "\n";
295         }
296         return oss.str();
297     }
298 }
```

```
299 public:
300     StackTraceException(const std::string& message)
301         : std::runtime_error(message), error_message(message) {}
302
303     void add_frame(const std::string& file, int line, const std::string&
304         function) {
305         stack_trace.push_back({file, line, function});
306     }
307
308     const char* what() const noexcept override {
309         static std::string formatted;
310         formatted = format_full_message();
311         return formatted.c_str();
312     }
313
314     const std::vector<StackFrame>& get_stack_trace() const { return
315         stack_trace; }
316
317 // Macro to add current location to stack trace
318 #define ADD_STACK_FRAME(exception) \
319     exception.add_frame(__FILE__, __LINE__, __FUNCTION__)
320
321 void level3_function() {
322     throw StackTraceException("Critical error in deepest function");
323 }
324
325 void level2_function() {
326     try {
327         level3_function();
328     }
329     catch (StackTraceException& e) {
330         ADD_STACK_FRAME(e);
331         throw; // Rethrow
332     }
333 }
334
335 void level1_function() {
336     try {
337         level2_function();
338     }
339     catch (StackTraceException& e) {
340         ADD_STACK_FRAME(e);
341         throw; // Rethrow
342     }
343 }
344
345 void demonstrate_stack_trace() {
346     std::cout << "\n==== 4. STACK TRACE SIMULATION ===" << std::endl;
347
348     try {
349         std::cout << "\nCalling nested functions..." << std::endl;
350         level1_function();
351     }
```

```
351     catch (const StackTraceException& e) {
352         std::cout << "\nCaught exception with stack trace:" << std::endl;
353         std::cout << e.what() << std::endl;
354     }
355 }
356
357 // =====
358 // 5. C++20 std::source_location (MODERN APPROACH)
359 // =====
360
361 #if __cpp_lib_source_location >= 201907L
362 #include <source_location>
363
364 class ModernLocationException : public std::runtime_error {
365 private:
366     std::source_location location;
367     std::string error_message;
368
369     std::string format_message() const {
370         std::string filename = location.file_name();
371         size_t last_slash = filename.find_last_of("/\\");
372         if (last_slash != std::string::npos) {
373             filename = filename.substr(last_slash + 1);
374         }
375
376         std::ostringstream oss;
377         oss << filename << "->Line #" << location.line()
378             << " in " << location.function_name()
379             << ": " << error_message;
380         return oss.str();
381     }
382
383 public:
384     ModernLocationException(const std::string& message,
385                             std::source_location loc = std::source_location::
386                             current())
387         : std::runtime_error(message),
388         location(loc),
389         error_message(message) {}
390
391     const char* what() const noexcept override {
392         static std::string formatted;
393         formatted = format_message();
394         return formatted.c_str();
395     }
396
397     const std::source_location& get_location() const { return location; }
398 };
399
400 void modern_function_that_fails() {
401     throw ModernLocationException("Modern exception with automatic location
402         capture");
403 }
```

```
403 void demonstrate_modern_location() {
404     std::cout << "\n==== 5. C++20 std::source_location ===" << std::endl;
405     std::cout << "Automatic location capture without macros!" << std::endl;
406
407     try {
408         std::cout << "\nThrowing modern exception..." << std::endl;
409         modern_function_that_fails();
410     }
411     catch (const ModernLocationException& e) {
412         std::cout << "Caught: " << e.what() << std::endl;
413         auto loc = e.get_location();
414         std::cout << "  File: " << loc.file_name() << std::endl;
415         std::cout << "  Line: " << loc.line() << std::endl;
416         std::cout << "  Column: " << loc.column() << std::endl;
417         std::cout << "  Function: " << loc.function_name() << std::endl;
418     }
419 }
420 #else
421 void demonstrate_modern_location() {
422     std::cout << "\n==== 5. C++20 std::source_location ===" << std::endl;
423     std::cout << "  std::source_location not available (requires C++20)" <<
424         std::endl;
425     std::cout << "Compiler support needed: GCC 11+, Clang 16+, MSVC 2019+" <<
426         std::endl;
427 }
428 #endif
429 // =====
430 // 6. PRACTICAL EXAMPLE: IMAGE PROCESSING WITH LOCATION TRACKING
431 // =====
432 class ImageProcessingException : public std::runtime_error {
433 private:
434     std::string module;
435     int line;
436     std::string operation;
437     std::string details;
438
439 public:
440     ImageProcessingException(const std::string& file, int ln,
441                             const std::string& op, const std::string& det)
442         : std::runtime_error(""),
443         module(extract_filename(file)),
444         line(ln),
445         operation(op),
446         details(det) {}
447
448     const char* what() const noexcept override {
449         static std::string msg;
450         msg = module + "->Line #" + std::to_string(line) +
451             " [" + operation + "]: " + details;
452         return msg.c_str();
453     }
454 }
```

```
455     static std::string extract_filename(const std::string& path) {
456         size_t pos = path.find_last_of("//\\");
457         return (pos != std::string::npos) ? path.substr(pos + 1) : path;
458     }
459 };
460
461 #define THROW_IMAGE_ERROR(operation, details) \
462     throw ImageProcessingException(__FILE__, __LINE__, operation, details)
463
464 class ImageProcessor {
465 public:
466     void load_image(const std::string& filename) {
467         if (filename.empty()) {
468             THROW_IMAGE_ERROR("LoadImage", "Filename cannot be empty");
469         }
470         // Simulate file not found
471         THROW_IMAGE_ERROR("LoadImage", "File not found: " + filename);
472     }
473
474     void resize_image(int width, int height) {
475         if (width <= 0 || height <= 0) {
476             THROW_IMAGE_ERROR("ResizeImage",
477                               "Invalid dimensions: " + std::to_string(width) + "x" + std::
478                               to_string(height));
479         }
480     }
481
482     void apply_filter(const std::string& filter_name) {
483         std::vector<std::string> valid_filters = {"blur", "sharpen", "edge_detect"};
484         bool found = false;
485         for (const auto& f : valid_filters) {
486             if (f == filter_name) {
487                 found = true;
488                 break;
489             }
490         }
491         if (!found) {
492             THROW_IMAGE_ERROR("ApplyFilter", "Unknown filter: " + filter_name)
493         }
494     }
495 };
496
497 void demonstrate_practical_example() {
498     std::cout << "\n==== 6. PRACTICAL EXAMPLE: IMAGE PROCESSING ===" << std::endl;
499
500     ImageProcessor processor;
501
502     // Test 1: Load with empty filename
503     try {
504         std::cout << "\nTest 1: Loading image with empty filename..." << std::endl;
```

```
504     processor.load_image("");
505 }
506 catch (const ImageProcessingException& e) {
507     std::cout << " Error: " << e.what() << std::endl;
508 }
509
510 // Test 2: Invalid resize dimensions
511 try {
512     std::cout << "\nTest 2: Resizing with invalid dimensions..." << std::endl;
513     processor.resize_image(-100, 200);
514 }
515 catch (const ImageProcessingException& e) {
516     std::cout << " Error: " << e.what() << std::endl;
517 }
518
519 // Test 3: Unknown filter
520 try {
521     std::cout << "\nTest 3: Applying unknown filter..." << std::endl;
522     processor.apply_filter("invalid_filter");
523 }
524 catch (const ImageProcessingException& e) {
525     std::cout << " Error: " << e.what() << std::endl;
526 }
527
528 // Test 4: File not found
529 try {
530     std::cout << "\nTest 4: Loading non-existent file..." << std::endl;
531     processor.load_image("nonexistent_image.png");
532 }
533 catch (const ImageProcessingException& e) {
534     std::cout << " Error: " << e.what() << std::endl;
535 }
536 }
537
538 // =====
539 // 7. DEBUGGING HELPER: EXCEPTION LOGGING
540 // =====
541
542 class LoggedException : public std::runtime_error {
543 private:
544     std::string module;
545     int line;
546     std::string function;
547     std::string message;
548
549 public:
550     LoggedException(const std::string& file, int ln, const std::string& func,
551                     const std::string& msg)
552         : std::runtime_error(msg),
553           module(extract_filename(file)),
554           line(ln),
555           function(func),
556           message(msg) {
```

```
557     log_exception();
558 }
559
560 void log_exception() const {
561     std::cerr << "\n"                                     n" << std::endl;
562     std::cerr << " EXCEPTION THROWN" << std::endl;
563     std::cerr << "                                     " << std::endl;
564     std::cerr << " Module:    " << module << std::endl;
565     std::cerr << " Line:      #" << line << std::endl;
566     std::cerr << " Function:   " << function << "()" << std::endl;
567     std::cerr << " Message:    " << message << std::endl;
568     std::cerr << "                                     " << std::endl;
569 }
570
571 const char* what() const noexcept override {
572     static std::string formatted;
573     formatted = module + "->Line #" + std::to_string(line) + ": " +
574         message;
575     return formatted.c_str();
576 }
577
578 static std::string extract_filename(const std::string& path) {
579     size_t pos = path.find_last_of("/\\");
580     return (pos != std::string::npos) ? path.substr(pos + 1) : path;
581 }
582
583 #define THROW_AND_LOG(message) \
584     throw LoggedException(__FILE__, __LINE__, __FUNCTION__, message)
585
586 void function_with_logging() {
587     THROW_AND_LOG("Critical error detected - system state corrupted");
588 }
589
590 void demonstrate_logging() {
591     std::cout << "\n== 7. EXCEPTION LOGGING ==" << std::endl;
592
593     try {
594         std::cout << "\nCalling function that logs exception details..." <<
595             std::endl;
596         function_with_logging();
597     }
598     catch (const LoggedException& e) {
599         std::cout << "\nException caught in main handler" << std::endl;
600         std::cout << "Formatted: " << e.what() << std::endl;
601     }
602 }
603 // =====
604 // MAIN FUNCTION
605 // =====
606
607 int main() {
608     std::cout << "\n
```

```
=====
  std::endl;
609 std::cout << "  EXCEPTION SOURCE LOCATION TRACKING" << std::endl;
610 std::cout << "
=====
  std::endl;
611 std::cout << "Format: ModuleName->Line #XXX: Error message" << std::endl;
612
613 demonstrate_basic_location();
614 demonstrate_detailed_location();
615 demonstrate_exception_hierarchy();
616 demonstrate_stack_trace();
617 demonstrate_modern_location();
618 demonstrate_practical_example();
619 demonstrate_logging();
620
621 std::cout << "\n
=====
  std::endl;
622 std::cout << "  IMPLEMENTATION TECHNIQUES" << std::endl;
623 std::cout << "
=====
  std::endl;
624
625 std::cout << "\n1  PREPROCESSOR MACROS (C++11/14/17):" << std::endl;
626 std::cout << "  __FILE__      - Source file path" << std::endl;
627 std::cout << "  __LINE__      - Line number" << std::endl;
628 std::cout << "  __FUNCTION__  - Function name (compiler extension)" << std
       ::endl;
629 std::cout << "\n  Usage:" << std::endl;
630 std::cout << "#define THROW_WITH_LOC(msg) \\\" << std::endl;
631 std::cout << "          throw MyException(__FILE__, __LINE__, msg)" << std::
       endl;
632
633 std::cout << "\n2  std::source_location (C++20):" << std::endl;
634 std::cout << "  •  Automatic location capture" << std::endl;
635 std::cout << "  •  No macros needed" << std::endl;
636 std::cout << "  •  Default parameter trick" << std::endl;
637 std::cout << "\n  Usage:" << std::endl;
638 std::cout << "  throw MyException(message, std::source_location::current
       ());" << std::endl;
639
640 std::cout << "\n3  CUSTOM EXCEPTION CLASS:" << std::endl;
641 std::cout << "  class MyException : public std::runtime_error {" << std::
       endl;
642 std::cout << "      std::string file;" << std::endl;
643 std::cout << "      int line;" << std::endl;
644 std::cout << "  public:" << std::endl;
645 std::cout << "      MyException(const string& f, int l, const string& msg
       );" << std::endl;
646 std::cout << "      const char* what() const noexcept override;" << std::
       endl;
647 std::cout << "  };" << std::endl;
648
```

```
649 std::cout << "\n4 MESSAGE FORMATTING:" << std::endl;
650 std::cout << "    Format: \"ModuleName->Line #123: Error message\"" << std
651     ::endl;
652 std::cout << " • Extract filename from path" << std::endl;
653 std::cout << " • Use std::ostringstream for formatting" << std::endl;
654 std::cout << " • Store formatted message in exception" << std::endl;
655
656 std::cout << "\n
657     ====="
658 std::endl;
659 std::cout << "    BEST PRACTICES" << std::endl;
660 std::cout << "
661     ====="
662 std::endl;
663 std::cout << "\n DO:" << std::endl;
664 std::cout << " • Use macros to automatically capture location" << std::
665     endl;
666 std::cout << " • Extract filename only (not full path)" << std::endl;
667 std::cout << " • Include function name for better context" << std::endl;
668 std::cout << " • Format consistently: Module->Line #XXX" << std::endl;
669 std::cout << " • Store location info in exception members" << std::endl;
670 std::cout << " • Use std::source_location in C++20 (no macros!)" << std
671     ::endl;
672
673 std::cout << "\n DON'T:" << std::endl;
674 std::cout << " • Don't include full file paths (too verbose)" << std::
675     endl;
676 std::cout << " • Don't hardcode file/line (will be wrong)" << std::endl;
677 std::cout << " • Don't forget to extract filename from path" << std::
678     endl;
679 std::cout << " • Don't use exceptions for control flow" << std::endl;
680
681 std::cout << "\n COMPARISON:" << std::endl;
682 std::cout << "\nMacro Approach (C++11-17):" << std::endl;
683 std::cout << "    Works with older compilers" << std::endl;
684 std::cout << "    Full control over formatting" << std::endl;
685 std::cout << "    Requires macros (ugly)" << std::endl;
686 std::cout << "    Manual __FILE__, __LINE__ passing" << std::endl;
687
688 std::cout << "\nstd::source_location (C++20):" << std::endl;
689 std::cout << "    No macros needed" << std::endl;
690 std::cout << "    Automatic capture via default parameter" << std::endl;
691 std::cout << "    Type-safe and elegant" << std::endl;
692 std::cout << "    Requires C++20 compiler support" << std::endl;
693
694 std::cout << "\n REAL-WORLD USAGE:" << std::endl;
695 std::cout << "\nDefensive Programming:" << std::endl;
696 std::cout << "    if (!validate(data)) {" << std::endl;
697 std::cout << "        THROW_WITH_LOCATION(\"Invalid data format\");" << std
698     ::endl;
699 std::cout << "    }" << std::endl;
700
701 std::cout << "\nResource Management:" << std::endl;
```

```
693     std::cout << "    if (!file.open()) {" << std::endl;
694     std::cout << "        THROW_WITH_LOCATION(\"Failed to open: \\" + filename);
695     std::cout << "    }" << std::endl;
696
697     std::cout << "\nAPI Boundary Checking:" << std::endl;
698     std::cout << "    if (index >= size()) {" << std::endl;
699     std::cout << "        THROW_WITH_LOCATION(\"Index out of range\");" << std
700         ::endl;
700     std::cout << "    }" << std::endl;
701
702     std::cout << "\n
702         =====\n" << std::endl;
703
704     return 0;
705 }
```

## 29 Source Code: FindCountOfCommonNodes.cpp

File: src/FindCountOfCommonNodes.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <array>
3 #include <string>
4
5 using namespace std;
6
7 struct ListNode {
8     int val;
9     ListNode *next;
10    ListNode(int x = 0) : val(x), next(NULL) {}
11 };
12
13 class Solution {
14 public:
15     void PrintList(ListNode *pListNode) {
16         if (pListNode != NULL) {
17             cout << pListNode->val << endl;
18             PrintList(pListNode->next);
19         } else {
20             cout << "NULL\n" << endl;
21         }
22     }
23
24     int getLengthOfList(const ListNode *pListNode) {
25         int length = 0;
26
27         while (pListNode != NULL) {
28             pListNode = pListNode->next;
29             length++;
30         }
31
32         return length;
33     }
34
35     int FindNumberOfCommonNodes(ListNode *pHead1, ListNode *pHead2) {
36
37         ListNode* current1 = pHead1;
38         ListNode* current2 = pHead2;
39
40         int count = 0;
41
42         // traverse list A till the end of list
43         while (current1 != NULL) {
44
45             // traverse list B till the end of list
46             while (current2 != NULL) {
47
48                 // if data is match then count increase
49                 if (current1->val == current2->val) {
```

```
50         count++;
51     }
52
53     // increase current pointer for next node
54     current2 = current2->next;
55 }
56
57 // increase current pointer of first list
58 current1 = current1->next;
59
60 // initialize starting point for second list
61 current2 = pHead2;
62 }
63
64 return count;
65 }
66 };
67
68 int main () {
69     Solution testSolution;
70     std::array<int, 12> intArray1 {{1, 3, 4, 6, 7, 10, 12, 13, 14, 15, 16,
71     17}};
72     std::array<int, 10> intArray2 = {2, 4, 5, 6, 7, 8, 11, 13, 14, 16};
73
74     ListNode *pMHead = NULL;
75     ListNode *pCurrent = NULL;
76     ListNode *pNHead = NULL;
77     auto i = 0;
78
79     for (i = 0; i < intArray1.size(); i++) {
80         ListNode *pTemp = new ListNode(intArray1[i]);
81
82         if (i == 0) {
83             pMHead = pCurrent = pTemp;
84         } else {
85             pCurrent->next = pTemp;
86             pCurrent = pCurrent->next; // pCurrent->next == pTemp
87         }
88     }
89
90     for (i = 0; i < intArray2.size(); i++) {
91         ListNode *pTemp = new ListNode(intArray2[i]);
92
93         if (i == 0) {
94             pNHead = pCurrent = pTemp;
95         } else {
96             pCurrent->next = pTemp;
97             pCurrent = pCurrent->next; // pCurrent->next == pTemp
98         }
99
100    // Print Linkedlist
101    cout << "PrintList(pMHead)" << endl;
102    testSolution.PrintList(pMHead);
```

```
103     cout << "PrintList(pNHead)" << endl;
104     testSolution.PrintList(pNHead);
105
106     int commonNodesCount = testSolution.FindNumberOfCommonNodes(pMHead, pNHead
107         );
108
109
110     cout << "commonNodesCount = " << commonNodesCount << endl;
111
112
113     return 0;
114 }
```

## 30 Source Code: FindFirstCommonNode.cpp

File: src/FindFirstCommonNode.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <array>
3 #include <string>
4
5 using namespace std;
6
7 struct ListNode {
8     int val;
9     ListNode *next;
10    ListNode(int x = 0) : val(x), next(NULL) {}
11 };
12
13 class Solution {
14 public:
15     void PrintList(ListNode *pListNode) {
16         if (pListNode != NULL) {
17             cout << pListNode->val << endl;
18             PrintList(pListNode->next);
19         } else {
20             cout << "NULL\n" << endl;
21         }
22     }
23
24     int getLengthOfList(const ListNode *pListNode) {
25         int length = 0;
26
27         while (pListNode != NULL) {
28             pListNode = pListNode->next;
29             length++;
30         }
31
32         return length;
33     }
34
35     ListNode* FindFirstCommonNode(ListNode *pHead1, ListNode *pHead2) {
36
37         ListNode* current1 = pHead1;
38         ListNode* current2 = pHead2;
39
40         // traverse list A till the end of list
41         while (current1 != NULL) {
42
43             // traverse list B till the end of list
44             while (current2 != NULL) {
45
46                 // if data is match then count increase
47                 if (current1->val == current2->val) {
48
49                     return current1;
50                 }
51             }
52         }
53     }
54 }
```

```
50     }
51
52     // increase current pointer for next node
53     current2 = current2->next;
54 }
55
56 // increase current pointer of first list
57 current1 = current1->next;
58
59 // initialize starting point for second list
60 current2 = pHead2;
61 }
62 }
63 };
64
65 int main () {
66     Solution testSolution;
67     std::array<int, 12> intArray1 {{1, 3, 4, 6, 7, 10, 12, 13, 14, 15, 16,
68     17}};
69     std::array<int, 10> intArray2 = {2, 4, 5, 6, 7, 8, 11, 13, 14, 16};
70
71     ListNode *pMHead = NULL;
72     ListNode *pCurrent = NULL;
73     ListNode *pNHead = NULL;
74     auto i = 0;
75
76     for (i = 0; i < intArray1.size(); i++) {
77         ListNode *pTemp = new ListNode(intArray1[i]);
78
79         if (i == 0) {
80             pMHead = pCurrent = pTemp;
81         } else {
82             pCurrent->next = pTemp;
83             pCurrent = pCurrent->next; // pCurrent->next == pTemp
84         }
85     }
86
87     for (i = 0; i < intArray2.size(); i++) {
88         ListNode *pTemp = new ListNode(intArray2[i]);
89
90         if (i == 0) {
91             pNHead = pCurrent = pTemp;
92         } else {
93             pCurrent->next = pTemp;
94             pCurrent = pCurrent->next; // pCurrent->next == pTemp
95         }
96
97     // Print Linkedlist
98     cout << "PrintList(pMHead)" << endl;
99     testSolution.PrintList(pMHead);
100
101    cout << "PrintList(pNHead)" << endl;
102    testSolution.PrintList(pNHead);
```

```
103     ListNode *pTarget = testSolution.FindFirstCommonNode(pMHead, pNHead);  
104  
105     // Print Returned Node  
106     if (pTarget != NULL)  
107         cout << "pTarget->val = " << pTarget->val << endl;  
108     cout << endl;  
109  
110     return 0;  
111 }  
112 }
```

## 31 Source Code: FindMToLastElement.cpp

File: src/FindMToLastElement.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <array>
3 #include <string>
4
5 using namespace std;
6
7 struct ListNode {
8     int val;
9     ListNode *next;
10    ListNode(int x = 0) : val(x), next(NULL) {}
11 };
12
13 class Solution {
14 public:
15     void PrintList(ListNode *pListNode) {
16         if (pListNode != NULL) {
17             cout << pListNode->val << endl;
18             PrintList(pListNode->next);
19         } else {
20             cout << "NULL\n" << endl;
21         }
22     }
23
24     int getLengthOfList(const ListNode *pListNode) {
25         int length = 0;
26
27         while (pListNode != NULL) {
28             pListNode = pListNode->next;
29             length++;
30         }
31
32         return length;
33     }
34
35     ListNode* FindMthToLastElement(ListNode *pHead1, int valueM) {
36
37         ListNode* current = pHead1;
38
39         // traverse list A till the end of list
40         while (current != NULL) {
41
42             // if data is match then count increase
43             if (current->val == valueM) {
44                 return current;
45             }
46
47             // increase current pointer of first list
48             current = current->next;
49         }
50     }
51 }
```

```
50     }
51
52     return NULL;
53 }
54 };
55
56 int main () {
57     Solution testSolution;
58     std::array<int, 12> intArray1 {{1, 3, 4, 6, 7, 10, 12, 13, 14, 15, 16,
59     17}};
60
61     ListNode *pMHead = NULL;
62     ListNode *pCurrent = NULL;
63     ListNode *pNHead = NULL;
64     auto i = 0;
65
66     for (i = 0; i < intArray1.size(); i++) {
67         ListNode *pTemp = new ListNode(intArray1[i]);
68
69         if (i == 0) {
70             pMHead = pCurrent = pTemp;
71         } else {
72             pCurrent->next = pTemp;
73             pCurrent = pCurrent->next; // pCurrent->next == pTemp
74         }
75
76         // Print Linkedlist
77         cout << "PrintList(pMHead)" << endl;
78         testSolution.PrintList(pMHead);
79
80         int valueOfM = 4;
81
82         ListNode *pTarget = testSolution.FindMthToLastElement(pMHead, valueOfM);
83
84         // Print Returned Node
85         if (pTarget != NULL) {
86             testSolution.PrintList(pMHead);
87         } else {
88             cout << "There is no element that has value equal to " << valueOfM <<
89             endl;
90         }
91
92         valueOfM = 5;
93
94         pTarget = testSolution.FindMthToLastElement(pMHead, valueOfM);
95
96         // Print Returned Node
97         if (pTarget != NULL) {
98             testSolution.PrintList(pMHead);
99         } else {
100            cout << "There is no element that has value equal to " << valueOfM <<
101            endl;
102        }
103    }
104 }
```

```
101
102
103     return 0;
104 }
```

## 32 Source Code: FindMaxNumberOfConsecutiveOnesFromIntArray.cpp

File: src/FindMaxNumberOfConsecutiveOnesFromIntArray.cpp

Repository: [View on GitHub](#)

```
1  /*
2  Instructions:
3
4  Write a function that returns the maximum number of consecutive 'ones' in an
5  integer array of length N.
6
7  You can assume the input is an array containing integers 0 and 1 only.
8
9  For example, for the following input the function should return 7.
10
11
12 This can be written in pseudo-code, Java, or C/C++. Syntax mistakes are not
13  important.
14 */
15
16 // Example program
17 #include <iostream>
18 #include <string>
19 #include <array>
20 #include <algorithm>
21
22 int main()
23 {
24     int cArray[] =
25         {0,1,0,0,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,0};
26
27     const int sizeArray = sizeof(cArray)/sizeof(cArray[0]);
28
29     std::array<int, sizeArray> cppArray;
30
31     std::move(std::begin(cArray), std::end(cArray), cppArray.begin());
32
33     auto maxiConsecutiveOnes = 0;
34     auto runningMaximumOnes= 0;
35
36     for(const auto& i : cppArray) {
37
38         if(i == 1) { // Fixed: use i directly, not cppArray[i]
39             runningMaximumOnes++;
40         } else {
41
42             if(maxiConsecutiveOnes < runningMaximumOnes) {
43                 maxiConsecutiveOnes = runningMaximumOnes;
44             }
45             runningMaximumOnes = 0;
46         }
47     }
48 }
```

```
46
47     std::cout << " maximum consecutive ones = " << maxiConsecutiveOnes << std::
48     endl;
}
```

### 33 Source Code: FragileBaseClass.cpp

File: src/FragileBaseClass.cpp

Repository: [View on GitHub](#)

```
1 // FragileBaseClass.cpp
2 // Demonstrates the Fragile Base Class Problem in C++ and its solutions
3 //
4 // The Fragile Base Class problem occurs when changes to a base class,
5 // which seem safe in isolation, inadvertently break the functionality
6 // of its derived (child) classes.
7 //
8 // KEY CONCEPTS:
9 // 1. Changes in Method Calls - base class refactoring breaks derived classes
10 // 2. Altering Data Layout - ABI compatibility issues
11 // 3. Virtual Function Overriding - unexpected behavior changes
12 //
13 // SOLUTIONS:
14 // 1. Composition over Inheritance
15 // 2. Hide Implementation Details (Encapsulation)
16 // 3. Use 'final' keyword
17 // 4. Design for Inheritance Explicitly
18 // 5. Prefer Interfaces/Abstract Classes
19
20 #include <iostream>
21 #include <string>
22 #include <vector>
23 #include <memory>
24
25 // =====
26 // SECTION 1: THE PROBLEM - Fragile Base Class
27 // =====
28
29 namespace fragile_example {
30
31 // Version 1: Original base class
32 class Counter_V1 {
33 protected:
34     int count_ = 0;
35
36 public:
37     virtual void add(int value) {
38         count_ += value;
39         std::cout << "    [Base] Added " << value << ", count = " << count_ <<
40         "\n";
41     }
42
43     void addMultiple(const std::vector<int>& values) {
44         for (int val : values) {
45             count_ += val; // Direct implementation
46         }
47     }
48 }
```

```
45     }
46     std::cout << "      [Base] Added multiple values, count = " << count_ <<
47     "\n";
48 }
49 int getCount() const { return count_; }
50 };
51
52 // Derived class works fine with V1
53 class LoggingCounter_V1 : public Counter_V1 {
54 private:
55     std::vector<std::string> log_;
56
57 public:
58     void add(int value) override {
59         log_.push_back("Adding " + std::to_string(value));
60         Counter_V1::add(value);
61     }
62
63     void printLog() const {
64         std::cout << "      Log entries: " << log_.size() << "\n";
65         for (const auto& entry : log_) {
66             std::cout << "      - " << entry << "\n";
67         }
68     }
69 };
70
71 // =====
72 // Now developer "optimizes" the base class by refactoring
73 // =====
74
75 class Counter_V2 {
76 protected:
77     int count_ = 0;
78
79 public:
80     virtual void add(int value) {
81         count_ += value;
82         std::cout << "      [Base] Added " << value << ", count = " << count_ <<
83         "\n";
84     }
85
86     // FRAGILE CHANGE: Refactored to use add() for "code reuse"
87     void addMultiple(const std::vector<int>& values) {
88         for (int val : values) {
89             add(val); // Now calls virtual add()!
90         }
91         std::cout << "      [Base] Added multiple values, count = " << count_ <<
92         "\n";
93     }
94 }
```

```
92     int getCount() const { return count_; }
93 };
94
95 // Same derived class, now BROKEN!
96 class LoggingCounter_V2 : public Counter_V2 {
97 private:
98     std::vector<std::string> log_;
99
100 public:
101     void add(int value) override {
102         log_.push_back("Adding " + std::to_string(value));
103         Counter_V2::add(value);
104     }
105
106     void printLog() const {
107         std::cout << "    Log entries: " << log_.size() << "\n";
108         for (const auto& entry : log_) {
109             std::cout << "        - " << entry << "\n";
110         }
111     }
112 }
113
114 void demonstrate() {
115     std::cout << "\n" << std::string(70, '=') << "\n";
116     std::cout << "==== SECTION 1: The Fragile Base Class Problem ===\n";
117     std::cout << std::string(70, '=') << "\n\n";
118
119     std::cout << "Scenario: Developer refactors base class to call virtual
120         method\n\n";
121
122     std::cout << "1. Original Version (V1) - Works as expected:\n";
123     LoggingCounter_V1 counter1;
124     counter1.addMultiple({1, 2, 3});
125     counter1.printLog();
126     std::cout << "    Expected: No log entries (addMultiple bypasses add())\n";
127     std::cout << "    Actual: " << (counter1.printLog(), " Works!\n");
128
129     std::cout << "\n2. After Refactoring (V2) - BROKEN:\n";
130     LoggingCounter_V2 counter2;
131     counter2.addMultiple({1, 2, 3});
132     counter2.printLog();
133     std::cout << "    Problem: Now creates log entries unexpectedly!\n";
134     std::cout << "    The derived class behavior changed without any
135         modification!\n";
136 }
137 } // namespace fragile_example
138
139 // =====
140 // SECTION 2: SOLUTION 1 - Composition Over Inheritance
141 //
```

```
=====
142
143 namespace composition_solution {
144
145 // Internal implementation (not exposed for inheritance)
146 class CounterImpl {
147 private:
148     int count_ = 0;
149
150 public:
151     void add(int value) {
152         count_ += value;
153     }
154
155     void addMultiple(const std::vector<int>& values) {
156         for (int val : values) {
157             add(val); // Safe - not virtual
158         }
159     }
160
161     int getCount() const { return count_; }
162 };
163
164 // Use composition instead of inheritance
165 class LoggingCounter {
166 private:
167     CounterImpl counter_; // Composition
168     std::vector<std::string> log_;
169
170 public:
171     void add(int value) {
172         log_.push_back("Adding " + std::to_string(value));
173         counter_.add(value);
174     }
175
176     void addMultiple(const std::vector<int>& values) {
177         for (int val : values) {
178             counter_.add(val); // Direct delegation, no logging
179         }
180     }
181
182     int getCount() const { return counter_.getCount(); }
183
184     void printLog() const {
185         std::cout << "    Log entries: " << log_.size() << "\n";
186     }
187 };
188
189 void demonstrate() {
190     std::cout << "\n" << std::string(70, '=') << "\n";
191     std::cout << "==== SECTION 2: Solution 1 - Composition Over Inheritance
192         ===\n";
193     std::cout << std::string(70, '=') << "\n\n";
```

```
193     std::cout << "  Using composition - behavior is explicit and controlled\n"
194     ;
195     LoggingCounter counter;
196     counter.add(5);
197     counter.addMultiple({1, 2, 3});
198     std::cout << "  Count: " << counter.getCount() << "\n";
199     counter.printLog();
200     std::cout << "  Benefit: No fragility - implementation changes don't break
201     us\n";
202 }
203 } // namespace composition_solution
204
205 // =====
206 // SECTION 3: SOLUTION 2 - Hide Implementation Details
207 // =====
208
209 namespace encapsulation_solution {
210
211 class Counter {
212 private:
213     int count_ = 0;
214
215     // Private helper - derived classes cannot rely on this
216     void incrementInternal(int value) {
217         count_ += value;
218     }
219
220 protected:
221     // Protected interface for derived classes
222     void setCount(int value) { count_ = value; }
223     int getCountInternal() const { return count_; }
224
225 public:
226     virtual void add(int value) {
227         incrementInternal(value);
228     }
229
230     // Non-virtual - cannot be overridden
231     void addMultiple(const std::vector<int>& values) {
232         for (int val : values) {
233             incrementInternal(val); // Calls private method
234         }
235     }
236
237     int getCount() const { return count_; }
238 };
239
240 class LoggingCounter : public Counter {
```

```
241 private:
242     std::vector<std::string> log_;
243
244 public:
245     void add(int value) override {
246         log_.push_back("Adding " + std::to_string(value));
247         Counter::add(value);
248     }
249
250     void printLog() const {
251         std::cout << "    Log entries: " << log_.size() << "\n";
252     }
253 };
254
255 void demonstrate() {
256     std::cout << "\n" << std::string(70, '=') << "\n";
257     std::cout << "==== SECTION 3: Solution 2 - Hide Implementation Details ===\
258         n";
259     std::cout << std::string(70, '=') << "\n\n";
260
261     std::cout << "  Private implementation, public/protected interface only\n"
262         ;
263     LoggingCounter counter;
264     counter.add(5);
265     counter.addMultiple({1, 2, 3});
266     std::cout << "  Count: " << counter.getCount() << "\n";
267     counter.printLog();
268     std::cout << "  Benefit: addMultiple() is non-virtual and uses private
269         method\n";
270 }
271
272 } // namespace encapsulation_solution
273
274 // SECTION 4: SOLUTION 3 - Use 'final' Keyword
275 // =====
276
277 namespace final_solution {
278
279 // Concrete class marked as final - cannot be inherited
280 class Counter final {
281     private:
282         int count_ = 0;
283
284     public:
285         void add(int value) {
286             count_ += value;
287         }
288
289         void addMultiple(const std::vector<int>& values) {
```

```
288     for (int val : values) {
289         add(val);
290     }
291 }
292
293     int getCount() const { return count_; }
294 };
295
296 // Base class with final methods
297 class ExtensibleCounter {
298 protected:
299     int count_ = 0;
300
301 public:
302     virtual ~ExtensibleCounter() = default;
303
304     // This method can be overridden
305     virtual void add(int value) {
306         count_ += value;
307     }
308
309     // This method CANNOT be overridden - marked final
310     virtual void addMultiple(const std::vector<int>& values) final {
311         for (int val : values) {
312             add(val);
313         }
314     }
315
316     int getCount() const { return count_; }
317 };
318
319 class LoggingCounter : public ExtensibleCounter {
320 private:
321     std::vector<std::string> log_;
322
323 public:
324     void add(int value) override {
325         log_.push_back("Adding " + std::to_string(value));
326         ExtensibleCounter::add(value);
327     }
328
329     // Cannot override addMultiple() - it's final
330     // void addMultiple(...) {} // ERROR!
331
332     void printLog() const {
333         std::cout << "    Log entries: " << log_.size() << "\n";
334     }
335 };
336
337 void demonstrate() {
338     std::cout << "\n" << std::string(70, '=') << "\n";
339     std::cout << "==== SECTION 4: Solution 3 - Use 'final' Keyword ===\n";
340     std::cout << std::string(70, '=') << "\n\n";
341 }
```

```
342     std::cout << " Using 'final' to prevent inheritance or method override\n\
343     n";
344
345     std::cout << "1. Final class (Counter) - cannot be inherited\n";
346     Counter counter1;
347     counter1.add(5);
348     std::cout << "  Count: " << counter1.getCount() << "\n";
349     // class Derived : public Counter { }; // ERROR: cannot inherit from final
350     // class
351
352     std::cout << "\n2. Final method (addMultiple) - cannot be overridden\n";
353     LoggingCounter counter2;
354     counter2.add(5);
355     counter2.addMultiple({1, 2, 3});
356     std::cout << "  Count: " << counter2.getCount() << "\n";
357     counter2.printLog();
358     std::cout << "  Benefit: Base class controls critical behavior\n";
359 }
360
361 // =====
362 // SECTION 5: SOLUTION 4 - Template Method Pattern
363 // =====
364
365 namespace template_method_solution {
366
367 // Base class designed for inheritance with clear extension points
368 class Counter {
369 protected:
370     int count_ = 0;
371
372     // Hook for derived classes - well-documented extension point
373     virtual void onBeforeAdd([[maybe_unused]] int value) {
374         // Default: do nothing
375     }
376
377     virtual void onAfterAdd([[maybe_unused]] int value) {
378         // Default: do nothing
379     }
380
381 private:
382     // Template method - defines the algorithm structure
383     void addImpl(int value) {
384         onBeforeAdd(value);
385         count_ += value;
386         onAfterAdd(value);
387     }
388
389 public:
```

```
390     virtual ~Counter() = default;
391
392     // Public interface - calls template method
393     void add(int value) {
394         addImpl(value);
395     }
396
397     // Non-virtual - uses template method internally
398     virtual void addMultiple(const std::vector<int>& values) final {
399         for (int val : values) {
400             addImpl(val); // Each add goes through hooks
401         }
402     }
403
404     int getCount() const { return count_; }
405 };
406
407 class LoggingCounter : public Counter {
408 private:
409     std::vector<std::string> log_;
410
411 protected:
412     void onBeforeAdd(int value) override {
413         log_.push_back("Adding " + std::to_string(value));
414     }
415
416 public:
417     void printLog() const {
418         std::cout << "    Log entries: " << log_.size() << "\n";
419     }
420 };
421
422 void demonstrate() {
423     std::cout << "\n" << std::string(70, '=') << "\n";
424     std::cout << "==== SECTION 5: Solution 4 - Template Method Pattern ===\n";
425     std::cout << std::string(70, '=') << "\n\n";
426
427     std::cout << " Clear extension points with template method pattern\n";
428     LoggingCounter counter;
429     counter.add(5);
430     counter.addMultiple({1, 2, 3});
431     std::cout << " Count: " << counter.getCount() << "\n";
432     counter.printLog();
433     std::cout << " Benefit: Hooks are called for both add() and addMultiple()
434         \n";
435     std::cout << " Base class controls algorithm, derived class controls
436         hooks\n";
437 }
438
439 } // namespace template_method_solution
440
441 // =====
```

```
440 // SECTION 6: SOLUTION 5 - Prefer Interfaces (Pure Abstract Classes)
441 //
442 =====
443
443 namespace interface_solution {
444
445 // Pure interface - no implementation details to break
446 class ICounter {
447 public:
448     virtual ~ICounter() = default;
449     virtual void add(int value) = 0;
450     virtual void addMultiple(const std::vector<int>& values) = 0;
451     virtual int getCount() const = 0;
452 };
453
454 // Concrete implementation
455 class BasicCounter : public ICounter {
456 private:
457     int count_ = 0;
458
459 public:
460     void add(int value) override {
461         count_ += value;
462     }
463
464     void addMultiple(const std::vector<int>& values) override {
465         for (int val : values) {
466             count_ += val; // Direct implementation
467         }
468     }
469
470     int getCount() const override { return count_; }
471 };
472
473 // Logging decorator using composition
474 class LoggingCounter : public ICounter {
475 private:
476     std::unique_ptr<ICounter> wrapped_;
477     std::vector<std::string> log_;
478
479 public:
480     explicit LoggingCounter(std::unique_ptr<ICounter> counter)
481         : wrapped_(std::move(counter)) {}
482
483     void add(int value) override {
484         log_.push_back("Adding " + std::to_string(value));
485         wrapped_->add(value);
486     }
487
488     void addMultiple(const std::vector<int>& values) override {
489         // No logging for batch operations
490         wrapped_->addMultiple(values);
491     }

```

```
492     int getCount() const override { return wrapped_->getCount(); }
493
494     void printLog() const {
495         std::cout << "    Log entries: " << log_.size() << "\n";
496     }
497 }
498
499
500 void demonstrate() {
501     std::cout << "\n" << std::string(70, '=') << "\n";
502     std::cout << "==== SECTION 6: Solution 5 - Prefer Interfaces ===\n";
503     std::cout << std::string(70, '=') << "\n\n";
504
505     std::cout << "  Pure interface + composition (Decorator pattern)\n";
506     auto counter = std::make_unique<LoggingCounter>(
507         std::make_unique<BasicCounter>()
508     );
509     counter->add(5);
510     counter->addMultiple({1, 2, 3});
511     std::cout << "  Count: " << counter->getCount() << "\n";
512     counter->printLog();
513     std::cout << "  Benefit: No implementation inheritance - no fragility\n";
514 }
515
516 } // namespace interface_solution
517
518 // =====
519 // SECTION 7: ABI Stability Example
520 // =====
521
522 namespace abi_example {
523
524 // Version 1: Library shipped to customers
525 struct Device_V1 {
526     int id;
527     std::string name;
528
529     virtual ~Device_V1() = default;
530     virtual void process() { std::cout << "    Processing " << name << "\n"; }
531 };
532
533 struct Sensor_V1 : public Device_V1 {
534     double reading;
535
536     void process() override {
537         std::cout << "    Sensor " << name << " reading: " << reading << "\n";
538     }
539 };
540
541 // Version 2: Developer adds a field (BREAKS ABI)
```

```
542 struct Device_V2 {
543     int id;
544     std::string name;
545     bool enabled; // NEW FIELD - changes memory layout!
546
547     virtual ~Device_V2() = default;
548     virtual void process() {
549         if (enabled) std::cout << "    Processing " << name << "\n";
550     }
551 };
552
553 // Customer code (compiled against V1) now has wrong offsets!
554
555 void demonstrate() {
556     std::cout << "\n" << std::string(70, '=') << "\n";
557     std::cout << "==== SECTION 7: ABI Stability and Memory Layout ===\n";
558     std::cout << std::string(70, '=') << "\n\n";
559
560     std::cout << "Problem: Adding fields to base class changes memory layout\n"
561         "\n";
562
563     std::cout << "Device_V1 layout:\n";
564     Device_V1 d1;
565     d1.id = 1;
566     d1.name = "Device1";
567     std::cout << "    sizeof(Device_V1) = " << sizeof(Device_V1) << " bytes\n";
568     std::cout << "    &id offset = " << (void*)&d1.id << "\n";
569     std::cout << "    &name offset = " << (void*)&d1.name << "\n";
570
571     std::cout << "\nSensor_V1 (derived) layout:\n";
572     Sensor_V1 s1;
573     s1.id = 2;
574     s1.name = "Sensor1";
575     s1.reading = 42.5;
576     std::cout << "    sizeof(Sensor_V1) = " << sizeof(Sensor_V1) << " bytes\n";
577     std::cout << "    &reading offset = " << (void*)&s1.reading << "\n";
578
579     std::cout << "\nDevice_V2 layout (after adding 'enabled' field):\n";
580     Device_V2 d2;
581     std::cout << "    sizeof(Device_V2) = " << sizeof(Device_V2) << " bytes\n";
582     std::cout << "    Problem: All derived class member offsets have shifted!\n"
583         ;
584     std::cout << "    Solution: Never add non-static data members to base
585         classes\n";
586     std::cout << "                Use Pimpl idiom or version interfaces\n";
587 }
588
589 } // namespace abi_example
590
591 // =====
592
593 // SECTION 8: Best Practices Summary
594 //
```

```
=====
591 void show_best_practices() {
592     std::cout << "\n" << std::string(70, '=') << "\n";
593     std::cout << "==== Best Practices to Avoid Fragile Base Class ===\n";
594     std::cout << std::string(70, '=') << "\n\n";
595
596     std::cout << "1. PREFER COMPOSITION OVER INHERITANCE\n";
597     std::cout << "    Use 'has-a' instead of 'is-a' when possible\n";
598     std::cout << "    Decouple implementation details\n";
599     std::cout << "    More flexible and testable\n\n";
600
601     std::cout << "2. DESIGN FOR INHERITANCE OR PROHIBIT IT\n";
602     std::cout << "    Make classes 'final' by default\n";
603     std::cout << "    Only allow inheritance for classes explicitly designed
604         for it\n";
605     std::cout << "    Document extension points clearly\n\n";
606
607     std::cout << "3. USE THE 'final' KEYWORD\n";
608     std::cout << "    Mark concrete classes as 'final'\n";
609     std::cout << "    Mark critical methods as 'final' to prevent override
610        ";
611     std::cout << "    Prevents accidental misuse\n\n";
612
613     std::cout << "4. HIDE IMPLEMENTATION DETAILS\n";
614     std::cout << "    Make data members private\n";
615     std::cout << "    Use private helper methods\n";
616     std::cout << "    Expose minimal protected interface\n\n";
617
618     std::cout << "5. PREFER INTERFACES (PURE ABSTRACT CLASSES)\n";
619     std::cout << "    Define contracts, not implementations\n";
620     std::cout << "    Use composition + decorator pattern\n";
621     std::cout << "    Eliminates implementation inheritance fragility\n\n";
622
623     std::cout << "6. USE TEMPLATE METHOD PATTERN\n";
624     std::cout << "    Base class controls algorithm structure\n";
625     std::cout << "    Derived classes override hooks only\n";
626     std::cout << "    Clear extension points\n\n";
627
628     std::cout << "7. FOR LIBRARY DEVELOPERS - MAINTAIN ABI STABILITY\n";
629     std::cout << "    Never add non-static data members to base classes\n";
630     std::cout << "    Use Pimpl idiom for implementation hiding\n";
631     std::cout << "    Version your interfaces\n\n";
632
633     std::cout << "REMEMBER: \"Prefer composition over inheritance\" - Gang of
634         Four\n";
635 }
636
637 // MAIN FUNCTION
638 //
```

=====

```
=====
638
639 int main() {
640     std::cout << "\n";
641     std::cout << "
642         std::cout << "      Fragile Base Class Problem in C++ and Solutions
643             "
644             std::cout << "
645             // Demonstrate the problem
646             fragile_example::demonstrate();
647
648             // Show solutions
649             composition_solution::demonstrate();
650             encapsulation_solution::demonstrate();
651             final_solution::demonstrate();
652             template_method_solution::demonstrate();
653             interface_solution::demonstrate();
654
655             // ABI considerations
656             abi_example::demonstrate();
657
658             // Best practices
659             show_best_practices();
660
661             std::cout << "\n" << std::string(70, '=') << "\n";
662             std::cout << "All demonstrations completed!\n";
663             std::cout << std::string(70, '=') << "\n\n";
664
665             return 0;
666 }
```

## 34 Source Code: FunctionalSafetyISO26262.cpp

**File:** src/FunctionalSafetyISO26262.cpp

**Repository:** [View on GitHub](#)

```
1 // FunctionalSafetyISO26262.cpp
2 // Comprehensive demonstration of ISO 26262 Functional Safety principles in
3 // Modern C++
4 // Covers ASIL levels, MISRA C++, redundancy, watchdogs, safe data types, and
5 // deterministic patterns
6
7 #include <iostream>
8 #include <cstdint>
9 #include <array>
10 #include <limits>
11 #include <chrono>
12 #include <functional>
13 #include <algorithm>
14 #include <cstring>
15
16 // SECTION 1: ASIL Levels and Safety Requirements
17 // ISO 26262 defines 4 Automotive Safety Integrity Levels (ASIL A-D)
18 // ASIL D = highest safety requirements (e.g., braking systems)
19 // ASIL A = lowest safety requirements (e.g., rear lights)
20
21 enum class ASILLevel : uint8_t {
22     QM = 0,    // Quality Management (no ASIL)
23     A = 1,     // Lowest safety integrity
24     B = 2,
25     C = 3,
26     D = 4     // Highest safety integrity
27 };
28
29 // Safety state for system degradation
30 enum class SafetyState : uint8_t {
31     NORMAL_OPERATION = 0,
32     DEGRADED_MODE = 1,
33     SAFE_STATE = 2,
34     EMERGENCY_STOP = 3
35 };
36
37 // Safety-critical function return type (no exceptions!)
38 enum class SafetyResult : uint8_t {
39     OK = 0,
40     WARNING = 1,
41     ERROR = 2,
42     CRITICAL_FAILURE = 3
43 };
```

```

44
45 void demonstrate_asil_levels() {
46     std::cout << "\n==== 1. ASIL Levels and Safety Requirements ===\n";
47
48     struct SafetyRequirement {
49         const char* component;
50         ASILLevel asil;
51         const char* requirement;
52     };
53
54     constexpr std::array<SafetyRequirement, 5> requirements = {{
55         {"Brake-by-wire", ASILLevel::D, "Dual redundancy + voting"}, 
56         {"Electric power steering", ASILLevel::C, "Plausibility checks + 
57             monitoring"}, 
58         {"Airbag control", ASILLevel::D, "Watchdog + memory protection"}, 
59         {"Engine control", ASILLevel::B, "Runtime checks + diagnostics"}, 
60         {"Interior lighting", ASILLevel::QM, "Basic error handling"} 
61     }};
62
63     std::cout << "\nASIL Classification Examples:\n";
64     for (const auto& req : requirements) {
65         std::cout << "    " << req.component << " [ASIL-";
66         switch (req.asil) {
67             case ASILLevel::QM: std::cout << "QM"; break;
68             case ASILLevel::A: std::cout << "A"; break;
69             case ASILLevel::B: std::cout << "B"; break;
70             case ASILLevel::C: std::cout << "C"; break;
71             case ASILLevel::D: std::cout << "D"; break;
72         }
73         std::cout << "] : " << req.requirement << "\n";
74     }
75
76     std::cout << "\nKey ISO 26262 Principles:\n";
77     std::cout << "    No dynamic memory allocation in safety functions\n";
78     std::cout << "    No exceptions in safety-critical paths (use error codes) 
79         \n";
80     std::cout << "    Deterministic timing (WCET - Worst Case Execution Time)\n";
81     std::cout << "    Redundancy and voting for ASIL-D\n";
82     std::cout << "    Comprehensive diagnostics and fault detection\n";
83 }
84
85 // =====
86 // SECTION 2: Safe Data Types with Overflow Protection
87 // =====
88
89 // MISRA C++ requires explicit bounds checking and overflow protection
90
91 template<typename T>
92 class SafeInteger {
93     private:

```

```
91     T value_;
92     bool valid_;
93
94 public:
95     constexpr SafeInteger() noexcept : value_(0), valid_(true) {}
96     constexpr explicit SafeInteger(T val) noexcept : value_(val), valid_(true)
97         {}
98
99     // Safe addition with overflow detection
100    SafeInteger add(T other) const noexcept {
101        SafeInteger result;
102
103        // Check for overflow before operation (MISRA requirement)
104        if (other > 0 && value_ > std::numeric_limits<T>::max() - other) {
105            result.valid_ = false; // Overflow detected
106            result.value_ = std::numeric_limits<T>::max();
107        } else if (other < 0 && value_ < std::numeric_limits<T>::min() - other)
108            {
109                result.valid_ = false; // Underflow detected
110                result.value_ = std::numeric_limits<T>::min();
111            } else {
112                result.value_ = value_ + other;
113                result.valid_ = true;
114            }
115
116        return result;
117    }
118
119    // Safe multiplication with overflow detection
120    SafeInteger multiply(T other) const noexcept {
121        SafeInteger result;
122
123        if (other == 0 || value_ == 0) {
124            result.value_ = 0;
125            result.valid_ = true;
126        } else if (value_ > std::numeric_limits<T>::max() / other) {
127            result.valid_ = false; // Overflow
128            result.value_ = std::numeric_limits<T>::max();
129        } else {
130            result.value_ = value_ * other;
131            result.valid_ = true;
132        }
133
134        return result;
135    }
136
137    constexpr T get() const noexcept { return value_; }
138    constexpr bool is_valid() const noexcept { return valid_; }
139};
140
141 void demonstrate_safe_data_types() {
142     std::cout << "\n== 2. Safe Data Types with Overflow Protection ==\n";
143
144     SafeInteger<int32_t> speed(100);
```

```
143     SafeInteger<int32_t> delta(50);
144
145     auto new_speed = speed.add(delta.get());
146     std::cout << "\nSafe Addition: 100 + 50 = " << new_speed.get()
147             << " (valid: " << (new_speed.is_valid() ? "" : "") << ")\n";
148
149     // Demonstrate overflow detection
150     SafeInteger<int32_t> large_val(std::numeric_limits<int32_t>::max() - 10);
151     auto overflow_result = large_val.add(100);
152     std::cout << "\nOverflow Detection: MAX-10 + 100 = " << overflow_result.
153             get()
154             << " (valid: " << (overflow_result.is_valid() ? "" : " OVERFLOW
155                 DETECTED") << ")\n";
156
157     std::cout << "\nMISRA C++ Guidelines:\n";
158     std::cout << "    All arithmetic checked for overflow/underflow\n";
159     std::cout << "    Explicit error flags instead of exceptions\n";
160     std::cout << "    constexpr for compile-time safety verification\n";
161     std::cout << "    noexcept guarantees for safety-critical functions\n";
162 }
163
164 // -----
165 // SECTION 3: Redundancy and Voting Mechanisms (ASIL-D Requirement)
166 // -----
167
168 // Dual-channel or triple-modular redundancy for critical computations
169
170 template<typename T>
171 class DualChannelComputation {
172 private:
173     std::function<T()> channel1_;
174     std::function<T()> channel2_;
175     T tolerance_;
176
177 public:
178     DualChannelComputation(std::function<T()> ch1, std::function<T()> ch2, T
179                           tol) noexcept
180             : channel1_(ch1), channel2_(ch2), tolerance_(tol) {}
181
182     // Execute both channels and compare results
183     SafetyResult compute(T& result) noexcept {
184         T result1 = channel1_();
185         T result2 = channel2_();
186
187         // Calculate absolute difference
188         T diff = (result1 > result2) ? (result1 - result2) : (result2 -
189             result1);
190
191         if (diff <= tolerance_) {
192             // Results agree within tolerance
193             result = (result1 + result2) / 2; // Use average
194         }
195     }
196 }
```

```
189         return SafetyResult::OK;
190     } else {
191         // Mismatch detected - critical failure
192         result = 0; // Safe default
193         return SafetyResult::CRITICAL_FAILURE;
194     }
195 }
196 };
197
198 // Triple Modular Redundancy (TMR) with voting
199 template<typename T>
200 class TripleModularRedundancy {
201 private:
202     std::array<std::function<T()>, 3> channels_;
203     T tolerance_;
204
205 public:
206     TripleModularRedundancy(std::function<T()> ch1, std::function<T()> ch2,
207                             std::function<T()> ch3, T tol) noexcept
208         : channels_{ch1, ch2, ch3}, tolerance_(tol) {}
209
210     // 2-out-of-3 voting
211     SafetyResult compute(T& result) noexcept {
212         std::array<T, 3> results;
213         for (size_t i = 0; i < 3; ++i) {
214             results[i] = channels_[i]();
215         }
216
217         // Check if any two results agree within tolerance
218         for (size_t i = 0; i < 3; ++i) {
219             for (size_t j = i + 1; j < 3; ++j) {
220                 T diff = (results[i] > results[j]) ?
221                         (results[i] - results[j]) : (results[j] - results[i]);
222
223                 if (diff <= tolerance_) {
224                     // Two channels agree
225                     result = (results[i] + results[j]) / 2;
226                     return (i == 2 || j == 2) ? SafetyResult::WARNING :
227                         SafetyResult::OK;
228                 }
229             }
230
231         // No agreement - critical failure
232         result = 0;
233         return SafetyResult::CRITICAL_FAILURE;
234     }
235 };
236
237 void demonstrate_redundancy_voting() {
238     std::cout << "\n== 3. Redundancy and Voting Mechanisms ==\n";
239
240     // Simulate brake pressure sensor readings
```

```
241     auto sensor_channel_1 = []() -> int32_t { return 850; }; // 850 kPa
242     auto sensor_channel_2 = []() -> int32_t { return 852; }; // 852 kPa (
243         slight variation)
244
245     DualChannelComputation<int32_t> brake_pressure(sensor_channel_1,
246             sensor_channel_2, 5);
247
248     int32_t pressure = 0;
249     SafetyResult status = brake_pressure.compute(pressure);
250
251     std::cout << "\nDual-Channel Brake Pressure Sensor:\n";
252     std::cout << "    Channel 1: 850 kPa\n";
253     std::cout << "    Channel 2: 852 kPa\n";
254     std::cout << "    Result: " << pressure << " kPa (avg)\n";
255     std::cout << "    Status: " << (status == SafetyResult::OK ? " OK" : " FAILURE") << "\n";
256
257     // Triple modular redundancy example
258     auto tmr_ch1 = []() -> int32_t { return 100; };
259     auto tmr_ch2 = []() -> int32_t { return 101; };
260     auto tmr_ch3 = []() -> int32_t { return 99; }; // All agree within
261         tolerance
262
263     TripleModularRedundancy<int32_t> steering_angle(tmr_ch1, tmr_ch2, tmr_ch3,
264             5);
265
266     int32_t angle = 0;
267     SafetyResult tmr_status = steering_angle.compute(angle);
268
269     std::cout << "\nTriple Modular Redundancy (Steering Angle):\n";
270     std::cout << "    Channel 1: 100°\n";
271     std::cout << "    Channel 2: 101°\n";
272     std::cout << "    Channel 3: 99°\n";
273     std::cout << "    Voted Result: " << angle << "°\n";
274     std::cout << "    Status: " << (tmr_status == SafetyResult::OK ? " OK (2-of
275         -3 agree)" : " FAILURE") << "\n";
276
277 }
278
279 // =====
280 // SECTION 4: Watchdog Timer and Heartbeat Monitoring
281 // =====
282
283 // Detect software hangs and stuck states
284
285 class WatchdogTimer {
```

```
285 | private:
286 |     uint32_t timeout_ms_;
287 |     uint32_t last_kick_time_;
288 |     bool expired_;
289 |
290 | public:
291 |     explicit WatchdogTimer(uint32_t timeout_ms) noexcept
292 |         : timeout_ms_(timeout_ms), last_kick_time_(0), expired_(false) {}
293 |
294 |     // Kick the watchdog (reset timer)
295 |     void kick(uint32_t current_time_ms) noexcept {
296 |         last_kick_time_ = current_time_ms;
297 |         expired_ = false;
298 |     }
299 |
300 |     // Check if watchdog expired
301 |     bool check_expired(uint32_t current_time_ms) noexcept {
302 |         if (current_time_ms - last_kick_time_ > timeout_ms_) {
303 |             expired_ = true;
304 |         }
305 |         return expired_;
306 |     }
307 |
308 |     bool is_expired() const noexcept { return expired_; }
309 | };
310 |
311 | class SafetyMonitor {
312 | private:
313 |     WatchdogTimer watchdog_;
314 |     uint32_t heartbeat_count_;
315 |     SafetyState state_;
316 |
317 | public:
318 |     explicit SafetyMonitor(uint32_t watchdog_timeout) noexcept
319 |         : watchdog_(watchdog_timeout), heartbeat_count_(0), state_(SafetyState
320 |             ::NORMAL_OPERATION) {}
321 |
322 |     void heartbeat(uint32_t current_time) noexcept {
323 |         watchdog_.kick(current_time);
324 |         ++heartbeat_count_;
325 |     }
326 |
327 |     SafetyResult check_safety(uint32_t current_time) noexcept {
328 |         if (watchdog_.check_expired(current_time)) {
329 |             state_ = SafetyState::EMERGENCY_STOP;
330 |             return SafetyResult::CRITICAL_FAILURE;
331 |         }
332 |         return SafetyResult::OK;
333 |     }
334 |
335 |     SafetyState get_state() const noexcept { return state_; }
336 |     uint32_t get_heartbeat_count() const noexcept { return heartbeat_count_; }
337 | };
```

```
338 void demonstrate_watchdog_monitoring() {
339     std::cout << "\n==== 4. Watchdog Timer and Heartbeat Monitoring ===\n";
340
341     SafetyMonitor monitor(100); // 100ms timeout
342
343     std::cout << "\nNormal Operation:\n";
344     for (uint32_t time = 0; time <= 250; time += 50) {
345         monitor.heartbeat(time);
346         SafetyResult status = monitor.check_safety(time);
347         std::cout << " Time " << time << "ms: Heartbeat #" << monitor.
348             get_heartbeat_count()
349             << " - Status: " << (status == SafetyResult::OK ? " OK" : "
350                 FAILURE") << "\n";
351
352     // Simulate watchdog expiration (no heartbeat for >100ms)
353     std::cout << "\nSimulating Watchdog Timeout (no heartbeat):\n";
354     SafetyMonitor timeout_monitor(100);
355     timeout_monitor.heartbeat(0);
356
357     for (uint32_t time = 50; time <= 200; time += 50) {
358         // Skip heartbeat to simulate hang
359         SafetyResult status = timeout_monitor.check_safety(time);
360         std::cout << " Time " << time << "ms: ";
361         if (status == SafetyResult::CRITICAL_FAILURE) {
362             std::cout << " WATCHDOG EXPIRED - EMERGENCY STOP\n";
363             break;
364         } else {
365             std::cout << " OK\n";
366         }
367     }
368
369     std::cout << "\nWatchdog Best Practices:\n";
370     std::cout << "     Timeout based on WCET (Worst Case Execution Time)\n";
371     std::cout << "     Hardware watchdog preferred for fail-safe\n";
372     std::cout << "     Heartbeat every control cycle\n";
373     std::cout << "     Transition to safe state on expiration\n";
374
375 // =====
376 // SECTION 5: Memory Safety and Bounds Checking
377 // =====
378 // MISRA C++ requires explicit bounds checking, no buffer overruns
379
380 template<typename T, size_t N>
381 class SafeArray {
382     private:
383         std::array<T, N> data_;
384
385     public:
```

```
386     constexpr SafeArray() noexcept : data_{} {}  
387  
388     // Safe access with bounds checking (no exceptions!)  
389     SafetyResult get(size_t index, T& value) const noexcept {  
390         if (index >= N) {  
391             return SafetyResult::ERROR;  
392         }  
393         value = data_[index];  
394         return SafetyResult::OK;  
395     }  
396  
397     SafetyResult set(size_t index, T value) noexcept {  
398         if (index >= N) {  
399             return SafetyResult::ERROR;  
400         }  
401         data_[index] = value;  
402         return SafetyResult::OK;  
403     }  
404  
405     constexpr size_t size() const noexcept { return N; }  
406  
407     // Safe iteration  
408     T* begin() noexcept { return data_.data(); }  
409     T* end() noexcept { return data_.data() + N; }  
410     const T* begin() const noexcept { return data_.data(); }  
411     const T* end() const noexcept { return data_.data() + N; }  
412 };  
413  
414 // Safe string operations (fixed-size, no dynamic allocation)  
415 template<size_t N>  
416 class SafeString {  
417     private:  
418         std::array<char, N> buffer_;  
419         size_t length_;  
420  
421     public:  
422         constexpr SafeString() noexcept : buffer_{}, length_(0) {  
423             buffer_[0] = '\0';  
424         }  
425  
426         SafetyResult copy_from(const char* src) noexcept {  
427             if (src == nullptr) {  
428                 return SafetyResult::ERROR;  
429             }  
430  
431             size_t i = 0;  
432             while (i < N - 1 && src[i] != '\0') {  
433                 buffer_[i] = src[i];  
434                 ++i;  
435             }  
436             buffer_[i] = '\0';  
437             length_ = i;  
438  
439             return (src[i] == '\0') ? SafetyResult::OK : SafetyResult::WARNING;  
440 }
```

```

440         // Truncated
441     }
442
443     SafetyResult append(const char* src) noexcept {
444         if (src == nullptr) {
445             return SafetyResult::ERROR;
446         }
447
448         size_t i = 0;
449         while (length_ < N - 1 && src[i] != '\0') {
450             buffer_[length_++] = src[i++];
451         }
452         buffer_[length_] = '\0';
453
454         return (src[i] == '\0') ? SafetyResult::OK : SafetyResult::WARNING;
455     }
456
457     const char* c_str() const noexcept { return buffer_.data(); }
458     size_t length() const noexcept { return length_; }
459     constexpr size_t capacity() const noexcept { return N - 1; }
460 };
461
462 void demonstrate_memory_safety() {
463     std::cout << "\n==== 5. Memory Safety and Bounds Checking ====\n";
464
465     // Safe array with compile-time size
466     SafeArray<int32_t, 10> sensor_readings;
467
468     std::cout << "\nSafe Array Operations:\n";
469     for (size_t i = 0; i < sensor_readings.size(); ++i) {
470         sensor_readings.set(i, static_cast<int32_t>(i * 100));
471     }
472
473     int32_t value = 0;
474     SafetyResult result = sensor_readings.get(5, value);
475     std::cout << " Reading[5] = " << value << " (status: "
476                 << (result == SafetyResult::OK ? " OK" : " ERROR")) << "\n";
477
478     // Attempt out-of-bounds access
479     result = sensor_readings.get(15, value);
480     std::cout << " Reading[15] = " << (result == SafetyResult::OK ? " Valid" :
481                                         " OUT OF BOUNDS") << "\n";
482
483     // Safe string operations
484     SafeString<32> device_id;
485     device_id.copy_from("BRAKE_CTRL_");
486     device_id.append("ECU001");
487
488     std::cout << "\nSafe String Operations:\n";
489     std::cout << " Device ID: " << device_id.c_str() << "\n";
490     std::cout << " Length: " << device_id.length() << "/" << device_id.
491                 capacity() << "\n";
492
493     std::cout << "\nMemory Safety Rules:\n";

```

```
491     std::cout << "    No dynamic allocation (new/malloc) in safety functions\n"
492     "    ";
493     std::cout << "    Fixed-size buffers with compile-time bounds\n";
494     std::cout << "    All array accesses bounds-checked\n";
495     std::cout << "    No pointer arithmetic without validation\n";
496     std::cout << "    Stack-allocated containers only\n";
497 }
498 // =====
499 // SECTION 6: Runtime Diagnostics and Self-Test
500 // =====
501 // Continuous self-monitoring and fault detection
502
503 class DiagnosticMonitor {
504 private:
505     struct DiagnosticCounter {
506         uint32_t total_checks;
507         uint32_t failures;
508         uint32_t warnings;
509     };
510
511     DiagnosticCounter counters_;
512     bool system_healthy_;
513
514 public:
515     DiagnosticMonitor() noexcept
516         : counters_{0, 0, 0}, system_healthy_(true) {}
517
518     // RAM test (simple pattern check)
519     SafetyResult test_ram(uint32_t* test_area, size_t size) noexcept {
520         ++counters_.total_checks;
521
522         // Write pattern
523         constexpr uint32_t PATTERN1 = 0x55555555;
524         constexpr uint32_t PATTERN2 = 0xAAAAAAA;
525
526         for (size_t i = 0; i < size; ++i) {
527             test_area[i] = PATTERN1;
528         }
529
530         // Verify pattern 1
531         for (size_t i = 0; i < size; ++i) {
532             if (test_area[i] != PATTERN1) {
533                 ++counters_.failures;
534                 system_healthy_ = false;
535                 return SafetyResult::CRITICAL_FAILURE;
536             }
537         }
538
539         // Write and verify pattern 2
```

```
540     for (size_t i = 0; i < size; ++i) {
541         test_area[i] = PATTERN2;
542     }
543
544     for (size_t i = 0; i < size; ++i) {
545         if (test_area[i] != PATTERN2) {
546             ++counters_.failures;
547             system_healthy_ = false;
548             return SafetyResult::CRITICAL_FAILURE;
549         }
550     }
551
552     return SafetyResult::OK;
553 }
554
555 // Plausibility check for sensor values
556 SafetyResult check_plausibility(int32_t value, int32_t min, int32_t max)
557     noexcept {
558     ++counters_.total_checks;
559
560     if (value < min || value > max) {
561         ++counters_.failures;
562         return SafetyResult::ERROR;
563     }
564
565     // Warning range (10% from limits)
566     int32_t warning_margin = (max - min) / 10;
567     if (value < min + warning_margin || value > max - warning_margin) {
568         ++counters_.warnings;
569         return SafetyResult::WARNING;
570     }
571
572     return SafetyResult::OK;
573 }
574
575 void get_diagnostics(uint32_t& total, uint32_t& failures, uint32_t&
576 warnings) const noexcept {
577     total = counters_.total_checks;
578     failures = counters_.failures;
579     warnings = counters_.warnings;
580 }
581
582 bool is_healthy() const noexcept { return system_healthy_; }
583 };
584
585 void demonstrate_runtime_diagnostics() {
586     std::cout << "\n==== 6. Runtime Diagnostics and Self-Test ====\n";
587
588     DiagnosticMonitor diagnostics;
589
590     // RAM self-test
591     std::array<uint32_t, 256> test_ram;
592     std::cout << "\nRAM Self-Test:\n";
593     SafetyResult ram_result = diagnostics.test_ram(test_ram.data(), test_ram.
```

```
    size());
592 std::cout << "  Pattern Test (0x55555555/0xAAAAAAA): "
593     << (ram_result == SafetyResult::OK ? "  PASS" : "  FAIL") << "\n"
594     ;
595
596 // Plausibility checks
597 std::cout << "\nSensor Plausibility Checks:\n";
598
599 struct SensorTest {
600     const char* name;
601     int32_t value;
602     int32_t min;
603     int32_t max;
604 };
605
606 constexpr std::array<SensorTest, 4> tests = {{
607     {"Wheel speed", 1200, 0, 3000},           // OK
608     {"Brake pressure", 850, 0, 1000},          // OK
609     {"Steering angle", 950, -900, 900},        // WARNING (near limit)
610     {"Throttle position", 1500, 0, 1000}      // ERROR (out of range)
611 }};
612
613 for (const auto& test : tests) {
614     SafetyResult result = diagnostics.check_plausibility(test.value, test.
615         min, test.max);
616     std::cout << "  " << test.name << " = " << test.value
617         << " [" << test.min << ".." << test.max << "]: ";
618
619     switch (result) {
620         case SafetyResult::OK:
621             std::cout << "  OK\n";
622             break;
623         case SafetyResult::WARNING:
624             std::cout << "  WARNING (near limit)\n";
625             break;
626         case SafetyResult::ERROR:
627             std::cout << "  OUT OF RANGE\n";
628             break;
629         default:
630             std::cout << "  CRITICAL\n";
631             break;
632     }
633 }
634
635 // Diagnostic summary
636 uint32_t total, failures, warnings;
637 diagnostics.get_diagnostics(total, failures, warnings);
638
639 std::cout << "\nDiagnostic Summary:\n";
640 std::cout << "  Total checks: " << total << "\n";
641 std::cout << "  Failures: " << failures << "\n";
642 std::cout << "  Warnings: " << warnings << "\n";
643 std::cout << "  System Health: " << (diagnostics.is_healthy() ? "  HEALTHY"
644     " : "  DEGRADED") << "\n";
```

```
642     std::cout << "\nDiagnostic Coverage:\n";
643     std::cout << "    RAM pattern test (startup + periodic)\n";
644     std::cout << "    Plausibility checks (range validation)\n";
645     std::cout << "    Stuck-at fault detection\n";
646     std::cout << "    Diagnostic Trouble Code (DTC) logging\n";
647 }
648
649
650 // =====
651 // SECTION 7: Safe State Transitions and Fault Handling
652 //
653 // =====
654
655 class BrakingController {
656 private:
657     enum class BrakeState : uint8_t {
658         INIT = 0,
659         NORMAL = 1,
660         DEGRADED = 2,
661         FAILSAFE = 3,
662         EMERGENCY = 4
663     };
664
665     BrakeState current_state_;
666     BrakeState previous_state_;
667     uint32_t fault_count_;
668
669 public:
670     BrakingController() noexcept
671         : current_state_(BrakeState::INIT),
672           previous_state_(BrakeState::INIT),
673           fault_count_(0) {}
674
675     SafetyResult initialize() noexcept {
676         if (current_state_ != BrakeState::INIT) {
677             return SafetyResult::ERROR;
678         }
679
680         // Perform self-tests
681         // ... (RAM test, sensor check, actuator test)
682
683         previous_state_ = current_state_;
684         current_state_ = BrakeState::NORMAL;
685         return SafetyResult::OK;
686     }
687
688     SafetyResult handle_fault(SafetyResult fault_severity) noexcept {
689         ++fault_count_;
690         previous_state_ = current_state_;
691     }

```

```
692     switch (fault_severity) {
693         case SafetyResult::WARNING:
694             // Stay in current state but log warning
695             return SafetyResult::WARNING;
696
697         case SafetyResult::ERROR:
698             // Transition to degraded mode
699             if (current_state_ == BrakeState::NORMAL) {
700                 current_state_ = BrakeState::DEGRADED;
701             }
702             return SafetyResult::ERROR;
703
704         case SafetyResult::CRITICAL_FAILURE:
705             // Immediate transition to emergency
706             current_state_ = BrakeState::EMERGENCY;
707             return SafetyResult::CRITICAL_FAILURE;
708
709         default:
710             return SafetyResult::OK;
711     }
712 }
713
714 SafetyResult apply_brakes(int32_t pressure) noexcept {
715     switch (current_state_) {
716         case BrakeState::INIT:
717             return SafetyResult::ERROR; // Not initialized
718
719         case BrakeState::NORMAL:
720             // Full functionality
721             return apply_normal_braking(pressure);
722
723         case BrakeState::DEGRADED:
724             // Limited functionality (e.g., one channel failed)
725             return apply_degraded_braking(pressure / 2);
726
727         case BrakeState::FAILSAFE:
728         case BrakeState::EMERGENCY:
729             // Maximum braking regardless of input
730             return apply_emergency_braking();
731     }
732
733     return SafetyResult::ERROR;
734 }
735
736 const char* get_state_name() const noexcept {
737     switch (current_state_) {
738         case BrakeState::INIT: return "INIT";
739         case BrakeState::NORMAL: return "NORMAL";
740         case BrakeState::DEGRADED: return "DEGRADED";
741         case BrakeState::FAILSAFE: return "FAILSAFE";
742         case BrakeState::EMERGENCY: return "EMERGENCY";
743         default: return "UNKNOWN";
744     }
745 }
```

```
746     uint32_t get_fault_count() const noexcept { return fault_count_; }
```

```
747
748
749 private:
750     SafetyResult apply_normal_braking(int32_t pressure) noexcept {
751         // Normal braking logic
752         (void)pressure; // Suppress unused warning
753         return SafetyResult::OK;
754     }
755
756     SafetyResult apply_degraded_braking(int32_t pressure) noexcept {
757         // Degraded mode (e.g., mechanical backup)
758         (void)pressure;
759         return SafetyResult::WARNING;
760     }
761
762     SafetyResult apply_emergency_braking() noexcept {
763         // Maximum braking effort
764         return SafetyResult::CRITICAL_FAILURE;
765     }
766 };
767
768 void demonstrate_safe_state_transitions() {
769     std::cout << "\n==== 7. Safe State Transitions and Fault Handling ===\n";
770
771     BrakingController brake_system;
772
773     std::cout << "\nBraking System State Machine:\n";
774     std::cout << "  Initial State: " << brake_system.get_state_name() << "\n";
775
776     // Initialize
777     brake_system.initialize();
778     std::cout << "  After Init: " << brake_system.get_state_name() << "\n";
779
780     // Normal operation
781     SafetyResult result = brake_system.apply_brakes(500);
782     std::cout << "  Apply 500 kPa: " << (result == SafetyResult::OK ? "  OK" :
783                               "  ERROR") << "\n";
784
785     // Simulate fault
786     std::cout << "\n  Simulating sensor fault...\n";
787     brake_system.handle_fault(SafetyResult::ERROR);
788     std::cout << "  State after fault: " << brake_system.get_state_name() << "\n";
789
790     result = brake_system.apply_brakes(500);
791     std::cout << "  Apply 500 kPa in degraded: "
792           << (result == SafetyResult::WARNING ? "  LIMITED" : "  ERROR") <<
793           "\n";
794
795     // Critical fault
796     std::cout << "\n  Simulating critical failure...\n";
797     brake_system.handle_fault(SafetyResult::CRITICAL_FAILURE);
798     std::cout << "  State after critical: " << brake_system.get_state_name()
```

```
    << "\n";
797  std::cout << "  Total faults handled: " << brake_system.get_fault_count()
    << "\n";
798
799  std::cout << "\nState Machine Principles:\n";
800  std::cout << "  Deterministic transitions (no race conditions)\n";
801  std::cout << "  Always transition to safe state on fault\n";
802  std::cout << "  No invalid states reachable\n";
803  std::cout << "  Fail-safe defaults (e.g., emergency braking)\n";
804 }
805
806 // =====
807 // SECTION 8: Deterministic Timing and WCET
808 // =====
809 // Predictable execution time for real-time safety functions
810
811 class ExecutionTimeMonitor {
812 private:
813     std::chrono::steady_clock::time_point start_time_;
814     uint32_t wcet_us_; // Worst-case execution time in microseconds
815     bool timingViolation_;
816
817 public:
818     explicit ExecutionTimeMonitor(uint32_t wcet_us) noexcept
819         : wcet_us_(wcet_us), timingViolation_(false) {}
820
821     void start() noexcept {
822         start_time_ = std::chrono::steady_clock::now();
823         timingViolation_ = false;
824     }
825
826     SafetyResult check() noexcept {
827         auto end_time = std::chrono::steady_clock::now();
828         auto elapsed_us = std::chrono::duration_cast<std::chrono::microseconds>(
829             end_time - start_time_).count();
830
831         if (elapsed_us > wcet_us_) {
832             timingViolation_ = true;
833             return SafetyResult::ERROR;
834         }
835
836         return SafetyResult::OK;
837     }
838
839     bool hasViolation() const noexcept { return timingViolation_; }
840 };
841
842 // Safety-critical control loop with deterministic timing
843 SafetyResult safety_control_cycle(ExecutionTimeMonitor& timer) noexcept {
```

```
844     timer.start();  
845  
846     // Read sensors (deterministic)  
847     volatile int32_t sensor1 = 100;  
848     volatile int32_t sensor2 = 200;  
849  
850     // Compute control output (no branches, no loops with variable iterations)  
851     volatile int32_t output = (sensor1 + sensor2) / 2;  
852  
853     // Write actuators  
854     (void)output; // Suppress unused warning  
855  
856     // Check timing constraint  
857     return timer.check();  
858 }  
859  
860 void demonstrate_deterministic_timing() {  
861     std::cout << "\n==== 8. Deterministic Timing and WCET ====\n";  
862  
863     // Set WCET budget to 100 microseconds  
864     ExecutionTimeMonitor timer(100);  
865  
866     std::cout << "\nControl Cycle Timing (WCET = 100 \u00b5s):\n";  
867  
868     for (int i = 0; i < 5; ++i) {  
869         SafetyResult result = safety_control_cycle(timer);  
870         std::cout << " Cycle " << (i + 1) << ":"  
871             << (result == SafetyResult::OK ? " Within WCET" : " TIMING  
872                 VIOLATION") << "\n";  
873     }  
874  
875     std::cout << "\nDeterminism Requirements:\n";  
876     std::cout << "    No dynamic memory allocation (non-deterministic)\n";  
877     std::cout << "    No unbounded loops (use fixed iteration count)\n";  
878     std::cout << "    No recursive functions (stack usage)\n";  
879     std::cout << "    Disable interrupts in critical sections\n";  
880     std::cout << "    Priority-based scheduling (preemptive RTOS)\n";  
881     std::cout << "    WCET analysis tools (e.g., aiT, RapiTime)\n";  
882 }  
883 //  
=====  
884 // SECTION 9: MISRA C++ Compliance Patterns  
885 //  
=====  
886  
887 void demonstrate_misra_compliance() {  
888     std::cout << "\n==== 9. MISRA C++ Compliance Patterns ====\n";  
889  
890     std::cout << "\nKey MISRA C++ 2023 Rules:\n\n";  
891  
892     std::cout << "Rule 5-0-3: Prohibited types and features:\n";
```

```
893     std::cout << "    Avoid: exceptions, RTTI (typeid), dynamic_cast in safety
894                 code\n";
895     std::cout << "    Use: Error codes, static polymorphism (templates)\n\n";
896
897     std::cout << "Rule 5-2-12: No dynamic memory allocation:\n";
898     std::cout << "    Avoid: new, delete, malloc, free, std::vector (dynamic)\n
899                 ";
900     std::cout << "    Use: std::array, static buffers, placement new (if
901                 needed)\n\n";
902
903
904     std::cout << "Rule 6-4-5: Unconditional throw/goto prohibited:\n";
905     std::cout << "    Avoid: throw exceptions in safety functions\n";
906     std::cout << "    Use: Return error codes (SafetyResult enum)\n\n";
907
908
909     std::cout << "Rule 8-0-1: All functions have one exit point:\n";
910     std::cout << "    Avoid: Multiple return statements\n";
911     std::cout << "    Use: Single return with result variable\n\n";
912
913
914     std::cout << "Rule 18-0-3: No library functions with undefined behavior:\n
915                 ";
916     std::cout << "    Avoid: atoi, gets, sprintf (unsafe)\n";
917     std::cout << "    Use: strtol with error checking, snprintf\n\n";
918
919
920     std::cout << "Rule 27-0-1: All includes have include guards:\n";
921     std::cout << "    Use: #pragma once or #ifndef guards\n\n";
922 }
923
924 ==
925 // SECTION 10: Safety Checklist and Best Practices Summary
926 // ==
927
928 void demonstrate_safety_checklist() {
929     std::cout << "\n== 10. ISO 26262 Safety Checklist ==\n";
930
931     std::cout << "\n DESIGN PHASE:\n";
932     std::cout << "    [] ASIL classification completed\n";
933     std::cout << "    [] Hazard analysis and risk assessment (HARA)\n";
934     std::cout << "    [] Safety goals defined (fail-safe behavior)\n";
935     std::cout << "    [] Redundancy strategy selected (dual/TMR)\n";
936     std::cout << "    [] WCET analysis performed\n";
937     std::cout << "    [] Diagnostic coverage targets set\n\n";
938 }
```

```

939 std::cout << " IMPLEMENTATION PHASE:\n";
940 std::cout << " [] MISRA C++ guidelines followed\n";
941 std::cout << " [] No dynamic memory allocation\n";
942 std::cout << " [] No exceptions in safety functions\n";
943 std::cout << " [] All arithmetic overflow-checked\n";
944 std::cout << " [] All arrays bounds-checked\n";
945 std::cout << " [] Watchdog timer implemented\n";
946 std::cout << " [] Safe state transitions verified\n";
947 std::cout << " [] Deterministic timing validated\n\n";
948
949 std::cout << " VERIFICATION PHASE:\n";
950 std::cout << " [] Static analysis (PC-lint, Coverity, PVS-Studio)\n";
951 std::cout << " [] Dynamic testing (functional + fault injection)\n";
952 std::cout << " [] Code reviews (multiple reviewers)\n";
953 std::cout << " [] Requirements traceability matrix\n";
954 std::cout << " [] Coverage analysis (MC/DC for ASIL-D)\n";
955 std::cout << " [] Safety case documentation\n\n";
956
957 std::cout << " VALIDATION PHASE:\n";
958 std::cout << " [] Hardware-in-the-loop (HIL) testing\n";
959 std::cout << " [] Environmental stress testing\n";
960 std::cout << " [] EMC/EMI validation\n";
961 std::cout << " [] Long-term reliability testing\n";
962 std::cout << " [] Safety audit completed\n\n";
963
964 std::cout << "KEY TAKEAWAYS:\n";
965 std::cout << " 1. Safety is a process, not just code\n";
966 std::cout << " 2. Redundancy and diagnostics are essential for ASIL-C/D\n";
967 std::cout << " ";
968 std::cout << " 3. Deterministic behavior is mandatory\n";
969 std::cout << " 4. Always transition to safe state on fault\n";
970 std::cout << " 5. Document everything (traceability is critical)\n";
971 std::cout << " 6. Use certified tools and compilers for final builds\n";
972 std::cout << " 7. Continuous monitoring in production (field data)\n";
973 }
974 // =====
975 // MAIN FUNCTION
976 // =====
977
978 int main() {
979     std::cout << "                               \n";
980     std::cout << " ISO 26262 Functional Safety - Modern C++ Demonstration\n";
981     std::cout << "                               \n";
982     std::cout << "                               \n";
983     std::cout << " Comprehensive examples of safety-critical automotive code\n";
984     std::cout << "                               \n";
985     std::cout << " covering ASIL levels, MISRA C++, redundancy, and more\n";
986     std::cout << "                               \n";

```

```
984     std::cout << "\n";
985
986     demonstrate_asil_levels();
987     demonstrate_safe_data_types();
988     demonstrate_redundancy_voting();
989     demonstrate_watchdog_monitoring();
990     demonstrate_memory_safety();
991     demonstrate_runtime_diagnostics();
992     demonstrate_safe_state_transitions();
993     demonstrate_deterministic_timing();
994     demonstrate_misra_compliance();
995     demonstrate_safety_checklist();
996
997     std::cout << "\n" << std::string(68, '=') << "\n";
998     std::cout << "All safety demonstrations completed successfully!\n";
999     std::cout << "Remember: Safety certification requires formal verification
1000           ,\n";
1000     std::cout << "           extensive testing, and compliance with ISO 26262
1001           process.\n";
1001     std::cout << std::string(68, '=') << "\n\n";
1002
1003     return 0;
1004 }
```

## 35 Source Code: FuturePromiseAsync.cpp

File: src/FuturePromiseAsync.cpp

Repository: [View on GitHub](#)

```
1 // FuturePromiseAsync.cpp
2 // Comprehensive guide to C++ concurrency primitives:
3 // - std::future and std::promise
4 // - std::packaged_task
5 // - std::async
6 // And how they relate to ASIO
7 //
8 // KEY CONCEPTS:
9 // 1. These are NOT alternatives to ASIO - they serve different purposes
10 // 2. future/promise/async are for TASK-BASED concurrency
11 // 3. ASIO is for ASYNCHRONOUS I/O and event-driven programming
12 // 4. They can be used together!
13 //
14 // WHEN TO USE WHAT:
15 // - std::async: Simplest, for fire-and-forget async tasks
16 // - std::promise/future: Manual control, producer-consumer pattern
17 // - std::packaged_task: Wrap callable, control execution timing
18 // - ASIO: Network I/O, timers, serial ports, event-driven architecture
19
20 #include <iostream>
21 #include <future>
22 #include <thread>
23 #include <chrono>
24 #include <functional>
25 #include <vector>
26 #include <queue>
27 #include <random>
28 #include <iomanip>
29
30 using namespace std::chrono_literals;
31
32 //
33 // SECTION 1: std::async - The Simplest Way
34 //
35
36 namespace async_examples {
37
38 // Simple async task
39 int calculate_sum(int a, int b) {
40     std::cout << " [async] Calculating " << a << " + " << b
41             << " on thread " << std::this_thread::get_id() << "\n";
42     std::this_thread::sleep_for(500ms);
43     return a + b;
44 }
```

```
46 // Async task with exception
47 int divide(int a, int b) {
48     std::cout << "  [async] Dividing " << a << " / " << b << "\n";
49     std::this_thread::sleep_for(300ms);
50     if (b == 0) {
51         throw std::invalid_argument("Division by zero!");
52     }
53     return a / b;
54 }
55
56 void demonstrate() {
57     std::cout << "\n" << std::string(70, '=') << "\n";
58     std::cout << "==== SECTION 1: std::async - The Simplest Way ===\n";
59     std::cout << std::string(70, '=') << "\n\n";
60
61     std::cout << "Main thread ID: " << std::this_thread::get_id() << "\n\n";
62
63     // 1. Launch async task (may run in new thread or deferred)
64     std::cout << "1. Basic async (default policy):\n";
65     auto future1 = std::async(calculate_sum, 10, 20);
66     std::cout << "  Task launched, doing other work...\n";
67     std::this_thread::sleep_for(200ms);
68     std::cout << "  Getting result: " << future1.get() << "\n\n";
69
70     // 2. Force async execution in new thread
71     std::cout << "2. Force async (std::launch::async):\n";
72     auto future2 = std::async(std::launch::async, calculate_sum, 15, 25);
73     std::cout << "  Task running in parallel...\n";
74     std::cout << "  Result: " << future2.get() << "\n\n";
75
76     // 3. Deferred execution (lazy evaluation)
77     std::cout << "3. Deferred execution (std::launch::deferred):\n";
78     std::cout << "  Launching deferred task...\n";
79     auto future3 = std::async(std::launch::deferred, calculate_sum, 5, 10);
80     std::cout << "  Task not started yet!\n";
81     std::cout << "  Calling get()... (task runs NOW on this thread)\n";
82     std::cout << "  Result: " << future3.get() << "\n\n";
83
84     // 4. Exception handling
85     std::cout << "4. Exception handling with async:\n";
86     auto future4 = std::async(std::launch::async, divide, 100, 0);
87     try {
88         std::cout << "  Getting result...\n";
89         int result = future4.get();
90         std::cout << "  Result: " << result << "\n";
91     } catch (const std::exception& e) {
92         std::cout << "  Caught exception: " << e.what() << "\n";
93     }
94
95     std::cout << "\n WHEN TO USE std::async:\n";
96     std::cout << "  •  Quick fire-and-forget async operations\n";
97     std::cout << "  •  Simple parallel computations\n";
98     std::cout << "  •  Don't need fine control over thread management\n";
99     std::cout << "  •  Want automatic exception propagation\n";
```

```
100  }
101
102 } // namespace async_examples
103
104 // =====
105 // SECTION 2: std::promise and std::future - Producer-Consumer Pattern
106 // =====
107
108 namespace promise_future_examples {
109
110 // Producer thread sets value via promise
111 void produce_value(std::promise<int> promise, int value) {
112     std::cout << "  [Producer] Starting work...\n";
113     std::this_thread::sleep_for(1s);
114     std::cout << "  [Producer] Producing value: " << value << "\n";
115     promise.set_value(value); // Set the value
116     std::cout << "  [Producer] Value set, exiting\n";
117 }
118
119 // Producer that fails
120 void produce_with_error(std::promise<int> promise) {
121     std::cout << "  [Producer] Starting work...\n";
122     std::this_thread::sleep_for(500ms);
123     try {
124         throw std::runtime_error("Production failed!");
125     } catch (...) {
126         std::cout << "  [Producer] Error occurred, setting exception\n";
127         promise.set_exception(std::current_exception());
128     }
129 }
130
131 // Multiple consumers waiting for same value
132 void consume_value(std::shared_future<int> future, int consumer_id) {
133     std::cout << "  [Consumer " << consumer_id << "] Waiting for value...\n";
134     int value = future.get(); // Blocks until value is ready
135     std::cout << "  [Consumer " << consumer_id << "] Got value: " << value <<
136     "\n";
137 }
138
139 void demonstrate() {
140     std::cout << "\n" << std::string(70, '=') << "\n";
141     std::cout << "==== SECTION 2: std::promise and std::future ===\n";
142     std::cout << std::string(70, '=') << "\n\n";
143
144     // 1. Basic promise-future pair
145     std::cout << "1. Basic producer-consumer with promise/future:\n";
146     std::promise<int> promise1;
147     std::future<int> future1 = promise1.get_future();
148
149     std::thread producer1(produce_value, std::move(promise1), 42);
```

```
149
150     std::cout << "  [Consumer] Waiting for result...\n";
151     int result1 = future1.get();
152     std::cout << "  [Consumer] Received: " << result1 << "\n\n";
153     producer1.join();
154
155     // 2. Exception propagation
156     std::cout << "2. Exception handling with promise/future:\n";
157     std::promise<int> promise2;
158     std::future<int> future2 = promise2.get_future();
159
160     std::thread producer2(produce_with_error, std::move(promise2));
161
162     try {
163         std::cout << "  [Consumer] Waiting for result...\n";
164         int result2 = future2.get();
165         std::cout << "  [Consumer] Received: " << result2 << "\n";
166     } catch (const std::exception& e) {
167         std::cout << "  [Consumer] Caught exception: " << e.what() << "\n";
168     }
169     producer2.join();
170     std::cout << "\n";
171
172     // 3. Multiple consumers with shared_future
173     std::cout << "3. Multiple consumers with shared_future:\n";
174     std::promise<int> promise3;
175     std::shared_future<int> shared_future = promise3.get_future().share();
176
177     // Launch multiple consumers
178     std::vector<std::thread> consumers;
179     for (int i = 1; i <= 3; ++i) {
180         consumers.emplace_back(consume_value, shared_future, i);
181     }
182
183     std::this_thread::sleep_for(500ms);
184     std::cout << "  [Main] Setting value 100\n";
185     promise3.set_value(100);
186
187     for (auto& t : consumers) {
188         t.join();
189     }
190
191     std::cout << "\n WHEN TO USE std::promise/future:\n";
192     std::cout << " • Need manual control over when value is set\n";
193     std::cout << " • Producer-consumer pattern across threads\n";
194     std::cout << " • Complex synchronization scenarios\n";
195     std::cout << " • Want to set value from a different location than task
196         creation\n";
197     std::cout << " • Need shared_future for multiple consumers\n";
198 }
199 } // namespace promise_future_examples
200 //
```

```
=====
202 // SECTION 3: std::packaged_task - Wrap Callable, Control Execution
203 //
=====

204
205 namespace packaged_task_examples {
206
207 // Function to be wrapped
208 int compute_factorial(int n) {
209     std::cout << "  [Task] Computing factorial of " << n << "\n";
210     std::this_thread::sleep_for(500ms);
211     int result = 1;
212     for (int i = 2; i <= n; ++i) {
213         result *= i;
214     }
215     return result;
216 }
217
218 // Task queue for thread pool simulation
219 class SimpleThreadPool {
220     std::vector<std::thread> threads_;
221     std::queue<std::function<void()>> tasks_;
222     std::mutex mutex_;
223     std::condition_variable cv_;
224     bool stop_ = false;
225
226 public:
227     SimpleThreadPool(size_t num_threads) {
228         for (size_t i = 0; i < num_threads; ++i) {
229             threads_.emplace_back([this, i] {
230                 std::cout << "  [Worker " << i << "] Started\n";
231                 while (true) {
232                     std::function<void()> task;
233                     {
234                         std::unique_lock<std::mutex> lock(mutex_);
235                         cv_.wait(lock, [this] { return stop_ || !tasks_.empty()
236                             (); });
237
238                         if (stop_ && tasks_.empty()) {
239                             std::cout << "  [Worker " << i << "] Stopping\n";
240                             return;
241                         }
242
243                         task = std::move(tasks_.front());
244                         tasks_.pop();
245                     }
246                     std::cout << "  [Worker " << i << "] Executing task\n";
247                     task();
248                 }
249             });
250         }
251     }
252 }
```

```
251
252     ~SimpleThreadPool() {
253     {
254         std::unique_lock<std::mutex> lock(mutex_);
255         stop_ = true;
256     }
257     cv_.notify_all();
258     for (auto& t : threads_) {
259         t.join();
260     }
261 }
262
263 template<typename F>
264 void enqueue(F&& task) {
265 {
266     std::unique_lock<std::mutex> lock(mutex_);
267     tasks_.push(std::forward<F>(task));
268 }
269     cv_.notify_one();
270 }
271 };
272
273 void demonstrate() {
274     std::cout << "\n" << std::string(70, '=') << "\n";
275     std::cout << "==== SECTION 3: std::packaged_task ===\n";
276     std::cout << std::string(70, '=') << "\n\n";
277
278     // 1. Basic packaged_task
279     std::cout << "1. Basic packaged_task:\n";
280     std::packaged_task<int(int)> task1(compute_factorial);
281     std::future<int> future1 = task1.get_future();
282
283     std::cout << "    Task created but not executed yet\n";
284     std::cout << "    Launching task on new thread...\n";
285     std::thread t1(std::move(task1), 5);
286
287     std::cout << "    Waiting for result...\n";
288     std::cout << "    Factorial(5) = " << future1.get() << "\n";
289     t1.join();
290     std::cout << "\n";
291
292     // 2. Using packaged_task with thread pool
293     std::cout << "2. Thread pool with packaged_task:\n";
294     SimpleThreadPool pool(2);
295
296     std::vector<std::future<int>> futures;
297
298     for (int i = 3; i <= 6; ++i) {
299         std::packaged_task<int(int)> task(compute_factorial);
300         futures.push_back(task.get_future());
301
302         // Enqueue task to thread pool - use shared_ptr for copy-ability
303         auto task_ptr = std::make_shared<std::packaged_task<int(int)>>(std::move(task));
```

```

304     int value = i;
305     pool.enqueue([task_ptr, value]() {
306         (*task_ptr)(value);
307     });
308 }
309
310 std::cout << "\n  All tasks enqueued, waiting for results...\n\n";
311
312 for (size_t i = 0; i < futures.size(); ++i) {
313     int result = futures[i].get();
314     std::cout << "  Result " << (i + 3) << ": factorial = " << result << "
315         "\n";
316 }
317
318 std::cout << "\n WHEN TO USE std::packaged_task:\n";
319 std::cout << " • Wrapping callable objects for later execution\n";
320 std::cout << " • Implementing thread pools or task queues\n";
321 std::cout << " • Need to separate task creation from execution\n";
322 std::cout << " • Want to store tasks in containers\n";
323 std::cout << " • Building custom scheduling systems\n";
324 }
325 } // namespace packaged_task_examples
326
327 // =====
328 // SECTION 4: Comparison Summary
329 // =====
330
331 void show_comparison() {
332     std::cout << "\n" << std::string(70, '=') << "\n";
333     std::cout << "==== SECTION 4: When to Use What? ===\n";
334     std::cout << std::string(70, '=') << "\n\n";
335
336     std::cout << "                                     \n";
337     std::cout << "     MECHANISM          USE CASE          \n";
338     std::cout << "                                     \n";
339     std::cout << "     std::async  •      Simplest way to run async tasks\n";
340     std::cout << "     \n";
341     std::cout << "     •                  Fire-and-forget operations\n";
342     std::cout << "     \n";
343     std::cout << "     •                  Quick parallel computations\n";
344     std::cout << "     \n";
345     std::cout << "     •                  Don't need thread control\n";
346     std::cout << "     \n";
347     std::cout << "     •                  Best for: Simple async operations\n";
348     std::cout << "     \n";
349     std::cout << "     promise/future  •  Producer-consumer pattern\n";
350     std::cout << "     \n";

```

```

346     std::cout << "    •          Set value from different location
347     std::cout << "    •          Complex synchronization
348     std::cout << "    •          \n";
349     std::cout << "    •          Multiple consumers (shared_future)
350     std::cout << "    •          \n";
351     std::cout << "    •          packaged_task •      Thread pools / task queues
352     std::cout << "    •          \n";
353     std::cout << "    •          Separate creation from execution
354     std::cout << "    •          \n";
355     std::cout << "    •          Store tasks in containers
356     std::cout << "    •          \n";
357     std::cout << "    •          Custom scheduling systems
358     std::cout << "    •          \n";
359     std::cout << "    •          Best for: Building task systems
360     std::cout << "    •          \n";
361     std::cout << "    •          \n";
362
363 // SECTION 5: Relationship with ASIO
364 // =====
365
366 void explain_asio_relationship() {
367     std::cout << "\n" << std::string(70, '=') << "\n";
368     std::cout << "==== SECTION 5: Are They Alternatives to ASIO? ====\n";
369     std::cout << std::string(70, '=') << "\n\n";
370
371     std::cout << "    •          SHORT ANSWER: NO - They serve DIFFERENT purposes!\n\n";
372
373     std::cout << "    •          \n";
374     std::cout << "    •          future/promise/async (Standard Library Concurrency)
375     std::cout << "    •          \n";
376     std::cout << "    •          PURPOSE:          \n";
377
378     std::cout << "    •          Task-based concurrency          \n";
379     std::cout << "    •          CPU-bound computations          \n";
380     std::cout << "    •          Running functions asynchronously          \n";
381     std::cout << "    •          Getting results from other threads          \n";
382
383     std::cout << "    •          \n";
384     std::cout << "    •          EXAMPLES:          \n";

```

```
380     std::cout << " • Parallel calculations\n";
381     std::cout << " • Image processing in background\n";
382     std::cout << " • File compression\n";
383     std::cout << " • Any CPU-intensive work\n";
384     std::cout << "\n";
385     std::cout << " LIMITATIONS:\n";
386     std::cout << " • Not designed for I/O operations\n";
387     std::cout << " • No built-in event loop\n";
388     std::cout << " • No socket/network abstractions\n";
389     std::cout << " • Limited scalability for many I/O operations\n";
390     std::cout << "\n\n";
391
392     std::cout << " ASIO (Asynchronous I/O Library)\n";
393     std::cout << " • PURPOSE:\n";
394     std::cout << " • Asynchronous I/O operations\n";
395     std::cout << " • Network programming (sockets, TCP/UDP)\n";
396     std::cout << " • Event-driven architecture\n";
397     std::cout << " • Timers and signals\n";
398     std::cout << " • Serial ports\n";
399
400     std::cout << "\n";
401     std::cout << " EXAMPLES:\n";
402     std::cout << " • HTTP/REST servers and clients\n";
403     std::cout << " • WebSocket servers\n";
404     std::cout << " • Chat applications\n";
405     std::cout << " • Real-time data streaming\n";
406     std::cout << " • Timers and periodic tasks\n";
407     std::cout << "\n";
408     std::cout << "\n";
```

```
409     std::cout << "    STRENGTHS:                                     \n";
410     std::cout << "        Optimized for I/O-bound operations      \n";
411     std::cout << "        Event loop / io_context                   \n";
412     std::cout << "        Handles thousands of connections efficiently \n";
413     std::cout << "        Cross-platform I/O abstractions           \n";
414     std::cout << "        Proactor pattern                          \n";
415     std::cout << "                                         \n\n";
416
417     std::cout << "                                         \n";
418     std::cout << "    THEY ARE COMPLEMENTARY - Use Together!      \n";
419     std::cout << "                                         \n";
420     std::cout << "                                         \n";
421     std::cout << "    EXAMPLE 1: ASIO for I/O + async for CPU work \n";
422     std::cout << "        • Use ASIO to handle HTTP requests          \n";
423     std::cout << "        • Use std::async to process images in parallel \n";
424     std::cout << "        • Return result via ASIO response          \n";
425     std::cout << "                                         \n";
426     std::cout << "    EXAMPLE 2: ASIO for networking + packaged_task for jobs \n";
427     std::cout << "        • ASIO receives network messages           \n";
428     std::cout << "        • Queue CPU-intensive tasks with packaged_task \n";
429     std::cout << "        • Thread pool processes tasks              \n";
430     std::cout << "        • ASIO sends results back over network     \n";
431     std::cout << "                                         \n";
432     std::cout << "    EXAMPLE 3: ASIO timers + promise/future for results \n";
433     std::cout << "        • ASIO timer for periodic checks          \n";
434     std::cout << "        • Use promise/future to coordinate between timers \n";
435     std::cout << "        • Combine I/O events with task results     \n";
436     std::cout << "                                         \n";
437     std::cout << "                                         \n\n";
```

```
438     std::cout << "DECISION TREE:\n";
439     std::cout << "    Need async I/O (network, files, timers)?    → Use ASIO\n";
440     std::cout << "    \n";
441     std::cout << "    Need CPU-bound parallel computation?    → Use std::async\n";
442     std::cout << "    \n";
443     std::cout << "    Need producer-consumer pattern?    → Use promise/\n";
444     std::cout << "    future\n";
445     std::cout << "    \n";
446     std::cout << "    Building thread pool / task queue?    → Use\n";
447     std::cout << "    packaged_task\n";
448     std::cout << "    \n";
449     std::cout << "    Need both I/O AND computation?    → Use ASIO + future\n";
450     std::cout << "    /async\n";
451 }
452 // -----
453 // SECTION 6: Practical Example - Combining Them
454 // -----
455
456 namespace combined_example {
457
458 // Simulate a web server scenario
459 class RequestProcessor {
460 public:
461     // Simulate receiving HTTP request (would use ASIO in real app)
462     static std::string receive_request() {
463         std::cout << "    [ASIO would be here] Received HTTP request\n";
464         return "process_image:photo.jpg";
465     }
466
467     // CPU-intensive image processing (use std::async)
468     static std::string process_image(const std::string& filename) {
469         std::cout << "    [Worker] Processing image: " << filename << "\n";
470         std::this_thread::sleep_for(1s); // Simulate heavy processing
471         return "processed_" + filename;
472     }
473
474     // Send response (would use ASIO in real app)
475     static void send_response(const std::string& result) {
476         std::cout << "    [ASIO would be here] Sending response: " << result <<
477         "\n";
478     }
479 };
480
481 void demonstrate() {
482     std::cout << "\n" << std::string(70, '=') << "\n";
483     std::cout << "==== SECTION 6: Practical Example - ASIO + async ===\n";
484     std::cout << std::string(70, '=') << "\n\n";
```

```
483     std::cout << "Scenario: Web server handling image processing requests\n\n";
484     ;
485
486     // 1. ASIO would receive request (simulated)
487     std::string request = RequestProcessor::receive_request();
488
489     // 2. Parse request
490     auto pos = request.find(':');
491     std::string filename = request.substr(pos + 1);
492
493     // 3. Use std::async for CPU-intensive work (doesn't block I/O thread)
494     std::cout << "  Launching async image processing...\n";
495     auto future = std::async(std::launch::async,
496                             RequestProcessor::process_image,
497                             filename);
498
499     std::cout << "  ASIO thread is free to handle other requests!\n";
500     std::this_thread::sleep_for(200ms);
501     std::cout << "  (Handling other requests...)\n";
502     std::this_thread::sleep_for(300ms);
503
504     // 4. Get result and send response
505     std::cout << "  Waiting for processing to complete...\n";
506     std::string result = future.get();
507
508     // 5. ASIO would send response (simulated)
509     RequestProcessor::send_response(result);
510
511     std::cout << "\n  This is how you combine them:\n";
512     std::cout << "  •  ASIO handles network I/O (non-blocking)\n";
513     std::cout << "  •  std::async handles CPU work (parallel)\n";
514     std::cout << "  •  Best of both worlds!\n";
515 }
516
517 } // namespace combined_example
518 //
519 // =====
520 // MAIN FUNCTION
521 //
522 // =====
523
524 int main() {
525     std::cout << "\n";
526     std::cout << "                               \n";
527     std::cout << "           C++ Concurrency: future, promise, async, packaged_task\n";
528     std::cout << "                               \n";
529     std::cout << "           And their relationship with ASIO\n";
530     std::cout << "                               \n";
531     std::cout << "                               \n";
```

```
530 // Section 1: std::async
531 async_examples::demonstrate();
532
533 // Section 2: promise/future
534 promise_future_examples::demonstrate();
535
536 // Section 3: packaged_task
537 packaged_task_examples::demonstrate();
538
539 // Section 4: Comparison
540 show_comparison();
541
542 // Section 5: ASIO relationship
543 explain_asio_relationship();
544
545 // Section 6: Combined example
546 combined_example::demonstrate();
547
548 std::cout << "\n" << std::string(70, '=') << "\n";
549 std::cout << "All demonstrations completed!\n";
550 std::cout << std::string(70, '=') << "\n\n";
551
552 std::cout << "KEY TAKEAWAYS:\n";
553 std::cout << "1. future/promise/async are for TASK-BASED concurrency\n";
554 std::cout << "2. ASIO is for ASYNCHRONOUS I/O and event-driven programming
      \n";
555 std::cout << "3. They are NOT alternatives - use them TOGETHER!\n";
556 std::cout << "4. std::async: simplest for parallel tasks\n";
557 std::cout << "5. promise/future: manual control, producer-consumer\n";
558 std::cout << "6. packaged_task: thread pools, task queues\n";
559 std::cout << "7. ASIO: network I/O, timers, scalable servers\n\n";
560
561 return 0;
562 }
```

## 36 Source Code: GenericLambdas.cpp

File: src/GenericLambdas.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <algorithm>
5 #include <numeric>
6 #include <functional>
7 #include <map>
8 #include <memory>
9
10 // =====
11 // 1. BASIC GENERIC LAMBDA (C++14)
12 // =====
13 void example_basic_generic_lambda() {
14     std::cout << "\n== 1. BASIC GENERIC LAMBDA (C++14) ==" << std::endl;
15
16     // Generic lambda that works with any type
17     auto identity = [] (auto x) {
18         return x;
19     };
20
21     std::cout << "Identity int: " << identity(42) << std::endl;
22     std::cout << "Identity double: " << identity(3.14) << std::endl;
23     std::cout << "Identity string: " << identity(std::string("Hello")) << std::endl;
24     std::cout << "Identity char: " << identity('A') << std::endl;
25 }
26
27 // =====
28 // 2. GENERIC LAMBDA WITH TYPE OPERATIONS
29 // =====
30 void example_generic_lambda_operations() {
31     std::cout << "\n== 2. GENERIC LAMBDA WITH TYPE OPERATIONS ==" << std::endl;
32
33     // Generic lambda that adds two values
34     auto add = [] (auto a, auto b) {
35         return a + b;
36     };
37
38     std::cout << "Add ints: " << add(10, 20) << std::endl;
39     std::cout << "Add doubles: " << add(3.14, 2.86) << std::endl;
40     std::cout << "Add strings: " << add(std::string("Hello "), std::string("World")) << std::endl;
41
42     // Generic lambda for multiplication
43     auto multiply = [] (auto a, auto b) {
44         return a * b;
45     };
46 }
```

```
47     std::cout << "Multiply ints: " << multiply(5, 6) << std::endl;
48     std::cout << "Multiply doubles: " << multiply(2.5, 4.0) << std::endl;
49 }
50
51 // =====
52 // 3. GENERIC LAMBDA WITH CONTAINERS
53 // =====
54 void example_generic_lambda_containers() {
55     std::cout << "\n==== 3. GENERIC LAMBDA WITH CONTAINERS ===" << std::endl;
56
57     // Generic lambda to print container elements
58     auto print_container = [] (const auto& container, const std::string& name)
59     {
60         std::cout << name << ": ";
61         for (const auto& elem : container) {
62             std::cout << elem << " ";
63         }
64         std::cout << std::endl;
65     };
66
67     std::vector<int> vec = {1, 2, 3, 4, 5};
68     std::vector<std::string> vec_str = {"Hello", "World", "C++14"};
69
70     print_container(vec, "Vector<int>");
71     print_container(vec_str, "Vector<string>");
72 }
73
74 // =====
75 // 4. GENERIC LAMBDA WITH STL ALGORITHMS
76 // =====
77 void example_generic_lambda_algorithms() {
78     std::cout << "\n==== 4. GENERIC LAMBDA WITH STL ALGORITHMS ===" << std::endl;
79
80     std::vector<int> numbers = {5, 2, 8, 1, 9, 3};
81
82     // Generic lambda for comparison
83     auto greater_than = [] (auto a, auto b) {
84         return a > b;
85     };
86
87     std::sort(numbers.begin(), numbers.end(), greater_than);
88
89     std::cout << "Sorted (descending): ";
90     for (auto n : numbers) {
91         std::cout << n << " ";
92     }
93     std::cout << std::endl;
94
95     // Generic lambda for transformation
96     auto square = [] (auto x) {
97         return x * x;
98     };

```

```
99     std::vector<int> squared(numbers.size());
100    std::transform(numbers.begin(), numbers.end(), squared.begin(), square);
101
102    std::cout << "Squared: ";
103    for (auto n : squared) {
104        std::cout << n << " ";
105    }
106    std::cout << std::endl;
107 }
108
109 // =====
110 // 5. GENERIC LAMBDA WITH MULTIPLE AUTO PARAMETERS
111 // =====
112 void example_multiple_auto_parameters() {
113     std::cout << "\n==== 5. GENERIC LAMBDA WITH MULTIPLE AUTO PARAMETERS ==="
114         << std::endl;
115
116     // Generic lambda with three different types
117     auto combine = [] (auto a, auto b, auto c) {
118         std::cout << "Types can differ: " << a << ", " << b << ", " << c <<
119             std::endl;
120         return true;
121     };
122
123     combine(42, 3.14, "Hello");
124     combine(std::string("World"), 100, 'X');
125 }
126
127 // =====
128 // 6. GENERIC LAMBDA WITH VARIADIC PARAMETERS
129 // =====
130 void example_generic_lambda_variadic() {
131     std::cout << "\n==== 6. GENERIC LAMBDA WITH VARIADIC PARAMETERS ===" << std::
132         ::endl;
133
134     // Generic lambda that sums any number of arguments
135     auto sum_all = [] (auto... args) {
136         return (args + ...); // C++17 fold expression
137     };
138
139     std::cout << "Sum of ints: " << sum_all(1, 2, 3, 4, 5) << std::endl;
140     std::cout << "Sum of doubles: " << sum_all(1.1, 2.2, 3.3) << std::endl;
141     std::cout << "Sum mixed: " << sum_all(1, 2.5, 3) << std::endl;
142 }
143
144 // =====
145 // 7. GENERIC LAMBDA FOR FACTORY PATTERN
146 // =====
147 void example_generic_lambda_factory() {
148     std::cout << "\n==== 7. GENERIC LAMBDA FOR FACTORY PATTERN ===" << std::
149         endl;
150
151     // Generic factory lambda
152     auto make_unique_ptr = [] (auto value) {
```

```
149     using T = decltype(value);
150     return std::make_unique<T>(value);
151 }
152
153 auto int_ptr = make_unique_ptr(42);
154 auto str_ptr = make_unique_ptr(std::string("Hello"));
155 auto double_ptr = make_unique_ptr(3.14);
156
157 std::cout << "Int unique_ptr: " << *int_ptr << std::endl;
158 std::cout << "String unique_ptr: " << *str_ptr << std::endl;
159 std::cout << "Double unique_ptr: " << *double_ptr << std::endl;
160 }
161
162 // =====
163 // 8. GENERIC LAMBDA WITH MAP OPERATIONS
164 // =====
165 void example_generic_lambda_map() {
166     std::cout << "\n== 8. GENERIC LAMBDA WITH MAP OPERATIONS ==" << std::endl;
167
168     std::map<std::string, int> age_map = {
169         {"Alice", 30},
170         {"Bob", 25},
171         {"Charlie", 35}
172     };
173
174     // Generic lambda to print key-value pairs
175     auto print_map = [] (const auto& map, const std::string& name) {
176         std::cout << name << ":" << std::endl;
177         for (const auto& [key, value] : map) {
178             std::cout << " " << key << " -> " << value << std::endl;
179         }
180     };
181
182     print_map(age_map, "Age Map");
183
184     // Generic lambda for filtering
185     auto filter_by_value = [] (const auto& map, auto threshold) {
186         std::vector<typename std::decay<decltype(map)>::type::key_type> result
187         ;
188         for (const auto& [key, value] : map) {
189             if (value > threshold) {
190                 result.push_back(key);
191             }
192         }
193         return result;
194     };
195
196     auto filtered = filter_by_value(age_map, 28);
197     std::cout << "People older than 28: ";
198     for (const auto& name : filtered) {
199         std::cout << name << " ";
200     }
201     std::cout << std::endl;
```

```
201 }
202
203 // =====
204 // 9. GENERIC LAMBDA AS PREDICATE
205 // =====
206 void example_generic_lambda_predicate() {
207     std::cout << "\n==== 9. GENERIC LAMBDA AS PREDICATE ===" << std::endl;
208
209     std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
210
211     // Generic lambda predicate
212     auto is_even = [] (auto n) {
213         return n % 2 == 0;
214     };
215
216     auto count = std::count_if(numbers.begin(), numbers.end(), is_even);
217     std::cout << "Even numbers count: " << count << std::endl;
218
219     // Find first even number
220     auto it = std::find_if(numbers.begin(), numbers.end(), is_even);
221     if (it != numbers.end()) {
222         std::cout << "First even number: " << *it << std::endl;
223     }
224 }
225
226 // =====
227 // 10. GENERIC LAMBDA WITH RETURN TYPE DEDUCTION
228 // =====
229 void example_return_type_deduction() {
230     std::cout << "\n==== 10. GENERIC LAMBDA WITH RETURN TYPE DEDUCTION ===" <<
231         std::endl;
232
233     // Lambda with auto return type - deduced from expression
234     auto double_value = [] (auto x) {
235         return x * 2;
236     };
237
238     // Lambda with explicit return type
239     auto to_string_value = [] (auto x) -> std::string {
240         return std::to_string(x);
241     };
242
243     std::cout << "Double int: " << double_value(21) << std::endl;
244     std::cout << "Double double: " << double_value(3.14) << std::endl;
245     std::cout << "To string (int): " << to_string_value(42) << std::endl;
246     std::cout << "To string (double): " << to_string_value(3.14159) << std::endl;
247 }
248
249 // =====
250 // MAIN FUNCTION
251 // =====
252 int main() {
253     std::cout << "\n======" << std::endl;
```

```
253     endl;
254     std::cout << "      C++14 GENERIC LAMBDAS EXAMPLES" << std::endl;
255     std::cout << "===== ===== ===== ===== =====" << std::
256     endl;
257
258     example_basic_generic_lambda();
259     example_generic_lambda_operations();
260     example_generic_lambda_containers();
261     example_generic_lambda_algorithms();
262     example_multiple_auto_parameters();
263     example_generic_lambda_variadic();
264     example_generic_lambda_factory();
265     example_generic_lambda_map();
266     example_generic_lambda_predicate();
267     example_return_type_deduction();
268
269     std::cout << "\n===== ===== ===== ===== =====" << std::
270     endl;
271     std::cout << "      ALL EXAMPLES COMPLETED" << std::endl;
272     std::cout << "===== ===== ===== ===== =====\n" << std::
273     endl;
274
275     return 0;
276 }
```

## 37 Source Code: InheritanceTypes.cpp

File: src/InheritanceTypes.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <memory>
5 #include <algorithm>
6
7 // =====
8 // PRIVATE, PROTECTED, AND PUBLIC INHERITANCE IN C++
9 // =====
10 // =====
11 // 1. PUBLIC INHERITANCE - "IS-A" RELATIONSHIP
12 // =====
13
14
15 class Animal {
16 protected:
17     std::string name;
18
19 public:
20     Animal(const std::string& n) : name(n) {}
21     virtual ~Animal() = default;
22
23     virtual void makeSound() const {
24         std::cout << name << " makes a sound" << std::endl;
25     }
26
27     void eat() const {
28         std::cout << name << " is eating" << std::endl;
29     }
30 };
31
32 // Public inheritance: Dog IS-A Animal
33 // All public members of Animal remain public in Dog
34 class Dog : public Animal {
35 public:
36     Dog(const std::string& n) : Animal(n) {}
37
38     void makeSound() const override {
39         std::cout << name << " barks: Woof!" << std::endl;
40     }
41 };
42
43 void example_public_inheritance() {
44     std::cout << "\n== 1. PUBLIC INHERITANCE (IS-A) ==" << std::endl;
45     std::cout << "Use case: Dog IS-A Animal (polymorphism, substitutability)\n"
46     " << std::endl;
47
48     Dog dog("Buddy");
49     dog.makeSound(); // Can call inherited method
```

```
49     dog.eat();           //  Can call inherited method
50
51 //  Polymorphism works with public inheritance
52 Animal* animal_ptr = &dog;
53 animal_ptr->makeSound(); // Calls Dog::makeSound()
54
55 std::cout << "\n PUBLIC INHERITANCE:" << std::endl;
56 std::cout << " • Dog IS-A Animal" << std::endl;
57 std::cout << " • Public members stay public" << std::endl;
58 std::cout << " • Protected members stay protected" << std::endl;
59 std::cout << " • Polymorphism works (can upcast to base)" << std::endl;
60 std::cout << " • Use for: Substitutability, \"is-a\" relationships" <<
61     std::endl;
62 }
63
64 // =====
65 // 2. PRIVATE INHERITANCE - "IMPLEMENTED-IN-TERMS-OF"
66 // =====
67
68 // Base class providing timer functionality
69 class Timer {
70 private:
71     int ticks = 0;
72
73 public:
74     void tick() { ++ticks; }
75     void reset() { ticks = 0; }
76     int getTicks() const { return ticks; }
77
78     void display() const {
79         std::cout << "Timer: " << ticks << " ticks" << std::endl;
80     }
81 };
82
83 // Private inheritance: Widget is IMPLEMENTED using Timer
84 // Timer's public interface becomes private in Widget
85 class Widget : private Timer { // "implemented-in-terms-of"
86 private:
87     std::string id;
88
89 public:
90     Widget(const std::string& i) : id(i) {}
91
92     // Widget controls which Timer methods to expose
93     void update() {
94         tick(); // Can call Timer::tick() internally
95         std::cout << "Widget " << id << " updated" << std::endl;
96     }
97
98     void resetState() {
99         reset(); // Can call Timer::reset() internally
100    }
101
102 // Expose only specific functionality
```

```
102     void showUpdateCount() const {
103         std::cout << "Widget " << id << " has been updated "
104                         << getTicks() << " times" << std::endl;
105     }
106
107     // Note: Timer::display() is NOT exposed
108 };
109
110 void example_private_inheritance() {
111     std::cout << "\n==== 2. PRIVATE INHERITANCE (IMPLEMENTED-IN-TERMS-OF) ==="
112                         << std::endl;
113     std::cout << "Use case: Widget is IMPLEMENTED using Timer (code reuse, not
114                         substitutability)\n" << std::endl;
115
116     Widget widget("W1");
117     widget.update();
118     widget.update();
119     widget.update();
120     widget.showUpdateCount();
121
122     // Cannot access Timer's public methods from outside
123     // widget.tick();      // Compile error! tick() is private in Widget
124     // widget.display();  // Compile error! display() is private in Widget
125
126     // Cannot upcast to Timer (no polymorphism)
127     // Timer* timer_ptr = &widget; // Compile error! Private inheritance
128                         breaks is-a
129
130     std::cout << "\n PRIVATE INHERITANCE:" << std::endl;
131     std::cout << " • Widget is IMPLEMENTED using Timer" << std::endl;
132     std::cout << " • All base members become private" << std::endl;
133     std::cout << " • No polymorphism (cannot upcast)" << std::endl;
134     std::cout << " • Base class is an implementation detail" << std::endl;
135     std::cout << " • Use for: Code reuse without exposing interface" << std
136                         ::endl;
137 }
138
139 // =====
140 // 3. PROTECTED INHERITANCE - "IMPLEMENTED-IN-TERMS-OF" FOR FURTHER DERIVATION
141 // =====
142
143 class Logger {
144 public:
145     void log(const std::string& msg) const {
146         std::cout << "[LOG] " << msg << std::endl;
147     }
148
149     void debug(const std::string& msg) const {
150         std::cout << "[DEBUG] " << msg << std::endl;
151     }
152 };
153
154 // Protected inheritance: Component uses Logger, allows derived classes to
155                         access it
```

```
151 class Component : protected Logger {
152 protected:
153     std::string name;
154
155 public:
156     Component(const std::string& n) : name(n) {
157         log("Component " + name + " created"); // Can use Logger internally
158     }
159
160     void doWork() {
161         debug("Component " + name + " is working");
162     }
163 };
164
165 // Derived class can access Logger through protected inheritance
166 class AdvancedComponent : public Component {
167 public:
168     AdvancedComponent(const std::string& n) : Component(n) {
169         // Can access Logger methods because of protected inheritance
170         log("Advanced features initialized");
171     }
172
173     void advancedWork() {
174         debug("Advanced component processing"); // Logger accessible here
175         doWork();
176     }
177 };
178
179 void example_protected_inheritance() {
180     std::cout << "\n==== 3. PROTECTED INHERITANCE (IMPLEMENTED-IN-TERMS-OF +
181             DERIVATION) ===" << std::endl;
182     std::cout << "Use case: Component uses Logger, derived classes can also
183             use it\n" << std::endl;
184
185     AdvancedComponent comp("AC1");
186     comp.advancedWork();
187
188     // Cannot access Logger from outside
189     // comp.log("test"); // Compile error! log() is protected in Component
190     // comp.debug("test"); // Compile error! debug() is protected in Component
191
192     // Cannot upcast to Logger
193     // Logger* logger_ptr = &comp; // Compile error! Protected inheritance
194
195     std::cout << "\n PROTECTED INHERITANCE:" << std::endl;
196     std::cout << " • Component is IMPLEMENTED using Logger" << std::endl;
197     std::cout << " • All base public members become protected" << std::endl;
198     std::cout << " • Derived classes can access base interface" << std::endl
199         ;
200     std::cout << " • No polymorphism (cannot upcast)" << std::endl;
201     std::cout << " • Use for: Implementation detail shared with derived
202             classes" << std::endl;
203 }
```

```
201 // =====
202 // 4. REAL-WORLD EXAMPLE: STACK IMPLEMENTED USING VECTOR
203 // =====
204
205 // Private inheritance: Stack is implemented using vector
206 template<typename T>
207 class Stack : private std::vector<T> {
208     using Base = std::vector<T>;
209
210 public:
211     // Expose only stack operations
212     void push(const T& value) {
213         Base::push_back(value);
214     }
215
216     void push(T&& value) {
217         Base::push_back(std::move(value));
218     }
219
220     void pop() {
221         if (!Base::empty()) {
222             Base::pop_back();
223         }
224     }
225
226     const T& top() const {
227         return Base::back();
228     }
229
230     bool empty() const {
231         return Base::empty();
232     }
233
234     size_t size() const {
235         return Base::size();
236     }
237
238     // Note: We DON'T expose vector's random access, insert, erase, etc.
239     // This enforces stack semantics (LIFO)
240 };
241
242 void example_stack_private_inheritance() {
243     std::cout << "\n==== 4. REAL-WORLD: STACK USING PRIVATE INHERITANCE ===" <<
244         std::endl;
245     std::cout << "Use case: Implement Stack using vector, but hide vector's
246         interface\n" << std::endl;
247
248     Stack<int> stack;
249     stack.push(10);
250     stack.push(20);
251     stack.push(30);
252
253     std::cout << "Stack top: " << stack.top() << std::endl;
254     std::cout << "Stack size: " << stack.size() << std::endl;
```

```
253     stack.pop();
254     std::cout << "After pop, top: " << stack.top() << std::endl;
255
256     // Vector operations are hidden
257     // stack[0] = 100;           // Compile error! operator[] not accessible
258     // stack.insert(...);      // Compile error! insert() not accessible
259     // stack.at(0);           // Compile error! at() not accessible
260
261     std::cout << "\n BENEFIT:" << std::endl;
262     std::cout << " • Reuses vector's implementation" << std::endl;
263     std::cout << " • Enforces stack semantics (LIFO only)" << std::endl;
264     std::cout << " • Users cannot break stack invariants" << std::endl;
265 }
266
267 // =====
268 // 5. COMPARISON: PRIVATE INHERITANCE VS COMPOSITION
269 // =====
270
271 // Option 1: Private inheritance
272 class WindowPrivate : private Timer {
273 public:
274     void refresh() {
275         tick();
276         std::cout << "Window refreshed " << getTicks() << " times" << std::endl;
277     }
278 };
279
280 // Option 2: Composition (often preferred)
281 class WindowComposition {
282 private:
283     Timer timer; // Has-a Timer
284
285 public:
286     void refresh() {
287         timer.tick();
288         std::cout << "Window refreshed " << timer.getTicks() << " times" << std::endl;
289     }
290 };
291
292 void example_private_vs_composition() {
293     std::cout << "\n== 5. PRIVATE INHERITANCE VS COMPOSITION ==" << std::endl;
294
295     std::cout << "\nPrivate inheritance:" << std::endl;
296     WindowPrivate win1;
297     win1.refresh();
298     win1.refresh();
299
300     std::cout << "\nComposition:" << std::endl;
301     WindowComposition win2;
302     win2.refresh();
```

```
304     win2.refresh();
305
306     std::cout << "\n WHEN TO USE PRIVATE INHERITANCE:" << std::endl;
307     std::cout << "      Need to override virtual functions from base" << std::endl;
308     std::cout << "      Need access to protected members of base" << std::endl;
309     std::cout << "      Want Empty Base Optimization (EBO) for zero-size bases"
310           << std::endl;
311
312     std::cout << "\n WHEN TO USE COMPOSITION (USUALLY PREFERRED):" << std::endl;
313     std::cout << "      Don't need to override virtual functions" << std::endl;
314     std::cout << "      Don't need access to protected members" << std::endl;
315     std::cout << "      Want to hold multiple instances" << std::endl;
316     std::cout << "      Want more explicit \"has-a\" relationship" << std::endl;
317           ;
318     std::cout << "      Better encapsulation and flexibility" << std::endl;
319 }
320
321 // =====
322 // 6. OVERRIDING VIRTUAL FUNCTIONS WITH PRIVATE INHERITANCE
323 // =====
324
325 class Observable {
326 public:
327     virtual ~Observable() = default;
328
329     virtual void onEvent() {
330         std::cout << "Observable: Event occurred" << std::endl;
331     }
332
333     void triggerEvent() {
334         std::cout << "Triggering event..." << std::endl;
335         onEvent();
336     }
337 };
338
339 // Private inheritance to customize behavior
340 class Sensor : private Observable {
341 private:
342     std::string sensorId;
343     int reading = 0;
344
345     // Override virtual function from Observable
346     void onEvent() override {
347         reading++;
348         std::cout << "Sensor " << sensorId << " received event, reading: " <<
349             reading << std::endl;
350     }
351
352 public:
353     Sensor(const std::string& id) : sensorId(id) {}
354
355     void simulate() {
```

```
353     // Call base class method that will call our overridden onEvent()
354     triggerEvent();
355 }
356 };
357
358 void example_override_with_private() {
359     std::cout << "\n==== 6. OVERRIDING VIRTUAL FUNCTIONS (PRIVATE INHERITANCE)
360     ===" << std::endl;
361     std::cout << "Use case: Customize base class behavior without exposing
362     base interface\n" << std::endl;
363
364     Sensor sensor("TEMP-01");
365     sensor.simulate();
366     sensor.simulate();
367     sensor.simulate();
368
369     std::cout << "\n KEY POINT:" << std::endl;
370     std::cout << " • Can override virtual functions even with private
371     inheritance" << std::endl;
372     std::cout << " • Base class doesn't need to know about derived class" <<
373     std::endl;
374     std::cout << " • This is the main reason to use private inheritance over
375     composition" << std::endl;
376 }
377
378 // =====
379 // 7. USING DECLARATION TO SELECTIVELY EXPOSE MEMBERS
380 // =====
381
382 class Engine {
383 public:
384     void start() { std::cout << "Engine started" << std::endl; }
385     void stop() { std::cout << "Engine stopped" << std::endl; }
386     void diagnose() { std::cout << "Running diagnostics..." << std::endl; }
387     void internalMaintenance() { std::cout << "Internal maintenance" << std::
388         endl; }
389 };
390
391 class Car : private Engine {
392 public:
393     // Selectively expose only specific Engine methods
394     using Engine::start; // Make start() public
395     using Engine::stop; // Make stop() public
396
397     // Note: diagnose() and internalMaintenance() remain private
398
399     void drive() {
400         start();
401         std::cout << "Car is driving" << std::endl;
402     }
403 };
404
405 void example_using_declaration() {
406     std::cout << "\n==== 7. USING DECLARATION - SELECTIVE EXPOSURE ===" << std
```

```
        ::endl;
401 std::cout << "Use case: Expose only specific base class methods\n" << std
        ::endl;

402
403 Car car;
404 car.start();      // Works - exposed via using declaration
405 car.drive();
406 car.stop();      // Works - exposed via using declaration
407
408 // These are not exposed
409 // car.diagnose();           // Compile error!
410 // car.internalMaintenance(); // Compile error!

411
412 std::cout << "\n BENEFIT:" << std::endl;
413 std::cout << " • Fine-grained control over interface" << std::endl;
414 std::cout << " • Can expose some methods while hiding others" << std::
        endl;
415 std::cout << " • More maintainable than forwarding each method manually"
        << std::endl;
416 }
417
418 // =====
419 // 8. ACCESS LEVEL SUMMARY TABLE
420 // =====
421
422 void example_access_summary() {
423     std::cout << "\n== 8. ACCESS LEVEL SUMMARY ==" << std::endl;
424
425     std::cout << "\n" << std::endl;
426     std::cout << " Base Class Member" << " Public" << " Protected"
427         " Private" << std::endl;
428     std::cout << " Access Level" << " Inheritance" << " Inheritance"
429         " Inheritance" << std::endl;
430     std::cout << " " << std::endl;
431     std::cout << " public" << " public" << " protected"
432         " private" << std::endl;
433     std::cout << " protected" << " protected" << " protected"
434         " private" << std::endl;
435     std::cout << " private" << " (hidden)" << " (hidden)" << " ("
436         " hidden)" << std::endl;
437     std::cout << " " << std::endl;
438 }
439
440 // =====
441 // MAIN FUNCTION
442 // =====
443
444 int main() {
445     std::cout << "\n"
446         "===== PRIVATE, PROTECTED, AND PUBLIC INHERITANCE IN C++ =====" <<
447         std::endl;
448     std::cout << " PRIVATE, PROTECTED, AND PUBLIC INHERITANCE IN C++" << std
449         ::endl;
450     std::cout << "
```

```
=====
        std::endl;

443   example_public_inheritance();
444   example_private_inheritance();
445   example_protected_inheritance();
446   example_stack_private_inheritance();
447   example_private_vs_composition();
448   example_override_with_private();
449   example_using_declaration();
450   example_access_summary();

452
453   std::cout << "\n"
454   =====
455   std::cout << "  DECISION GUIDE: WHICH INHERITANCE TYPE TO USE?" << std::endl;
456   std::cout << "  ====="
457   std::cout << "\n  USE PUBLIC INHERITANCE WHEN:" << std::endl;
458   std::cout << "  •  Derived class IS-A base class" << std::endl;
459   std::cout << "  •  You need polymorphism and substitutability" << std::endl;
460   std::cout << "  •  Liskov Substitution Principle applies" << std::endl;
461   std::cout << "  •  Example: Dog is-a Animal, Circle is-a Shape" << std::endl;
462   std::cout << "  •  Usage: ~95% of all inheritance cases" << std::endl;
463
464   std::cout << "\n  USE PRIVATE INHERITANCE WHEN:" << std::endl;
465   std::cout << "  •  Derived class is IMPLEMENTED using base class" << std::endl;
466   std::cout << "  •  Need to override virtual functions from base" << std::endl;
467   std::cout << "  •  Need access to protected members of base" << std::endl;
468   std::cout << "  •  Want Empty Base Optimization (EBO)" << std::endl;
469   std::cout << "  •  Example: Stack implemented using vector" << std::endl;
470   std::cout << "  •  Usage: <5% of inheritance cases" << std::endl;
471   std::cout << "  •  Alternative: Prefer composition if possible" << std::endl;
472
473   std::cout << "\n  USE PROTECTED INHERITANCE WHEN:" << std::endl;
474   std::cout << "  •  Same as private inheritance, but..." << std::endl;
475   std::cout << "  •  Further derived classes need access to base interface"
476   << std::endl;
476   std::cout << "  •  Creating a hierarchy of implementation details" << std::endl;
477   std::cout << "  •  Example: Component hierarchy sharing Logger" << std::endl;
478   std::cout << "  •  Usage: <1% of inheritance cases (very rare)" << std::endl;
479   std::cout << "  •  Alternative: Usually better to use composition" << std::endl;
```

```
480
481     std::cout << "\n COMPOSITION VS PRIVATE INHERITANCE:" << std::endl;
482     std::cout << " • Default to COMPOSITION (has-a relationship)" << std::
483         endl;
484     std::cout << " • Use private inheritance ONLY if:" << std::endl;
485     std::cout << "     - Need to override virtual functions, OR" << std::endl;
486     std::cout << "     - Need access to protected members, OR" << std::endl;
487     std::cout << "     - Need Empty Base Optimization" << std::endl;
488     std::cout << " • Composition is more explicit and flexible" << std::endl
489         ;
490
491     std::cout << "\n MODERN C++ BEST PRACTICES:" << std::endl;
492     std::cout << " • Public inheritance: Use for polymorphism" << std::endl;
493     std::cout << " • Private inheritance: Rare, prefer composition" << std::
494         endl;
495     std::cout << " • Protected inheritance: Almost never use" << std::endl;
496     std::cout << " • If unsure: Choose composition over private inheritance"
497         << std::endl;
498     std::cout << " • Scott Meyers: \"Prefer composition to private
499         inheritance\""
500         << std::endl;
501
502     std::cout << "\n"
503         =====\n"
504         << std::endl;
505
506     return 0;
507 }
```

## 38 Source Code: InsertAndDeleteNodes.cpp

File: src/InsertAndDeleteNodes.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 using namespace std;
3
4 struct ListNode {
5     int val;
6     ListNode *next;
7     ListNode(int x) : val(x), next(NULL) {}
8 };
9
10 class Solution
11 {
12 public:
13     void PrintListNode(ListNode* inListNode) {
14         if (inListNode != NULL) {
15             cout << inListNode->val << " -> ";
16             PrintListNode(inListNode->next);
17         } else {
18             cout << "NULL" << endl;
19         }
20     }
21
22     void Test(ListNode* inListNode) {
23         delete inListNode;
24         inListNode = NULL;
25         cout << inListNode << endl;
26
27         PrintListNode(inListNode);
28     }
29
30     // Insert node at the beginning of the list
31     ListNode* InsertNodeAtHead(ListNode* pHead, int value) {
32         cout << "Entry: InsertNodeAtHead(value=" << value << ")" << endl;
33
34         ListNode* pNewNode = new ListNode(value);
35         pNewNode->next = pHead;
36
37         cout << "Exit: InsertNodeAtHead" << endl;
38         return pNewNode; // New head
39     }
40
41     // Insert node at the end of the list
42     ListNode* InsertNodeAtTail(ListNode* pHead, int value) {
43         cout << "Entry: InsertNodeAtTail(value=" << value << ")" << endl;
44
45         ListNode* pNewNode = new ListNode(value);
46
47         // Empty list case
48         if (pHead == NULL) {
49             cout << "Exit: InsertNodeAtTail (empty list)" << endl;
```

```
50         return pNewNode;
51     }
52
53     // Traverse to the end
54     ListNode* pCur = pHead;
55     while (pCur->next != NULL) {
56         pCur = pCur->next;
57     }
58
59     pCur->next = pNewNode;
60
61     cout << "Exit: InsertNodeAtTail" << endl;
62     return pHead;
63 }
64
65 // Insert node after a specific node
66 void InsertNodeAfter(ListNode* pPrevNode, int value) {
67     cout << "Entry: InsertNodeAfter(value=" << value << ")" << endl;
68
69     if (pPrevNode == NULL) {
70         cout << "Error: Previous node cannot be NULL" << endl;
71         return;
72     }
73
74     ListNode* pNewNode = new ListNode(value);
75     pNewNode->next = pPrevNode->next;
76     pPrevNode->next = pNewNode;
77
78     cout << "Exit: InsertNodeAfter" << endl;
79 }
80
81 // Insert node at a specific position (0-indexed)
82 ListNode* InsertNodeAtPosition(ListNode* pHead, int position, int value) {
83     cout << "Entry: InsertNodeAtPosition(pos=" << position << ", value="
84         << value << ")" << endl;
85
86     // Insert at head
87     if (position == 0) {
88         return InsertNodeAtHead(pHead, value);
89     }
90
91     // Traverse to position-1
92     ListNode* pCur = pHead;
93     for (int i = 0; i < position - 1 && pCur != NULL; i++) {
94         pCur = pCur->next;
95     }
96
97     if (pCur == NULL) {
98         cout << "Error: Position out of bounds" << endl;
99         return pHead;
100    }
101
102    InsertNodeAfter(pCur, value);
```

```
103     cout << "Exit: InsertNodeAtPosition" << endl;
104     return pHead;
105 }
106
107 // Insert node in sorted list (ascending order)
108 ListNode* InsertNodeSorted(ListNode* pHead, int value) {
109     cout << "Entry: InsertNodeSorted(value=" << value << ")" << endl;
110
111     ListNode* pNewNode = new ListNode(value);
112
113     // Empty list or insert at head
114     if (pHead == NULL || pHead->val >= value) {
115         pNewNode->next = pHead;
116         cout << "Exit: InsertNodeSorted (at head)" << endl;
117         return pNewNode;
118     }
119
120     // Find insertion point
121     ListNode* pCur = pHead;
122     while (pCur->next != NULL && pCur->next->val < value) {
123         pCur = pCur->next;
124     }
125
126     pNewNode->next = pCur->next;
127     pCur->next = pNewNode;
128
129     cout << "Exit: InsertNodeSorted" << endl;
130     return pHead;
131 }
132
133 void DeleteNode(ListNode* pInHead, ListNode* pToBeDeleted) {
134     cout << "Entry void DeleteNode(ListNode* pInHead, ListNode*"
135           pToBeDeleted)\n" << endl;
136
137     // Check NULL always
138     if (pInHead == NULL || pToBeDeleted == NULL) {
139         cout << "Abort void DeleteNode(ListNode* pInHead, ListNode*"
140           pToBeDeleted)\n" << endl;
141         return;
142     }
143
144     // PrintListNode(pInHead);
145
146     // Delete non-tail node including head node
147     if (pToBeDeleted->next != NULL) {
148         ListNode* pNext = pToBeDeleted->next;
149         pToBeDeleted->val = pNext->val;
150         pToBeDeleted->next = pNext->next;
151
152         delete pNext;
153         pNext = NULL;
154     } else { // Delete tail
155         ListNode* pPre = pInHead;
```

```
155     while (pPre->next != pToBeDeleted && pPre != NULL) {
156         pPre = pPre->next;
157     }
158     if (pPre == NULL)
159         return;
160
161     pPre->next = NULL;
162     delete pToBeDeleted;
163     pToBeDeleted = NULL;
164 }
165
166 cout << "Exit void DeleteNode(ListNode* pInHead, ListNode*  
    pToBeDeleted)\n" << endl;
167 }
168
169 // Delete node by value
170 ListNode* DeleteNodeByValue(ListNode* pHead, int value) {
171     cout << "Entry: DeleteNodeByValue(value=" << value << ")" << endl;
172
173     if (pHead == NULL) {
174         cout << "Error: Empty list" << endl;
175         return NULL;
176     }
177
178     // Delete head node
179     if (pHead->val == value) {
180         ListNode* pTemp = pHead;
181         pHead = pHead->next;
182         delete pTemp;
183         cout << "Exit: DeleteNodeByValue (deleted head)" << endl;
184         return pHead;
185     }
186
187     // Find and delete node
188     ListNode* pCur = pHead;
189     while (pCur->next != NULL && pCur->next->val != value) {
190         pCur = pCur->next;
191     }
192
193     if (pCur->next == NULL) {
194         cout << "Error: Value not found" << endl;
195         return pHead;
196     }
197
198     ListNode* pToDelete = pCur->next;
199     pCur->next = pCur->next->next;
200     delete pToDelete;
201
202     cout << "Exit: DeleteNodeByValue" << endl;
203     return pHead;
204 }
205
206 };
207
```

```
208 void TestInsertOperations() {
209     cout << "\n" << string(70, '=') << endl;
210     cout << "TESTING INSERT OPERATIONS\n";
211     cout << string(70, '=') << endl;
212
213     Solution solution;
214     ListNode* pHead = NULL;
215
216     // Test 1: Insert at head (empty list)
217     cout << "\n--- Test 1: Insert at head (empty list) ---" << endl;
218     pHead = solution.InsertNodeAtHead(pHead, 10);
219     solution.PrintListNode(pHead);
220
221     // Test 2: Insert at head (non-empty list)
222     cout << "\n--- Test 2: Insert at head (non-empty list) ---" << endl;
223     pHead = solution.InsertNodeAtHead(pHead, 5);
224     solution.PrintListNode(pHead);
225
226     // Test 3: Insert at tail
227     cout << "\n--- Test 3: Insert at tail ---" << endl;
228     pHead = solution.InsertNodeAtTail(pHead, 20);
229     pHead = solution.InsertNodeAtTail(pHead, 25);
230     solution.PrintListNode(pHead);
231
232     // Test 4: Insert after specific node
233     cout << "\n--- Test 4: Insert after second node ---" << endl;
234     solution.InsertNodeAfter(pHead->next, 15);
235     solution.PrintListNode(pHead);
236
237     // Test 5: Insert at position
238     cout << "\n--- Test 5: Insert at position 2 ---" << endl;
239     pHead = solution.InsertNodeAtPosition(pHead, 2, 12);
240     solution.PrintListNode(pHead);
241
242     // Clean up
243     while (pHead != NULL) {
244         ListNode* temp = pHead;
245         pHead = pHead->next;
246         delete temp;
247     }
248 }
249
250 void TestInsertSorted() {
251     cout << "\n" << string(70, '=') << endl;
252     cout << "TESTING INSERT IN SORTED LIST\n";
253     cout << string(70, '=') << endl;
254
255     Solution solution;
256     ListNode* pHead = NULL;
257
258     // Insert values in random order
259     int values[] = {30, 10, 50, 20, 40};
260
261     for (int val : values) {
```

```
262     cout << "\nInserting " << val << " into sorted list:" << endl;
263     pHead = solution.InsertNodeSorted(pHead, val);
264     solution.PrintListNode(pHead);
265 }
266
267 // Clean up
268 while (pHead != NULL) {
269     ListNode* temp = pHead;
270     pHead = pHead->next;
271     delete temp;
272 }
273 }
274
275 void TestDeleteOperations() {
276     cout << "\n" << string(70, '=') << endl;
277     cout << "TESTING DELETE OPERATIONS\n";
278     cout << string(70, '=') << endl;
279
280     Solution testSolution;
281     int count = 5;
282
283     for (int k = 0; k <= count; k++) {
284         ListNode* pHead = NULL;
285         ListNode* pCur = NULL;
286         ListNode* pDel = NULL;
287
288         cout << "\n--- Creating list and deleting node at position " << k << " ---" << endl;
289
290         for (int i = 0; i < count; i++) {
291             ListNode* pTemp = new ListNode(i);
292
293             if (i == 0) {
294                 pHead = pCur = pTemp;
295             }
296             else {
297                 pCur->next = pTemp;
298                 pCur = pCur->next; // pCur->next == pTemp
299             }
300
301             if (i == k)
302                 pDel = pTemp;
303         }
304
305         cout << "Original list: ";
306         testSolution.PrintListNode(pHead);
307
308         testSolution.DeleteNode(pHead, pDel);
309
310         cout << "After deletion: ";
311         testSolution.PrintListNode(pHead);
312
313         // Clean up remaining nodes
314         while (pHead != NULL) {
```

```
315     ListNode* temp = pHead;
316     pHead = pHead->next;
317     delete temp;
318 }
319 }
320 }
321
322 void TestDeleteByValue() {
323     cout << "\n" << string(70, '=') << endl;
324     cout << "TESTING DELETE BY VALUE\n";
325     cout << string(70, '=') << endl;
326
327     Solution solution;
328     ListNode* pHead = NULL;
329
330     // Create list: 10 -> 20 -> 30 -> 40 -> 50
331     pHead = solution.InsertNodeAtTail(pHead, 10);
332     pHead = solution.InsertNodeAtTail(pHead, 20);
333     pHead = solution.InsertNodeAtTail(pHead, 30);
334     pHead = solution.InsertNodeAtTail(pHead, 40);
335     pHead = solution.InsertNodeAtTail(pHead, 50);
336
337     cout << "\nOriginal list: ";
338     solution.PrintListNode(pHead);
339
340     // Delete middle node
341     cout << "\nDeleting 30:" << endl;
342     pHead = solution.DeleteNodeByValue(pHead, 30);
343     solution.PrintListNode(pHead);
344
345     // Delete head node
346     cout << "\nDeleting 10 (head):" << endl;
347     pHead = solution.DeleteNodeByValue(pHead, 10);
348     solution.PrintListNode(pHead);
349
350     // Delete tail node
351     cout << "\nDeleting 50 (tail):" << endl;
352     pHead = solution.DeleteNodeByValue(pHead, 50);
353     solution.PrintListNode(pHead);
354
355     // Clean up
356     while (pHead != NULL) {
357         ListNode* temp = pHead;
358         pHead = pHead->next;
359         delete temp;
360     }
361 }
362
363 int main () {
364     cout << "\n";
365     cout << "                                     \n";
366     cout << "           LINKED LIST INSERT AND DELETE OPERATIONS
367     \n";
368     cout << "
```

```
368     cout << "    Demonstrates various insertion and deletion techniques
369     \n";
370     cout << "                                \n";
371     TestInsertOperations();
372     TestInsertSorted();
373     TestDeleteOperations();
374     TestDeleteByValue();
375
376     cout << "\n" << string(70, '=') << endl;
377     cout << "ALL TESTS COMPLETED\n";
378     cout << string(70, '=') << endl;
379     cout << endl;
380
381     return 0;
382 }
```

## 39 Source Code: LambdaCaptures.cpp

File: src/LambdaCaptures.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <memory>
4 #include <functional>
5 #include <vector>
6 #include <future>
7 #include <thread>
8
9 // =====
10 // 1. NO CAPTURE []
11 // =====
12 void example_no_capture() {
13     std::cout << "\n== 1. NO CAPTURE [] ==" << std::endl;
14
15     int global_counter = 100; // Global-like variable
16
17     // Lambda can only access its parameters and static/global variables
18     auto lambda = []() {
19         static int static_var = 42;
20         std::cout << "Static variable: " << static_var++ << std::endl;
21         return static_var;
22     };
23
24     std::cout << "Call 1: " << lambda() << std::endl; // 43
25     std::cout << "Call 2: " << lambda() << std::endl; // 44
26 }
27
28 // =====
29 // 2. CAPTURE ALL BY VALUE [=]
30 // =====
31 void example_capture_all_by_value() {
32     std::cout << "\n== 2. CAPTURE ALL BY VALUE [=] ==" << std::endl;
33
34     int x = 10;
35     int y = 20;
36     int z = 30; // Not referenced, so not captured
37
38     auto lambda = [=]() { // Captures x and y by value
39         std::cout << "x = " << x << ", y = " << y << std::endl;
40         // Cannot modify: x and y are const copies
41         // x = 100; // ERROR!
42         return x + y;
43     };
44
45     x = 100; // Change original
46     std::cout << "Lambda result: " << lambda() << std::endl; // Still 30
47         // (10+20)
48     std::cout << "Original x after: " << x << std::endl; // 100
49 }
```

```
49 // =====
50 // 3. CAPTURE ALL BY REFERENCE [&]
51 // =====
52
53 void example_capture_all_by_reference() {
54     std::cout << "\n== 3. CAPTURE ALL BY REFERENCE [&] ==" << std::endl;
55
56     int x = 10;
57     int y = 20;
58
59     auto lambda = [&]() { // Captures x and y by reference
60         std::cout << "Before modify: x = " << x << ", y = " << y << std::endl;
61         x = 100; // Modifies original
62         y = 200; // Modifies original
63         return x + y;
64     };
65
66     std::cout << "Before lambda: x=" << x << ", y=" << y << std::endl; // 10,
67     20
68     std::cout << "Lambda result: " << lambda() << std::endl; // 300
69     std::cout << "After lambda: x=" << x << ", y=" << y << std::endl; //
70     100, 200
71 }
72
73 // =====
74 // 4. CAPTURE SPECIFIC BY VALUE [var]
75 // =====
76
77 void example_specific_capture_by_value() {
78     std::cout << "\n== 4. CAPTURE SPECIFIC BY VALUE [var] ==" << std::endl;
79
80     int a = 1, b = 2, c = 3;
81
82     auto lambda = [a, c]() { // Only captures a and c by value
83         std::cout << "a = " << a << ", c = " << c << std::endl;
84         // b is not accessible here
85         return a + c;
86     };
87
88     a = 100; // Change original
89     std::cout << "Lambda result: " << lambda() << std::endl; // 4 (1+3)
90     std::cout << "Original a after: " << a << std::endl; // 100
91 }
92
93 // =====
94 // 5. CAPTURE SPECIFIC BY REFERENCE [&var]
95 // =====
96
97 void example_specific_capture_by_reference() {
98     std::cout << "\n== 5. CAPTURE SPECIFIC BY REFERENCE [&var] ==" << std::endl;
99
100    int a = 1, b = 2, c = 3;
101
102    auto lambda = [&a, &c]() { // Only captures a and c by reference
103        std::cout << "Before modify: a = " << a << ", c = " << c << std::endl;
```

```
100     a = 10; // Modifies original a
101     c = 30; // Modifies original c
102     return a + c;
103 }
104
105 std::cout << "Before lambda: a=" << a << ", c=" << c << std::endl; // 1,
106     3
107 std::cout << "Lambda result: " << lambda() << std::endl; // 40
108 std::cout << "After lambda: a=" << a << ", c=" << c << std::endl; // 10,
109     30
110 }
111
112 // =====
113 // 6. MIXED CAPTURE [=, &var]
114 // =====
115 void example_mixed_capture_value_default() {
116     std::cout << "\n==== 6. MIXED CAPTURE [=, &var] ===" << std::endl;
117
118     int x = 10, y = 20, z = 30;
119
120     auto lambda = [=, &z]() { // x,y by value, z by reference
121         std::cout << "x=" << x << ", y=" << y << ", z=" << z << std::endl;
122         // x = 100; // ERROR: captured by value
123         z = 300; // OK: captured by reference
124         return x + y + z;
125     };
126
127     std::cout << "Before lambda: z=" << z << std::endl; // 30
128     std::cout << "Lambda result: " << lambda() << std::endl; // 330
129     std::cout << "After lambda: z=" << z << std::endl; // 300
130 }
131
132 // =====
133 // 7. MIXED CAPTURE [&, var]
134 // =====
135 void example_mixed_capture_reference_default() {
136     std::cout << "\n==== 7. MIXED CAPTURE [&, var] ===" << std::endl;
137
138     int x = 10, y = 20, z = 30;
139
140     auto lambda = [&, y]() { // x,z by reference, y by value
141         std::cout << "x=" << x << ", y=" << y << ", z=" << z << std::endl;
142         x = 100; // OK: captured by reference
143         // y = 200; // ERROR: captured by value
144         z = 300; // OK: captured by reference
145         return x + y + z;
146     };
147
148     std::cout << "Before lambda: x=" << x << ", y=" << y << ", z=" << z << std
149         ::endl; // 10,20,30
150     std::cout << "Lambda result: " << lambda() << std::endl; // 420
151     std::cout << "After lambda: x=" << x << ", y=" << y << ", z=" << z << std
152         ::endl; // 100,20,300
153 }
```

```
150
151 // =====
152 // 8. CAPTURE THIS POINTER [this]
153 // =====
154 class ExampleClass {
155 private:
156     int member_var = 42;
157     std::string name = "TestClass";
158
159 public:
160     void example_capture_this() {
161         std::cout << "\n== 8. CAPTURE THIS POINTER [this] ==" << std::endl;
162
163         int local_var = 100;
164
165         auto lambda = [this, local_var]() {
166             std::cout << "Member var: " << member_var << std::endl;
167             std::cout << "Name: " << name << std::endl;
168             std::cout << "Local var: " << local_var << std::endl;
169             member_var = 999; // Can modify member through this
170             return member_var + local_var;
171         };
172
173         std::cout << "Before lambda: member_var=" << member_var << std::endl;
174             // 42
175         std::cout << "Lambda result: " << lambda() << std::endl;
176             // 1099
177         std::cout << "After lambda: member_var=" << member_var << std::endl;
178             // 999
179     }
180 }
181
182 // =====
183 // 9. CAPTURE *THIS [*this] (C++17)
184 // =====
185 class ExampleClass2 {
186 private:
187     int value = 42;
188
189 public:
190     void example_capture_star_this() {
191         std::cout << "\n== 9. CAPTURE *THIS [*this] (C++17) ==" << std::endl
192             ;
193
194         std::cout << "Original object value: " << value << std::endl;
195
196         // Capture a copy of the entire object
197         auto lambda = [*this]() mutable {
198             std::cout << "Copied object value: " << value << std::endl;
199             value = 100; // Modifies the copy
200             return value;
201         };
202
203         // Simulate async usage
```

```
200     auto future = std::async(std::launch::async, lambda);
201
202     std::cout << "Original unchanged during async: " << value << std::endl
203         ; // 42
204     std::cout << "Async result (copy modified): " << future.get() << std::endl
205         ; // 100
206     std::cout << "Original still: " << value << std::endl; // 42
207 }
208
209 // =====
210 // 10. MUTABLE LAMBDA
211 // =====
212 void example Mutable_lambda() {
213     std::cout << "\n==== 10. MUTABLE LAMBDA ===" << std::endl;
214
215     int x = 10;
216     int counter = 0;
217
218     // Without mutable - cannot modify captured values
219     // auto lambda1 = [x, counter]() {
220     //     counter++; // ERROR
221     //     return x + counter;
222     // };
223
224     // With mutable - can modify captured copies
225     auto lambda2 = [x, counter]() mutable {
226         std::cout << "Before increment: x=" << x << ", counter=" << counter <<
227             std::endl;
228         x += 5; // Can modify captured copy
229         counter++; // Can modify captured copy
230         std::cout << "After increment: x=" << x << ", counter=" << counter <<
231             std::endl;
232         return x + counter;
233     };
234
235     std::cout << "Before lambda: counter=" << counter << std::endl; // 0
236     std::cout << "Lambda result 1: " << lambda2() << std::endl; // 16
237         (15+1)
238     std::cout << "Lambda result 2: " << lambda2() << std::endl; // 22
239         (20+2)
240     std::cout << "Original unchanged: counter=" << counter << std::endl; // 0
241 }
242
243 // =====
244 // 11. CAPTURE WITH INITIALIZATION (C++14)
245 // =====
246 void example Capture_with_initialization() {
247     std::cout << "\n==== 11. CAPTURE WITH INITIALIZATION (C++14) ===" << std::endl;
248
249     int x = 10;
250     std::string str = "Hello";
```

```

247 // C++14: Capture with initialization expressions
248 auto lambda1 = [value = x * 2, message = str + " World"]() {
249     std::cout << "value = " << value << std::endl;           // 20
250     std::cout << "message = " << message << std::endl; // "Hello World"
251     return value;
252 };
253
254 auto lambda2 = [&ref = x]() { // Capture reference with custom name
255     ref = 100; // Modifies original x
256     return ref;
257 };
258
259 std::cout << "Lambda1 result: " << lambda1() << std::endl; // 20
260 std::cout << "Before lambda2: x=" << x << std::endl;           // 10
261 std::cout << "Lambda2 result: " << lambda2() << std::endl; // 100
262 std::cout << "After lambda2: x=" << x << std::endl;           // 100
263
264 // Capture unique_ptr
265 auto lambda3 = [ptr = std::make_unique<int>(42)]() {
266     return *ptr;
267 };
268 std::cout << "Lambda3 with unique_ptr: " << lambda3() << std::endl; // 42
269 }
270
271 // =====
272 // 12. MULTIPLE SPECIFIC CAPTURES
273 // =====
274 void example_multiple_specific_captures() {
275     std::cout << "\n==== 12. MULTIPLE SPECIFIC CAPTURES ===" << std::endl;
276
277     int a = 1, b = 2, c = 3, d = 4, e = 5;
278
279     auto lambda = [a, &b, c, &d]() { // Mix of value and reference
280         std::cout << "Captured: a=" << a << ", b=" << b
281             << ", c=" << c << ", d=" << d << std::endl;
282         b = 20; // OK: reference
283         d = 40; // OK: reference
284         // a = 10; // ERROR: value capture
285         return a + b + c + d;
286     };
287
288     std::cout << "Before lambda: a=" << a << ", b=" << b
289             << ", c=" << c << ", d=" << d << std::endl; // 1,2,3,4
290     std::cout << "Lambda result: " << lambda() << std::endl;           // 66
291             (1+20+3+40)
292     std::cout << "After lambda: a=" << a << ", b=" << b
293             << ", c=" << c << ", d=" << d << std::endl; // 1,20,3,40
294 }
295
296 // =====
297 // 13. PRACTICAL EXAMPLE: EVENT HANDLER
298 // =====
299 void example_event_handler() {
300     std::cout << "\n==== 13. PRACTICAL EXAMPLE: EVENT HANDLER ===" << std::endl

```

```
300 ;
301
302 class Button {
303 private:
304     std::vector<std::function<void()>> click_handlers;
305     std::string label;
306
307 public:
308     Button(const std::string& lbl) : label(lbl) {}
309
310     void add_handler(std::function<void()> handler) {
311         click_handlers.push_back(handler);
312     }
313
314     void click() {
315         std::cout << "Button '" << label << "' clicked!" << std::endl;
316         for (auto& handler : click_handlers) {
317             handler();
318         }
319     }
320 };
321
322 std::string username = "Alice";
323 int click_count = 0;
324
325 Button button("Submit");
326
327 // Capture username by value, click_count by reference
328 button.add_handler([username, &click_count]() {
329     std::cout << "Hello " << username << "!" << std::endl;
330     click_count++;
331     std::cout << "Click count: " << click_count << std::endl;
332 });
333
334 button.click(); // Hello Alice! Click count: 1
335 button.click(); // Hello Alice! Click count: 2
336
337 username = "Bob"; // Change doesn't affect captured value
338
339 button.click(); // Still: Hello Alice! Click count: 3
340
341 // =====
342 // 14. DANGEROUS EXAMPLE: DANGLING REFERENCE
343 // =====
344 void example_dangling_reference() {
345     std::cout << "\n== 14. DANGEROUS: DANGLING REFERENCE ==" << std::endl;
346
347     std::function<void()> dangerous_lambda;
348
349 {
350     int local_variable = 42;
351
352     // CAPTURE BY REFERENCE - DANGER!
```

```
353     dangerous_lambda = [&local_variable]() {
354         std::cout << "Local variable: " << local_variable << std::endl;
355         // UNDEFINED!
356     };
357
358     // local_variable will be destroyed when scope ends
359 }
360
361 std::cout << "WARNING: Calling lambda with dangling reference..." << std::endl;
362 // dangerous_lambda(); // UNDEFINED BEHAVIOR - might crash
363
364 // SAFE ALTERNATIVE
365 std::function<void()> safe_lambda;
366 {
367     int local_variable = 42;
368     safe_lambda = [local_variable]() { // Capture by value
369         std::cout << "Safe: captured value = " << local_variable << std::endl;
370     };
371 }
372
373 safe_lambda(); // Safe! Outputs: 42
374 }
375 // =====
376 // 15. COMPREHENSIVE EXAMPLE: ALL IN ONE
377 // =====
378 void comprehensive_example() {
379     std::cout << "\n==== 15. COMPREHENSIVE EXAMPLE ===" << std::endl;
380
381     int x = 1, y = 2, z = 3;
382     static int static_var = 100;
383
384     std::cout << "Creating multiple lambdas with different captures:" << std::endl;
385
386     // 1. No capture
387     auto lambda1 = []() {
388         std::cout << " Lambda1: No capture, static=" << static_var++ << std::endl;
389     };
390
391     // 2. Capture all by value
392     auto lambda2 = [=]() {
393         std::cout << " Lambda2: All by value, x=" << x << ", y=" << y << std::endl;
394     };
395
396     // 3. Capture all by reference
397     auto lambda3 = [&]() {
398         std::cout << " Lambda3: All by ref, before x=" << x;
399         x = 10;
400         std::cout << ", after x=" << x << std::endl;
```

```
401 };
```

```
402
403 // 4. Specific captures
404 auto lambda4 = [x, &y]() {
405     std::cout << " Lambda4: x by value=" << x << ", y by ref=" << y <<
406         std::endl;
407     y = 20;
408 };
409
410 // 5. Mixed capture
411 auto lambda5 = [=, &z]() {
412     std::cout << " Lambda5: x,y by value, z by ref, z=" << z << std::endl
413         ;
414     z = 30;
415 };
416
417 // 6. Mutable
418 auto lambda6 = [x]() mutable {
419     std::cout << " Lambda6: Mutable, x before=" << x;
420     x = 100; // Modifies copy
421     std::cout << ", x after=" << x << std::endl;
422 };
423
424 // Execute all
425 lambda1();
426 lambda2();
427 lambda3();
428 lambda4();
429 lambda5();
430 lambda6();
431
432 std::cout << "\nFinal values: x=" << x << ", y=" << y << ", z=" << z <<
433     std::endl;
434 }
435
436 // =====
437 // MAIN FUNCTION
438 // =====
439 int main() {
440     std::cout << "
441         =====
442         << std::endl;
443     std::cout << "COMPLETE C++ LAMBDA CAPTURE EXAMPLES IN ONE FILE" << std::
444         endl;
445     std::cout << "
446         =====
447         << std::endl;
448
449 // Run all examples
450 example_no_capture();
451 example_capture_all_by_value();
452 example_capture_all_by_reference();
453 example_specific_capture_by_value();
454 example_specific_capture_by_reference();
```

```
447     example_mixed_capture_value_default();
448     example_mixed_capture_reference_default();
449
450     ExampleClass obj1;
451     obj1.example_capture_this();
452
453     ExampleClass2 obj2;
454     obj2.example_capture_star_this();
455
456     example_mutable_lambda();
457     example_capture_with_initialization();
458     example_multiple_specific_captures();
459     example_event_handler();
460     example_dangling_reference();
461     comprehensive_example();
462
463     std::cout << "\n"
464     =====
465     << std::endl;
466     std::cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!" << std::endl;
467     std::cout << "
468     =====
469     << std::endl;
470
471     return 0;
472 }
```

## 40 Source Code: MISRACppDemo.cpp

File: src/MISRACppDemo.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // MISRA C++ CODING GUIDELINES DEMONSTRATION
3 // =====
4 // Demonstrates key MISRA C++ rules for safety-critical systems
5 // MISRA C++:2008 and MISRA C++:2023 guidelines
6 //
7 // MISRA C++ focuses on:
8 // - Avoiding undefined behavior
9 // - Avoiding implementation-defined behavior
10 // - Defensive programming practices
11 // - Code clarity and maintainability
12 // - Safety in automotive, aerospace, medical devices
13 //
14 // Build: g++ -std=c++20 -Wall -Wextra -Wpedantic -O2 -o MISRACppDemo
15 // MISRACppDemo.cpp
16 // =====
17
18 #include <iostream>
19 #include <cstdint>
20 #include <array>
21 #include <limits>
22 #include <memory>
23 #include <string>
24 #include <algorithm> // for std::copy_n
25 // =====
26 // RULE CATEGORY 1: TYPES AND DECLARATIONS
27 // =====
28
29 namespace types_and_declarations {
30
31 // MISRA Rule 3-9-1: Use fixed-width integer types from <cstdint>
32 // Rationale: Ensures portability across platforms
33
34 // BAD: Platform-dependent types
35 void bad_types() {
36     std::cout << "\n BAD: Platform-dependent types\n";
37
38     int x = 100; // Size undefined (16, 32, or 64 bits?)
39     long y = 1000; // Size undefined (32 or 64 bits?)
40     unsigned int z = 500; // Size undefined
41
42     std::cout << "    int size: " << sizeof(x) << " bytes (undefined!)\n";
43     std::cout << "    long size: " << sizeof(y) << " bytes (undefined!)\n";
44 }
45
46 // GOOD: Fixed-width types
47 void good_types() {
48     std::cout << "\n GOOD: Fixed-width types\n";
```

```
49
50     int32_t x = 100;           // Always 32 bits
51     int64_t y = 1000;          // Always 64 bits
52     uint32_t z = 500;          // Always unsigned 32 bits
53
54     std::cout << "    int32_t: " << x << " (always 32 bits)\n";
55     std::cout << "    int64_t: " << y << " (always 64 bits)\n";
56     std::cout << "    uint32_t: " << z << " (always unsigned 32 bits)\n";
57 }
58
59 // MISRA Rule 4-5-1: Expressions with type bool shall not be used with
60 // operators other than
61 // ==, !=, !=, !, &&, ||
62
63 // BAD: Arithmetic on bool
64 void bad_bool_usage() {
65     std::cout << "\n BAD: Arithmetic on bool\n";
66
67     bool flag1 = true;
68     bool flag2 = false;
69
70     // int result = flag1 + flag2; // MISRA violation!
71     // int value = flag1 * 5;      // MISRA violation!
72
73     std::cout << "    Arithmetic on bool is prohibited by MISRA\n";
74 }
75
76 // GOOD: Proper bool usage
77 void good_bool_usage() {
78     std::cout << "\n GOOD: Proper bool usage\n";
79
80     bool flag1 = true;
81     bool flag2 = false;
82
83     bool result = flag1 && flag2; // OK: logical operator
84     bool equal = (flag1 == flag2); // OK: comparison
85     bool negate = !flag1;        // OK: logical NOT
86
87     std::cout << "    flag1 && flag2 = " << result << "\n";
88     std::cout << "    flag1 == flag2 = " << equal << "\n";
89 }
90
91 void demonstrate() {
92     std::cout << "\n" << std::string(70, '=') << "\n";
93     std::cout << "MISRA CATEGORY 1: TYPES AND DECLARATIONS\n";
94     std::cout << std::string(70, '=') << "\n";
95
96     bad_types();
97     good_types();
98     bad_bool_usage();
99     good_bool_usage();
100 }
```

```
102 // =====
103 // RULE CATEGORY 2: EXPRESSIONS AND OPERATORS
104 // =====
105
106
107 namespace expressions_and_operators {
108
109 // MISRA Rule 5-0-21: Bitwise operators shall only be applied to operands of
110 // unsigned type
111
112 // BAD: Bitwise operations on signed integers
113 void bad_bitwise() {
114     std::cout << "\n BAD: Bitwise operations on signed integers\n";
115
116     int32_t signed_val = -1;
117     // int32_t result = signed_val << 2; // MISRA violation! Undefined
118     // behavior
119     // int32_t mask = signed_val & 0xFF; // MISRA violation!
120
121     std::cout << "    Bitwise ops on signed integers can cause undefined
122     behavior\n";
123 }
124
125 // GOOD: Bitwise operations on unsigned integers
126 void good_bitwise() {
127     std::cout << "\n GOOD: Bitwise operations on unsigned integers\n";
128
129     uint32_t unsigned_val = 0xF0;
130     uint32_t result = unsigned_val << 2; // OK: unsigned
131     uint32_t mask = unsigned_val & 0xFF; // OK: unsigned
132
133     std::cout << "    unsigned_val << 2 = 0x" << std::hex << result << "\n";
134     std::cout << "    unsigned_val & 0xFF = 0x" << mask << std::dec << "\n";
135 }
136
137 // MISRA Rule 5-0-5: There shall be no implicit floating-integral conversions
138
139 // BAD: Implicit conversions
140 void bad_conversions() {
141     std::cout << "\n BAD: Implicit floating-integral conversions\n";
142
143     float f = 3.14f;
144     // int32_t i = f; // MISRA violation! Implicit conversion
145     // float result = 5 + f; // MISRA violation! Mixed type arithmetic
146
147     std::cout << "    Implicit float-int conversions lose precision\n";
148 }
149
150 // GOOD: Explicit conversions
151 void good_conversions() {
152     std::cout << "\n GOOD: Explicit conversions\n";
153
154     float f = 3.14f;
155     int32_t i = static_cast<int32_t>(f); // OK: explicit cast
```

```
153     float result = static_cast<float>(5) + f; // OK: explicit cast
154
155     std::cout << "    static_cast<int32_t>(3.14f) = " << i << "\n";
156     std::cout << "    Explicit casts make intent clear\n";
157 }
158
159 // MISRA Rule 5-0-6: An implicit integral or floating-point conversion shall
160 // not reduce
161 // the size of the underlying type
162
163 // BAD: Narrowing conversions
164 void bad_narrowing() {
165     std::cout << "\n BAD: Narrowing conversions\n";
166
167     int64_t large = 1000000;
168     // int16_t small = large; // MISRA violation! Data loss possible
169
170     std::cout << "    Narrowing conversions can lose data\n";
171 }
172
173 // GOOD: Safe conversions or explicit narrowing
174 void good_narrowing() {
175     std::cout << "\n GOOD: Safe conversions with range checks\n";
176
177     int64_t large = 1000;
178
179     // Check before narrowing
180     if (large <= std::numeric_limits<int16_t>::max() &&
181         large >= std::numeric_limits<int16_t>::min()) {
182         int16_t small = static_cast<int16_t>(large); // OK: checked and
183         explicit
184         std::cout << "    Safe narrowing: " << small << "\n";
185     } else {
186         std::cout << "    Value out of range for int16_t\n";
187     }
188 }
189
190 void demonstrate() {
191     std::cout << "\n" << std::string(70, '=') << "\n";
192     std::cout << "MISRA CATEGORY 2: EXPRESSIONS AND OPERATORS\n";
193     std::cout << std::string(70, '=') << "\n";
194
195     bad_bitwise();
196     good_bitwise();
197     bad_conversions();
198     good_conversions();
199     bad_narrowing();
200     good_narrowing();
201 }
202
203 } // namespace expressions_and_operators
204
205 // =====
206 // RULE CATEGORY 3: STATEMENTS
```

```
205 // =====
206
207 namespace statements {
208
209 // MISRA Rule 6-4-1: An if (condition) construct shall be followed by a
210 // compound statement
211 // MISRA Rule 6-4-4: A switch statement shall have at least two case clauses
212
213 // BAD: Single-statement if without braces
214 void bad_if_statements() {
215     std::cout << "\n BAD: Single-statement if without braces\n";
216
217     int32_t value = 5;
218
219     // if (value > 0)
220     //     std::cout << "Positive\n"; // MISRA violation! No braces
221
222     std::cout << "    Missing braces can lead to errors during maintenance\n";
223 }
224
225 // GOOD: Always use braces
226 void good_if_statements() {
227     std::cout << "\n GOOD: Always use braces\n";
228
229     int32_t value = 5;
230
231     if (value > 0) { // OK: compound statement
232         std::cout << "    Positive\n";
233     } else {
234         std::cout << "    Non-positive\n";
235     }
236
237 // MISRA Rule 6-4-5: An unconditional throw or break shall terminate every non
238 // -empty
239 // switch clause
240
241 // BAD: Fall-through in switch
242 void bad_switch(int32_t value) {
243     std::cout << "\n BAD: Fall-through in switch\n";
244
245     // switch (value) {
246     //     case 1:
247     //         std::cout << "One\n";
248     //         // Fall-through! MISRA violation
249     //     case 2:
250     //         std::cout << "Two\n";
251     //         break;
252     //     default:
253     //         std::cout << "Other\n";
254     // }
255
256     std::cout << "    Fall-through is error-prone\n";
257 }
```

```
257
258 //  GOOD: Explicit break in every case
259 void good_switch(int32_t value) {
260     std::cout << "\n GOOD: Explicit break in every case\n";
261
262     switch (value) {
263         case 1: {
264             std::cout << "    One\n";
265             break;
266         }
267         case 2: {
268             std::cout << "    Two\n";
269             break;
270         }
271         default: {
272             std::cout << "    Other\n";
273             break;
274         }
275     }
276 }
277
278 // MISRA Rule 6-6-1, 6-6-2: Any label referenced by a goto shall be in the
279 // same block
280 // or enclosing block
281
282 //  BAD: goto usage (generally prohibited)
283 void bad_goto() {
284     std::cout << "\n BAD: goto usage\n";
285     std::cout << "    MISRA strongly discourages or prohibits goto\n";
286     std::cout << "    Use structured programming instead\n";
287 }
288
289 //  GOOD: Structured programming
290 void good_structured() {
291     std::cout << "\n GOOD: Structured programming\n";
292
293     bool error_occurred = false;
294
295     // Instead of goto for error handling, use early return
296     if (error_occurred) {
297         std::cout << "    Error handled with early return\n";
298         return;
299     }
300
301     std::cout << "    Normal execution path\n";
302 }
303
304 void demonstrate() {
305     std::cout << "\n" << std::string(70, '=')
306     << "\n";
307     std::cout << "MISRA CATEGORY 3: STATEMENTS\n";
308     std::cout << std::string(70, '=')
309     << "\n";
310
311     bad_if_statements();
312     good_if_statements();
```

```
310     bad_switch(1);
311     good_switch(1);
312     bad_goto();
313     good_structured();
314 }
315
316 } // namespace statements
317
318 // =====
319 // RULE CATEGORY 4: FUNCTIONS
320 // =====
321
322 namespace functions {
323
324 // MISRA Rule 8-4-2: All exit paths from a function with non-void return type
325 // shall have
326 // an explicit return statement
327
328 // BAD: Missing return in some paths
329 // int32_t bad_function(int32_t value) {
330 //     if (value > 0) {
331 //         return value;
332 //     }
333 //     // MISRA violation! Missing return for value <= 0
334 // }
335
336 // GOOD: All paths have explicit return
337 int32_t good_function(int32_t value) {
338     if (value > 0) {
339         return value;
340     } else {
341         return 0;
342     }
343
344 // MISRA Rule 8-4-4: A function with no parameters shall be declared with
345 // parameter (void)
346
347 // BAD: Empty parameter list in C++
348 // void bad_no_params(); // In C++, this is OK, but MISRA prefers explicit
349 // void
350
351 // GOOD: Explicit void for clarity (C-style, but clearer)
352 void good_no_params(void) {
353     std::cout << "    Function with explicit void parameter list\n";
354
355 // MISRA Rule 7-1-1: A variable that is not modified shall be const qualified
356
357 // BAD: Non-const variable that isn't modified
358 void bad_const_usage() {
359     std::cout << "\n BAD: Non-const variable that isn't modified\n";
360
361     int32_t value = 100; // MISRA violation! Should be const
```

```
361     std::cout << "    Value: " << value << "\n";
362 }
363
364 //  GOOD: const-qualified for immutable data
365 void good_const_usage() {
366     std::cout << "\n GOOD: const-qualified for immutable data\n";
367
368     const int32_t value = 100; // OK: const qualifier
369     std::cout << "    Value: " << value << "\n";
370 }
371
372 void demonstrate() {
373     std::cout << "\n" << std::string(70, '=') << "\n";
374     std::cout << "MISRA CATEGORY 4: FUNCTIONS\n";
375     std::cout << std::string(70, '=') << "\n";
376
377     std::cout << "\n GOOD: All paths return a value\n";
378     std::cout << "    good_function(5) = " << good_function(5) << "\n";
379     std::cout << "    good_function(-5) = " << good_function(-5) << "\n";
380
381     good_no_params();
382
383     bad_const_usage();
384     good_const_usage();
385 }
386
387 } // namespace functions
388
389 // =====
390 // RULE CATEGORY 5: ARRAYS AND POINTERS
391 // =====
392
393 namespace arrays_and_pointers {
394
395 // MISRA Rule 5-0-15: Array indexing shall be the only form of pointer
396 // arithmetic
397
398 // BAD: Pointer arithmetic
399 void bad_pointer_arithmetic() {
400     std::cout << "\n BAD: Pointer arithmetic\n";
401
402     int32_t arr[5] = {1, 2, 3, 4, 5};
403     int32_t* ptr = arr;
404
405     // ptr++;           // MISRA violation!
406     // int32_t val = *(ptr + 2); // MISRA violation!
407
408     std::cout << "    Pointer arithmetic is error-prone\n";
409 }
410
411 //  GOOD: Array indexing
412 void good_array_indexing() {
413     std::cout << "\n GOOD: Array indexing\n";
```

```
414     std::array<int32_t, 5> arr = {1, 2, 3, 4, 5};  
415  
416     for (size_t i = 0; i < arr.size(); ++i) {  
417         std::cout << "    arr[" << i << "] = " << arr[i] << "\n";  
418     }  
419 }  
420  
421 // MISRA Rule 5-0-16: The result of pointer subtraction shall not be used  
422  
423 // BAD: Pointer subtraction  
424 void bad_pointer_subtraction() {  
425     std::cout << "\n BAD: Pointer subtraction\n";  
426  
427     int32_t arr[5] = {1, 2, 3, 4, 5};  
428     int32_t* p1 = &arr[0];  
429     int32_t* p2 = &arr[3];  
430  
431     // ptrdiff_t diff = p2 - p1; // MISRA violation!  
432  
433     std::cout << "    Pointer subtraction should be avoided\n";  
434 }  
435  
436 // GOOD: Use indices instead  
437 void good_index_difference() {  
438     std::cout << "\n GOOD: Use indices instead\n";  
439  
440     std::array<int32_t, 5> arr = {1, 2, 3, 4, 5};  
441     size_t idx1 = 0;  
442     size_t idx2 = 3;  
443  
444     size_t diff = idx2 - idx1;  
445     std::cout << "    Index difference: " << diff << "\n";  
446 }  
447  
448 // MISRA Rule: Use std::array instead of C-style arrays  
449  
450 // BAD: C-style arrays  
451 void bad_c_arrays() {  
452     std::cout << "\n BAD: C-style arrays\n";  
453  
454     int32_t arr[10]; // MISRA prefers std::array  
455     arr[0] = 1;  
456  
457     std::cout << "    C-style arrays lack bounds checking\n";  
458 }  
459  
460 // GOOD: std::array with bounds checking  
461 void good_std_array() {  
462     std::cout << "\n GOOD: std::array with bounds checking\n";  
463  
464     std::array<int32_t, 10> arr{};  
465     arr.at(0) = 1; // Bounds-checked access  
466  
467     std::cout << "    arr.at(0) = " << arr.at(0) << "\n";
```

```
468     std::cout << "    std::array provides bounds checking with at()\n";
469 }
470
471 void demonstrate() {
472     std::cout << "\n" << std::string(70, '=') << "\n";
473     std::cout << "MISRA CATEGORY 5: ARRAYS AND POINTERS\n";
474     std::cout << std::string(70, '=') << "\n";
475
476     bad_pointer_arithmetic();
477     good_array_indexing();
478     bad_pointer_subtraction();
479     good_index_difference();
480     bad_c_arrays();
481     good_std_array();
482 }
483
484 } // namespace arrays_and_pointers
485
486 // =====
487 // RULE CATEGORY 6: CLASSES
488 // =====
489
490 namespace classes {
491
492 // MISRA Rule 12-8-1: A copy constructor shall only initialize its base
493 // classes and the
494 // non-static members of the class of which it is a member
495
496 // GOOD: Proper class design
497 class Resource {
498 private:
499     int32_t* data_;
500     size_t size_;
501
502 public:
503     // Constructor
504     explicit Resource(size_t size)
505         : data_(new int32_t[size]), size_(size) {
506         std::cout << "    [Resource] Constructor: allocated " << size_ <<
507             " ints\n";
508     }
509
510     // Destructor
511     ~Resource() {
512         delete[] data_;
513         std::cout << "    [Resource] Destructor: freed memory\n";
514     }
515
516     // MISRA Rule: Define all special members or none (Rule of Five)
517     Resource(const Resource& other)
518         : data_(new int32_t[other.size_]), size_(other.size_) {
519             std::copy_n(other.data_, size_, data_);
520             std::cout << "    [Resource] Copy constructor\n";
521     }
```

```
520
521     Resource& operator=(const Resource& other) {
522         if (this != &other) {
523             delete[] data_;
524             size_ = other.size_;
525             data_ = new int32_t[size_];
526             std::copy_n(other.data_, size_, data_);
527         }
528         std::cout << "    [Resource] Copy assignment\n";
529         return *this;
530     }
531
532     Resource(Resource&& other) noexcept
533         : data_(other.data_), size_(other.size_) {
534         other.data_ = nullptr;
535         other.size_ = 0;
536         std::cout << "    [Resource] Move constructor\n";
537     }
538
539     Resource& operator=(Resource&& other) noexcept {
540         if (this != &other) {
541             delete[] data_;
542             data_ = other.data_;
543             size_ = other.size_;
544             other.data_ = nullptr;
545             other.size_ = 0;
546         }
547         std::cout << "    [Resource] Move assignment\n";
548         return *this;
549     }
550
551     size_t size() const { return size_; }
552 };
553
554 // MISRA Rule: Virtual destructor for base classes
555
556 // BAD: Base class without virtual destructor
557 class BadBase {
558 public:
559     ~BadBase() { // MISRA violation! Should be virtual
560         std::cout << "    [BadBase] Destructor\n";
561     }
562 };
563
564 // GOOD: Virtual destructor for polymorphic base
565 class GoodBase {
566 public:
567     virtual ~GoodBase() {
568         std::cout << "    [GoodBase] Virtual destructor\n";
569     }
570
571     virtual void process() = 0;
572 };
573
```

```
574 class Derived : public GoodBase {
575 public:
576     void process() override {
577         std::cout << "    [Derived] Processing\n";
578     }
579
580     ~Derived() override {
581         std::cout << "    [Derived] Destructor\n";
582     }
583 };
584
585 void demonstrate() {
586     std::cout << "\n" << std::string(70, '=') << "\n";
587     std::cout << "MISRA CATEGORY 6: CLASSES\n";
588     std::cout << std::string(70, '=') << "\n";
589
590     std::cout << "\n GOOD: Rule of Five implementation\n";
591 {
592     Resource r1(5);
593     Resource r2 = r1; // Copy
594     Resource r3 = std::move(r1); // Move
595 }
596
597 std::cout << "\n GOOD: Virtual destructor for polymorphism\n";
598 {
599     std::unique_ptr<GoodBase> ptr = std::make_unique<Derived>();
600     ptr->process();
601 }
602 }
603
604 } // namespace classes
605
606 // =====
607 // RULE CATEGORY 7: EXCEPTION HANDLING
608 // =====
609
610 namespace exception_handling {
611
612 // MISRA Rule 15-1-1: Only throw exceptions for exceptional conditions
613 // MISRA Rule 15-3-1: Exceptions shall be used only for error handling
614
615 // BAD: Using exceptions for control flow
616 void bad_exception_usage() {
617     std::cout << "\n BAD: Using exceptions for control flow\n";
618
619     // try {
620     //     // Don't use exceptions for normal flow!
621     //     throw 42;
622     // } catch (int value) {
623     //     // Control flow via exception (BAD!)
624     // }
625
626     std::cout << "    Exceptions should only be used for errors\n";
627 }
```

```
628
629 //  GOOD: Exceptions for error handling only
630 void good_exception_usage() {
631     std::cout << "\n GOOD: Exceptions for error handling\n";
632
633     try {
634         // Simulate error condition
635         throw std::runtime_error("Genuine error occurred");
636     } catch (const std::exception& e) {
637         std::cout << "    Caught exception: " << e.what() << "\n";
638     }
639 }
640
641 // MISRA Note: Many safety-critical systems disable exceptions entirely
642 void no_exceptions_approach() {
643     std::cout << "\n ALTERNATIVE: Error codes (no exceptions)\n";
644     std::cout << "    Many MISRA-compliant projects compile with -fno-
645         exceptions\n";
646     std::cout << "    Use error codes, std::optional, or std::expected instead\n";
647 }
648
649 void demonstrate() {
650     std::cout << "\n" << std::string(70, '=') << "\n";
651     std::cout << "MISRA CATEGORY 7: EXCEPTION HANDLING\n";
652     std::cout << std::string(70, '=') << "\n";
653
654     bad_exception_usage();
655     good_exception_usage();
656     no_exceptions_approach();
657 }
658 } // namespace exception_handling
659
660 // =====
661 // MAIN - Demonstrate All MISRA C++ Categories
662 // =====
663
664 int main() {
665     std::cout << "\n";
666     std::cout << "                                \n";
667     std::cout << "                                MISRA C++ CODING GUIDELINES
668                                \n";
669     std::cout << "                                Safety-Critical Software Development
670                                \n";
671     std::cout << "                                \n";
672
673     try {
674         types_and_declarations::demonstrate();
675         expressions_and_operators::demonstrate();
676         statements::demonstrate();
677         functions::demonstrate();
678         arrays_and_pointers::demonstrate();
679         classes::demonstrate();
680     }
```

```
678     exception_handling::demonstrate();  
679  
680     // Summary  
681     std::cout << "\n" << std::string(70, '=') << "\n";  
682     std::cout << "SUMMARY: KEY MISRA C++ PRINCIPLES\n";  
683     std::cout << std::string(70, '=') << "\n\n";  
684  
685     std::cout << " MISRA C++ OBJECTIVES:\n";  
686     std::cout << " • Avoid undefined behavior\n";  
687     std::cout << " • Avoid implementation-defined behavior\n";  
688     std::cout << " • Maximize portability\n";  
689     std::cout << " • Enhance code clarity and maintainability\n";  
690     std::cout << " • Enable thorough code review and testing\n\n";  
691  
692     std::cout << " KEY PRACTICES:\n";  
693     std::cout << " 1. Use fixed-width types (int32_t, uint32_t)\n";  
694     std::cout << " 2. Always use braces for if/while/for\n";  
695     std::cout << " 3. No pointer arithmetic (use array indexing)\n";  
696     std::cout << " 4. Explicit conversions (avoid implicit casts)\n";  
697     std::cout << " 5. const-qualify immutable data\n";  
698     std::cout << " 6. Virtual destructors for polymorphic classes\n";  
699     std::cout << " 7. Rule of Five for resource-managing classes\n";  
700     std::cout << " 8. Prefer std::array over C-style arrays\n\n";  
701  
702     std::cout << " APPLICATION DOMAINS:\n";  
703     std::cout << " • Automotive (ISO 26262)\n";  
704     std::cout << " • Aerospace (DO-178C)\n";  
705     std::cout << " • Medical devices (IEC 62304)\n";  
706     std::cout << " • Industrial control systems\n";  
707     std::cout << " • Railway (EN 50128)\n\n";  
708  
709     std::cout << " NOTE:\n";  
710     std::cout << " Many MISRA-compliant projects also:\n";  
711     std::cout << " • Disable exceptions (-fno-exceptions)\n";  
712     std::cout << " • Disable RTTI (-fno-rtti)\n";  
713     std::cout << " • Limit or prohibit dynamic memory allocation\n";  
714     std::cout << " • Use static analysis tools (PC-lint, Coverity, etc.)  
715     \n\n";  
716     std::cout << " \n";  
717     std::cout << " ALL MISRA C++ CATEGORIES DEMONSTRATED!  
718     \n";  
719  
720 } catch (const std::exception& e) {  
721     std::cerr << " Error: " << e.what() << "\n";  
722     return 1;  
723 }  
724  
725     return 0;  
726 }
```

## 41 Source Code: MockInterview.cpp

File: src/MockInterview.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // COMPLETE C++ INTERVIEW QUESTIONS & ANSWERS
3 // =====
4 // Comprehensive collection of C++ interview questions with detailed answers
5 // covering fundamentals, memory management, OOP, templates, concurrency,
6 // and optimization techniques.
7 //
8 // Topics covered:
9 // - C++ Fundamentals (pointers, references, const)
10 // - Memory Management (smart pointers, allocators, alignment)
11 // - OOP & Design Patterns (virtual dispatch, CRTP, PIMPL)
12 // - Templates & Metaprogramming (SFINAE, variadic, compile-time)
13 // - Concurrency (lock-free, thread pools, atomics)
14 // - Performance & Optimization (cache-friendly, branch prediction)
15 //
16 // Build: g++ -std=c++20 -pthread -O2 -o MockInterview MockInterview.cpp
17 // Run: ./MockInterview
18 // =====
19
20 #include <iostream>
21 #include <memory>
22 #include <thread>
23 #include <mutex>
24 #include <atomic>
25 #include <vector>
26 #include <list>
27 #include <unordered_map>
28 #include <optional>
29 #include <functional>
30 #include <queue>
31 #include <condition_variable>
32 #include <shared_mutex>
33 #include <future>
34 #include <array>
35 #include <type_traits>
36 #include <variant>
37 #include <any>
38 #include <tuple>
39 #include <string>
40 #include <algorithm>
41 #include <numeric>
42 #include <random>
43 #include <chrono>
44 #include <cstdio>
45 #include <stdexcept>
46
47 using namespace std;
48
49 // =====
```

```
50 // SECTION 1: C++ FUNDAMENTALS
51 // =====
52
53 // ===== Q1.1: Pointers vs References =====
54 void pointers_vs_references() {
55     cout << "\n== POINTERS VS REFERENCES ==\n";
56
57     int x = 10;
58     int y = 20;
59
60     // Reference - alias, must be initialized, cannot be null
61     int& ref = x;
62     ref = 30; // Changes x
63
64     // Pointer - stores address, can be null, can be reassigned
65     int* ptr = &x;
66     *ptr = 40; // Changes x
67     ptr = &y; // Now points to y
68
69     cout << "x = " << x << ", y = " << y << endl;
70     cout << "ref = " << ref << ", *ptr = " << *ptr << endl;
71
72     // Key differences:
73     cout << "\nKey Differences:" << endl;
74     cout << "1. References must be initialized, pointers can be null" << endl;
75     cout << "2. References cannot be reassigned, pointers can" << endl;
76     cout << "3. sizeof(reference) = sizeof(object), sizeof(pointer) = platform
77         dependent" << endl;
78     cout << "4. References have cleaner syntax, pointers more flexible" <<
79         endl;
80 }
81
82 // ===== Q1.2: Const Correctness =====
83 void const_correctness() {
84     cout << "\n== CONST CORRECTNESS ==\n";
85
86     class Data {
87         mutable int counter; // Can be modified even in const methods
88         int value;
89
90         public:
91             Data(int v) : counter(0), value(v) {}
92
93             // 1. Non-const getter - can only be called on non-const objects
94             int getValue() {
95                 value = 44;
96                 cout << "Non-const getValue, Changed value = ";
97                 return value;
98             }
99
100            // 2. Const getter - can be called on both const and non-const objects
101            int getValue() const {
102                 cout << "Const getValue, value = ";
103                 counter++; // OK because counter is mutable
104             }
105     }
106 }
```

```
102         return value;
103     }
104
105     // 3. Const reference return - prevents modification
106     const int& getValueRef() const {
107         cout << "Const reference getter" << endl;
108         return value;
109     }
110 };
111
112 Data d1(42);
113 const Data d2(43);
114
115 cout << "d1 " << std::to_string(d1.getValue()) << endl; // Calls non-
116     const version
117 cout << "d2 " << std::to_string(d2.getValue()) << endl; // Calls const
118     version
119
120 // d2.getValueRef() = 50; // ERROR: Cannot modify through const reference
121
122 cout << "\nConst Rules:" << endl;
123 cout << "1. const methods can be called on const objects" << endl;
124 cout << "2. const methods cannot modify non-mutable members" << endl;
125 cout << "3. Return const references to prevent modification" << endl;
126 cout << "4. Use mutable for members that don't affect logical state" <<
127     endl;
128 }
129 // ===== Q1.3: RAII Pattern =====
130 class FileHandler {
131     FILE* file;
132
133 public:
134     FileHandler(const char* filename, const char* mode) {
135         file = fopen(filename, mode);
136         if (!file) {
137             throw runtime_error("Failed to open file");
138         }
139         cout << "File opened: " << filename << endl;
140     }
141
142     ~FileHandler() {
143         if (file) {
144             fclose(file);
145             cout << "File closed" << endl;
146         }
147     }
148
149     // Delete copy operations
150     FileHandler(const FileHandler&) = delete;
151     FileHandler& operator=(const FileHandler&) = delete;
152 }
```

```
153 // Allow move operations
154 FileHandler(FileHandler&& other) noexcept : file(other.file) {
155     other.file = nullptr;
156 }
157
158 FileHandler& operator=(FileHandler&& other) noexcept {
159     if (this != &other) {
160         if (file) fclose(file);
161         file = other.file;
162         other.file = nullptr;
163     }
164     return *this;
165 }
166
167 void write(const string& data) {
168     if (file) {
169         fputs(data.c_str(), file);
170     }
171 }
172 };
173
174 void raii_demo() {
175     cout << "\n==== RAI (Resource Acquisition Is Initialization) ===\n";
176
177     try {
178         FileHandler file("test_interview.txt", "w");
179         file.write("Hello RAI!\n");
180         // File automatically closed when scope ends
181     } catch (const exception& e) {
182         cout << "Exception: " << e.what() << endl;
183     }
184
185     cout << "\nRAII Principles:" << endl;
186     cout << "1. Acquire resource in constructor" << endl;
187     cout << "2. Release resource in destructor" << endl;
188     cout << "3. Use stack unwinding for exception safety" << endl;
189     cout << "4. Smart pointers, locks, containers use RAI" << endl;
190     cout << "5. Use automatic (in stack) std::lock_guard and std::unique_lock
191         in member functions" << endl;
192 }
193 // =====
194 // SECTION 2: MEMORY MANAGEMENT
195 // =====
196
197 // ===== Q2.1: Smart Pointers Deep Dive =====
198 void smart_pointers_advanced() {
199     cout << "\n==== SMART POINTERS ADVANCED ===\n";
200
201     // 1. Custom deleters - Why and when to use them
202     {
203         cout << "\n1. Custom Deleters - Use Cases:" << endl;
204         cout << " • File handles (FILE*) - need fclose()" << endl;
205         cout << " • C arrays - need delete[] instead of delete" << endl;
```

```

206     cout << " • Resource cleanup - network sockets, locks, etc." << endl
207     ;
208     cout << " • Custom allocators - memory pools" << endl;
209     cout << " • Logging/debugging - track object lifetime\n" << endl;
210
211     // Example 1: FILE* with custom deleter
212     auto file_deleter = [](FILE* f) {
213         if (f) {
214             cout << "    Closing file with fclose()" << endl;
215             fclose(f);
216         }
217     };
218     shared_ptr<FILE> file(fopen("temp.txt", "w"), file_deleter);
219     if (file) {
220         fprintf(file.get(), "Custom deleter example\n");
221     }
222
223     // Example 2: Array with delete[]
224     shared_ptr<int> arr(new int[10], [](int* p) {
225         cout << "    Deleting array with delete[]" << endl;
226         delete[] p;
227     });
228     // Note: Better to use unique_ptr<int[]> or vector<int> for arrays
229
230     // Example 3: unique_ptr with custom deleter type
231     auto int_deleter = [](int* p) {
232         cout << "    Custom delete: " << *p << endl;
233         delete p;
234     };
235     unique_ptr<int, decltype(int_deleter)> ptr1(new int(42), int_deleter);
236 }
237
238 // 2. weak_ptr - Breaking circular references
239 {
240     cout << "\n2. weak_ptr - Safe Non-Owning Observation:" << endl;
241     cout << "    Purpose: Observe shared_ptr without affecting reference
242         count" << endl;
243     cout << "    Use cases:" << endl;
244     cout << " • Breaking circular references (parent-child relationships
245         )" << endl;
246     cout << " • Cache implementations (entries can be evicted)" << endl;
247     cout << " • Observer pattern (observers shouldn't own subject)" <<
248         endl;
249     cout << " • Factory tracking (don't prevent object deletion)\n" <<
250         endl;
251
252     // Example: Parent-Child circular reference problem
253     class Child;
254     class Parent {
255     public:
256         shared_ptr<Child> child;
257         ~Parent() { cout << "    ~Parent()" << endl; }
258     };

```

```
255     class Child {
256     public:
257         weak_ptr<Parent> parent; // Use weak_ptr to break cycle!
258         ~Child() { cout << "    ~Child()" << endl; }
259     };
260
261     cout << "    Creating parent-child relationship:" << endl;
262     {
263         auto parent = make_shared<Parent>();
264         auto child = make_shared<Child>();
265
266         parent->child = child;
267         child->parent = parent; // weak_ptr doesn't increase ref count
268
269         cout << "    Parent use_count: " << parent.use_count() << endl; // 1
270         cout << "    Child use_count: " << child.use_count() << endl; // 2
271
272         // Safe weak_ptr access pattern
273         if (auto parent_ptr = child->parent.lock()) { // Convert weak to
274             shared
275                 cout << "    Parent is still alive, use_count: " << parent_ptr.
276                 use_count() << endl;
277         } else {
278             cout << "    Parent has been destroyed" << endl;
279         }
280
281         cout << "    Scope ending..." << endl;
282     } // Both Parent and Child are properly destroyed!
283     cout << "    Both objects cleaned up (no memory leak)\n" << endl;
284
285     // Example: Safe weak_ptr usage pattern
286     shared_ptr<int> strong = make_shared<int>(100);
287     weak_ptr<int> weak = strong;
288
289     cout << "    weak_ptr safe access pattern:" << endl;
290     cout << "    • expired(): Check if object still exists" << endl;
291     cout << "    • lock(): Get shared_ptr if object exists" << endl;
292     cout << "    • use_count(): Get current ref count\n" << endl;
293
294     cout << "    weak.expired(): " << (weak.expired() ? "true" : "false")
295         << endl;
296     cout << "    weak.use_count(): " << weak.use_count() << endl;
297
298     if (auto sp = weak.lock()) {
299         cout << "    Successfully locked: " << *sp << endl;
300     }
301
302     strong.reset(); // Destroy the object
303     cout << "    After strong.reset():" << endl;
304     cout << "    weak.expired(): " << (weak.expired() ? "true" : "false")
305         << endl;
```

```

303     if (auto sp = weak.lock()) {
304         cout << "    Got shared_ptr" << endl;
305     } else {
306         cout << "    lock() returned nullptr (object destroyed)" << endl;
307     }
308 }
309
310 // 3. make_shared is ONE-WAY - Cannot convert back to unique_ptr
311 {
312     cout << "\n3. make_shared is ONE-WAY (Irreversible):" << endl;
313     cout << "    unique_ptr → shared_ptr: YES (via std::move)" << endl;
314     cout << "    shared_ptr → unique_ptr: NO (impossible)\n" << endl;
315
316     // Forward conversion: unique → shared (OK)
317     unique_ptr<int> unique = make_unique<int>(42);
318     cout << "    unique_ptr created: " << *unique << endl;
319
320     shared_ptr<int> shared = move(unique); // Transfer ownership
321     cout << "    Moved to shared_ptr: " << *shared << endl;
322     cout << "    unique_ptr is now: " << (unique ? "valid" : "nullptr") <<
323         endl;
324
325     // Backward conversion: shared → unique (IMPOSSIBLE)
326     cout << "\n    Why shared_ptr → unique_ptr is IMPOSSIBLE:" << endl;
327     cout << "    • shared_ptr allows multiple owners (ref count)" << endl;
328     cout << "    • unique_ptr requires single ownership" << endl;
329     cout << "    • Cannot guarantee no other shared_ptrs exist" << endl;
330     cout << "    • No std::unique_ptr(std::shared_ptr) constructor" << endl
331         ;
332     cout << "    • No way to 'steal' ownership from shared_ptr\n" << endl;
333
334     // // This would NOT compile:
335     // unique_ptr<int> back_to_unique = move(shared); // ERROR!
336     // unique_ptr<int> back_to_unique(shared.get()); // DANGEROUS!
337     //     Double delete!
338
339     cout << "    Design implication: Choose wisely at creation!" << endl;
340     cout << "    • Use unique_ptr by default (can upgrade later)" << endl;
341     cout << "    • Use shared_ptr only when shared ownership needed" <<
342         endl;
343     cout << "    • Prefer make_unique/make_shared for exception safety" <<
344         endl;
345 }
346
347 // 4. Aliasing constructor
348 {
349     cout << "\n4. Aliasing Constructor:" << endl;
350     struct Data {
351         int x = 10;
352         int y = 20;
353     };
354
355     auto data_ptr = make_shared<Data>();
356     auto x_ptr = shared_ptr<int>(data_ptr, &data_ptr->x);

```

```

352     auto y_ptr = shared_ptr<int>(data_ptr, &data_ptr->y);
353
354     cout << "    Use count: " << data_ptr.use_count() << endl;
355     cout << "    x_ptr points to x but shares ownership of Data" << endl;
356 }
357
358 // 5. enable_shared_from_this
359 {
360     cout << "\n5. enable_shared_from_this:" << endl;
361     class Widget : public enable_shared_from_this<Widget> {
362     public:
363         shared_ptr<Widget> get_shared() {
364             return shared_from_this(); // Safe even if multiple
365             shared_ptrs exist
366         }
367     };
368
369     auto widget = make_shared<Widget>();
370     auto another_ref = widget->get_shared();
371     cout << "    Use count: " << widget.use_count() << endl;
372 }
373
374 // ===== Q2.2: Memory Alignment =====
375 void memory_alignment() {
376     cout << "\n==== MEMORY ALIGNMENT ===\n";
377
378     struct BadLayout {
379         char c;           // 1 byte
380         int i;            // 4 bytes (may need 3 bytes padding after c)
381         double d;          // 8 bytes
382         char c2;          // 1 byte (7 bytes padding at end)
383         // Total: 1 + 3(pad) + 4 + 8 + 1 + 7(pad) = 24 bytes
384     };
385
386     struct GoodLayout {
387         double d;          // 8 bytes
388         int i;            // 4 bytes
389         char c;           // 1 byte
390         char c2;          // 1 byte (2 bytes padding at end)
391         // Total: 8 + 4 + 1 + 1 + 2(pad) = 16 bytes
392     };
393
394     cout << "sizeof(BadLayout): " << sizeof(BadLayout) << " bytes" << endl;
395     cout << "sizeof(GoodLayout): " << sizeof(GoodLayout) << " bytes" << endl;
396     cout << "Savings: " << (sizeof(BadLayout) - sizeof(GoodLayout)) << " bytes"
397         << endl;
398
399     cout << "\nAlignment Rules:" << endl;
400     cout << "1. Data type must be aligned to its size boundary" << endl;
401     cout << "2. struct alignment = largest member alignment" << endl;
402     cout << "3. Reorder members from largest to smallest" << endl;
403     cout << "4. Use alignas() for custom alignment" << endl;
404     cout << "5. Cache line size typically 64 bytes" << endl;

```

```

404 }
405
406 // =====
407 // SECTION 3: OOP & DESIGN PATTERNS
408 // =====
409
410 // ===== Q3.1: Virtual Dispatch Mechanics =====
411 void virtual_dispatch_details() {
412     cout << "\n==== VIRTUAL DISPATCH MECHANICS ===\n";
413
414     class Base {
415     public:
416         virtual void func1() { cout << "Base::func1" << endl; }
417         virtual void func2() { cout << "Base::func2" << endl; }
418         virtual ~Base() {}
419     };
420
421     class Derived : public Base {
422     public:
423         void func1() override { cout << "Derived::func1" << endl; }
424         void func3() { cout << "Derived::func3" << endl; } // Not in vtable
425     };
426
427     // How virtual call works:
428     Base* b = new Derived();
429     b->func1();
430
431     cout << "\nVirtual call process:" << endl;
432     cout << "1. Object contains vptr (pointer to vtable)" << endl;
433     cout << "2. vtable contains function pointers" << endl;
434     cout << "3. b->func1() becomes: (*b->vptr[0])()" << endl;
435     cout << "4. vptr[0] points to Derived::func1" << endl;
436
437     // Multiple inheritance
438     class Base2 {
439     public:
440         virtual void func4() { cout << "Base2::func4" << endl; }
441         virtual ~Base2() {}
442     };
443
444     class MultiDerived : public Base, public Base2 {
445     public:
446         void func1() override { cout << "MultiDerived::func1" << endl; }
447         void func4() override { cout << "MultiDerived::func4" << endl; }
448     };
449
450     MultiDerived md;
451     Base* b1 = &md;
452     Base2* b2 = &md;
453
454     cout << "\nMultiple inheritance layout:" << endl;
455     cout << "MultiDerived object has TWO vptrs" << endl;
456     cout << "b1 and b2 point to different subobjects" << endl;
457     cout << "Address difference: " << (long long)b2 - (long long)b1 << " bytes"

```

```
        " << endl;

458     delete b;
459 }
460
461 // ===== Q3.2: CRTP with Mixins =====
462 template<typename Derived>
463 class Printable {
464 public:
465     void print() const {
466         cout << static_cast<const Derived*>(this)->to_string() << endl;
467     }
468 };
469
470 template<typename Derived>
471 class Comparable {
472 public:
473     bool operator==(const Comparable& other) const {
474         return static_cast<const Derived*>(this)->get_id() ==
475                 static_cast<const Derived*>(&other)->get_id();
476     }
477 };
478
479
480 class Widget : public Printable<Widget>, public Comparable<Widget> {
481     int id;
482     string name;
483
484 public:
485     Widget(int i, string n) : id(i), name(move(n)) {}
486
487     string to_string() const {
488         return "Widget[" + std::to_string(id) + ", " + name + "]";
489     }
490
491     int get_id() const { return id; }
492 };
493
494 void crtp_mixins_demo() {
495     cout << "\n==== CRTP WITH MIXINS ===\n";
496
497     Widget w1(1, "First");
498     Widget w2(2, "Second");
499     Widget w3(1, "Another");
500
501     w1.print();
502     w2.print();
503
504     cout << "w1 == w2: " << (w1 == w2) << endl;
505     cout << "w1 == w3: " << (w1 == w3) << endl;
506
507     cout << "\nCRTP Mixin Benefits:" << endl;
508     cout << "1. Compile-time polymorphism" << endl;
509     cout << "2. No vtable overhead" << endl;
510     cout << "3. Can combine multiple behaviors" << endl;
511 }
```

```
511     cout << "4. Type-safe at compile time" << endl;
512 }
513
514 // =====
515 // SECTION 4: TEMPLATES & METAPROGRAMMING
516 // =====
517
518 // ===== Q4.1: SFINAE Techniques =====
519 template<typename T>
520 auto print_value_impl(const T& value, int)
521     -> decltype(cout << value, void{}) {
522     cout << "Printable: " << value << endl;
523 }
524
525 template<typename T>
526 void print_value_impl(const T&, long) {
527     cout << "Not printable" << endl;
528 }
529
530 template<typename T>
531 void print_value(const T& value) {
532     print_value_impl(value, 0);
533 }
534
535 // SFINAE with enable_if
536 template<typename T>
537 typename enable_if<is_integral<T>::value, void>::type
538 process(T value) {
539     cout << "Processing integral: " << value << endl;
540 }
541
542 template<typename T>
543 typename enable_if<is_floating_point<T>::value, void>::type
544 process(T value) {
545     cout << "Processing float: " << value << endl;
546 }
547
548 void sfinae_demo() {
549     cout << "\n==== SFINAE TECHNIQUES ====\n";
550
551     print_value(42);
552     print_value(vector<int>{1, 2, 3});
553
554     process(10);      // Integral version
555     process(3.14);   // Float version
556 }
557
558 // ===== Q4.2: Variadic Templates =====
559 template<typename... Args>
560 void print_all(Args&&... args) {
561     (cout << ... << args) << endl; // C++17 fold expression
562 }
563
564 void variadic_demo() {
```

```
565     cout << "\n==== VARIADIC TEMPLATES ===\n";
566
567     print_all("Hello", " ", "World", " ", 2024, "!");
568
569     cout << "\nVariadic Benefits:" << endl;
570     cout << "1. Type-safe variadic functions" << endl;
571     cout << "2. Compile-time expansion" << endl;
572     cout << "3. Perfect forwarding support" << endl;
573 }
574
575 // ===== Q4.3: Compile-Time Computation =====
576 constexpr int factorial(int n) {
577     if (n <= 1) return 1;
578     return n * factorial(n - 1);
579 }
580
581 template<int N>
582 struct Factorial {
583     static constexpr int value = N * Factorial<N-1>::value;
584 };
585
586 template<>
587 struct Factorial<0> {
588     static constexpr int value = 1;
589 };
590
591 void compile_time_demo() {
592     cout << "\n==== COMPILE-TIME COMPUTATION ===\n";
593
594     constexpr int fact = factorial(5);
595     cout << "Factorial(5) at compile time: " << fact << endl;
596
597     cout << "Template factorial(5): " << Factorial<5>::value << endl;
598
599     // Static asserts
600     static_assert(factorial(5) == 120, "Compile-time factorial failed");
601     static_assert(Factorial<5>::value == 120, "Template factorial failed");
602
603     cout << "\nCompile-Time Benefits:" << endl;
604     cout << "1. Zero runtime overhead" << endl;
605     cout << "2. Computed during compilation" << endl;
606     cout << "3. Can be used in constant expressions" << endl;
607 }
608
609 // =====
610 // SECTION 5: CONCURRENCY & MULTITHREADING
611 // =====
612
613 // ===== Q5.1: Thread-Safe Singleton =====
614 class ThreadSafeSingleton {
615     static mutex instance_mutex;
616     static unique_ptr<ThreadSafeSingleton> instance;
617
618     ThreadSafeSingleton() {
```

```

619         cout << "Singleton created" << endl;
620     }
621
622 public:
623     static ThreadSafeSingleton& getInstance() {
624         // C++11 guarantees thread-safe static initialization
625         static ThreadSafeSingleton instance;
626         return instance;
627     }
628
629     // Better: use call_once
630     static ThreadSafeSingleton& getInstanceCallOnce() {
631         static once_flag flag;
632         call_once(flag, []() {
633             cout << "Initialized with call_once" << endl;
634         });
635         static ThreadSafeSingleton instance;
636         return instance;
637     }
638
639     void doSomething() {
640         cout << "Singleton method called" << endl;
641     }
642
643     // Delete copy/move
644     ThreadSafeSingleton(const ThreadSafeSingleton&) = delete;
645     ThreadSafeSingleton& operator=(const ThreadSafeSingleton&) = delete;
646 };
647
648 void singleton_demo() {
649     cout << "\n==== THREAD-SAFE SINGLETON ===\n";
650
651     vector<thread> threads;
652     for (int i = 0; i < 5; ++i) {
653         threads.emplace_back([]() {
654             ThreadSafeSingleton::getInstance().doSomething();
655         });
656     }
657
658     for (auto& t : threads) {
659         t.join();
660     }
661
662     cout << "\nSingleton Patterns:" << endl;
663     cout << "1. Meyer's Singleton (C++11 static init)" << endl;
664     cout << "2. std::call_once for initialization" << endl;
665     cout << "3. Double-checked locking (avoid!)" << endl;
666     cout << "4. Thread-local storage (per-thread instances)" << endl;
667 }
668
669 // ===== Q5.2: Producer-Consumer with Condition Variables =====
670 class ProducerConsumer {
671     queue<int> buffer;
672     mutex mtx;

```

```
673     condition_variable cv_producer, cv_consumer;
674     const size_t max_size = 10;
675     bool done = false;
676
677 public:
678     void produce(int id, int count) {
679         for (int i = 0; i < count; ++i) {
680             unique_lock lock(mtx);
681             cv_producer.wait(lock, [this] {
682                 return buffer.size() < max_size || done;
683             });
684
685             if (done) break;
686
687             buffer.push(id * 100 + i);
688             cout << "Produced: " << (id * 100 + i) << " (buffer size: " <<
689                 buffer.size() << ")" << endl;
690
691             cv_consumer.notify_one();
692             this_thread::sleep_for(chrono::milliseconds(50));
693         }
694     }
695
696     void consume(int id, int count) {
697         for (int i = 0; i < count; ++i) {
698             unique_lock lock(mtx);
699             cv_consumer.wait(lock, [this] {
700                 return !buffer.empty() || done;
701             });
702
703             if (done && buffer.empty()) break;
704
705             int value = buffer.front();
706             buffer.pop();
707             cout << "Consumed: " << value << " by consumer " << id << " (
708                 buffer size: " << buffer.size() << ")" << endl;
709
710             cv_producer.notify_one();
711             this_thread::sleep_for(chrono::milliseconds(100));
712         }
713     }
714
715     void stop() {
716         {
717             lock_guard lock(mtx);
718             done = true;
719         }
720         cv_producer.notify_all();
721         cv_consumer.notify_all();
722     }
723 };
724
725 void producer_consumer_demo() {
726     cout << "\n==== PRODUCER-CONSUMER PATTERN ===\n";
```

```
725
726     ProducerConsumer pc;
727
728     thread producer1([&pc]() { pc.produce(1, 5); });
729     thread producer2([&pc]() { pc.produce(2, 5); });
730     thread consumer1([&pc]() { pc.consume(1, 5); });
731     thread consumer2([&pc]() { pc.consume(2, 5); });
732
733     producer1.join();
734     producer2.join();
735     consumer1.join();
736     consumer2.join();
737
738     pc.stop();
739
740     cout << "\nCondition Variable Best Practices:" << endl;
741     cout << "1. Always use with a mutex" << endl;
742     cout << "2. Use wait() with predicate to avoid spurious wakeups" << endl;
743     cout << "3. notify_one() vs notify_all()" << endl;
744     cout << "4. Consider separate CVs for different conditions" << endl;
745 }
746
747 // ===== Q5.3: Atomic Operations =====
748 void atomic_operations_demo() {
749     cout << "\n== ATOMIC OPERATIONS ==\n";
750
751     atomic<int> counter{0};
752     atomic<bool> ready{false};
753
754     vector<thread> threads;
755     for (int i = 0; i < 10; ++i) {
756         threads.emplace_back([&counter, &ready]() {
757             while (!ready.load(memory_order_acquire)) {
758                 this_thread::yield();
759             }
760
761             for (int j = 0; j < 1000; ++j) {
762                 counter.fetch_add(1, memory_order_relaxed);
763             }
764         });
765     }
766
767     ready.store(true, memory_order_release);
768
769     for (auto& t : threads) {
770         t.join();
771     }
772
773     cout << "Final counter value: " << counter.load() << " (expected: 10000)"
774         << endl;
775
776     cout << "\nAtomic Operation Types:" << endl;
777     cout << "1. load/store - read/write atomic value" << endl;
778     cout << "2. fetch_add/fetch_sub - atomic arithmetic" << endl;
```

```
778     cout << "3. compare_exchange_weak/strong - CAS operations" << endl;
779     cout << "4. Memory ordering: relaxed, acquire, release, seq_cst" << endl;
780 }
781 // =====
782 // SECTION 6: PERFORMANCE & OPTIMIZATION
783 // =====
784 // ===== Q6.1: Move Semantics Performance =====
785
786 class LargeObject {
787     vector<int> data;
788
789 public:
790     LargeObject(size_t size) : data(size, 42) {
791         cout << "Constructor: " << size << " elements" << endl;
792     }
793
794     // Copy constructor (expensive)
795     LargeObject(const LargeObject& other) : data(other.data) {
796         cout << "Copy constructor: " << data.size() << " elements" << endl;
797     }
798
799     // Move constructor (cheap)
800     LargeObject(LargeObject&& other) noexcept : data(move(other.data)) {
801         cout << "Move constructor" << endl;
802     }
803
804     // Copy assignment
805     LargeObject& operator=(const LargeObject& other) {
806         if (this != &other) {
807             data = other.data;
808             cout << "Copy assignment: " << data.size() << " elements" << endl;
809         }
810         return *this;
811     }
812
813     // Move assignment
814     LargeObject& operator=(LargeObject&& other) noexcept {
815         if (this != &other) {
816             data = move(other.data);
817             cout << "Move assignment" << endl;
818         }
819         return *this;
820     }
821 }
822 };
823
824 void move_semantics_performance() {
825     cout << "\n==== MOVE SEMANTICS PERFORMANCE ===\n";
826
827     {
828         cout << "\nCopy version:" << endl;
829         LargeObject obj1(1000000);
830         LargeObject obj2 = obj1; // Copy
831     }
```

```
832
833     {
834         cout << "\nMove version:" << endl;
835         LargeObject obj1(1000000);
836         LargeObject obj2 = move(obj1); // Move
837     }
838
839     cout << "\nMove Semantics Guidelines:" << endl;
840     cout << "1. Return by value (compiler uses RVO/NRVO)" << endl;
841     cout << "2. Use std::move() for rvalues" << endl;
842     cout << "3. Mark move constructors noexcept" << endl;
843     cout << "4. Don't move from const objects" << endl;
844     cout << "5. Perfect forwarding with std::forward" << endl;
845 }
846
847 // ===== Q6.2: RVO and Copy Elision =====
848 class Expensive {
849 public:
850     Expensive() {
851         cout << "Default constructor" << endl;
852     }
853
854     Expensive(const Expensive&)
855         cout << "Copy constructor" << endl;
856     }
857
858     Expensive(Expensive&&) noexcept {
859         cout << "Move constructor" << endl;
860     }
861 };
862
863 Expensive createExpensive() {
864     return Expensive{}; // RVO applies
865 }
866
867 Expensive createExpensiveNamed() {
868     Expensive obj;
869     return obj; // NRVO applies
870 }
871
872 void copy_elision_demo() {
873     cout << "\n== COPY ELISION (RVO/NRVO) ==\n";
874
875     cout << "Creating with RVO:" << endl;
876     Expensive e1 = createExpensive();
877
878     cout << "\nCreating with NRVO:" << endl;
879     Expensive e2 = createExpensiveNamed();
880
881     cout << "\nCopy Elision Rules:" << endl;
882     cout << "1. RVO: Return Value Optimization" << endl;
883     cout << "2. NRVO: Named Return Value Optimization" << endl;
884     cout << "3. Guaranteed in C++17 for prvalue" << endl;
885     cout << "4. Compiler may apply in other cases" << endl;
```

```
886     cout << "5. Don't use std::move() on return values" << endl;
887 }
888
889 // =====
890 // MAIN - Demonstrate All Interview Topics
891 // =====
892
893 int main() {
894     cout << "\n";
895     cout << "                                \n";
896     cout << "          COMPLETE C++ INTERVIEW QUESTIONS & ANSWERS
897          \n";
898     cout << "          Mock Interview Preparation Guide
899          \n";
900     cout << "                                \n";
901
902     try {
903         // Section 1: C++ Fundamentals
904         cout << "\n" << string(70, '=') << "\n";
905         cout << "SECTION 1: C++ FUNDAMENTALS\n";
906         cout << string(70, '=') << "\n";
907
908         pointers_vs_references();
909         const_correctness();
910         raii_demo();
911
912         // Section 2: Memory Management
913         cout << "\n" << string(70, '=') << "\n";
914         cout << "SECTION 2: MEMORY MANAGEMENT\n";
915         cout << string(70, '=') << "\n";
916
917         smart_pointers_advanced();
918         memory_alignment();
919
920         // Section 3: OOP & Design Patterns
921         cout << "\n" << string(70, '=') << "\n";
922         cout << "SECTION 3: OOP & DESIGN PATTERNS\n";
923         cout << string(70, '=') << "\n";
924
925         virtual_dispatch_details();
926         crtp_mixins_demo();
927
928         // Section 4: Templates & Metaprogramming
929         cout << "\n" << string(70, '=') << "\n";
930         cout << "SECTION 4: TEMPLATES & METAPROGRAMMING\n";
931         cout << string(70, '=') << "\n";
932
933         sfinae_demo();
934         variadic_demo();
935         compile_time_demo();
936
937         // Section 5: Concurrency & Multithreading
938         cout << "\n" << string(70, '=') << "\n";
939         cout << "SECTION 5: CONCURRENCY & MULTITHREADING\n";
```

```
938     cout << string(70, '=') << "\n";
939
940     singleton_demo();
941     producer_consumer_demo();
942     atomic_operations_demo();
943
944     // Section 6: Performance & Optimization
945     cout << "\n" << string(70, '=') << "\n";
946     cout << "SECTION 6: PERFORMANCE & OPTIMIZATION\n";
947     cout << string(70, '=') << "\n";
948
949     move_semantics_performance();
950     copy_elision_demo();
951
952     // Summary
953     cout << "\n" << string(70, '=') << "\n";
954     cout << "SUMMARY: KEY INTERVIEW TOPICS COVERED\n";
955     cout << string(70, '=') << "\n\n";
956
957     cout << "  FUNDAMENTALS:\n";
958     cout << "  •  Pointers vs References\n";
959     cout << "  •  Const correctness\n";
960     cout << "  •  RAII pattern\n\n";
961
962     cout << "  MEMORY MANAGEMENT:\n";
963     cout << "  •  Smart pointers (unique_ptr, shared_ptr, weak_ptr)\n";
964     cout << "  •  Memory alignment and padding\n";
965     cout << "  •  Custom allocators\n\n";
966
967     cout << "  OOP & DESIGN:\n";
968     cout << "  •  Virtual dispatch mechanics\n";
969     cout << "  •  CRTP with mixins\n";
970     cout << "  •  PIMPL idiom\n\n";
971
972     cout << "  TEMPLATES:\n";
973     cout << "  •  SFINAE techniques\n";
974     cout << "  •  Variadic templates\n";
975     cout << "  •  Compile-time computation\n\n";
976
977     cout << "  CONCURRENCY:\n";
978     cout << "  •  Thread-safe singleton\n";
979     cout << "  •  Producer-consumer pattern\n";
980     cout << "  •  Atomic operations\n\n";
981
982     cout << "  PERFORMANCE:\n";
983     cout << "  •  Move semantics\n";
984     cout << "  •  Copy elision (RVO/NRVO)\n";
985     cout << "  •  Cache-friendly design\n\n";
986
987     cout << "  COMMON INTERVIEW QUESTIONS ANSWERED:\n";
988     cout << "    1. What's the difference between pointers and references?\n";
989     cout << "    2. Explain smart pointers and when to use each\n";
990     cout << "    3. How does virtual dispatch work internally?\n";
```

```
991     cout << " 4. What is SFINAE and how is it used?\n";
992     cout << " 5. How do you implement thread-safe singleton?\n";
993     cout << " 6. What's the difference between RVO and move semantics?\n";
994     cout << " 7. Explain memory ordering in atomic operations\n";
995     cout << " 8. How does CRTP provide compile-time polymorphism?\n\n";
996
997     cout << " ADDITIONAL TOPICS TO STUDY:\n";
998     cout << " • Lambda expressions and closures\n";
999     cout << " • Concepts (C++20)\n";
1000    cout << " • Coroutines (C++20)\n";
1001    cout << " • Ranges library (C++20)\n";
1002    cout << " • std::expected (C++23)\n";
1003    cout << " • Design patterns (Factory, Observer, Strategy)\n\n";
1004
1005    cout << "                                     \n";
1006    cout << "           ALL INTERVIEW TOPICS SUCCESSFULLY DEMONSTRATED!
1007           \n";
1008    cout << "                                     \n\n";
1009
1010 } catch (const exception& e) {
1011     cerr << " Error: " << e.what() << "\n";
1012     return 1;
1013 }
1014
1015 return 0;
}
```

## 42 Source Code: MoveSemantics.cpp

File: src/MoveSemantics.cpp

Repository: [View on GitHub](#)

```

1 // =====
2 // std::move AND MOVE SEMANTICS - COMPREHENSIVE GUIDE
3 // =====
4 // Topics covered:
5 // 1. Lvalues vs Rvalues - What they are and why they matter
6 // 2. std::move - What it actually does (cast to rvalue reference)
7 // 3. Move constructor vs Copy constructor - Performance comparison
8 // 4. Why move semantics are important - Real-world scenarios
9 // 5. Common pitfalls and best practices
10 // 6. Perfect forwarding with std::forward
11 // =====
12
13 #include <iostream>
14 #include <string>
15 #include <vector>
16 #include <memory>
17 #include <chrono>
18 #include <utility> // for std::move, std::forward
19
20 // =====
21 // 1. LVALUES vs RVALUES - Understanding the Basics
22 // =====
23
24 void demonstrate_lvalue_rvalue() {
25     std::cout << "\n== 1. LVALUES vs RVALUES ==" << std::endl;
26
27     std::cout << "\n Theory:" << std::endl;
28     std::cout << "    LVALUE = 'Locator value' - has a name, addressable,
29         persists" << std::endl;
30     std::cout << "    RVALUE = 'Right-hand value' - temporary, about to expire"
31         << std::endl;
32
33     // LVALUES - have names, can take address
34     int x = 42; // x is an lvalue
35     int* ptr = &x; // Can take address of x
36     std::string name = "John"; // name is an lvalue
37
38     std::cout << "\n LVALUES (have names, addressable):" << std::endl;
39     std::cout << "    int x = 42; // x is lvalue, address: " << &x <<
40         std::endl;
41     std::cout << "    string name = \"John\"; // name is lvalue, address: " <<
42         &name << std::endl;
43
44     // RVALUES - temporaries, no name, about to be destroyed
45     std::cout << "\n RVALUES (temporaries, no name):" << std::endl;
46     std::cout << "    42 // literal rvalue" << std::endl;
47     std::cout << "    x + 5 // expression result is rvalue" << std::
48         endl;
49     std::cout << "    string(\"hi\") // temporary object is rvalue" << std::
50         endl;
51 }
```

```
        endl;

45    // x = 42;           // OK: x is lvalue, 42 is rvalue
46    // 42 = x;          // ERROR: 42 is rvalue, can't assign to rvalue
47    // &42;              // ERROR: can't take address of rvalue
48
49
50    std::cout << "\n Key insight:" << std::endl;
51    std::cout << " • Lvalues appear on LEFT of assignment: x = ..." << std::
52        endl;
53    std::cout << " • Rvalues appear on RIGHT of assignment: ... = 42" << std
54        ::endl;
55    std::cout << " • Rvalues are about to die → safe to 'steal' their
56        resources!" << std::endl;
57
58 }

59
60 // =====
61 // 2. std::move - What It Actually Does
62 // =====
63
64 void demonstrate_std_move() {
65     std::cout << "\n== 2. WHAT IS std::move? ==" << std::endl;
66
67     std::cout << "\n std::move is NOT:" << std::endl;
68     std::cout << "    Moving anything" << std::endl;
69     std::cout << "    Transferring ownership" << std::endl;
70     std::cout << "    Doing any work" << std::endl;
71
72     std::cout << "\n std::move IS:" << std::endl;
73     std::cout << " • A CAST from lvalue to rvalue reference" << std::endl;
74     std::cout << " • Tells compiler: 'This object is about to expire'" <<
75         std::endl;
76     std::cout << " • Enables move constructor/assignment to be called" <<
77         std::endl;
78
79     std::cout << "\n Conceptual implementation:" << std::endl;
80     std::cout << "    template<typename T>" << std::endl;
81     std::cout << "    typename remove_reference<T>::type&& move(T&& arg) {" <<
82         std::endl;
83     std::cout << "        return static_cast<typename remove_reference<T>::type
84         &&>(arg);" << std::endl;
85     std::cout << "    }" << std::endl;
86
87     std::string str1 = "Hello";
88
89     std::cout << "\n Example:" << std::endl;
90     std::cout << "    string str1 = \"Hello\"; // str1 is lvalue" << std::endl
91         ;
92     std::cout << "    string str2 = str1;      // COPY: str1 is lvalue → copy
93         constructor" << std::endl;
94     std::cout << "    string str3 = std::move(str1); // MOVE: std::move(str1)
95         is rvalue → move constructor" << std::endl;
96
97     std::string str2 = str1;           // Copy
98     std::string str3 = std::move(str1); // Move (str1 now in valid but
```

```
        unspecified state)

88
89     std::cout << "\n  After move:" << std::endl;
90     std::cout << "    str1 = \"" << str1 << "\"" (moved-from, valid but empty)"
91         << std::endl;
92     std::cout << "    str2 = \"" << str2 << "\"" (copy, unchanged)" << std::endl
93         ;
94     std::cout << "    str3 = \"" << str3 << "\"" (move, got str1's data)" << std
95         ::endl;
96 }
97
98 // =====
99 // 3. COPY vs MOVE - Performance Comparison
100 // =====
101
102 class LargeObject {
103 private:
104     int* data;
105     size_t size;
106     std::string name;
107
108 public:
109     // Constructor
110     LargeObject(const std::string& n, size_t s)
111         : size(s), name(n) {
112         data = new int[size];
113         for (size_t i = 0; i < size; ++i) {
114             data[i] = static_cast<int>(i);
115         }
116         std::cout << "    Constructor: " << name
117             << " (" << size << " elements)" << std::endl;
118     }
119
120     // Destructor
121     ~LargeObject() {
122         delete[] data;
123         std::cout << "    Destructor: " << name << std::endl;
124     }
125
126     // Copy constructor (EXPENSIVE)
127     LargeObject(const LargeObject& other)
128         : size(other.size), name(other.name + "_copy") {
129         data = new int[size];
130         // Copy every element - EXPENSIVE!
131         for (size_t i = 0; i < size; ++i) {
132             data[i] = other.data[i];
133         }
134         std::cout << "    COPY constructor: " << name
135             << " (" << size << " elements copied)" << std::endl;
136     }
137
138     // Move constructor (CHEAP)
139     LargeObject(LargeObject&& other) noexcept
140         : data(other.data), size(other.size), name(other.name + "_moved") {
```

```
138     // Steal the data pointer - CHEAP!
139     other.data = nullptr;
140     other.size = 0;
141     std::cout << "      MOVE constructor: " << name
142             << " (pointer stolen, zero copy!)" << std::endl;
143 }
144
145 // Copy assignment (EXPENSIVE)
146 LargeObject& operator=(const LargeObject& other) {
147     if (this != &other) {
148         delete[] data;
149         size = other.size;
150         name = other.name + "_copy_assigned";
151         data = new int[size];
152         for (size_t i = 0; i < size; ++i) {
153             data[i] = other.data[i];
154         }
155         std::cout << "      COPY assignment: " << name << std::endl;
156     }
157     return *this;
158 }
159
160 // Move assignment (CHEAP)
161 LargeObject& operator=(LargeObject&& other) noexcept {
162     if (this != &other) {
163         delete[] data;
164         data = other.data;
165         size = other.size;
166         name = other.name + "_move_assigned";
167         other.data = nullptr;
168         other.size = 0;
169         std::cout << "      MOVE assignment: " << name << std::endl;
170     }
171     return *this;
172 }
173
174     size_t get_size() const { return size; }
175 };
176
177 void demonstrate_copy_vs_move() {
178     std::cout << "\n--- 3. COPY vs MOVE CONSTRUCTORS ---" << std::endl;
179
180     std::cout << "\n--- Copy Constructor (expensive) ---" << std::endl;
181 {
182     LargeObject obj1("obj1", 1000);
183     LargeObject obj2 = obj1; // COPY: obj1 is lvalue
184     std::cout << "      Result: obj1 still has data, obj2 copied it" << std::endl;
185 }
186
187 std::cout << "\n--- Move Constructor (cheap) ---" << std::endl;
188 {
189     LargeObject obj3("obj3", 1000);
190     LargeObject obj4 = std::move(obj3); // MOVE: std::move(obj3) is
```

```

191     rvalue
192     std::cout << "    Result: obj3 empty (moved-from), obj4 stole its data"
193             << std::endl;
194 }
195
196 std::cout << "\n--- Assignment operators ---" << std::endl;
197 {
198     LargeObject obj5("obj5", 500);
199     LargeObject obj6("obj6", 500);
200     LargeObject obj7("obj7", 500);
201
202     std::cout << "\nCopy assignment:" << std::endl;
203     obj6 = obj5; // Copy assignment
204
205     std::cout << "\nMove assignment:" << std::endl;
206     obj7 = std::move(obj5); // Move assignment
207 }
208
209 std::cout << "\n Performance impact:" << std::endl;
210 std::cout << " • Copy: O(n) - copies every element" << std::endl;
211 std::cout << " • Move: O(1) - just copies pointers" << std::endl;
212 std::cout << " • For 1,000,000 elements: Copy = 1M ops, Move = 3 ops!"
213
214 // =====
215 // 4. WHY MOVE SEMANTICS ARE IMPORTANT
216 // =====
217
218 // Scenario 1: Returning large objects from functions
219 std::vector<int> create_large_vector_without_move() {
220     std::vector<int> vec(1000000, 42);
221     // Before C++11: Copy entire vector on return (SLOW!)
222     // With C++11: RVO or move (FAST!)
223     return vec; // Move constructor called (or RVO)
224 }
225
226 // Scenario 2: Storing move-only types in containers
227
228 class MoveOnlyType {
229     std::unique_ptr<int> ptr;
230
231 public:
232     MoveOnlyType(int value) : ptr(std::make_unique<int>(value)) {
233         std::cout << "    MoveOnlyType created with value: " << *ptr << std::endl;
234     }
235
236     // Delete copy constructor and copy assignment
237     MoveOnlyType(const MoveOnlyType&) = delete;
238     MoveOnlyType& operator=(const MoveOnlyType&) = delete;
239
240     // Enable move constructor and move assignment

```

```
241     MoveOnlyType(MoveOnlyType&&) noexcept = default;  
242     MoveOnlyType& operator=(MoveOnlyType&&) noexcept = default;  
243  
244     int get_value() const { return *ptr; }  
245 };  
246  
247 void demonstrate_why_move_important() {  
248     std::cout << "\n==== 4. WHY MOVE SEMANTICS ARE IMPORTANT ===" << std::endl;  
249  
250     // Scenario 1: Return values  
251     std::cout << "\n Scenario 1: Returning large objects" << std::endl;  
252     std::cout << "    Before C++11: Expensive copy on return" << std::endl;  
253     std::cout << "    With C++11: Move or RVO (Return Value Optimization)" <<  
254         std::endl;  
255  
256     auto start = std::chrono::high_resolution_clock::now();  
257     auto vec = create_large_vector_without_move();  
258     auto end = std::chrono::high_resolution_clock::now();  
259     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end  
260         - start);  
261  
262     std::cout << "    Created vector with " << vec.size()  
263         << " elements in " << duration.count() << " s" << std::endl;  
264     std::cout << "    (Would be much slower without move/RVO)" << std::endl;  
265  
266     // Scenario 2: Move-only types  
267     std::cout << "\n Scenario 2: Move-only types (unique_ptr, thread, etc.)"  
268         << std::endl;  
269     {  
270         MoveOnlyType obj1(42);  
271         // MoveOnlyType obj2 = obj1; // ERROR: Copy deleted  
272         MoveOnlyType obj2 = std::move(obj1); // OK: Move  
273         std::cout << "    Moved unique ownership from obj1 to obj2" << std::endl;  
274     }  
275  
276     // Scenario 3: Container operations  
277     std::cout << "\n Scenario 3: Container optimizations" << std::endl;  
278     {  
279         std::vector<std::string> names;  
280         names.reserve(3);  
281  
282         std::string name1 = "Alice";  
283         std::string name2 = "Bob";  
284  
285         std::cout << "    push_back(name1): COPY (name1 still needed)" << std::endl;  
286         names.push_back(name1); // Copy  
287  
288         std::cout << "    push_back(std::move(name2)): MOVE (name2 not needed)" << std::endl;  
289         names.push_back(std::move(name2)); // Move  
290  
291         std::cout << "    push_back(\"Charlie\"): MOVE (temporary rvalue)" <<
```

```

        std::endl;
289     names.push_back("Charlie"); // Move from temporary
290
291     std::cout << "\n  After operations:" << std::endl;
292     std::cout << "    name1 = \" " << name1 << "\" (still valid, was copied)
293         " << std::endl;
294     std::cout << "    name2 = \" " << name2 << "\" (moved-from, empty)" <<
295         std::endl;
296
297 // Scenario 4: Swapping
298 std::cout << "\n Scenario 4: Efficient swapping" << std::endl;
299 {
300     std::vector<int> vec1(1000, 1);
301     std::vector<int> vec2(2000, 2);
302
303     std::cout << "    Before swap: vec1.size()=" << vec1.size()
304         << ", vec2.size()=" << vec2.size() << std::endl;
305
306     // std::swap uses move semantics internally
307     std::swap(vec1, vec2); // O(1) with moves, not O(n) with copies!
308
309     std::cout << "    After swap: vec1.size()=" << vec1.size()
310         << ", vec2.size()=" << vec2.size() << std::endl;
311     std::cout << "    (Swap is O(1) thanks to move semantics)" << std::endl
312         ;
313 }
314
315 // =====
316 // 5. COMMON PITFALLS AND BEST PRACTICES
317 // =====
318 void demonstrate_pitfalls() {
319     std::cout << "\n== 5. COMMON PITFALLS ==" << std::endl;
320
321     // Pitfall 1: Using moved-from object
322     std::cout << "\n Pitfall 1: Using moved-from object" << std::endl;
323     {
324         std::string str1 = "Hello";
325         std::string str2 = std::move(str1);
326
327         std::cout << "    str1 after move: \" " << str1 << "\" << std::endl;
328         std::cout << "    str1 is in valid but unspecified state!" << std::endl;
329         std::cout << "    Safe: Check if empty or reassign" << std::endl;
330         std::cout << "    Unsafe: Assume it has specific content" << std::endl;
331
332         // Safe operations after move:
333         str1.clear(); // OK
334         str1 = "New value"; // OK
335         if (str1.empty()) {} // OK
336         // str1[0]; // UNSAFE! May be undefined behavior

```

```
337 }
338
339 // Pitfall 2: Moving const objects
340 std::cout << "\n Pitfall 2: Can't move from const" << std::endl;
341 {
342     const std::string str1 = "Hello";
343     std::string str2 = std::move(str1); // Calls COPY, not move!
344
345     std::cout << "    const objects can't be moved" << std::endl;
346     std::cout << "    std::move on const → copy constructor called" << std
347         ::endl;
348 }
349
350 // Pitfall 3: Unnecessary moves
351 std::cout << "\n Pitfall 3: Unnecessary std::move on return" << std::endl
352     ;
353 std::cout << R"(

// BAD - prevents RVO (Return Value Optimization)
353 string bad_function() {
354     string result = "value";
355     return std::move(result); // Don't do this!
356 }

// GOOD - enables RVO
358 string good_function() {
359     string result = "value";
360     return result; // Compiler automatically moves/optimizes
361 }
362 )" << std::endl;
364
365 // Best practices
366 std::cout << "\n BEST PRACTICES:" << std::endl;
367 std::cout << "    1. Mark move constructors 'noexcept'" << std::endl;
368 std::cout << "    →      Enables optimizations (vector resize)" << std::endl;
369 std::cout << "    2. Don't std::move on return values" << std::endl;
370 std::cout << "    →      Compiler does RVO or automatic move" << std::endl;
371 std::cout << "    3. Move when you know object won't be used again" << std
372         ::endl;
373 std::cout << "    →      push_back(std::move(obj))" << std::endl;
374 std::cout << "    4. Don't use moved-from objects" << std::endl;
375 std::cout << "    →      Unless reassigning or destroying" << std::endl;
376 std::cout << "    5. Move-only types (unique_ptr) can't be copied" << std::
377         endl;
378 std::cout << "    →      Use std::move to transfer ownership" << std::endl;
379 }
380
381 // =====
382 // 6. std::forward - PERFECT FORWARDING
383 // =====
384
385 void process(int& x) {
386     std::cout << "    process(int&): lvalue " << x << std::endl;
387 }
```

```
387 void process(int&& x) {
388     std::cout << "    process(int&&): rvalue " << x << std::endl;
389 }
390
391 // Without perfect forwarding - loses rvalue-ness
392 template<typename T>
393 void wrapper_bad(T&& arg) {
394     // arg is always an lvalue inside the function (even if T&& is rvalue ref)
395     process(arg); // Always calls process(int&)
396 }
397
398 // With perfect forwarding - preserves value category
399 template<typename T>
400 void wrapper_good(T&& arg) {
401     // std::forward preserves lvalue/rvalue-ness
402     process(std::forward<T>(arg)); // Calls correct overload
403 }
404
405 void demonstrate_perfect_forwarding() {
406     std::cout << "\n==== 6. std::forward - PERFECT FORWARDING ===" << std::endl
407         ;
408
409     std::cout << "\n Problem: Template functions lose value category" << std
410         ::endl;
411     std::cout << "    Template parameter T&& is 'forwarding reference'" << std
412         ::endl;
413     std::cout << "    But inside function, arg is always lvalue!" << std::endl;
414
415     int x = 42;
416
417     std::cout << "\n Without std::forward:" << std::endl;
418     wrapper_bad(x); // lvalue
419     wrapper_bad(100); // rvalue, but treated as lvalue!
420
421     std::cout << "\n With std::forward:" << std::endl;
422     wrapper_good(x); // Forwards as lvalue
423     wrapper_good(200); // Forwards as rvalue
424
425     std::cout << "\n Key differences:" << std::endl;
426     std::cout << "    • std::move:    ALWAYS casts to rvalue" << std::endl;
427     std::cout << "    • std::forward: Casts to rvalue ONLY if original was
428         rvalue" << std::endl;
429     std::cout << "                                (preserves value category)" << std::endl;
430
431     std::cout << "\n When to use:" << std::endl;
432     std::cout << "    • std::move:    When you know object is done being used"
433         << std::endl;
434     std::cout << "    • std::forward: In template functions forwarding
435         arguments" << std::endl;
436 }
437
438 // =====
439 // 7. REAL-WORLD EXAMPLE: Factory Pattern
440 // =====
```

```
435
436 class Resource {
437     std::string name;
438     std::vector<int> data;
439
440 public:
441     Resource(const std::string& n) : name(n) {
442         data.resize(1000, 42);
443         std::cout << "    Resource created: " << name << std::endl;
444     }
445
446     Resource(const Resource& other)
447         : name(other.name + "_copy"), data(other.data) {
448         std::cout << "    Resource COPIED: " << name << std::endl;
449     }
450
451     Resource(Resource&& other) noexcept
452         : name(std::move(other.name)), data(std::move(other.data)) {
453         std::cout << "    Resource MOVED: " << name << std::endl;
454     }
455
456     const std::string& get_name() const { return name; }
457 };
458
459 class ResourceFactory {
460 public:
461     // Return by value - move semantics makes this efficient
462     static Resource create_resource(const std::string& name) {
463         Resource res(name);
464         // Return local object - move or RVO kicks in
465         return res; // Don't use std::move here!
466     }
467
468     // Store in container - use move to avoid copy
469     static void store_in_container(std::vector<Resource>& container, Resource
470         && res) {
471         std::cout << "    Storing in container..." << std::endl;
472         container.push_back(std::move(res)); // Move into container
473     }
474
475     void demonstrate_real_world() {
476         std::cout << "\n== 7. REAL-WORLD: Factory Pattern ==" << std::endl;
477
478         std::vector<Resource> resources;
479         resources.reserve(3);
480
481         std::cout << "\n--- Creating and storing resources ---" << std::endl;
482
483         // Create and store - efficient with move semantics
484         auto res1 = ResourceFactory::create_resource("Resource1");
485         ResourceFactory::store_in_container(resources, std::move(res1));
486
487         // Direct temporary - even more efficient
```

```
488     ResourceFactory::store_in_container(
489         resources,
490         ResourceFactory::create_resource("Resource2")
491     );
492
493     std::cout << "\n Container now has " << resources.size()
494         << " resources" << std::endl;
495     std::cout << "    All stored efficiently with move semantics!" << std::endl
496         ;
497 }
498 // =====
499 // MAIN
500 // =====
501
502 int main() {
503     std::cout << "\n
504         =====" <<
505         std::endl;
506     std::cout << "    std::move AND MOVE SEMANTICS - COMPREHENSIVE GUIDE" << std
507         ::endl;
508     std::cout << "
509         =====" <<
510         std::endl;
511
512     demonstrate_lvalue_rvalue();
513     demonstrate_std_move();
514     demonstrate_copy_vs_move();
515     demonstrate_why_move_important();
516     demonstrate_pitfalls();
517     demonstrate_perfect_forwarding();
518     demonstrate_real_world();
519
520     std::cout << "\n
521         =====" <<
522         std::endl;
523     std::cout << "    SUMMARY: std::move and Move Semantics" << std::endl;
524     std::cout << "
525         =====" <<
526         std::endl;
527
528     std::cout << "\n1  KEY CONCEPTS:" << std::endl;
529     std::cout << "\n1  Lvalue vs Rvalue:" << std::endl;
530     std::cout << "    •    Lvalue: Has name, addressable, persists" << std::endl;
531     std::cout << "    •    Rvalue: Temporary, about to expire, no name" << std::
532         endl;
533
534     std::cout << "\n2  std::move:" << std::endl;
535     std::cout << "    •    Just a CAST from lvalue to rvalue reference" << std::
536         endl;
537     std::cout << "    •    Doesn't actually move anything" << std::endl;
538     std::cout << "    •    Tells compiler 'safe to steal resources'" << std::endl;
539
540     std::cout << "\n3  Move Constructor/Assignment:" << std::endl;
```

```
530     std::cout << " • Steals resources from source object" << std::endl;
531     std::cout << " • O(1) instead of O(n) for large objects" << std::endl;
532     std::cout << " • Should be marked 'noexcept'" << std::endl;
533
534     std::cout << "\n4 Why Important:" << std::endl;
535     std::cout << " • Performance: O(1) vs O(n) for copies" << std::endl;
536     std::cout << " • Enables move-only types (unique_ptr, thread)" << std::endl;
537     std::cout << " • Efficient return values and container ops" << std::endl;
538     std::cout << " • Essential for modern C++" << std::endl;
539
540     std::cout << "\n5 Best Practices:" << std::endl;
541     std::cout << " • DO: Mark move operations noexcept" << std::endl;
542     std::cout << " • DO: Move when object not needed anymore" << std::endl;
543     std::cout << " • DO: Use std::forward in template forwarding" << std::endl;
544     std::cout << " • DON'T: std::move on return values (blocks RVO)" << std::endl;
545     std::cout << " • DON'T: Use moved-from objects (unless reassigning)" << std::endl;
546     std::cout << " • DON'T: Move const objects (silently copies)" << std::endl;
547
548     std::cout << "\n PERFORMANCE IMPACT:" << std::endl;
549     std::cout << " • vector<int>(1,000,000):" << std::endl;
550     std::cout << " • Copy: ~4,000,000 bytes copied, ~1ms" << std::endl;
551     std::cout << " • Move: ~24 bytes copied, ~1s (1000x faster!)" << std::endl;
552
553     std::cout << "\n WHEN TO USE:" << std::endl;
554     std::cout << " • Returning local objects from functions" << std::endl;
555     std::cout << " • Inserting into containers (push_back)" << std::endl;
556     std::cout << " • Transferring unique ownership (unique_ptr)" << std::endl;
557     std::cout << " • Implementing swap operations" << std::endl;
558     std::cout << " • Last use of a variable before destruction" << std::endl;
559     std::cout << " • ";
560     std::cout << "\n";
561     std::cout << "=====\\n" << std::endl;
562
563 }
```

## 43 Source Code: MoveSemanticsPerfectForwarding.cpp

File: src/MoveSemanticsPerfectForwarding.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // IMPLEMENTING MOVE SEMANTICS AND PERFECT FORWARDING
3 // =====
4 // A comprehensive guide to C++11/14/17/20 move semantics and perfect
5 // forwarding
6 //
7 // Topics Covered:
8 // 1. Lvalues vs Rvalues - Understanding value categories
9 // 2. Move Semantics - Efficient resource transfer
10 // 3. std::move - What it really does
11 // 4. Perfect Forwarding - Preserving value categories
12 // 5. Rule of Zero/Three/Five
13 // 6. Real-World Examples
14 // 7. Performance Comparison
15 //
16 // Build: g++ -std=c++20 -Wall -Wextra -O2 -o MoveSemanticsPerfectForwarding
17 // MoveSemanticsPerfectForwarding.cpp
18 // =====
19
20 #include <iostream>
21 #include <vector>
22 #include <string>
23 #include <memory>
24 #include <utility> // std::move, std::forward
25 #include <chrono>
26 //
27 // =====
28 // SECTION 1: LVALUES vs RVALUES
29 // =====
30
31 namespace lvalues_rvalues {
32
33 void demonstrate() {
34     std::cout << "\n" << std::string(70, '=') << "\n";
35     std::cout << "SECTION 1: LVALUES vs RVALUES\n";
36     std::cout << std::string(70, '=') << "\n\n";
37
38     std::cout << "  THEORY:\n";
39     std::cout << "  Lvalue = Has a NAME and PERSISTENT ADDRESS\n";
40     std::cout << "  Rvalue = TEMPORARY object without persistent address\n\n";
41
42     // LVALUES - have names, can take address
43     int x = 42;
44     std::string name = "Hello";
45
46     std::cout << "  LVALUES (have names, addressable):\n";
47     std::cout << "    int x = 42;           // x is lvalue, address: " << &x
48     << "\n";
49 }
```

```

46     std::cout << "    string name = \"Hello\"; // name is lvalue, address: " <<
47     &name << "\n\n";
48
49     // RVALUES - temporaries
50     std::cout << "    RVALUES (temporaries, no persistent address):\n";
51     std::cout << "    42                      // Literal\n";
52     std::cout << "    x + 1                  // Expression result\n";
53     std::cout << "    string(\"World\")      // Temporary object\n\n";
54
55     // REFERENCE BINDING
56     std::cout << "    REFERENCE BINDING:\n\n";
57
58     int y = 10;
59
60     // Lvalue reference - binds to lvalue
61     int& lref = y;
62     std::cout << "    int& lref = y;           // OK: lvalue ref binds to
63     lvalue\n";
64     // int& lref2 = 42;        // ERROR: lvalue ref cannot bind to rvalue
65
66     // Rvalue reference - binds to rvalue
67     int&& rref = 42;
68     std::cout << "    int&& rref = 42;         // OK: rvalue ref binds to
69     rvalue\n";
70     // int&& rref2 = y;        // ERROR: rvalue ref cannot bind to lvalue
71
72     // Special case: const lvalue reference can bind to both!
73     const int& cref = 42;
74     std::cout << "    const int& cref = 42;  // OK: const lvalue ref extends
75     lifetime\n\n";
76 }
77
78 } // namespace lvalues_rvalues
79
80 // =====
81 // SECTION 2: MOVE SEMANTICS - Basic Implementation
82 // =====
83
84 namespace move_semantics {
85
86 class Resource {
87 private:
88     int* data_;
89     size_t size_;
90
91 public:
92     // Constructor
93     explicit Resource(size_t size = 0)
94         : data_(size > 0 ? new int[size] : nullptr), size_(size) {

```

```
95         std::cout << "    [Resource] Constructor: allocated " << size_ << "
96             ints\n";
97
98     // Destructor
99     ~Resource() {
100         if (data_) {
101             std::cout << "    [Resource] Destructor: freeing " << size_ << "
102                 ints\n";
103             delete [] data_;
104         }
105     }
106
107     // Copy Constructor - EXPENSIVE! (O(n))
108     Resource(const Resource& other) : size_(other.size_) {
109         std::cout << "    [Resource] Copy Constructor - EXPENSIVE! Copying "
110             << size_ << " ints\n";
111         if (size_ > 0) {
112             data_ = new int[size_];
113             std::copy_n(other.data_, size_, data_);
114         } else {
115             data_ = nullptr;
116         }
117     }
118
119     // Copy Assignment - EXPENSIVE! (O(n))
120     Resource& operator=(const Resource& other) {
121         std::cout << "    [Resource] Copy Assignment - EXPENSIVE! Copying "
122             << other.size_ << " ints\n";
123         if (this != &other) {
124             delete [] data_;
125             size_ = other.size_;
126             if (size_ > 0) {
127                 data_ = new int[size_];
128                 std::copy_n(other.data_, size_, data_);
129             } else {
130                 data_ = nullptr;
131             }
132         }
133         return *this;
134     }
135
136     // Move Constructor - FAST! (O(1))
137     Resource(Resource&& other) noexcept
138         : data_(other.data_), size_(other.size_) {
139             std::cout << "    [Resource] Move Constructor - FAST! O(1) pointer
140                 transfer\n";
141
142             // Leave source in valid state
143             other.data_ = nullptr;
144             other.size_ = 0;
145         }
146
147     // Move Assignment - FAST! (O(1))
```

```
146     Resource& operator=(Resource&& other) noexcept {
147         std::cout << "      [Resource] Move Assignment - FAST! O(1) pointer
148             transfer\n";
149         if (this != &other) {
150             // Clean up existing resource
151             delete[] data_;
152
153             // Transfer ownership
154             data_ = other.data_;
155             size_ = other.size_;
156
157             // Leave source in valid state
158             other.data_ = nullptr;
159             other.size_ = 0;
160         }
161         return *this;
162     }
163
164     size_t size() const { return size_; }
165
166     int* get_data() { return data_; }
167     const int* get_data() const { return data_; }
168 };
169
170 void demonstrate() {
171     std::cout << "\n" << std::string(70, '=') << "\n";
172     std::cout << "SECTION 2: MOVE SEMANTICS\n";
173     std::cout << std::string(70, '=') << "\n\n";
174
175     std::cout << "  THE PROBLEM: Expensive Copies\n";
176     std::cout << "  Copy constructor/assignment performs deep copy (O(n))\n";
177     std::cout << "  For large objects, this is 10x-1000x slower than
178         necessary!\n\n";
179
180     std::cout << "  Demonstrating COPY vs MOVE:\n\n";
181
182     // Copy Constructor
183     std::cout << " 1 COPY Constructor (expensive):\n";
184     Resource r1(5);
185     Resource r2 = r1; // Copy constructor
186     std::cout << "  Result: r1 still valid with " << r1.size() << " ints\n\n"
187         ;
188
189     // Move Constructor
190     std::cout << " 2 MOVE Constructor (fast):\n";
191     Resource r3(5);
192     Resource r4 = std::move(r3); // Move constructor
193     std::cout << "  Result: r3 moved-from (size=" << r3.size()
194         << "), r4 owns data (" << r4.size() << " ints)\n\n";
195
196     // Copy Assignment
197     std::cout << " 3 COPY Assignment (expensive):\n";
198     Resource r5(3);
199     Resource r6(7);
```

```
197     r6 = r5; // Copy assignment
198     std::cout << "    Result: Both valid, r6 has " << r6.size() << " ints\n\n";
199
200 // Move Assignment
201 std::cout << " 4 MOVE Assignment (fast):\n";
202 Resource r7(3);
203 Resource r8(7);
204 r8 = std::move(r7); // Move assignment
205 std::cout << "    Result: r7 moved-from (size=" << r7.size()
206             << "), r8 owns data (" << r8.size() << " ints)\n\n";
207
208 std::cout << "  KEY INSIGHT:\n";
209 std::cout << "    Move operations transfer ownership in O(1) time!\n";
210 std::cout << "    Copy operations duplicate data in O(n) time.\n";
211 std::cout << "    Always mark move operations 'noexcept' for std::vector
212             optimization!\n";
213 }
214 } // namespace move_semantics
215
216 // =====
217 // SECTION 3: UNDERSTANDING std::move
218 // =====
219
220 namespace understanding_move {
221
222 void demonstrate() {
223     std::cout << "\n" << std::string(70, '=') << "\n";
224     std::cout << "SECTION 3: UNDERSTANDING std::move\n";
225     std::cout << std::string(70, '=') << "\n\n";
226
227     std::cout << "  WHAT IS std::move?\n";
228     std::cout << "    std::move does NOT move anything!\n";
229     std::cout << "    It's just a CAST to rvalue reference (T&&)\n";
230     std::cout << "    It tells compiler: 'this object can be moved from'\n\n";
231
232     std::cout << "  Simplified implementation:\n";
233     std::cout << "    template<typename T>\n";
234     std::cout << "    T&& move(T&& t) noexcept {\n";
235     std::cout << "        return static_cast<T&&>(t);\n";
236     std::cout << "    }\n\n";
237
238     std::cout << "  DEMONSTRATION:\n\n";
239
240     // std::move with string
241     std::cout << " 1 std::move with std::string:\n";
242     std::string s1 = "Hello, World!";
243     std::cout << "    Before: s1 = \" " << s1 << "\"\n";
244
245     std::string s2 = std::move(s1); // Move constructor called
246     std::cout << "    After move:\n";
247     std::cout << "    s1 = \" " << s1 << "\" (moved-from state)\n";
248     std::cout << "    s2 = \" " << s2 << "\" (owns the data)\n\n";
249 }
```

```
250 // Moved-from state
251 std::cout << " 2 MOVED-FROM STATE:\n";
252 std::cout << "      After std::move, object is in 'valid but unspecified'
253 std::cout << "      state\n";
254 std::cout << "      SAFE: Destroy it, assign to it\n";
255 std::cout << "      UNSAFE: Don't use its value!\n\n";
256
257 s1 = "New Value"; // Safe: reassignment
258 std::cout << "      s1 reassigned: \" " << s1 << "\" (valid again)\n\n";
259
260 // Common mistakes
261 std::cout << "  COMMON MISTAKES:\n\n";
262
263 std::cout << "  MISTAKE 1: Using object after move\n";
264 std::cout << "      string s3 = \"test\";\n";
265 std::cout << "      string s4 = std::move(s3);\n";
266 std::cout << "      cout << s3; // WRONG! Undefined behavior!\n\n";
267
268 std::cout << "  MISTAKE 2: Returning std::move(local)\n";
269 std::cout << "      string func() {\n";
270 std::cout << "          string s = \"hello\";\n";
271 std::cout << "          return std::move(s); // Prevents RVO!\n";
272 std::cout << "      }\n";
273 std::cout << "      Correct: return s; // RVO or move, NOT copy\n\n";
274
275 std::cout << "  MISTAKE 3: Moving from const\n";
276 std::cout << "      const string cs = \"test\";\n";
277 std::cout << "      string s = std::move(cs); // Calls COPY, not move!\n\n";
278
279 std::cout << "  KEY TAKEAWAY:\n";
280 std::cout << "      std::move is a PERMISSION to move, not a command.\n";
281 std::cout << "      The actual move happens in the move constructor/
282 std::cout << "      assignment.\n";
283 }
284
285 // =====
286 // SECTION 4: PERFECT FORWARDING
287 // =====
288
289 namespace perfect_forwarding {
290
291 class Widget {
292 public:
293     Widget(const std::string& name) {
294         std::cout << "      [Widget] Constructed with LVALUE: \" " << name << "\"
295         (copied)\n";
296     }
297
298     Widget(std::string&& name) {
299         std::cout << "      [Widget] Constructed with RVALUE: \" " << name << "\"
300         (moved)\n";
301 }
```

```

299     }
300 };
301
302 // Bad: Always copies
303 template<typename T>
304 void bad_wrapper(T arg) {
305     // arg is always an lvalue here, even if rvalue was passed!
306     Widget w(arg); // Always calls Widget(const string&)
307 }
308
309 // Good: Perfect forwarding
310 template<typename T>
311 void good_wrapper(T&& arg) { // Universal/Forwarding reference
312     // std::forward preserves the value category
313     Widget w(std::forward<T>(arg));
314 }
315
316 void demonstrate() {
317     std::cout << "\n" << std::string(70, '=') << "\n";
318     std::cout << "SECTION 4: PERFECT FORWARDING\n";
319     std::cout << std::string(70, '=') << "\n\n";
320
321     std::cout << " THE PROBLEM:\n";
322     std::cout << " When passing arguments through wrapper functions,\n";
323     std::cout << " we lose information about whether they were lvalues or
324         rvalues.\n\n";
325
326     std::cout << " DEMONSTRATION:\n\n";
327
328     std::cout << " 1 WITHOUT Perfect Forwarding (bad_wrapper):\n";
329     std::string s1 = "lvalue";
330     bad_wrapper(s1); // Copies (expected)
331     bad_wrapper(std::string("rvalue")); // Still COPIES (not desired!)
332     std::cout << "\n";
333
334     std::cout << " 2 WITH Perfect Forwarding (good_wrapper):\n";
335     std::string s2 = "lvalue";
336     good_wrapper(s2); // Copies (as expected)
337     good_wrapper(std::string("rvalue")); // MOVES (optimized!)
338     std::cout << "\n";
339
340     std::cout << " UNIVERSAL REFERENCE (T&&):\n";
341     std::cout << "     template<typename T>\n";
342     std::cout << "     void func(T&& arg); // NOT an rvalue reference!\n\n";
343     std::cout << "     It's a UNIVERSAL REFERENCE (also called forwarding
344         reference)\n";
345     std::cout << "     Can bind to BOTH lvalues and rvalues\n\n";
346
347     std::cout << " Reference Collapsing Rules:\n";
348     std::cout << "     T& && -> T& (lvalue reference)\n";
349     std::cout << "     T&& && -> T&& (rvalue reference)\n\n";
350
351     std::cout << " WHEN TO USE:\n";
352     std::cout << "     std::move - UNCONDITIONAL cast to rvalue\n";

```

```

351     std::cout << "                                Use when you KNOW you have an lvalue to
352         transfer\n";
353     std::cout << "      std::forward - CONDITIONAL forwarding preserving value
354         category\n";
355     std::cout << "                                Use with T&& (universal reference) in
356         templates\n";
357
358 // Example: Factory function
359 template<typename T, typename... Args>
360 std::unique_ptr<T> make_unique_custom(Args&&... args) {
361     std::cout << "      [Factory] Creating object with perfect forwarding\n";
362     return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
363 }
364
365 void demonstrate_factory() {
366     std::cout << "\n3 Factory Pattern with Perfect Forwarding:\n";
367
368     std::string name = "Gadget";
369     auto w1 = make_unique_custom<Widget>(name);           // Forwards as
370         lvalue
371     auto w2 = make_unique_custom<Widget>("Gizmo");        // Forwards as
372         rvalue
373     auto w3 = make_unique_custom<Widget>(std::move(name)); // Forwards as
374         rvalue
375 }
376
377 } // namespace perfect_forwarding
378
379 // =====
380 // SECTION 5: RULE OF ZERO/THREE/FIVE
381 // =====
382
383 namespace rule_of_five {
384
385 void demonstrate() {
386     std::cout << "\n" << std::string(70, '=') << "\n";
387     std::cout << "SECTION 5: RULE OF ZERO/THREE/FIVE\n";
388     std::cout << std::string(70, '=') << "\n\n";
389
390     std::cout << "      RULE OF ZERO (PREFERRED!):\n";
391     std::cout << "      Don't manage resources manually!\n";
392     std::cout << "      Use std::vector, std::string, std::unique_ptr, etc.\n\n";
393
394     std::cout << "      class Widget {\n";
395     std::cout << "          std::string name_;           // Self-managing\n";
396     std::cout << "          std::vector<int> data_;     // Self-managing\n";
397     std::cout << "          // No special members needed!\n";
398     std::cout << "      };\n\n";
399
400     std::cout << "      RULE OF THREE (C++98):\n";
401     std::cout << "      If you define ONE, define ALL THREE:\n";
402     std::cout << "          1. Destructor\n";
403     std::cout << "          2. Copy Constructor\n";

```

```

399 std::cout << " 3. Copy Assignment Operator\n\n";
400
401 std::cout << " RULE OF FIVE (C++11+):\n";
402 std::cout << " If managing resources, define ALL FIVE:\n";
403 std::cout << " 1. Destructor\n";
404 std::cout << " 2. Copy Constructor\n";
405 std::cout << " 3. Copy Assignment Operator\n";
406 std::cout << " 4. Move Constructor (mark noexcept!)\n";
407 std::cout << " 5. Move Assignment Operator (mark noexcept!)\n\n";
408
409 std::cout << " class Resource {\n";
410 std::cout << " public:\n";
411 std::cout << "     ~Resource(); // 1\n";
412 std::cout << "     Resource(const Resource&); // 2\n";
413 std::cout << "     Resource& operator=(const Resource&); // 3\n";
414 std::cout << "     Resource(Resource&&) noexcept; // 4\n";
415 std::cout << "     Resource& operator=(Resource&&) noexcept; // 5\n";
416 std::cout << " };\n\n";
417
418 std::cout << " DELETE UNWANTED OPERATIONS:\n";
419 std::cout << " Make class non-copyable but movable:\n\n";
420 std::cout << " class NonCopyable {\n";
421 std::cout << " public:\n";
422 std::cout << "     NonCopyable(const NonCopyable&) = delete;\n";
423 std::cout << "     NonCopyable& operator=(const NonCopyable&) = delete;\n";
424 std::cout << "     NonCopyable(NonCopyable&&) = default;\n";
425 std::cout << "     NonCopyable& operator=(NonCopyable&&) = default;\n";
426 std::cout << " };\n\n";
427
428 std::cout << " BEST PRACTICE:\n";
429 std::cout << " 1. Prefer Rule of Zero (use standard library types)\n";
430 std::cout << " 2. If managing resources, use Rule of Five\n";
431 std::cout << " 3. Always mark move operations noexcept\n";
432 std::cout << " 4. Use =delete for unwanted operations\n";
433 std::cout << " 5. Use =default when compiler-generated is correct\n";
434 }
435
436 } // namespace rule_of_five
437
438 // =====
439 // SECTION 6: REAL-WORLD EXAMPLES
440 // =====
441
442 namespace real_world_examples {
443
444 void demonstrate_vector() {
445     std::cout << "\n" << std::string(70, '=') << "\n";
446     std::cout << "SECTION 6: REAL-WORLD EXAMPLES\n";
447     std::cout << std::string(70, '=') << "\n\n";
448
449     std::cout << " 1 std::vector Push Operations:\n\n";
450
451     std::vector<std::string> vec;

```

```
452     vec.reserve(3);
453
454     std::string s1 = "String1";
455     std::string s2 = "String2";
456
457     std::cout << "    vec.push_back(s1);           // Copy (s1 still valid)\n";
458     ;
459     vec.push_back(s1);
460     std::cout << "    s1 after: \" " << s1 << "\"\n\n";
461
462     std::cout << "    vec.push_back(std::move(s2)); // Move (s2 now empty)\n";
463     vec.push_back(std::move(s2));
464     std::cout << "    s2 after: \" " << s2 << "\" (moved-from)\n\n";
465
466     std::cout << "    vec.emplace_back(\"String3\"); // Construct in-place\n";
467     vec.emplace_back("String3");
468     std::cout << "\n";
469 }
470
471 void demonstrate_rvo() {
472     std::cout << " 2 Return Value Optimization (RVO):\n\n";
473
474     std::cout << "      CORRECT:\n";
475     std::cout << "      vector<int> create() {\n";
476     std::cout << "          vector<int> result(1000);\n";
477     std::cout << "          return result; // RVO or move (NOT copy!)\n";
478     std::cout << "      }\n\n";
479
480     std::cout << "      WRONG:\n";
481     std::cout << "      vector<int> create() {\n";
482     std::cout << "          vector<int> result(1000);\n";
483     std::cout << "          return std::move(result); // Prevents RVO!\n";
484     std::cout << "      }\n\n";
485 }
486
487 void demonstrate_unique_ptr() {
488     std::cout << " 3 Unique Ownership Transfer:\n\n";
489
490     std::cout << "     auto ptr1 = make_unique<int>(42);\n";
491     auto ptr1 = std::make_unique<int>(42);
492     std::cout << "     ptr1 owns: " << *ptr1 << "\n\n";
493
494     std::cout << "     auto ptr2 = std::move(ptr1); // Transfer ownership\n";
495     auto ptr2 = std::move(ptr1);
496     std::cout << "     ptr1 is now: " << (ptr1 ? "valid" : "nullptr") << "\n";
497     std::cout << "     ptr2 owns: " << *ptr2 << "\n\n";
498 }
499
500 void demonstrate_factory() {
501     std::cout << " 4 Factory Functions with Perfect Forwarding:\n\n";
502
503     std::cout << "     See Section 4 for detailed factory examples!\n\n";
504 }
```

```
505 void demonstrate_all() {
506     demonstrate_vector();
507     demonstrate_rvo();
508     demonstrate_unique_ptr();
509     demonstrate_factory();
510 }
511
512 } // namespace real_world_examples
513
514 // =====
515 // SECTION 7: PERFORMANCE COMPARISON
516 // =====
517
518 namespace performance_comparison {
519
520 std::vector<int> create_large_vector(size_t size) {
521     std::vector<int> result(size);
522     for (size_t i = 0; i < size; ++i) {
523         result[i] = static_cast<int>(i);
524     }
525     return result; // RVO or move
526 }
527
528 void benchmark_copy_vs_move() {
529     std::cout << "\n" << std::string(70, '=') << "\n";
530     std::cout << "SECTION 7: PERFORMANCE COMPARISON\n";
531     std::cout << std::string(70, '=') << "\n\n";
532
533     const size_t size = 1000000; // 1 million integers
534     const int iterations = 100;
535
536     std::cout << " Benchmark Configuration:\n";
537     std::cout << "     Buffer size: " << size << " integers\n";
538     std::cout << "     Iterations: " << iterations << "\n\n";
539
540     // Benchmark COPY
541     std::cout << " 1 Testing COPY operations...\n";
542     auto start_copy = std::chrono::high_resolution_clock::now();
543
544     std::vector<int> source = create_large_vector(size);
545     for (int i = 0; i < iterations; ++i) {
546         std::vector<int> dest = source; // Copy
547         (void)dest; // Prevent optimization
548     }
549
550     auto end_copy = std::chrono::high_resolution_clock::now();
551     auto copy_time = std::chrono::duration_cast<std::chrono::milliseconds>(
552         end_copy - start_copy).count();
553
554     // Benchmark MOVE
555     std::cout << " 2 Testing MOVE operations...\n";
556     auto start_move = std::chrono::high_resolution_clock::now();
557
558     for (int i = 0; i < iterations; ++i) {
```

```
559     std::vector<int> source_temp = create_large_vector(size);
560     std::vector<int> dest = std::move(source_temp); // Move
561     (void)dest; // Prevent optimization
562 }
563
564 auto end_move = std::chrono::high_resolution_clock::now();
565 auto move_time = std::chrono::duration_cast<std::chrono::milliseconds>(
566     end_move - start_move).count();
567
568 // Results
569 std::cout << "\n RESULTS:\n";
570 std::cout << "    Copy time: " << copy_time << " ms\n";
571 std::cout << "    Move time: " << move_time << " ms\n";
572 std::cout << "    Speedup: " << (double)copy_time / move_time << "x
573     faster!\n\n";
574
575 std::cout << "    INTERPRETATION:\n";
576 std::cout << "    Copy: O(n) - copies all " << size << " integers\n";
577 std::cout << "    Move: O(1) - just transfers pointer ownership\n";
578 std::cout << "    For large objects, move is 10x-1000x+ faster!\n";
579 }
580 } // namespace performance_comparison
581
582 // =====
583 // MAIN - Run All Demonstrations
584 // =====
585
586 int main() {
587     std::cout << "\n";
588     std::cout << "                                \n";
589     std::cout << "    IMPLEMENTING MOVE SEMANTICS AND PERFECT FORWARDING
590     \n";
591     std::cout << "                                Comprehensive Guide to Modern C++ Features
592     \n";
593     std::cout << "                                \n";
594
595     try {
596         // Section 1: Lvalues vs Rvalues
597         lvalues_rvalues::demonstrate();
598
599         // Section 2: Move Semantics
600         move_semantics::demonstrate();
601
602         // Section 3: Understanding std::move
603         understanding_move::demonstrate();
604
605         // Section 4: Perfect Forwarding
606         perfect_forwarding::demonstrate();
607         perfect_forwarding::demonstrate_factory();
608
609         // Section 5: Rule of Zero/Three/Five
610         rule_of_five::demonstrate();
611     }
612 }
```

```
610 // Section 6: Real-World Examples
611 real_world_examples::demonstrate_all();
612
613 // Section 7: Performance Comparison
614 performance_comparison::benchmark_copy_vs_move();
615
616 // Summary
617 std::cout << "\n" << std::string(70, '=') << "\n";
618 std::cout << "SUMMARY: KEY TAKEAWAYS\n";
619 std::cout << std::string(70, '=') << "\n\n";
620
621 std::cout << " Move Semantics:\n";
622 std::cout << " • Enables O(1) resource transfer instead of O(n) copy
623             \n";
624 std::cout << " • Always mark move operations noexcept\n";
625 std::cout << " • 10x-1000x+ performance improvement for large
626             objects\n\n";
627
628 std::cout << " Perfect Forwarding:\n";
629 std::cout << " • Preserves lvalue/rvalue category through templates\
630             \n";
631 std::cout << " • Use T&& (universal reference) with std::forward<T>\
632             \n";
633 std::cout << " • Essential for factory functions and wrappers\n\n";
634
635 std::cout << " Best Practices:\n";
636 std::cout << " • Prefer Rule of Zero (use std types)\n";
637 std::cout << " • If managing resources, implement Rule of Five\n";
638 std::cout << " • Return by value (trust RVO and move)\n";
639 std::cout << " • Use std::move for explicit ownership transfer\n";
640 std::cout << " • Use std::forward in template forwarding\n";
641 std::cout << " • Don't use objects after moving from them\n\n";
642
643 std::cout << "                                     \n";
644 std::cout << "                                     ALL EXAMPLES COMPLETED!
645                                     \n";
646 std::cout << "                                     \n\n";
647
648 } catch (const std::exception& e) {
649     std::cerr << " Error: " << e.what() << "\n";
650     return 1;
651 }
652
653 return 0;
654 }
```

## 44 Source Code: MultiThreadedMicroservices.cpp

File: src/MultiThreadedMicroservices.cpp

Repository: [View on GitHub](#)

```
1 // MultiThreadedMicroservices.cpp
2 // Demonstrates multi-threaded microservices architecture with different
3 // exception handling strategies: REST threads use pthread_exit(), core
4 // threads use abort()
5
6 #include <iostream>
7 #include <thread>
8 #include <mutex>
9 #include <condition_variable>
10 #include <queue>
11 #include <vector>
12 #include <string>
13 #include <memory>
14 #include <atomic>
15 #include <chrono>
16 #include <sstream>
17 #include <iomanip>
18 #include <cstring>
19 #include <csignal>
20 #include <functional>
21
22 #ifdef __linux__
23     #include <pthread.h>
24     #include <execinfo.h>
25     #include <cxxabi.h>
26 #elif _WIN32
27     #include <windows.h>
28     #include <dbghelp.h>
29 #endif
30
31 using namespace std::chrono;
32 using namespace std::chrono_literals;
33
34 // SECTION 1: Stack Trace and Logging Infrastructure
35 // -----
36
37 class Logger {
38 private:
39     static std::mutex log_mutex_;
40
41     static std::string get_timestamp() {
42         auto now = system_clock::now();
43         auto time = system_clock::to_time_t(now);
44         auto ms = duration_cast<milliseconds>(now.time_since_epoch()) % 1000;
```



```

98             std::string mangled(begin, end - begin);
99             int status;
100            char* demangled = abi::__cxa_demangle(mangled.c_str(),
101                                         nullptr, nullptr, &status);
102            if (status == 0 && demangled) {
103                frame = demangled;
104                free(demangled);
105            }
106        }
107        frames.push_back(frame);
108    }
109    free(symbols);
110 }
111 #elif _WIN32
112     void* buffer[32];
113     HANDLE process = GetCurrentProcess();
114     SymInitialize(process, NULL, TRUE);
115
116     WORD frame_count = CaptureStackBackTrace(0, 32, buffer, NULL);
117     SYMBOL_INFO* symbol = (SYMBOL_INFO*)calloc(sizeof(SYMBOL_INFO) + 256,
118                                               1);
119
120     if (symbol) {
121         symbol->MaxNameLen = 255;
122         symbol->SizeOfStruct = sizeof(SYMBOL_INFO);
123
124         for (WORD i = 0; i < frame_count && i < 10; ++i) {
125             if (SymFromAddr(process, (DWORD64)buffer[i], 0, symbol)) {
126                 frames.push_back(symbol->Name);
127             } else {
128                 std::ostringstream oss;
129                 oss << "0x" << std::hex << (uint64_t)buffer[i];
130                 frames.push_back(oss.str());
131             }
132         }
133         free(symbol);
134     }
135     SymCleanup(process);
136 #else
137     frames.push_back("[Stack trace not available]");
138 #endif
139
140     return frames;
141 }
142
143 static void log_exception_with_stack(const std::exception& e,
144                                     const std::string& thread_name) {
145     log(CRITICAL, std::string("EXCEPTION: ") + e.what(), thread_name);
146
147     auto stack = capture_stack_trace();
148     if (!stack.empty()) {
149         log(CRITICAL, "Stack trace:", thread_name);
150         for (size_t i = 0; i < stack.size(); ++i) {
151             log(CRITICAL, stack[i], thread_name);
152         }
153     }
154 }

```

```
150         std::ostringstream oss;
151         oss << " #" << i << ":" << stack[i];
152         log(CRITICAL, oss.str(), thread_name);
153     }
154 }
155 }
156 };
157
158 std::mutex Logger::log_mutex_;
159
160 // -----
161 // SECTION 2: Thread Type Identification and Exception Policies
162 // -----
163
164 enum class ThreadType {
165     CORE_SERVICE,      // Critical - abort() on exception
166     REST_SERVICE,      // Non-critical - pthread_exit() on exception
167     MONITORING         // Non-critical - pthread_exit() on exception
168 };
169
170 class ThreadContext {
171 private:
172     ThreadType type_;
173     std::string name_;
174
175 public:
176     ThreadContext(ThreadType type, const std::string& name)
177         : type_(type), name_(name) {}
178
179     ThreadType get_type() const { return type_; }
180     const std::string& get_name() const { return name_; }
181
182     bool is_core_service() const { return type_ == ThreadType::CORE_SERVICE; }
183     bool is_rest_service() const { return type_ == ThreadType::REST_SERVICE; }
184
185     void handle_exception(const std::exception& e) {
186         Logger::log_exception_with_stack(e, name_);
187
188         if (is_core_service()) {
189             Logger::log(Logger::CRITICAL,
190                         "CORE SERVICE FAILURE - Calling abort() to terminate
191                         application!",
192                         name_);
193             std::cerr << "\n";
194             std::cerr << "    CRITICAL: CORE SERVICE CRASHED\n";
195             std::cerr << "    Terminating entire application\n";
196             std::cerr << "\n";
197             std::abort(); // Terminate entire process
198         } else {
```

```
199     Logger::log(Logger::ERROR,
200                 "REST/MONITORING SERVICE FAILURE - Exiting thread only
201                 (core services continue)",
202                 name_);
203                 std::cerr << "\n";
204                 std::cerr << "    REST/Monitor thread exiting      \n";
205                 std::cerr << "    Core services still running OK      \n";
206                 std::cerr << "\n\n";
207 #ifdef __linux__
208         pthread_exit(nullptr); // Exit only this thread
209 #elif _WIN32
210         ExitThread(1); // Windows equivalent
211 #else
212         std::this_thread::sleep_for(1s);
213         throw; // Fallback
214 #endif
215     }
216 }
217 };
218
219 // Thread-local storage for thread context
220 thread_local std::unique_ptr<ThreadContext> g_thread_context;
221
222 // =====
223 // SECTION 3: Simple JSON Parser (for demonstration)
224 // =====
225
226 class JsonParseException : public std::runtime_error {
227 public:
228     explicit JsonParseException(const std::string& msg)
229         : std::runtime_error("JSON Parse Error: " + msg) {}
230 };
231
232 class SimpleJson {
233 private:
234     std::string data_;
235
236 public:
237     explicit SimpleJson(const std::string& json_str) : data_(json_str) {
238         validate();
239     }
240
241     void validate() const {
242         // Simple validation
243         if (data_.empty()) {
244             throw JsonParseException("Empty JSON string");
245         }
246
247         // Must start with { and end with }
```

```
248     size_t start = data_.find_first_not_of(" \t\n\r");
249     size_t end = data_.find_last_not_of(" \t\n\r");
250
251     if (start == std::string::npos || end == std::string::npos) {
252         throw JsonParseException("Invalid JSON: whitespace only");
253     }
254
255     if (data_[start] != '{' || data_[end] != '}') {
256         throw JsonParseException("Invalid JSON: must start with { and end
257             with }");
258     }
259
260     // Count braces
261     int brace_count = 0;
262     for (char c : data_) {
263         if (c == '{') brace_count++;
264         if (c == '}') brace_count--;
265         if (brace_count < 0) {
266             throw JsonParseException("Invalid JSON: unmatched closing
267                 brace");
268         }
269     }
270
271     if (brace_count != 0) {
272         throw JsonParseException("Invalid JSON: unmatched opening brace");
273     }
274
275     std::string get_field(const std::string& field_name) const {
276         std::string search = "\"" + field_name + "\"";
277         size_t pos = data_.find(search);
278
279         if (pos == std::string::npos) {
280             throw JsonParseException("Field not found: " + field_name);
281         }
282
283         // Find value after colon
284         size_t colon = data_.find(':', pos);
285         if (colon == std::string::npos) {
286             throw JsonParseException("Malformed field: " + field_name);
287         }
288
289         size_t value_start = data_.find_first_not_of(" \t\n\r", colon + 1);
290         size_t value_end = data_.find_first_of(",}", value_start);
291
292         if (value_start == std::string::npos || value_end == std::string::npos
293             ) {
294             throw JsonParseException("Cannot extract value for: " + field_name
295                 );
296         }
297
298         std::string value = data_.substr(value_start, value_end - value_start)
299         ;
300     }
```

```
297     // Remove quotes if present
298     if (value.front() == '\"' && value.back() == '\"') {
299         value = value.substr(1, value.length() - 2);
300     }
301
302     return value;
303 }
304
305     const std::string& raw() const { return data_; }
306 };
307
308 // -----
309 // SECTION 4: Core Services (Critical - abort on exception)
310 // -----
311
312 class DatabaseService {
313 private:
314     std::atomic<bool> running_{false};
315     std::atomic<int> query_count_{0};
316     std::mutex mutex_;
317     std::condition_variable cv_;
318     std::queue<std::string> query_queue_;
319
320 public:
321     void start() {
322         running_ = true;
323
324         g_thread_context = std::make_unique<ThreadContext>(
325             ThreadType::CORE_SERVICE, "DatabaseService");
326
327         Logger::log(Logger::INFO, "Database service started", "DatabaseService"
328                     );
329
330         try {
331             while (running_) {
332                 std::unique_lock<std::mutex> lock(mutex_);
333                 cv_.wait_for(lock, 500ms, [this] {
334                     return !query_queue_.empty() || !running_;
335                 });
336
337                 if (!running_) break;
338
339                 while (!query_queue_.empty()) {
340                     std::string query = query_queue_.front();
341                     query_queue_.pop();
342                     lock.unlock();
343
344                     execute_query(query);
345
346                     lock.lock();
347                 }
348             }
349         } catch (const std::exception& e) {
350             std::cerr << "DatabaseService::start() exception: " << e.what() << std::endl;
351             std::abort();
352         }
353     }
354
355     void stop() {
356         running_ = false;
357
358         cv_.notify_all();
359
360         std::this_thread::sleep_for(500ms);
361
362         std::unique_lock<std::mutex> lock(mutex_);
363         cv_.wait(lock, [this] {
364             return !query_queue_.empty();
365         });
366
367         std::string query = query_queue_.front();
368         query_queue_.pop();
369         lock.unlock();
370
371         execute_query(query);
372
373         lock.lock();
374     }
375
376     void execute_query(const std::string& query) {
377         std::unique_lock<std::mutex> lock(mutex_);
378
379         std::string result;
380
381         // Execute the query (implementation omitted)
382
383         result = "Query result: " + query;
384
385         std::unique_lock<std::mutex> lock(mutex_);
386         cv_.notify_all();
387
388         query_queue_.push(result);
389     }
390
391     std::string raw() const {
392         std::unique_lock<std::mutex> lock(mutex_);
393
394         return query_queue_.front();
395     }
396
397     std::string raw() const {
398         std::unique_lock<std::mutex> lock(mutex_);
399
400         return query_queue_.front();
401     }
402
403     std::string raw() const {
404         std::unique_lock<std::mutex> lock(mutex_);
405
406         return query_queue_.front();
407     }
408
409     std::string raw() const {
410         std::unique_lock<std::mutex> lock(mutex_);
411
412         return query_queue_.front();
413     }
414
415     std::string raw() const {
416         std::unique_lock<std::mutex> lock(mutex_);
417
418         return query_queue_.front();
419     }
420
421     std::string raw() const {
422         std::unique_lock<std::mutex> lock(mutex_);
423
424         return query_queue_.front();
425     }
426
427     std::string raw() const {
428         std::unique_lock<std::mutex> lock(mutex_);
429
430         return query_queue_.front();
431     }
432
433     std::string raw() const {
434         std::unique_lock<std::mutex> lock(mutex_);
435
436         return query_queue_.front();
437     }
438
439     std::string raw() const {
440         std::unique_lock<std::mutex> lock(mutex_);
441
442         return query_queue_.front();
443     }
444
445     std::string raw() const {
446         std::unique_lock<std::mutex> lock(mutex_);
447
448         return query_queue_.front();
449     }
450
451     std::string raw() const {
452         std::unique_lock<std::mutex> lock(mutex_);
453
454         return query_queue_.front();
455     }
456
457     std::string raw() const {
458         std::unique_lock<std::mutex> lock(mutex_);
459
460         return query_queue_.front();
461     }
462
463     std::string raw() const {
464         std::unique_lock<std::mutex> lock(mutex_);
465
466         return query_queue_.front();
467     }
468
469     std::string raw() const {
470         std::unique_lock<std::mutex> lock(mutex_);
471
472         return query_queue_.front();
473     }
474
475     std::string raw() const {
476         std::unique_lock<std::mutex> lock(mutex_);
477
478         return query_queue_.front();
479     }
480
481     std::string raw() const {
482         std::unique_lock<std::mutex> lock(mutex_);
483
484         return query_queue_.front();
485     }
486
487     std::string raw() const {
488         std::unique_lock<std::mutex> lock(mutex_);
489
490         return query_queue_.front();
491     }
492
493     std::string raw() const {
494         std::unique_lock<std::mutex> lock(mutex_);
495
496         return query_queue_.front();
497     }
498
499     std::string raw() const {
500         std::unique_lock<std::mutex> lock(mutex_);
501
502         return query_queue_.front();
503     }
504
505     std::string raw() const {
506         std::unique_lock<std::mutex> lock(mutex_);
507
508         return query_queue_.front();
509     }
510
511     std::string raw() const {
512         std::unique_lock<std::mutex> lock(mutex_);
513
514         return query_queue_.front();
515     }
516
517     std::string raw() const {
518         std::unique_lock<std::mutex> lock(mutex_);
519
520         return query_queue_.front();
521     }
522
523     std::string raw() const {
524         std::unique_lock<std::mutex> lock(mutex_);
525
526         return query_queue_.front();
527     }
528
529     std::string raw() const {
530         std::unique_lock<std::mutex> lock(mutex_);
531
532         return query_queue_.front();
533     }
534
535     std::string raw() const {
536         std::unique_lock<std::mutex> lock(mutex_);
537
538         return query_queue_.front();
539     }
540
541     std::string raw() const {
542         std::unique_lock<std::mutex> lock(mutex_);
543
544         return query_queue_.front();
545     }
546
547     std::string raw() const {
548         std::unique_lock<std::mutex> lock(mutex_);
549
550         return query_queue_.front();
551     }
552
553     std::string raw() const {
554         std::unique_lock<std::mutex> lock(mutex_);
555
556         return query_queue_.front();
557     }
558
559     std::string raw() const {
560         std::unique_lock<std::mutex> lock(mutex_);
561
562         return query_queue_.front();
563     }
564
565     std::string raw() const {
566         std::unique_lock<std::mutex> lock(mutex_);
567
568         return query_queue_.front();
569     }
570
571     std::string raw() const {
572         std::unique_lock<std::mutex> lock(mutex_);
573
574         return query_queue_.front();
575     }
576
577     std::string raw() const {
578         std::unique_lock<std::mutex> lock(mutex_);
579
580         return query_queue_.front();
581     }
582
583     std::string raw() const {
584         std::unique_lock<std::mutex> lock(mutex_);
585
586         return query_queue_.front();
587     }
588
589     std::string raw() const {
590         std::unique_lock<std::mutex> lock(mutex_);
591
592         return query_queue_.front();
593     }
594
595     std::string raw() const {
596         std::unique_lock<std::mutex> lock(mutex_);
597
598         return query_queue_.front();
599     }
599 }
```

```
346         }
347     }
348 }
349     catch (const std::exception& e) {
350         g_thread_context->handle_exception(e);
351     }
352
353     Logger::log(Logger::INFO, "Database service stopped", "DatabaseService
354         ");
355
356     void execute_query(const std::string& query) {
357         query_count_++;
358
359         // Simulate query execution
360         std::this_thread::sleep_for(50ms);
361
362         Logger::log(Logger::INFO,
363             "Executed query #" + std::to_string(query_count_.load()) +
364             ": " + query,
365             "DatabaseService");
366
367         // Simulate critical error in core service (triggered by special query
368         )
369         if (query.find("TRIGGER_CORE_FAILURE") != std::string::npos) {
370             Logger::log(Logger::ERROR,
371                 "SIMULATING CRITICAL DATABASE CORRUPTION!",
372                 "DatabaseService");
373             std::this_thread::sleep_for(100ms);
374             throw std::runtime_error("CRITICAL: Database corruption detected!
375                 Data integrity compromised!");
376         }
377     }
378
379     void submit_query(const std::string& query) {
380         std::lock_guard<std::mutex> lock(mutex_);
381         query_queue_.push(query);
382         cv_.notify_one();
383     }
384
385     void stop() {
386         running_ = false;
387         cv_.notify_all();
388     }
389
390     int get_query_count() const { return query_count_.load(); }
391
392     class CacheService {
393     private:
394         std::atomic<bool> running_{false};
395         std::atomic<int> cache_hits_{0};
396         std::atomic<int> cache_misses_{0};
397     };
398 }
```

```
396 public:
397     void start() {
398         running_ = true;
399
400         g_thread_context = std::make_unique<ThreadContext>(
401             ThreadType::CORE_SERVICE, "CacheService");
402
403         Logger::log(Logger::INFO, "Cache service started", "CacheService");
404
405     try {
406         while (running_) {
407             std::this_thread::sleep_for(1s);
408
409             // Simulate cache maintenance
410             if (cache_hits_ + cache_misses_ > 0) {
411                 int total = cache_hits_ + cache_misses_;
412                 float hit_rate = (100.0f * cache_hits_) / total;
413
414                 std::ostringstream oss;
415                 oss << "Cache stats: " << cache_hits_.load() << " hits, "
416                     << cache_misses_.load() << " misses (hit rate: "
417                     << std::fixed << std::setprecision(1) << hit_rate << "%)";
418
419                 Logger::log(Logger::INFO, oss.str(), "CacheService");
420             }
421         }
422     }
423     catch (const std::exception& e) {
424         g_thread_context->handle_exception(e);
425     }
426
427     Logger::log(Logger::INFO, "Cache service stopped", "CacheService");
428 }
429
430     bool get(const std::string& key) {
431         // Simulate cache lookup
432         bool hit = (std::hash<std::string>{}(key) % 3) == 0;
433
434         if (hit) {
435             cache_hits_++;
436         } else {
437             cache_misses_++;
438         }
439
440         return hit;
441     }
442
443     void stop() {
444         running_ = false;
445     }
446 };
447
448 //
```

```
=====
449 // SECTION 5: REST Service (Non-critical - pthread_exit on exception)
450 //
=====

451
452 class Rest ApiService {
453 private:
454     std::atomic<bool> running_{false};
455     std::atomic<int> request_count_{0};
456     std::atomic<int> error_count_{0};
457     DatabaseService& db_;
458     CacheService& cache_;
459
460 public:
461     Rest ApiService(DatabaseService& db, CacheService& cache)
462         : db_(db), cache_(cache) {}
463
464     void start() {
465         running_ = true;
466
467         g_thread_context = std::make_unique<ThreadContext>(
468             ThreadType::REST_SERVICE, "Rest ApiService");
469
470         Logger::log(Logger::INFO, "REST API service started on port 8080 (
471             simulated)",
472             "Rest ApiService");
473
474         try {
475             simulate_rest_requests();
476         }
477         catch (const std::exception& e) {
478             g_thread_context->handle_exception(e);
479         }
480
481         Logger::log(Logger::INFO, "REST API service stopped", "Rest ApiService"
482             );
483     }
484
485     void simulate_rest_requests() {
486         // Simulate various REST requests
487         std::vector<std::string> requests = {
488             R"({"action": "get_user", "user_id": "123"})",
489             R"({"action": "create_order", "product": "laptop", "quantity": 1})",
490             "",
491             R"({"action": "update_profile", "name": "John Doe"})",
492             R"({invalid json - missing closing brace})", // This will cause
493             // exception!
494             R"({"action": "delete_item", "item_id": "456"})",
495             R"()", // Empty - will cause exception!
496         };
497
498         int request_num = 0;
```

```
495
496     while (running_ && request_num < requests.size()) {
497         std::this_thread::sleep_for(800ms);
498
499         const std::string& request_body = requests[request_num];
500         request_count_++;
501
502         Logger::log(Logger::INFO,
503                     "Received REST request #" + std::to_string(
504                         request_count_.load()),
505                     "Rest ApiService");
506
507         try {
508             handle_request(request_body);
509         }
510         catch (const JsonParseException& e) {
511             error_count_++;
512
513             Logger::log(Logger::ERROR,
514                         "Invalid JSON in request #" + std::to_string(
515                             request_count_.load()),
516                         "Rest ApiService");
517
518             // This will trigger pthread_exit() through ThreadContext
519             throw;
520         }
521
522         request_num++;
523     }
524
525     // If we get here without exception, run indefinitely
526     while (running_) {
527         std::this_thread::sleep_for(1s);
528     }
529
530     void handle_request(const std::string& json_body) {
531         // Parse JSON (may throw JsonParseException)
532         SimpleJson json(json_body);
533
534         std::string action = json.get_field("action");
535
536         Logger::log(Logger::INFO, "Processing action: " + action, "Rest ApiService");
537
538         // Check cache
539         if (cache_.get(action)) {
540             Logger::log(Logger::INFO, "Cache HIT for action: " + action, "Rest ApiService");
541         } else {
542             Logger::log(Logger::INFO, "Cache MISS for action: " + action, "Rest ApiService");
543             // Submit query to database
544             db_.submit_query("SELECT * FROM actions WHERE action='"
545                             + action + "'"
546                             + " LIMIT 1");
547         }
548     }
549
550     // If we get here without exception, run indefinitely
551     while (running_) {
552         std::this_thread::sleep_for(1s);
553     }
554 }
```

```
        " :: ");
544    }
545 }
546
547 void stop() {
548     running_ = false;
549 }
550
551 int get_request_count() const { return request_count_.load(); }
552 int get_error_count() const { return error_count_.load(); }
553 };
554
555 // =====
556 // SECTION 6: Monitoring Service (Non-critical)
557 // =====
558
559 class MonitoringService {
560 private:
561     std::atomic<bool> running_{false};
562     DatabaseService& db_;
563     Rest ApiService& rest_;
564
565 public:
566     MonitoringService(DatabaseService& db, Rest ApiService& rest)
567         : db_(db), rest_(rest) {}
568
569     void start() {
570         running_ = true;
571
572         g_thread_context = std::make_unique<ThreadContext>(
573             ThreadType::MONITORING, "MonitoringService");
574
575         Logger::log(Logger::INFO, "Monitoring service started", "
576             MonitoringService");
577
578         try {
579             while (running_) {
580                 std::this_thread::sleep_for(2s);
581
582                 std::ostringstream oss;
583                 oss << "System Health: DB queries=" << db_.get_query_count()
584                     << ", REST requests=" << rest_.get_request_count()
585                     << ", REST errors=" << rest_.get_error_count();
586
587                 Logger::log(Logger::INFO, oss.str(), "MonitoringService");
588             }
589         } catch (const std::exception& e) {
590             g_thread_context->handle_exception(e);
591         }
592     }
593 }
```

```
592     Logger::log(Logger::INFO, "Monitoring service stopped", "  
593             MonitoringService");  
594 }  
595  
596 void stop() {  
597     running_ = false;  
598 }  
599 };  
600 //  
=====  
602 // SECTION 7: Microservices Orchestrator  
603 //  
=====  
604  
605 class MicroservicesOrchestrator {  
606 private:  
607     DatabaseService db_service_;  
608     CacheService cache_service_;  
609     Rest ApiService rest_service_;  
610     MonitoringService monitoring_service_;  
611  
612     std::vector<std::thread> threads_;  
613     bool simulate_core_failure_;  
614  
615 public:  
616     MicroservicesOrchestrator(bool simulate_core_failure = false)  
617         : rest_service_(db_service_, cache_service_),  
618         monitoring_service_(db_service_, rest_service_),  
619         simulate_core_failure_(simulate_core_failure) {}  
620  
621     void start() {  
622         std::cout << "\n";  
623         std::cout << "                                \n";  
624         std::cout << "      Multi-Threaded Microservices Architecture  
625                                \n";  
626         std::cout << "                                \n";  
627         std::cout << "      Core Services (abort on exception):  
628                                \n";  
629         std::cout << "          •      DatabaseService  
630                                \n";  
631         std::cout << "          •      CacheService  
632                                \n";  
633         std::cout << "                                \n";  
634         std::cout << "      REST Services (pthread_exit on exception):  
635                                \n";  
636         std::cout << "          •      Rest ApiService  
637                                \n";  
638         std::cout << "          •      MonitoringService  
639                                \n";
```

```
633     std::cout << "\n";
634
635     Logger::log(Logger::INFO, "Starting microservices orchestrator", "Orchestrator");
636
637     // Start core services first
638     threads_.emplace_back(&DatabaseService::start, &db_service_);
639     threads_.emplace_back(&CacheService::start, &cache_service_);
640
641     std::this_thread::sleep_for(500ms);
642
643     // Start REST and monitoring services
644     threads_.emplace_back(&Rest ApiService::start, &rest_service_);
645     threads_.emplace_back(&MonitoringService::start, &monitoring_service_);
646
647     Logger::log(Logger::INFO, "All services started", "Orchestrator");
648
649     // If simulating core failure, trigger it after a few seconds
650     if (simulate_core_failure_) {
651         std::this_thread::sleep_for(3s);
652         Logger::log(Logger::WARNING,
653                     "Triggering core service failure in 1 second...", "Orchestrator");
654         std::this_thread::sleep_for(1s);
655         db_service_.submit_query("TRIGGER_CORE_FAILURE - Simulated
656                                   corruption");
657     }
658 }
659
660 void wait_and_stop(int seconds) {
661     std::this_thread::sleep_for(std::chrono::seconds(seconds));
662
663     Logger::log(Logger::INFO, "Initiating graceful shutdown...", "Orchestrator");
664
665     // Stop services in reverse order
666     monitoring_service_.stop();
667     rest_service_.stop();
668     cache_service_.stop();
669     db_service_.stop();
670
671     // Wait for threads to finish
672     for (auto& thread : threads_) {
673         if (thread.joinable()) {
674             thread.join();
675         }
676     }
677
678     Logger::log(Logger::INFO, "All services stopped", "Orchestrator");
679 }
680 };
681 }
```

```
682 // =====
683 // SECTION 8: Demonstrations
684 // =====
685
686 void demonstrate_rest_service_exception() {
687     std::cout << "\n" << std::string(70, '=') << "\n";
688     std::cout << "==== Demonstration 1: REST Service Exception Handling ===\n";
689     std::cout << std::string(70, '=') << "\n\n";
690
691     std::cout << "Scenario:\n";
692     std::cout << " • REST service will receive invalid JSON requests\n";
693     std::cout << " • JsonParseException will be thrown\n";
694     std::cout << " • REST thread will call pthread_exit() and terminate\n";
695     std::cout << " • Core services (Database, Cache) will continue running\n\n";
696
697     std::cout << "Press Enter to start demonstration... \n";
698     std::cin.get();
699
700     MicroservicesOrchestrator orchestrator(false); // No core failure
701     orchestrator.start();
702
703     // Let it run for 10 seconds
704     // REST service will encounter invalid JSON around request #4
705     orchestrator.wait_and_stop(10);
706
707     std::cout << "\n" << std::string(70, '=') << "\n";
708     std::cout << " REST service terminated (pthread_exit) but core services
709     survived!\n";
710     std::cout << std::string(70, '=') << "\n\n";
711 }
712
713 void demonstrate_core_service_exception() {
714     std::cout << "\n" << std::string(70, '=') << "\n";
715     std::cout << "==== Demonstration 2: CORE Service Exception Handling ===\n";
716     std::cout << std::string(70, '=') << "\n\n";
717
718     std::cout << "Scenario:\n";
719     std::cout << " • Core service (Database) will encounter critical error\n";
720     std::cout << " • Exception will be thrown in DatabaseService\n";
721     std::cout << " • DatabaseService thread will call abort()\n";
722     std::cout << " • ENTIRE APPLICATION WILL TERMINATE \n\n";
723
724     std::cout << " WARNING: This will terminate the process with abort()!\n";
725     std::cout << "Press Enter to start demonstration... \n";
726     std::cin.get();
727
728     MicroservicesOrchestrator orchestrator(true); // Simulate core failure
```

```
728     orchestrator.start();  
729  
730     // Wait for core failure (will never return - abort() will be called)  
731     orchestrator.wait_and_stop(30);  
732  
733     // This line will never be reached  
734     std::cout << "\n This message should NEVER appear (abort() was called)\n"  
735 }  
736  
737 void demonstrate_best_practices() {  
738     std::cout << "\n" << std::string(70, '=') << "\n";  
739     std::cout << "==== Best Practices for Multi-Threaded Microservices ===\n";  
740     std::cout << std::string(70, '=') << "\n\n";  
741  
742     std::cout << " THREAD CLASSIFICATION:\n";  
743     std::cout << " 1. Core/Critical Services:\n";  
744     std::cout << " • Essential for application function\n";  
745     std::cout << " • Exception → abort() entire process\n";  
746     std::cout << " • Examples: Database, Message Queue, State Manager\n\n"  
747         ;  
748  
749     std::cout << " 2. REST/External Services:\n";  
750     std::cout << " • Handle external/untrusted input\n";  
751     std::cout << " • Exception → pthread_exit() current thread only\n";  
752     std::cout << " • Examples: HTTP servers, gRPC endpoints, WebSocket  
753         handlers\n\n";  
754  
755     std::cout << " 3. Monitoring/Auxiliary Services:\n";  
756     std::cout << " • Non-essential functionality\n";  
757     std::cout << " • Exception → pthread_exit() current thread only\n";  
758     std::cout << " • Examples: Metrics, logging, health checks\n\n";  
759  
760     std::cout << " EXCEPTION HANDLING STRATEGY:\n";  
761     std::cout << " 1. Always log exceptions with stack traces\n";  
762     std::cout << " 2. Validate all external input (JSON, XML, protobuf)\n";  
763     std::cout << " 3. Use thread-local context to identify thread type\n";  
764     std::cout << " 4. Implement graceful degradation for non-critical threads  
765         \n";  
766     std::cout << " 5. Use RAII for resource cleanup before pthread_exit()\n\n"  
767         ;  
768  
769     std::cout << " ISOLATION AND RESILIENCE:\n";  
770     std::cout << " 1. Run REST handlers in separate threads/thread pool\n";  
771     std::cout << " 2. Implement circuit breakers for failing services\n";  
772     std::cout << " 3. Use message queues to decouple services\n";  
773     std::cout << " 4. Monitor thread health and restart if needed\n";  
774     std::cout << " 5. Test exception paths thoroughly\n\n";  
775  
776     std::cout << " CRITICAL WARNINGS:\n";  
777     std::cout << " 1. pthread_exit() does NOT call destructors for local  
778         objects!\n";  
779     std::cout << " 2. Clean up resources manually before pthread_exit()\n";  
780     std::cout << " 3. Don't use pthread_exit() from main thread\n";
```

```
776     std::cout << " 4. On Windows, use ExitThread() instead of pthread_exit()\n";
777     std::cout << " 5. Consider std::terminate() as alternative to abort()\n";
778 }
779 //
780 //=====
781 // MAIN FUNCTION
782 //=====
783
784 int main() {
785     std::cout << "\n";
786     std::cout << "                               \n";
787     std::cout << "      Multi-Threaded Microservices with Exception Handling\n";
788     std::cout << "                               \n";
789     std::cout << "      Demonstrates different exception handling strategies:\n";
790     std::cout << "      \n";
791     std::cout << "      •      REST Service: pthread_exit() (thread-local\n";
792     std::cout << "          termination) \n";
793     std::cout << "      •      Core Service: abort() (process-wide termination)\n";
794     std::cout << "          \n";
795     std::cout << "          \n";
796     std::cout << "      Select demonstration:\n";
797     std::cout << "      1. REST Service Exception (pthread_exit - graceful)\n";
798     std::cout << "      2. CORE Service Exception (abort - terminates process)\n";
799     std::cout << "      3. Best Practices Guide (no execution)\n";
800     std::cout << "\nEnter choice (1-3): ";
801
802     int choice;
803     std::cin >> choice;
804     std::cin.ignore(); // Clear newline
805
806     switch (choice) {
807         case 1:
808             demonstrate_rest_service_exception();
809             demonstrate_best_practices();
810             break;
811
812         case 2:
813             demonstrate_core_service_exception();
814             // Will never reach here - abort() terminates process
815             break;
816
817         case 3:
818             demonstrate_best_practices();
819             break;
820
821         default:
```

```
820     std::cout << "\nInvalid choice. Running REST demonstration by
821         default.\n";
822     demonstrate_rest_service_exception();
823     demonstrate_best_practices();
824     break;
825
826     std::cout << "\n" << std::string(70, '=')
827     << "All demonstrations completed!\n";
828     std::cout << "\nKEY TAKEAWAYS:\n";
829     std::cout << "  1. REST threads handle untrusted input → pthread_exit() on
830         error\n";
831     std::cout << "  2. Core threads are critical → abort() on error\n";
832     std::cout << "  3. Always log exceptions with stack traces\n";
833     std::cout << "  4. Use thread-local context for thread identification\n";
834     std::cout << "  5. Implement graceful degradation for non-critical
835         services\n";
836     std::cout << std::string(70, '=')
837     << "\n\n";
838
839     return 0;
840 }
```

## 45 Source Code: NVIIdiomTemplateMethod.cpp

**File:** src/NVIIdiomTemplateMethod.cpp

**Repository:** [View on GitHub](#)

```
1 // NVIIdiomTemplateMethod.cpp
2 // Demonstrates the Non-Virtual Interface (NVI) Idiom and Template Method
3 // Pattern
4 //
5 // KEY CONCEPTS:
6 // 1. Public non-virtual methods define the interface (Template Method)
7 // 2. Private/protected virtual methods are customization points
8 // 3. Base class controls the algorithm flow
9 // 4. Derived classes customize specific steps
10 //
11 // HERB SUTTER'S GUIDELINES:
12 // Guideline #1: Prefer to make interfaces non-virtual (Template Method)
13 // Guideline #2: Prefer to make virtual functions private
14 // Guideline #3: Only if derived needs base implementation, make virtual
15 // protected
16 // Guideline #4: Destructor should be public+virtual OR protected+non-virtual
17 //
18 // WHEN TO USE NVI:
19 // Need invariant algorithm with customizable steps
20 // Want to enforce pre/post conditions
21 // Need to guarantee certain code always runs
22 // Want separation of interface from implementation
23 //
24 // WHEN TO AVOID NVI:
25 // Simple interfaces with no invariants
26 // Performance-critical code (tiny extra overhead)
27 // When derived classes need full control
28 // Pure abstract interfaces (use public virtual)
29
30 #include <iostream>
31 #include <string>
32 #include <vector>
33 #include <memory>
34 #include <chrono>
35 #include <fstream>
36 #include <stdexcept>
37
38 using namespace std::chrono_literals;
39 //
40 =====
41 // SECTION 1: Basic NVI Idiom - Template Method Pattern
42 // =====
43
44 namespace basic_nvi {
```

```
44
45 class DataProcessor {
46 public:
47     // PUBLIC NON-VIRTUAL: This is the Template Method
48     // Defines the algorithm structure that CANNOT be changed by derived
49     // classes
50     void process() {
51         std::cout << "  [DataProcessor::process] Algorithm started\n";
52
53         // Invariant: Always validate before processing
54         if (!validate()) {
55             std::cout << "    Validation failed, aborting\n";
56             return;
57         }
58
59         // Call customization points (private virtuals)
60         load();
61         transform();
62         save();
63
64         std::cout << "  [DataProcessor::process] Algorithm completed\n";
65     }
66
67     virtual ~DataProcessor() = default;
68
69 private:
70     // PRIVATE VIRTUAL: Customization points that derived classes override
71     // These are the "steps" of the algorithm
72
73     virtual bool validate() {
74         std::cout << "    [Base] Default validation\n";
75         return true; // Base default: always valid
76     }
77
78     virtual void load() = 0; // Pure virtual - must override
79     virtual void transform() = 0; // Pure virtual - must override
80     virtual void save() = 0; // Pure virtual - must override
81 };
82
83 class CSVProcessor : public DataProcessor {
84 private:
85     // Override private virtuals to provide CSV-specific behavior
86     void load() override {
87         std::cout << "    [CSV] Loading CSV file...\n";
88     }
89
90     void transform() override {
91         std::cout << "    [CSV] Transforming CSV data...\n";
92     }
93
94     void save() override {
95         std::cout << "    [CSV] Saving CSV results...\n";
96     }
97 };
```

```
97
98 class JSONProcessor : public DataProcessor {
99 private:
100     bool validate() override {
101         std::cout << "[JSON] Custom JSON validation\n";
102         return true;
103     }
104
105     void load() override {
106         std::cout << "[JSON] Loading JSON file...\n";
107     }
108
109     void transform() override {
110         std::cout << "[JSON] Transforming JSON data...\n";
111     }
112
113     void save() override {
114         std::cout << "[JSON] Saving JSON results...\n";
115     }
116 };
117
118 void demonstrate() {
119     std::cout << "\n" << std::string(70, '=') << "\n";
120     std::cout << "==== SECTION 1: Basic NVI Idiom - Template Method ===\n";
121     std::cout << std::string(70, '=') << "\n\n";
122
123     std::cout << "1. Processing CSV data:\n";
124     CSVProcessor csv;
125     csv.process(); // Client calls public non-virtual method
126
127     std::cout << "\n2. Processing JSON data:\n";
128     JSONProcessor json;
129     json.process();
130
131     std::cout << "\n Benefits:\n";
132     std::cout << " • Base class controls algorithm flow (validation always
133         runs)\n";
134     std::cout << " • Derived classes customize only the steps they need\n";
135     std::cout << " • Cannot accidentally skip validation or change algorithm\n
136         ";
137     std::cout << " • Clear separation: public interface vs private
138         implementation\n";
139 }
137
138 } // namespace basic_nvi
139
140 /**
141 =====
142 // SECTION 2: NVI with Pre/Post Conditions - Real Power
143 // =====
```

```
144 namespace preconditions_example {
145
146 class DatabaseConnection {
147     bool connected_ = false;
148     int transaction_count_ = 0;
149
150 public:
151     // PUBLIC NON-VIRTUAL: Template Method with pre/post conditions
152     void executeQuery(const std::string& query) {
153         std::cout << "  [executeQuery] Starting query execution\n";
154
155         // PRE-CONDITION: Must be connected
156         if (!connected_) {
157             std::cout << "    Error: Not connected to database!\n";
158             return;
159         }
160
161         std::cout << "  [Pre] Acquiring lock...\n";
162         std::cout << "  [Pre] Starting transaction " << ++transaction_count_
163             << "\n";
164
165         // Call customization point
166         try {
167             doExecuteQuery(query);
168
169             // POST-CONDITION: Commit transaction
170             std::cout << "  [Post] Committing transaction\n";
171             std::cout << "  [Post] Releasing lock\n";
172
173         } catch (const std::exception& e) {
174             // POST-CONDITION: Rollback on error
175             std::cout << "  [Post] Rolling back transaction due to error\n";
176             std::cout << "  [Post] Releasing lock\n";
177             throw;
178         }
179
180         std::cout << "  [executeQuery] Query completed successfully\n";
181     }
182
183     void connect() {
184         std::cout << "  [Connection] Connected to database\n";
185         connected_ = true;
186     }
187
188     virtual ~DatabaseConnection() = default;
189
190 private:
191     // PRIVATE VIRTUAL: Derived classes provide query logic
192     virtual void doExecuteQuery(const std::string& query) = 0;
193 };
194
195 class MySQLConnection : public DatabaseConnection {
196 private:
197     void doExecuteQuery(const std::string& query) override {
```

```

197     std::cout << "      [MySQL] Executing: " << query << "\n";
198     std::cout << "      [MySQL] Query executed successfully\n";
199 }
200 };
201
202 class PostgreSQLConnection : public DatabaseConnection {
203 private:
204     void doExecuteQuery(const std::string& query) override {
205         std::cout << "      [PostgreSQL] Executing: " << query << "\n";
206         if (query.find("DROP") != std::string::npos) {
207             throw std::runtime_error("DROP statements not allowed!");
208         }
209         std::cout << "      [PostgreSQL] Query executed successfully\n";
210     }
211 };
212
213 void demonstrate() {
214     std::cout << "\n" << std::string(70, '=') << "\n";
215     std::cout << "==== SECTION 2: NVI with Pre/Post Conditions ===\n";
216     std::cout << std::string(70, '=') << "\n\n";
217
218     std::cout << "1. MySQL query execution:\n";
219     MySQLConnection mysql;
220     mysql.connect();
221     mysql.executeQuery("SELECT * FROM users");
222
223     std::cout << "\n2. PostgreSQL with error handling:\n";
224     PostgreSQLConnection postgres;
225     postgres.connect();
226     try {
227         postgres.executeQuery("DROP TABLE users");
228     } catch (const std::exception& e) {
229         std::cout << "      Caught exception: " << e.what() << "\n";
230         std::cout << "      Transaction rolled back automatically!\n";
231     }
232
233     std::cout << "\n3. Attempting query without connection:\n";
234     MySQLConnection mysql2;
235     mysql2.executeQuery("SELECT * FROM users");
236
237     std::cout << "\n NVI ensures:\n";
238     std::cout << " • Pre-conditions always checked (connection, locks)\n";
239     std::cout << " • Post-conditions always executed (commit, rollback,
240         cleanup)\n";
241     std::cout << " • Derived classes cannot bypass these guarantees!\n";
242 }
243 } // namespace preconditions_example
244
245 ==
246 // SECTION 3: Protected Virtual - When Derived Needs Base Implementation
247 //

```

```
=====
248
249 namespace protected_virtual_example {
250
251 class Logger {
252 public:
253     // PUBLIC NON-VIRTUAL: Template Method
254     void log(const std::string& message) {
255         std::cout << " [log] Preparing log entry\n";
256         addTimestamp();
257         writeLog(message);
258         flush();
259     }
260
261     virtual ~Logger() = default;
262
263 protected:
264     // PROTECTED VIRTUAL: Derived classes may want to call base implementation
265     virtual void writeLog(const std::string& message) {
266         std::cout << " [Base Logger] " << message << "\n";
267     }
268
269 private:
270     void addTimestamp() {
271         std::cout << " [Timestamp] 2026-01-02 10:30:45\n";
272     }
273
274     void flush() {
275         std::cout << " [Flush] Log buffer flushed\n";
276     }
277 };
278
279 class FileLogger : public Logger {
280 protected:
281     void writeLog(const std::string& message) override {
282         // Call base implementation first
283         Logger::writeLog(message);
284
285         // Add file-specific logging
286         std::cout << " [FileLogger] Writing to file: logs.txt\n";
287     }
288 };
289
290 class NetworkLogger : public Logger {
291 protected:
292     void writeLog(const std::string& message) override {
293         // Call base implementation
294         Logger::writeLog(message);
295
296         // Send over network
297         std::cout << " [NetworkLogger] Sending to log server 192.168.1.100\
298             n";
299 }
```

```
299 };
```

```
300
301 void demonstrate() {
302     std::cout << "\n" << std::string(70, '=') << "\n";
303     std::cout << "==== SECTION 3: Protected Virtual (Guideline #3) ====\n";
304     std::cout << std::string(70, '=') << "\n\n";
305
306     std::cout << "1. File logger with base implementation:\n";
307     FileLogger file_logger;
308     file_logger.log("Application started");
309
310     std::cout << "\n2. Network logger with base implementation:\n";
311     NetworkLogger net_logger;
312     net_logger.log("User logged in");
313
314     std::cout << "\n Protected virtual when:\n";
315     std::cout << " • Derived classes need to call base implementation\n";
316     std::cout << " • Want to extend, not replace, base behavior\n";
317     std::cout << " • Common pattern: call base, then add specific logic\n";
318 }
319
320 } // namespace protected_virtual_example
321
322 // =====
323 // SECTION 4: When NOT to Use NVI - Alternatives
324 // =====
325
326 namespace when_not_to_use_nvi {
327
328 // ANTI-PATTERN: NVI overkill for simple interfaces
329 class BadExample_OverengineeredNVI {
330 public:
331     void getValue() {
332         doGetValue(); // Unnecessary indirection!
333     }
334
335 private:
336     virtual void doGetValue() = 0;
337 };
338
339 // BETTER: Simple public virtual for simple interfaces
340 class GoodExample_SimpleInterface {
341 public:
342     virtual int getValue() const = 0; // Direct and clear
343     virtual ~GoodExample_SimpleInterface() = default;
344 };
345
346 class SimpleImpl : public GoodExample_SimpleInterface {
347 public:
348     int getValue() const override { return 42; }
```

```
349 };
```

```
350
```

```
351 // WHEN TO USE PUBLIC VIRTUAL: Pure abstract interfaces (like Java interfaces)
```

```
352 class IDrawable {
pre>353 public:
pre>    virtual void draw() = 0;
pre>    virtual void move(int x, int y) = 0;
pre>    virtual ~IDrawable() = default;
pre>};
```

```
358
```

```
359 class Circle : public IDrawable {
pre>360 public:
pre>    void draw() override {
pre>        std::cout << "      Drawing circle\n";
pre>    }
pre>
pre>    void move(int x, int y) override {
pre>        std::cout << "      Moving circle to (" << x << ", " << y << ")\n";
pre>    }
pre>};
```

```
369
```

```
370 void demonstrate() {
pre>    std::cout << "\n" << std::string(70, '=') << "\n";
pre>    std::cout << "==== SECTION 4: When NOT to Use NVI ===\n";
pre>    std::cout << std::string(70, '=') << "\n\n";
pre>
pre>    std::cout << "1. Simple interface - public virtual is fine:\n";
pre>    SimpleImpl impl;
pre>    std::cout << "      Value: " << impl.getValue() << "\n";
pre>
pre>    std::cout << "\n2. Pure abstract interface - use public virtual:\n";
pre>    Circle circle;
pre>    circle.draw();
pre>    circle.move(10, 20);
pre>
pre>    std::cout << "\n DON'T use NVI when:\n";
pre>    std::cout << " • Simple getter/setter interfaces\n";
pre>    std::cout << " • No invariants to enforce\n";
pre>    std::cout << " • No pre/post conditions needed\n";
pre>    std::cout << " • Pure abstract interfaces (like Java interfaces)\n";
pre>    std::cout << " • Performance critical (tiny overhead of extra call)\n\n";
pre>
pre>    std::cout << " DO use NVI when:\n";
pre>    std::cout << " • Need to enforce pre/post conditions\n";
pre>    std::cout << " • Have invariant algorithm with customizable steps\n";
pre>    std::cout << " • Want to guarantee certain code always runs\n";
pre>    std::cout << " • Need clear separation of interface from implementation\n
pre>        ";
pre>};
```

```
397
```

```
398 } // namespace when_not_to_use_nvi
```

```
399
```

```
400 // =====
```

```
401 // SECTION 5: Real-World Example - HTTP Request Handler
402 //
403 //=====
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
```

```
// 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";
        }
    }
}
```

```
451     virtual ~HTTPRequestHandler() = default;
452
453 private:
454     // PRIVATE VIRTUAL: Customization points
455     virtual bool authenticate() {
456         return true; // Default: no auth
457     }
458
459     virtual bool validateInput([[maybe_unused]] const std::string& method,
460                               [[maybe_unused]] const std::string& path) {
461         return true; // Default: always valid
462     }
463
464     virtual bool checkRateLimit() {
465         return true; // Default: no limits
466     }
467
468     virtual void processRequest(const std::string& method,
469                               const std::string& path) = 0;
470
471     virtual void logAccess(const std::string& method,
472                           const std::string& path,
473                           int status_code) {
474         std::cout << "      [Log] " << method << " " << path
475         << " - " << status_code << "\n";
476     }
477 };
478
479 class UserAPIHandler : public HTTPRequestHandler {
480 private:
481     bool authenticate() override {
482         std::cout << "      [Auth] Checking JWT token...\n";
483         return true; // Simplified
484     }
485
486     bool validateInput(const std::string& method,
487                        const std::string& path) override {
488         if (path.find("/api/users") != 0) {
489             std::cout << "      [Validation] Invalid path\n";
490             return false;
491         }
492         return true;
493     }
494
495     void processRequest(const std::string& method,
496                           const std::string& path) override {
497         if (method == "GET") {
498             std::cout << "      [Handler] Fetching user data from database\n";
499             std::cout << "      [Handler] Returning user list\n";
500         } else if (method == "POST") {
501             std::cout << "      [Handler] Creating new user\n";
502             std::cout << "      [Handler] User created successfully\n";
503         }
504     }
505 }
```

```
505 };
```

```
506
```

```
507 class PublicAPIHandler : public HTTPRequestHandler {
```

```
508 private:
```

```
509     int request_count_ = 0;
```

```
510
```

```
511     bool checkRateLimit() override {
512         if (++request_count_ > 3) {
513             std::cout << "      [RateLimit] Exceeded rate limit!\n";
514             return false;
515         }
516         return true;
517     }
```

```
518
```

```
519     void processRequest([[maybe_unused]] const std::string& method,
520                         const std::string& path) override {
521         std::cout << "      [Handler] Processing public API request: " << path
522             << "\n";
523     }
524 };
```

```
525 void demonstrate() {
526     std::cout << "\n" << std::string(70, '=') << "\n";
527     std::cout << "==== SECTION 5: Real-World - HTTP Request Handler ===\n";
528     std::cout << std::string(70, '=') << "\n\n";
529
530     std::cout << "1. User API with authentication:\n";
531     UserAPIHandler user_api;
532     user_api.handleRequest("GET", "/api/users/123");
533
534     std::cout << "\n2. Public API with rate limiting:\n";
535     PublicAPIHandler public_api;
536     for (int i = 1; i <= 5; ++i) {
537         std::cout << "\n Request " << i << ":\n";
538         public_api.handleRequest("GET", "/api/public/data");
539         if (i >= 3) {
540             break; // Stop after rate limit
541         }
542     }
543
544     std::cout << "\n NVI guarantees:\n";
545     std::cout << " • Authentication always checked before processing\n";
546     std::cout << " • Input validation cannot be bypassed\n";
547     std::cout << " • Rate limiting enforced consistently\n";
548     std::cout << " • Logging always happens (success or error)\n";
549     std::cout << " • Derived classes customize logic, not security/logging!\n
550         ";
551 }
```

```
552 } // namespace http_handler_example
553
554 // =====
```

```
555 // SECTION 6: Guideline #4 - Destructor Rules
556 //
557 =====
558
559 namespace destructor_rules {
560
561 // CORRECT: Public virtual destructor for polymorphic base class
562 class PolymorphicBase {
563 public:
564     virtual ~PolymorphicBase() {
565         std::cout << "[PolymorphicBase] Destructor\n";
566     }
567
568     virtual void doSomething() = 0;
569 };
570
571 class DerivedFromPolymorphic : public PolymorphicBase {
572 public:
573     ~DerivedFromPolymorphic() override {
574         std::cout << "[DerivedFromPolymorphic] Destructor\n";
575     }
576
577     void doSomething() override {
578         std::cout << "[DerivedFromPolymorphic] Doing something\n";
579     }
580 };
581
582 // CORRECT: Protected non-virtual destructor for non-polymorphic base
583 class NonPolymorphicBase {
584 protected:
585     ~NonPolymorphicBase() {
586         std::cout << "[NonPolymorphicBase] Destructor\n";
587     }
588
589 public:
590     void doWork() {
591         std::cout << "[NonPolymorphicBase] Working\n";
592     }
593 };
594
595 class DerivedFromNonPolymorphic : public NonPolymorphicBase {
596 public:
597     ~DerivedFromNonPolymorphic() {
598         std::cout << "[DerivedFromNonPolymorphic] Destructor\n";
599     }
600
601 void demonstrate() {
602     std::cout << "\n" << std::string(70, '=') << "\n";
603     std::cout << "==== SECTION 6: Guideline #4 - Destructor Rules ===\n";
604     std::cout << std::string(70, '=') << "\n\n";
605
606     std::cout << "1. Polymorphic base (public virtual destructor):\n";
```

```
607 {
608     PolymorphicBase* ptr = new DerivedFromPolymorphic();
609     ptr->doSomething();
610     delete ptr; // Calls derived destructor first, then base
611 }
612
613 std::cout << "\n2. Non-polymorphic base (protected non-virtual destructor)
614 :\\n";
615 {
616     DerivedFromNonPolymorphic obj;
617     obj.doWork();
618     // Cannot do: NonPolymorphicBase* ptr = new DerivedFromNonPolymorphic
619     // ();
620     // delete ptr; // Would be undefined behavior!
621     // Protected destructor prevents polymorphic deletion
622 }
623
624 std::cout << "\\n Guideline #4 (Herb Sutter):\n";
625 std::cout << " • Polymorphic base class: public + virtual destructor\\n";
626 std::cout << " • Non-polymorphic base class: protected + non-virtual
627     destructor\\n";
628 std::cout << " • Protected destructor prevents polymorphic deletion\\n";
629 std::cout << " • Virtual destructor allows safe polymorphic deletion\\n";
630 }
631
632 // namespace destructor_rules
633
634 // SECTION 7: Summary - Complete Guidelines
635
636 void show_summary() {
637     std::cout << "\\n" << std::string(70, '=') << "\\n";
638     std::cout << "==== Herb Sutter's Virtuality Guidelines - Complete Summary
639     ===\\n";
640     std::cout << std::string(70, '=') << "\\n\\n";
641
642     std::cout << "GUIDELINE #1: Prefer non-virtual interfaces (Template Method
643     )\\n";
644     std::cout << " • Public non-virtual methods define the interface\\n";
645     std::cout << " • These are the \"Template Methods\" that orchestrate the
646     algorithm\\n";
647     std::cout << " • Benefits: Enforce invariants, pre/post conditions,
648     logging, etc.\\n\\n";
649
650     std::cout << "GUIDELINE #2: Prefer private virtual functions\\n";
651     std::cout << " • Private virtual methods are customization points\\n";
652     std::cout << " • Derived classes override to provide specific behavior\\n"
653     ;
654     std::cout << " • Cannot be called directly by client code or derived
```

```

    classes\n";
649 std::cout << " • Clearest separation of interface from implementation\n\n"
    ";
650
651 std::cout << "GUIDELINE #3: Make virtual protected if derived needs base
    impl\n";
652 std::cout << " • Use protected when derived classes need to call base
    version\n";
653 std::cout << " • Common pattern: call base, then add derived-specific
    logic\n";
654 std::cout << " • Still not part of public interface\n\n";
655
656 std::cout << "GUIDELINE #4: Destructor rules\n";
657 std::cout << " • Polymorphic base: public AND virtual destructor\n";
658 std::cout << " • Non-polymorphic base: protected AND non-virtual
    destructor\n";
659 std::cout << " • Public virtual: allows safe polymorphic deletion\n";
660 std::cout << " • Protected non-virtual: prevents polymorphic deletion\n\n"
    ";
661
662 std::cout << std::string(70, '-') << "\n\n";
663
664 std::cout << "WHEN TO USE NVI:\n";
665 std::cout << "    Need to enforce pre/post conditions\n";
666 std::cout << "    Have invariant algorithm with customizable steps\n";
667 std::cout << "    Want to guarantee certain code always runs\n";
668 std::cout << "    Need separation of interface from implementation\n";
669 std::cout << "    Building frameworks or libraries with hooks\n\n";
670
671 std::cout << "WHEN NOT TO USE NVI:\n";
672 std::cout << "    Simple interfaces with no invariants\n";
673 std::cout << "    Pure abstract interfaces (use public virtual)\n";
674 std::cout << "    No pre/post conditions needed\n";
675 std::cout << "    Performance-critical inner loops (tiny overhead)\n";
676 std::cout << "    Simple getter/setter methods\n\n";
677
678 std::cout << "COMPARISON:\n";
679 std::cout << "    Traditional (public virtual):\n";
680 std::cout << "    • Simpler for basic polymorphism\n";
681 std::cout << "    • No invariant enforcement\n";
682 std::cout << "    • Derived classes have full control\n\n";
683
684 std::cout << "    NVI Idiom (public non-virtual, private virtual):\n";
685 std::cout << "    • Enforces invariants and contracts\n";
686 std::cout << "    • Base controls algorithm flow\n";
687 std::cout << "    • Clear interface/implementation separation\n";
688 std::cout << "    • Better encapsulation\n\n";
689
690 std::cout << "REFERENCE:\n";
691 std::cout << "    Herb Sutter's \"Virtuality\" article:\n";
692 std::cout << "    http://www.gotw.ca/publications/mill18.htm\n";
693 }
694
695 //
```

```
=====
696 // MAIN FUNCTION
697 //
=====

698
699 int main() {
700     std::cout << "\n";
701     std::cout << "                                \n";
702     std::cout << "  Non-Virtual Interface (NVI) Idiom & Template Method
703         Pattern  \n";
704     std::cout << "                                Herb Sutter's Virtuality Guidelines
705         \n";
706     std::cout << "                                \n";
707
708     // Section 1: Basic NVI
709     basic_nvi::demonstrate();
710
711
712     // Section 2: Pre/post conditions
713     preconditions_example::demonstrate();
714
715     // Section 3: Protected virtual
716     protected_virtual_example::demonstrate();
717
718     // Section 4: When NOT to use NVI
719     when_not_to_use_nvi::demonstrate();
720
721     // Section 5: Real-world example
722     http_handler_example::demonstrate();
723
724     // Section 6: Destructor rules
725     destructor_rules::demonstrate();
726
727     // Section 7: Complete summary
728     show_summary();
729
730     std::cout << "\n" << std::string(70, '=') << "\n";
731     std::cout << "All demonstrations completed!\n";
732     std::cout << std::string(70, '=') << "\n\n";
733
734     return 0;
735 }
```

## 46 Source Code: NlohmannJsonExample.cpp

File: src/NlohmannJsonExample.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // NLOHMANN JSON LIBRARY EXAMPLE
3 // =====
4 // This example demonstrates the nlohmann/json library - a modern C++
5 // JSON library that provides an intuitive API similar to Python/JavaScript.
6 //
7 // TOPICS COVERED:
8 // 1. Basic JSON parsing and serialization
9 // 2. Working with JSON objects and arrays
10 // 3. Type conversions and safety
11 // 4. Custom struct serialization
12 // 5. JSON Merge Patch (RFC 7386)
13 // 6. JSON Pointer (RFC 6901)
14 // 7. CBOR/MessagePack/BSON/UBJSON support
15 // 8. Performance and embedded systems considerations
16 //
17 // WHAT IS NLOHMANN JSON?
18 // - Header-only C++ JSON library
19 // - Intuitive API similar to Python dictionaries
20 // - No external dependencies
21 // - Supports C++11 and later
22 // - MIT licensed, widely adopted
23 //
24 // WHY USE NLOHMANN JSON?
25 // Easy to use: json["key"] = value
26 // Header-only: Just #include <nlohmann/json.hpp>
27 // STL-like: Integrates seamlessly with C++ containers
28 // Type-safe: Strong typing with automatic conversions
29 // Standards-compliant: RFC 7159, RFC 6901, RFC 7386
30 // Modern C++: Uses C++11/14/17/20 features
31 //
32 // INSTALLATION:
33 // - Ubuntu: sudo apt-get install nlohmann-json3-dev
34 // - vcpkg: vcpkg install nlohmann-json
35 // - CMake: find_package(nlohmann_json REQUIRED)
36 // - Single header: Download json.hpp from GitHub
37 //
38 // =====
39
40 #include <iostream>
41 #include <fstream>
42 #include <iomanip>
43 #include <vector>
44 #include <map>
45 #include <set>
46 #include <nlohmann/json.hpp>
47
48 // For convenience
49 using json = nlohmann::json;
```

```
50 // =====
51 // EXAMPLE 1: BASIC JSON CREATION AND PARSING
52 // =====
53
54
55 void example_basic_json() {
56     std::cout << "==== Example 1: Basic JSON Creation and Parsing ===\n";
57
58     // Create JSON from initializer list (most readable)
59     json j = {
60         {"name", "Alice"},
61         {"age", 30},
62         {"city", "San Francisco"},
63         {"active", true}
64     };
65
66     std::cout << "Created JSON:\n" << j.dump(2) << "\n\n";
67
68     // Access values (Python-like syntax)
69     std::cout << "Name: " << j["name"] << "\n";
70     std::cout << "Age: " << j["age"] << "\n";
71
72     // Type-safe access with .get<T>()
73     std::string name = j["name"].get<std::string>();
74     int age = j["age"].get<int>();
75     std::cout << "Type-safe: " << name << " is " << age << " years old\n";
76
77     // Parse from string
78     std::string json_str = R"({\"device": "sensor_001", \"temperature": 23.5})";
79     json parsed = json::parse(json_str);
80     std::cout << "\nParsed JSON:\n" << parsed.dump(2) << "\n";
81
82     // Check if key exists
83     if (parsed.contains("temperature")) {
84         std::cout << "Temperature: " << parsed["temperature"] << "°C\n";
85     }
86
87     std::cout << "\n Easy syntax like Python/JavaScript\n";
88     std::cout << " Automatic type conversions\n";
89     std::cout << " contains() for safe key checking\n\n";
90 }
91
92 // =====
93 // EXAMPLE 2: WORKING WITH ARRAYS
94 // =====
95
96 void example_arrays() {
97     std::cout << "==== Example 2: JSON Arrays ===\n";
98
99     // Create array
100    json arr = json::array();
101    arr.push_back(10);
102    arr.push_back(20);
103    arr.push_back(30);
```

```
104     std::cout << "Array: " << arr.dump() << "\n";
105
106
107 // Create from vector
108 std::vector<int> vec = {1, 2, 3, 4, 5};
109 json j_vec = vec;
110 std::cout << "From vector: " << j_vec.dump() << "\n";
111
112 // Convert back to vector
113 std::vector<int> vec2 = j_vec.get<std::vector<int>>();
114 std::cout << "Back to vector: size = " << vec2.size() << "\n";
115
116 // Array of objects
117 json sensors = json::array();
118 for (int i = 0; i < 5; ++i) {
119     sensors.push_back({
120         {"id", "sensor_" + std::to_string(i)},
121         {"value", 20.0 + i * 0.5},
122         {"online", i % 2 == 0}
123     });
124 }
125
126 std::cout << "\nSensor array:\n" << sensors.dump(2) << "\n";
127
128 // Iterate over array
129 std::cout << "\nOnline sensors:\n";
130 for (const auto& sensor : sensors) {
131     if (sensor["online"].get<bool>()) {
132         std::cout << " " << sensor["id"] << ":" << sensor["value"] << "\n";
133     }
134 }
135
136 std::cout << "\n STL container integration\n";
137 std::cout << " Range-based for loops\n";
138 std::cout << " Array manipulation like JavaScript\n\n";
139 }
140
141 // =====
142 // EXAMPLE 3: CUSTOM STRUCT SERIALIZATION
143 // =====
144
145 struct SensorReading {
146     std::string device_id;
147     double temperature;
148     double humidity;
149     bool online;
150 };
151
152 // Define JSON serialization (method 1: macro)
153 NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(SensorReading, device_id, temperature,
154                                     humidity, online)
155 // Alternative custom struct with intrusive serialization
```

```
156 struct GpsCoordinate {
157     double latitude;
158     double longitude;
159     double altitude;
160
161     // Method 2: Member functions (more control)
162     NLOHMANN_DEFINE_TYPE_INTRUSIVE(GpsCoordinate, latitude, longitude,
163                                     altitude)
164 };
165
166 void example_custom_types() {
167     std::cout << "==== Example 3: Custom Struct Serialization ===\n";
168
169     // Create struct
170     SensorReading reading{
171         "sensor_001",
172         23.5,
173         65.3,
174         true
175     };
176
177     // Automatic conversion to JSON
178     json j = reading;
179     std::cout << "Struct to JSON:\n" << j.dump(2) << "\n";
180
181     // Automatic conversion from JSON
182     json j2 = R"({
183         "device_id": "sensor_002",
184         "temperature": 25.0,
185         "humidity": 70.5,
186         "online": false
187     })"_json; // User-defined literal
188
189     SensorReading reading2 = j2.get<SensorReading>();
190     std::cout << "\nJSON to struct:\n";
191     std::cout << "    Device: " << reading2.device_id << "\n";
192     std::cout << "    Temperature: " << reading2.temperature << "°C\n";
193     std::cout << "    Online: " << (reading2.online ? "yes" : "no") << "\n";
194
195     // Array of structs
196     std::vector<SensorReading> readings = {
197         {"sensor_001", 20.0, 60.0, true},
198         {"sensor_002", 21.5, 62.0, true},
199         {"sensor_003", 19.0, 58.0, false}
200     };
201
202     json j_array = readings;
203     std::cout << "\nVector of structs to JSON:\n" << j_array.dump(2) << "\n";
204
205     std::cout << "\n Automatic struct JSON conversion\n";
206     std::cout << " NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE macro\n";
207     std::cout << " Works with STL containers\n\n";
208 }
```

```
209 // =====
210 // EXAMPLE 4: FILE I/O
211 // =====
212
213 void example_file_io() {
214     std::cout << "==== Example 4: File I/O ===\n";
215
216     // Create configuration JSON
217     json config = {
218         {"version", "1.0.0"},  

219         {"server", {
220             {"host", "localhost"},  

221             {"port", 8080},  

222             {"ssl", true}
223         }},
224         {"sensors", json::array({
225             {"id", "temp_01"}, {"enabled", true},  

226             {"id", "humid_01"}, {"enabled", true}
227         })}
228     };
229
230     // Write to file
231     std::ofstream file("config.json");
232     file << std::setw(2) << config << std::endl;
233     file.close();
234     std::cout << " Written to config.json\n";
235
236     // Read from file
237     std::ifstream input("config.json");
238     json loaded_config;
239     input >> loaded_config;
240     input.close();
241
242     std::cout << " Loaded from config.json:\n";
243     std::cout << " Version: " << loaded_config["version"] << "\n";
244     std::cout << " Server port: " << loaded_config["server"]["port"] << "\n";
245     std::cout << " Sensors: " << loaded_config["sensors"].size() << "\n";
246
247     std::cout << "\n Easy file serialization\n";
248     std::cout << " Pretty printing with setw()\n";
249     std::cout << " Stream operators << and >>\n\n";
250 }
251
252 // =====
253 // EXAMPLE 5: JSON POINTER (RFC 6901)
254 // =====
255
256 void example_json_pointer() {
257     std::cout << "==== Example 5: JSON Pointer (RFC 6901) ===\n";
258
259     json data = {
260         {"user", {
261             {"name", "Alice"},  

262             {"address", {
```

```
263         {"city", "San Francisco"},  
264         {"zip", "94102"}  
265     }},  
266     {"scores", {85, 92, 78}}  
267 }  
268 };  
269  
270 // Access nested values using JSON Pointer  
271 std::cout << "Using JSON Pointer:\n";  
272 std::cout << "  /user/name: " << data["/user/name"_json_pointer] << "\n";  
273 std::cout << "  /user/address/city: " << data["/user/address/city"  
274     _json_pointer] << "\n";  
275 std::cout << "  /user/scores/0: " << data["/user/scores/0"_json_pointer]  
276     << "\n";  
277  
278 // Check if pointer exists  
279 if (data.contains(json::json_pointer("/user/address/country"))){  
280     std::cout << "  Country exists\n";  
281 } else {  
282     std::cout << "  Country does not exist\n";  
283 }  
284  
285 // Modify using pointer  
286 data["/user/address/country"_json_pointer] = "USA";  
287 std::cout << "\n Added country: " << data["/user/address/country"  
288     _json_pointer] << "\n";  
289  
290 std::cout << "\n Navigate nested structures easily\n";  
291 std::cout << "  RFC 6901 compliant\n";  
292 std::cout << "  Safe access with contains()\n\n";  
293 }  
294  
295 // =====  
296 // EXAMPLE 6: JSON MERGE PATCH (RFC 7386)  
297 // =====  
298  
299 void example_merge_patch() {  
300     std::cout << "==== Example 6: JSON Merge Patch (RFC 7386) ===\n";  
301  
302     json original = {  
303         {"name", "Alice"},  
304         {"age", 30},  
305         {"city", "San Francisco"},  
306         {"active", true}  
307     };  
308  
309     std::cout << "Original:\n" << original.dump(2) << "\n";  
310  
311     // Merge patch: update existing fields, add new fields  
312     json patch = {  
313         {"age", 31},           // Update  
314         {"city", nullptr},    // Delete  
315         {"country", "USA"}   // Add  
316     };
```

```
314     original.merge_patch(patch);
315
316     std::cout << "\nAfter merge patch:\n" << original.dump(2) << "\n";
317
318     std::cout << "\n Update multiple fields at once\n";
319     std::cout << "  null value deletes field\n";
320     std::cout << "  RFC 7386 compliant\n\n";
321 }
322
323
324 // =====
325 // EXAMPLE 7: BINARY FORMATS (CBOR, MessagePack, BSON)
326 // =====
327
328 void example_binary_formats() {
329     std::cout << "==== Example 7: Binary Formats ===\n";
330
331     json data = {
332         {"device", "sensor_001"},
333         {"temperature", 23.5},
334         {"humidity", 65.3},
335         {"readings", {20, 21, 22, 23, 24}}
336     };
337
338     // JSON string
339     std::string json_str = data.dump();
340     std::cout << "JSON size: " << json_str.size() << " bytes\n";
341
342     // CBOR (Concise Binary Object Representation)
343     std::vector<uint8_t> cbor = json::to_cbor(data);
344     std::cout << "CBOR size: " << cbor.size() << " bytes";
345     std::cout << " (" << std::fixed << std::setprecision(1)
346             << (1.0 - (double)cbor.size() / json_str.size()) * 100 << "%"
347             smaller)\n";
348
349     // MessagePack
350     std::vector<uint8_t> msgpack = json::to_msgpack(data);
351     std::cout << "MessagePack size: " << msgpack.size() << " bytes";
352     std::cout << " (" << (1.0 - (double)msgpack.size() / json_str.size()) *
353             100 << "% smaller)\n";
354
355     // BSON
356     std::vector<uint8_t> bson = json::to_bson(data);
357     std::cout << "BSON size: " << bson.size() << " bytes";
358     std::cout << " (" << (1.0 - (double)bson.size() / json_str.size()) * 100
359             << "% smaller)\n";
360
361     // UBJSON (Universal Binary JSON)
362     std::vector<uint8_t> ubjson = json::to_ubjson(data);
363     std::cout << "UBJSON size: " << ubjson.size() << " bytes\n";
364
365     // Deserialize CBOR
366     json restored = json::from_cbor(cbor);
367     std::cout << "\n Restored from CBOR:\n" << restored.dump(2) << "\n";
```

```
365     std::cout << "\n Multiple binary format support\n";
366     std::cout << " 30-50% smaller than JSON text\n";
367     std::cout << " Faster parsing than text JSON\n\n";
368 }
369
370 // =====
371 // EXAMPLE 8: ERROR HANDLING
372 // =====
373
374 void example_error_handling() {
375     std::cout << "== Example 8: Error Handling ==\n";
376
377     // Safe parsing with exception handling
378     std::string invalid_json = R"({\"name\": \"Alice\", \"age\": })";
379
380     try {
381         json j = json::parse(invalid_json);
382     } catch (const json::parse_error& e) {
383         std::cout << "Parse error caught:\n";
384         std::cout << "  Message: " << e.what() << "\n";
385         std::cout << "  Exception ID: " << e.id << "\n";
386         std::cout << "  Byte position: " << e.byte << "\n";
387     }
388
389     // Safe access with value()
390     json data = {{"name", "Alice"}, {"age", 30}};
391
392     std::string name = data.value("name", "Unknown");
393     std::string country = data.value("country", "Unknown"); // Default if
394     missing
395
396     std::cout << "\n Safe value() with default:\n";
397     std::cout << "  Name: " << name << "\n";
398     std::cout << "  Country: " << country << " (default)\n";
399
400     // Type checking
401     if (data["age"].is_number_integer()) {
402         int age = data["age"];
403         std::cout << "  Age is integer: " << age << "\n";
404     }
405
406     std::cout << "\n Exception-based error handling\n";
407     std::cout << "  value() method with defaults\n";
408     std::cout << "  Type checking with is_xxx()\n\n";
409 }
410
411 // =====
412 // EXAMPLE 9: ADVANCED STL CONTAINER CONVERSIONS
413 // =====
414
415 void example_stl_conversions() {
416     std::cout << "== Example 9: Advanced STL Container Conversions ==\n";
417 }
```

```
418 // Map conversions
419 std::map<std::string, int> scores = {
420     {"Alice", 95},
421     {"Bob", 87},
422     {"Charlie", 92}
423 };
424
425 json j_map = scores;
426 std::cout << "Map to JSON:\n" << j_map.dump(2) << "\n";
427
428 auto restored_map = j_map.get<std::map<std::string, int>>();
429 std::cout << "\n Restored map: " << restored_map.size() << " entries\n";
430
431 // Nested containers
432 std::vector<std::map<std::string, std::vector<int>>> complex = {
433     {{"data", {1, 2, 3}}, {"scores", {90, 85, 88}}},
434     {{"data", {4, 5, 6}}, {"scores", {92, 87, 91}}}
435 };
436
437 json j_complex = complex;
438 std::cout << "\nNested containers:\n" << j_complex.dump(2) << "\n";
439
440 // Set conversions
441 std::set<std::string> tags = {"cpp", "json", "modern", "c++17"};
442 json j_set = tags;
443 std::cout << "\nSet to JSON array: " << j_set.dump() << "\n";
444
445 // Tuple conversions
446 std::tuple<std::string, int, double> data = {"sensor_001", 42, 23.5};
447 json j_tuple = data;
448 std::cout << "Tuple to JSON array: " << j_tuple.dump() << "\n";
449
450 std::cout << "\n Automatic STL container serialization\n";
451 std::cout << " Works with nested containers\n";
452 std::cout << " Two-way conversion\n\n";
453 }
454
455 // =====
456 // EXAMPLE 10: SAX PARSING FOR LARGE FILES
457 // =====
458
459 void example_sax_parsing() {
460     std::cout << "== Example 10: SAX Parsing for Large Files ==\n";
461
462     // SAX (Simple API for XML) style parsing - event-driven
463     // Useful for large JSON files to avoid loading entire document
464
465     struct MySaxHandler {
466         bool null() {
467             std::cout << " Event: null\n";
468             return true;
469         }
470
471         bool boolean(bool val) {
```

```
472     std::cout << "  Event: boolean = " << std::boolalpha << val << "\n"
473     ";
474     return true;
475 }
476
477 bool number_integer(json::number_integer_t val) {
478     std::cout << "  Event: integer = " << val << "\n";
479     return true;
480 }
481
482 bool number_unsigned(json::number_unsigned_t val) {
483     std::cout << "  Event: unsigned = " << val << "\n";
484     return true;
485 }
486
487 bool number_float(json::number_float_t val, const std::string&) {
488     std::cout << "  Event: float = " << val << "\n";
489     return true;
490 }
491
492 bool binary(json::binary_t& val) {
493     std::cout << "  Event: binary (" << val.size() << " bytes)\n";
494     return true;
495 }
496
497 bool string(std::string& val) {
498     std::cout << "  Event: string = \" " << val << "\"\n";
499     return true;
500 }
501
502 bool start_object(std::size_t) {
503     std::cout << "  Event: start_object\n";
504     return true;
505 }
506
507 bool end_object() {
508     std::cout << "  Event: end_object\n";
509     return true;
510 }
511
512 bool start_array(std::size_t) {
513     std::cout << "  Event: start_array\n";
514     return true;
515 }
516
517 bool end_array() {
518     std::cout << "  Event: end_array\n";
519     return true;
520 }
521
522 bool key(std::string& val) {
523     std::cout << "  Event: key = \" " << val << "\"\n";
524     return true;
525 }
```

```

525
526     bool parse_error(std::size_t, const std::string&, const json::exception& ex) {
527         std::cout << " Parse error: " << ex.what() << "\n";
528         return false;
529     }
530 };
531
532 std::cout << "\nParsing JSON with SAX events:\n";
533 std::string json_str = R"({\"name\":\"Alice\", \"age\":30, \"scores\":[95,87,92]})";
534
535 MySaxHandler handler;
536 bool result = json::sax_parse(json_str, &handler);
537
538 std::cout << "\n SAX parsing " << (result ? "succeeded" : "failed") << "\n";
539
540 std::cout << "\n Memory-efficient for large files\n";
541 std::cout << " Event-driven processing\n";
542 std::cout << " Can stop parsing early\n";
543 std::cout << " Useful for streaming data\n\n";
544 }
545
546 // =====
547 // EXAMPLE 11: JSON SCHEMA VALIDATION
548 // =====
549
550 void example_json_schema() {
551     std::cout << "== Example 11: JSON Schema-like Validation ==\n";
552
553     // Note: nlohmann::json doesn't have built-in schema validation
554     // But we can implement basic validation checks
555
556     json schema = {
557         {"type", "object"},
558         {"required", {"name", "age", "email"}},
559         {"properties", {
560             {"name", {{"type", "string}}},
561             {"age", {{"type", "number"}, {"minimum", 0}, {"maximum", 150}}},
562             {"email", {{"type", "string"}}}
563         }}
564     };
565
566     auto validate = [] (const json& data, const json& schema) -> bool {
567         // Simple validation example
568         if (schema["type"] == "object") {
569             // Check required fields
570             for (const auto& field : schema["required"]) {
571                 if (!data.contains(field.get<std::string>())) {
572                     std::cout << " Missing required field: " << field << "\n";
573                     return false;
574                 }
575             }
576         }
577     }
578 }
```

```
576
577     // Check types
578     for (const auto& [key, prop_schema] : schema["properties"].items())
579     ) {
580         if (data.contains(key)) {
581             std::string expected_type = prop_schema["type"];
582             const auto& value = data[key];
583
584             if (expected_type == "string" && !value.is_string()) {
585                 std::cout << "    Field '" << key << "' should be
586                 string\n";
587                 return false;
588             }
589             if (expected_type == "number" && !value.is_number()) {
590                 std::cout << "    Field '" << key << "' should be
591                 number\n";
592                 return false;
593             }
594
595             // Check number constraints
596             if (value.is_number() && prop_schema.contains("minimum"))
597             {
598                 if (value.get<double>() < prop_schema["minimum"].get<
599                     double>()) {
600                     std::cout << "    Field '" << key << "' below
601                     minimum\n";
602                     return false;
603                 }
604             }
605             return true;
606         };
607
608         json valid_data = {
609             {"name", "Alice"},
610             {"age", 30},
611             {"email", "alice@example.com"}
612         };
613
614         json invalid_data = {
615             {"name", "Bob"},
616             {"age", "thirty"} // Wrong type
617         };
618
619         std::cout << "\nValidating correct data:\n";
620         if (validate(valid_data, schema)) {
621             std::cout << "    Validation passed\n";
622         }
623
624         std::cout << "\nValidating incorrect data:\n";
625         validate(invalid_data, schema);
626     }
627 }
```

```
624     std::cout << "\n For full JSON Schema support, use:\n";
625     std::cout << " • nlohmann/json-schema-validator library\n";
626     std::cout << " • valijson library\n\n";
627 }
628
629 // =====
630 // EXAMPLE 12: CUSTOM ALLOCATORS FOR EMBEDDED SYSTEMS
631 // =====
632
633 void example_custom_allocator() {
634     std::cout << "== Example 12: Custom Allocators for Embedded ==\n";
635
636     // nlohmann::json supports custom allocators via template parameter
637     // Useful for embedded systems with custom memory management
638
639     std::cout << "\nDefault allocator usage:\n";
640     json j = {{"sensor", "temp_01"}, {"value", 23.5}};
641     std::cout << " Created JSON: " << j.dump() << "\n";
642
643     // For embedded systems, you can use:
644     // 1. Compile with -DJSON_NOEXCEPTION (no exceptions)
645     // 2. Use custom allocator with basic_json template
646     // 3. Disable certain features to reduce code size
647
648     std::cout << "\n Embedded system optimizations:\n";
649     std::cout << " 1. Compile flags:\n";
650     std::cout << " • -DJSON_NOEXCEPTION - Disable exceptions\n";
651     std::cout << " • -DJSON_NO_IO - Disable file I/O\n";
652     std::cout << " • -DJSON_DIAGNOSTICS=0 - Smaller binary\n";
653     std::cout << " 2. Use binary formats (CBOR/MessagePack)\n";
654     std::cout << " 3. Use SAX parsing for large data\n";
655     std::cout << " 4. Pre-allocate with reserve()\n";
656     std::cout << " 5. Consider alternatives like ArduinoJson\n\n";
657
658     std::cout << "Memory footprint:\n";
659     std::cout << " • Header-only: ~20KB code size\n";
660     std::cout << " • Runtime: depends on JSON size\n";
661     std::cout << " • Stack usage: minimal (recursion depth)\n\n";
662 }
663
664 // =====
665 // EXAMPLE 13: PERFORMANCE TIPS
666 // =====
667
668 void example_performance() {
669     std::cout << "== Example 13: Performance Tips ==\n";
670
671     // Tip 1: Use references to avoid copies
672     json large_data = {
673         {"sensors", json::array()}
674     };
675
676     for (int i = 0; i < 100; ++i) {
677         large_data["sensors"].push_back({
```

```
678     {"id", i},
679     {"value", 20.0 + i * 0.1}
680 );
681 }
682
683 // BAD: Creates copies
684 // for (auto sensor : large_data["sensors"]) { ... }
685
686 // GOOD: Use const reference
687 int count = 0;
688 for (const auto& sensor : large_data["sensors"]){
689     if (sensor["value"].get<double>() > 25.0) {
690         ++count;
691     }
692 }
693 std::cout << "Found " << count << " sensors > 25.0 (using const ref)\n";
694
695 // Tip 2: Reserve array capacity
696 json arr = json::array();
697 // arr.get_ref<json::array_t&>().reserve(1000); // Pre-allocate
698
699 // Tip 3: Use binary formats for network/storage
700 std::vector<uint8_t> cbor = json::to_cbor(large_data);
701 std::cout << "Binary format: " << cbor.size() << " bytes\n";
702
703 // Tip 4: Disable exceptions for embedded systems
704 // Compile with -DJSON_NOEXCEPTION
705
706 std::cout << "\nPerformance tips:\n";
707 std::cout << "    Use const auto& in loops\n";
708 std::cout << "    Reserve array capacity\n";
709 std::cout << "    Use binary formats for I/O\n";
710 std::cout << "    Compile with -DJSON_NOEXCEPTION for embedded\n\n";
711 }
712
713 // =====
714 // EXAMPLE 14: COMPARISON WITH ALTERNATIVES
715 // =====
716
717 void comparison_with_alternatives() {
718     std::cout << "==== Comparison with Alternatives ===\n\n";
719
720     std::cout << "RapidJSON:\n";
721     std::cout << "    Faster parsing (2-3x)\n";
722     std::cout << "    Lower memory usage\n";
723     std::cout << "    More complex API\n";
724     std::cout << "    Manual memory management\n\n";
725
726     std::cout << "jsoncpp:\n";
727     std::cout << "    Mature and stable\n";
728     std::cout << "    Older API design\n";
729     std::cout << "    Slower than nlohmann\n";
730     std::cout << "    Requires compilation\n\n";
731 }
```

```
732     std::cout << "simdjson:\n";
733     std::cout << "    Extremely fast (SIMD optimized)\n";
734     std::cout << "    Read-only parsing\n";
735     std::cout << "    No JSON creation\n";
736     std::cout << "    Different API paradigm\n\n";
737
738     std::cout << "Protocol Buffers:\n";
739     std::cout << "    Smaller binary format\n";
740     std::cout << "    Faster parsing\n";
741     std::cout << "    Requires schema\n";
742     std::cout << "    Code generation needed\n\n";
743
744     std::cout << "When to use nlohmann/json:\n";
745     std::cout << "    Need easy-to-use API\n";
746     std::cout << "    Header-only library preferred\n";
747     std::cout << "    Configuration files\n";
748     std::cout << "    REST API communication\n";
749     std::cout << "    Rapid prototyping\n";
750     std::cout << "    Not for extreme performance (use RapidJSON/simdjson)\n\n";
751     ";
752 }
753 // =====
754 // MAIN FUNCTION
755 // =====
756
757 int main() {
758     std::cout << "=====\\n";
759     std::cout << "NLOHMANN JSON LIBRARY EXAMPLES\\n";
760     std::cout << "=====\\n\\n";
761
762     example_basic_json();
763     example_arrays();
764     example_custom_types();
765     example_file_io();
766     example_json_pointer();
767     example_merge_patch();
768     example_binary_formats();
769     example_error_handling();
770     example_stl_conversions();
771     example_sax_parsing();
772     example_json_schema();
773     example_custom_allocator();
774     example_performance();
775     comparison_with_alternatives();
776
777     std::cout << "=====\\n";
778     std::cout << "INSTALLATION:\\n";
779     std::cout << "=====\\n\\n";
780 }
```

```
781     std::cout << "Ubuntu/Debian:\n";
782     std::cout << "  sudo apt-get install nlohmann-json3-dev\n\n";
783
784     std::cout << "vcpkg:\n";
785     std::cout << "  vcpkg install nlohmann-json\n\n";
786
787     std::cout << "CMake:\n";
788     std::cout << "  find_package(nlohmann_json REQUIRED)\n";
789     std::cout << "  target_link_libraries(YourTarget nlohmann_json::
790     nlohmann_json)\n\n";
791
792     std::cout << "Single header:\n";
793     std::cout << "  Download: https://github.com/nlohmann/json/releases\n";
794     std::cout << "  Just #include \"json.hpp\"\n\n";
795
796     std::cout << "=====";
797
798     std::cout << "RESOURCES:\n";
799     std::cout << "=====";
800
801     std::cout << "GitHub: https://github.com/nlohmann/json\n";
802     std::cout << "Documentation: https://json.nlohmann.me/\n";
803     std::cout << "API Reference: https://json.nlohmann.me/api/basic\_json/\n\n";
804
805     ;
806
807     return 0;
808 }
```

## 47 Source Code: NoexceptBestPractices.cpp

File: src/NoexceptBestPractices.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // NOEXCEPT BEST PRACTICES IN MODERN C++
3 // =====
4 // Comprehensive guide to when and when NOT to use noexcept
5 //
6 // Topics Covered:
7 // 1. What is noexcept and why it matters
8 // 2. When to ALWAYS use noexcept
9 // 3. When to NEVER use noexcept
10 // 4. Conditional noexcept
11 // 5. Performance implications
12 // 6. std::vector optimization with noexcept
13 // 7. Exception safety guarantees
14 // 8. Best practices and guidelines
15 //
16 // Build: g++ -std=c++20 -Wall -Wextra -O2 -o NoexceptBestPractices
17 // NoexceptBestPractices.cpp
18 // =====
19
20 #include <iostream>
21 #include <vector>
22 #include <string>
23 #include <memory>
24 #include <utility>
25 #include <type_traits>
26 #include <chrono>
27 #include <cmath> // for std::log2
28 #include <algorithm> // for std::copy_n
29 // =====
30 // SECTION 1: WHAT IS NOEXCEPT?
31 // =====
32
33 namespace what_is_noexcept {
34
35 void demonstrate() {
36     std::cout << "\n" << std::string(70, '=') << "\n";
37     std::cout << "SECTION 1: WHAT IS NOEXCEPT?\n";
38     std::cout << std::string(70, '=') << "\n\n";
39
40     std::cout << " DEFINITION:\n";
41     std::cout << " noexcept is a specifier that tells the compiler:\n";
42     std::cout << " 'This function will NEVER throw an exception'\n\n";
43
44     std::cout << " WHAT HAPPENS IF YOU LIE?\n";
45     std::cout << " If a noexcept function throws:\n";
46     std::cout << " 1. std::terminate() is called immediately\n";
47     std::cout << " 2. Stack unwinding does NOT happen\n";
48     std::cout << " 3. Destructors are NOT called\n";
```

```
49     std::cout << "    4. Program terminates (no recovery possible)\n\n";
50
51     std::cout << " WHY USE NOEXCEPT?\n";
52     std::cout << "    1. Performance: Compiler can optimize more aggressively\n";
53     std::cout << "    2. std::vector uses move only if noexcept\n";
54     std::cout << "    3. Documents that function is exception-safe\n";
55     std::cout << "    4. Enables certain optimizations at call sites\n";
56 }
57
58 } // namespace what_is_noexcept
59
60 // =====
61 // SECTION 2: WHEN TO ALWAYS USE NOEXCEPT
62 // =====
63
64 namespace always_use_noexcept {
65
66 // RULE 1: DESTRUCTORS - Always noexcept (implicit in C++)
67 class Resource {
68 private:
69     int* data_;
70
71 public:
72     Resource() : data_(new int(42)) {
73         std::cout << "    [Resource] Constructor\n";
74     }
75
76     // Destructors are implicitly noexcept
77     ~Resource() noexcept { // Explicit is good for documentation
78         delete data_;
79         std::cout << "    [Resource] Destructor (noexcept)\n";
80     }
81 };
82
83 // RULE 2: MOVE CONSTRUCTORS AND MOVE ASSIGNMENT
84 // Critical for std::vector optimization!
85 class MoveableResource {
86 private:
87     int* data_;
88     size_t size_;
89
90 public:
91     explicit MoveableResource(size_t size)
92         : data_(new int[size]), size_(size) {
93         std::cout << "    [MoveableResource] Constructor\n";
94     }
95
96     ~MoveableResource() noexcept {
97         delete[] data_;
98     }
99
100    // Move constructor - ALWAYS mark noexcept
101    MoveableResource(MoveableResource&& other) noexcept
```

```
102     : data_(other.data_), size_(other.size_) {
103     other.data_ = nullptr;
104     other.size_ = 0;
105     std::cout << "    [MoveableResource] Move constructor (noexcept)\n";
106 }
107
108 // Move assignment - ALWAYS mark noexcept
109 MoveableResource& operator=(MoveableResource&& other) noexcept {
110     if (this != &other) {
111         delete[] data_;
112         data_ = other.data_;
113         size_ = other.size_;
114         other.data_ = nullptr;
115         other.size_ = 0;
116     }
117     std::cout << "    [MoveableResource] Move assignment (noexcept)\n";
118     return *this;
119 }
120
121 // Copy operations (not noexcept - can throw on allocation)
122 MoveableResource(const MoveableResource& other)
123     : data_(new int[other.size_]), size_(other.size_) {
124     std::copy_n(other.data_, size_, data_);
125     std::cout << "    [MoveableResource] Copy constructor (can throw)\n";
126 }
127
128 MoveableResource& operator=(const MoveableResource& other) {
129     if (this != &other) {
130         delete[] data_;
131         size_ = other.size_;
132         data_ = new int[size_]; // Can throw!
133         std::copy_n(other.data_, size_, data_);
134     }
135     std::cout << "    [MoveableResource] Copy assignment (can throw)\n";
136     return *this;
137 }
138 };
139
140 // RULE 3: SWAP FUNCTIONS - Always noexcept
141 class Swappable {
142 private:
143     int value_;
144
145 public:
146     explicit Swappable(int val) : value_(val) {}
147
148     // Swap should always be noexcept
149     friend void swap(Swappable& a, Swappable& b) noexcept {
150         using std::swap;
151         swap(a.value_, b.value_);
152         std::cout << "    [Swappable] Swapped (noexcept)\n";
153     }
154 };
```

```
156 // RULE 4: Simple getters and setters (no allocation, no complex logic)
157 class DataHolder {
158 private:
159     int value_;
160
161 public:
162     // Simple getter - noexcept
163     int getValue() const noexcept {
164         return value_;
165     }
166
167     // Simple setter - noexcept
168     void setValue(int val) noexcept {
169         value_ = val;
170     }
171 };
172
173 void demonstrate() {
174     std::cout << "\n" << std::string(70, '=') << "\n";
175     std::cout << "SECTION 2: WHEN TO ALWAYS USE NOEXCEPT\n";
176     std::cout << std::string(70, '=') << "\n\n";
177
178     std::cout << " ALWAYS USE NOEXCEPT FOR:\n\n";
179
180     std::cout << " 1 DESTRUCTORS (implicit noexcept):\n";
181 {
182     Resource r;
183 }
184 std::cout << "\n";
185
186 std::cout << " 2 MOVE OPERATIONS (critical for std::vector):\n";
187 MoveableResource mr1(10);
188 MoveableResource mr2 = std::move(mr1);
189 std::cout << "\n";
190
191 std::cout << " 3 SWAP FUNCTIONS:\n";
192 Swappable s1(10), s2(20);
193 swap(s1, s2);
194 std::cout << "\n";
195
196 std::cout << " 4 SIMPLE GETTERS/SETTERS (no allocation):\n";
197 DataHolder dh;
198 dh.setValue(42);
199 std::cout << "     getValue() = " << dh.getValue() << " (noexcept)\n\n";
200
201 std::cout << " WHY THESE MUST BE NOEXCEPT:\n";
202 std::cout << " • Destructors: Can't handle exceptions during cleanup\n";
203 std::cout << " • Move ops: std::vector falls back to copy if not
204 noexcept\n";
205 std::cout << " • Swap: Used in exception-safe code patterns\n";
206 std::cout << " • Simple ops: No reason to throw, performance benefit\n";
207 }
208 } // namespace always_use_noexcept
```

```
209
210 // =====
211 // SECTION 3: WHEN TO NEVER USE NOEXCEPT
212 // =====
213
214 namespace never_use_noexcept {
215
216 // NEVER mark noexcept if function can throw!
217
218 // DON'T use noexcept: Functions that allocate memory
219 class Container {
220 private:
221     std::vector<int> data_;
222
223 public:
224     // DON'T mark noexcept - can throw std::bad_alloc
225     void addElement(int value) { // No noexcept!
226         data_.push_back(value); // Can throw
227         std::cout << "    [Container] Added element (can throw)\n";
228     }
229
230     // DON'T mark noexcept - can throw on allocation
231     Container(size_t size) : data_(size) { // No noexcept!
232         std::cout << "    [Container] Constructor (can throw)\n";
233     }
234 };
235
236 // DON'T use noexcept: Functions that perform I/O
237 class FileHandler {
238 public:
239     // DON'T mark noexcept - I/O can fail
240     void writeToFile(const std::string& filename, const std::string& data) {
241         std::cout << "    [FileHandler] Writing to file (can throw)\n";
242         // File operations can throw
243         // if (!success) throw std::runtime_error("Write failed");
244     }
245
246     // DON'T mark noexcept - I/O can fail
247     std::string readFromFile(const std::string& filename) {
248         std::cout << "    [FileHandler] Reading from file (can throw)\n";
249         // File operations can throw
250         return "data";
251     }
252 };
253
254 // DON'T use noexcept: Functions that validate input
255 class Validator {
256 public:
257     // DON'T mark noexcept - validation can throw
258     void validateAge(int age) {
259         if (age < 0 || age > 150) {
260             throw std::invalid_argument("Invalid age");
261         }
262         std::cout << "    [Validator] Age validated (can throw)\n";
263 }
```

```
263     }
264
265     // DON'T mark noexcept - validation can throw
266     void checkNonEmpty(const std::string& str) {
267         if (str.empty()) {
268             throw std::invalid_argument("String cannot be empty");
269         }
270         std::cout << "    [Validator] String validated (can throw)\n";
271     }
272 };
273
274 // DON'T use noexcept: Functions calling other non-noexcept functions
275 class Processor {
276 private:
277     Container container_;
278
279 public:
280     Processor() : container_(10) {}
281
282     // DON'T mark noexcept - calls non-noexcept function
283     void process(int value) { // No noexcept!
284         container_.addElement(value); // Can throw
285         std::cout << "    [Processor] Processed (can throw)\n";
286     }
287 };
288
289 // DON'T use noexcept: Functions that throw std::runtime_error
290 class RuntimeErrorExample {
291 public:
292     // DANGEROUS: Marked noexcept but throws std::runtime_error
293     // void dangerousFunction() noexcept {
294     //     throw std::runtime_error("This causes std::terminate()!");
295     //     // If this runs, program terminates immediately!
296     // }
297
298     // CORRECT: NOT marked noexcept, can throw
299     void safeFunction(bool shouldFail) {
300         if (shouldFail) {
301             throw std::runtime_error("Safe to throw - not noexcept");
302         }
303         std::cout << "    [RuntimeErrorExample] Executed successfully\n";
304     }
305
306     // WRONG: Claims noexcept but allocates (can throw bad_alloc)
307     // std::vector<int> wrongNoexcept(size_t size) noexcept {
308     //     return std::vector<int>(size); // Can throw! Causes terminate!
309     // }
310
311     // CORRECT: Not marked noexcept
312     std::vector<int> correctVersion(size_t size) {
313         std::cout << "    [RuntimeErrorExample] Creating vector (can throw
314             bad_alloc)\n";
315         return std::vector<int>(size);
316     }
317 }
```

```
316 };
```

```
317
```

```
318 void demonstrate() {
319     std::cout << "\n" << std::string(70, '=') << "\n";
320     std::cout << "SECTION 3: WHEN TO NEVER USE NOEXCEPT\n";
321     std::cout << " 5 FUNCTIONS THAT THROW std::runtime_error:\n";
322     RuntimeErrorExample rte;
323     try {
324         rte.safeFunction(false);
325         std::cout << "    Success case handled\n";
326     } catch (const std::runtime_error& e) {
327         std::cout << "    Caught: " << e.what() << "\n";
328     }
329
330     auto vec = rte.correctVersion(10);
331     std::cout << "    Vector created with size: " << vec.size() << "\n\n";
332
333     std::cout << "    CRITICAL: If safeFunction() was marked noexcept:\n";
334     std::cout << "    • Throwing std::runtime_error would call std::terminate()
335         \n";
336     std::cout << "    • No catch block would execute\n";
337     std::cout << "    • No stack unwinding, no destructors called\n";
338     std::cout << "    • Program crashes immediately\n\n";
339
340     std::cout << std::string(70, '=') << "\n\n";
341
342     std::cout << "    NEVER USE NOEXCEPT FOR:\n\n";
343
344     std::cout << " 1 FUNCTIONS THAT ALLOCATE MEMORY:\n";
345     Container c(10);
346     c.addElement(42);
347     std::cout << "    Allocation can throw std::bad_alloc\n\n";
348
349     std::cout << " 2 FUNCTIONS THAT PERFORM I/O:\n";
350     FileHandler fh;
351     fh.writeToFile("test.txt", "data");
352     std::cout << "    I/O operations can fail and throw\n\n";
353
354     std::cout << " 3 FUNCTIONS THAT VALIDATE INPUT:\n";
355     Validator v;
356     try {
357         v.checkNonEmpty("hello");
358         std::cout << "    Validation passed\n";
359     } catch (...) {
360         std::cout << "    Validation can throw\n";
361     }
362     std::cout << "\n";
363
364     std::cout << " 4 FUNCTIONS CALLING NON-NOEXCEPT FUNCTIONS:\n";
365     Processor p;
366     p.process(100);
367     std::cout << "    Chain of calls inherits throwing behavior\n\n";
368
369     std::cout << "    WHY NEVER MARK THESE NOEXCEPT:\n";
```

```
369     std::cout << " • Lying about noexcept causes std::terminate()\n";
370     std::cout << " • No recovery possible if exception thrown\n";
371     std::cout << " • Better to propagate exception to caller\n";
372     std::cout << " • Allows proper error handling up the stack\n";
373 }
374 }
375 } // namespace never_use_noexcept
376
377 // =====
378 // SECTION 4: CONDITIONAL NOEXCEPT
379 // =====
380
381 namespace conditional_noexcept {
382
383 // Use conditional noexcept for template operations
384 template<typename T>
385 class Wrapper {
386 private:
387     T value_;
388
389 public:
390     explicit Wrapper(const T& val) : value_(val) {}
391
392     // Conditional noexcept based on T's move constructor
393     Wrapper(Wrapper&& other) noexcept(std::is_nothrow_move_constructible_v<T>)
394         : value_(std::move(other.value_)) {
395             std::cout << "    [Wrapper] Move constructor (conditionally noexcept)\n"
396             "";
397         }
398
399     // Conditional noexcept based on T's swap
400     void swap(Wrapper& other) noexcept(std::is_nothrow_swappable_v<T>) {
401         using std::swap;
402         swap(value_, other.value_);
403         std::cout << "    [Wrapper] Swap (conditionally noexcept)\n";
404     }
405 }
406
407 // Example: std::pair uses conditional noexcept
408 template<typename T1, typename T2>
409 class MyPair {
410 private:
411     T1 first_;
412     T2 second_;
413
414 public:
415     MyPair(const T1& f, const T2& s) : first_(f), second_(s) {}
416
417     // noexcept only if both T1 and T2 have noexcept move constructors
418     MyPair(MyPair&& other) noexcept(
419         std::is_nothrow_move_constructible_v<T1> &&
420         std::is_nothrow_move_constructible_v<T2>)
421         : first_(std::move(other.first_))
422         , second_(std::move(other.second_)) {
```

```
422         std::cout << "    [MyPair] Move (conditionally noexcept)\n";
423     }
424 };
425
426 void demonstrate() {
427     std::cout << "\n" << std::string(70, '=') << "\n";
428     std::cout << "SECTION 4: CONDITIONAL NOEXCEPT\n";
429     std::cout << std::string(70, '=') << "\n\n";
430
431     std::cout << "  CONDITIONAL NOEXCEPT:\n";
432     std::cout << "    Use noexcept(condition) for templates\n";
433     std::cout << "    noexcept status depends on template parameter\n\n";
434
435     std::cout << "  EXAMPLE 1: Wrapper with int (noexcept move):\n";
436     Wrapper<int> w1(42);
437     Wrapper<int> w2(std::move(w1));
438     std::cout << "    int move is noexcept: "
439                 << std::is_nothrow_move_constructible_v<int> << "\n\n";
440
441     std::cout << "  EXAMPLE 2: Wrapper with std::string (noexcept move):\n";
442     Wrapper<std::string> w3("hello");
443     Wrapper<std::string> w4(std::move(w3));
444     std::cout << "    string move is noexcept: "
445                 << std::is_nothrow_move_constructible_v<std::string> << "\n\n";
446
447     std::cout << "  EXAMPLE 3: MyPair with noexcept types:\n";
448     MyPair<int, int> p1(10, 20);
449     MyPair<int, int> p2(std::move(p1));
450     std::cout << "\n";
451
452     std::cout << "  WHEN TO USE CONDITIONAL NOEXCEPT:\n";
453     std::cout << "  •    Template classes wrapping user types\n";
454     std::cout << "  •    Want to preserve noexcept guarantee when possible\n";
455     std::cout << "  •    Standard library containers use this extensively\n";
456 }
457
458 } // namespace conditional_noexcept
459
460 // =====
461 // SECTION 5: STD::VECTOR OPTIMIZATION WITH NOEXCEPT
462 // =====
463
464 namespace vector_optimization {
465
466 // Class WITHOUT noexcept move constructor
467 class WithoutNoexcept {
468 private:
469     int* data_;
470
471 public:
472     explicit WithoutNoexcept(int val = 0) : data_(new int(val)) {
473         // std::cout << "    [WithoutNoexcept] Constructor\n";
474     }
475 }
```

```
476     ~WithoutNoexcept() {
477         delete data_;
478     }
479
480     // Move constructor WITHOUT noexcept
481     WithoutNoexcept(WithoutNoexcept&& other)
482         : data_(other.data_) {
483         other.data_ = nullptr;
484         std::cout << "[WithoutNoexcept] MOVE constructor (not noexcept)\n";
485     }
486
487     // Copy constructor
488     WithoutNoexcept(const WithoutNoexcept& other)
489         : data_(new int(*other.data_)) {
490         std::cout << "[WithoutNoexcept] COPY constructor\n";
491     }
492
493     WithoutNoexcept& operator=(WithoutNoexcept&&) = default;
494     WithoutNoexcept& operator=(const WithoutNoexcept&) = default;
495 };
496
497 // Class WITH noexcept move constructor
498 class WithNoexcept {
499 private:
500     int* data_;
501
502 public:
503     explicit WithNoexcept(int val = 0) : data_(new int(val)) {
504         // std::cout << "[WithNoexcept] Constructor\n";
505     }
506
507     ~WithNoexcept() {
508         delete data_;
509     }
510
511     // Move constructor WITH noexcept
512     WithNoexcept(WithNoexcept&& other) noexcept
513         : data_(other.data_) {
514         other.data_ = nullptr;
515         std::cout << "[WithNoexcept] MOVE constructor (noexcept)\n";
516     }
517
518     // Copy constructor
519     WithNoexcept(const WithNoexcept& other)
520         : data_(new int(*other.data_)) {
521         std::cout << "[WithNoexcept] COPY constructor\n";
522     }
523
524     WithNoexcept& operator=(WithNoexcept&&) noexcept = default;
525     WithNoexcept& operator=(const WithNoexcept&) = default;
526 };
527
528 void demonstrate() {
529     std::cout << "\n" << std::string(70, '=') << "\n";
530 }
```

```
530     std::cout << "SECTION 5: STD::VECTOR OPTIMIZATION WITH NOEXCEPT\n";
531     std::cout << std::string(70, '=') << "\n\n";
532
533     std::cout << "  THE CRITICAL DIFFERENCE:\n";
534     std::cout << "    std::vector uses MOVE only if noexcept\n";
535     std::cout << "    Otherwise, it uses COPY for exception safety\n\n";
536
537     std::cout << " WITHOUT noexcept move (vector uses COPY on resize):\n";
538 {
539     std::vector<WithoutNoexcept> vec;
540     vec.reserve(2);
541     vec.emplace_back(1);
542     vec.emplace_back(2);
543
544     std::cout << "    Resizing vector (triggers reallocation):\n";
545     vec.emplace_back(3); // Triggers resize - uses COPY!
546 }
547 std::cout << "\n";
548
549 std::cout << " WITH noexcept move (vector uses MOVE on resize):\n";
550 {
551     std::vector<WithNoexcept> vec;
552     vec.reserve(2);
553     vec.emplace_back(1);
554     vec.emplace_back(2);
555
556     std::cout << "    Resizing vector (triggers reallocation):\n";
557     vec.emplace_back(3); // Triggers resize - uses MOVE!
558 }
559 std::cout << "\n";
560
561 std::cout << " PERFORMANCE IMPACT:\n";
562 std::cout << "    Without noexcept: O(n) COPY during resize\n";
563 std::cout << "    With noexcept:    O(n) MOVE during resize (much faster!)\n";
564 std::cout << "    For 1000 objects: 1000x COPY vs 1000x MOVE\n";
565 std::cout << "    Speedup can be 10x-100x+ depending on object size!\n";
566 }
567
568 } // namespace vector_optimization
569
570 // =====
571 // SECTION 6: PERFORMANCE COMPARISON
572 // =====
573
574 namespace performance_comparison {
575
576 class HeavyObject {
577 private:
578     std::vector<int> data_;
579
580 public:
581     explicit HeavyObject(size_t size = 1000) : data_(size, 42) {}
582 }
```

```
583 // Copy constructor (expensive)
584 HeavyObject(const HeavyObject& other) : data_(other.data_) {}
585
586 // Move constructor (cheap) - WITH noexcept
587 HeavyObject(HeavyObject&& other) noexcept : data_(std::move(other.data_))
588 {}
589
590 HeavyObject& operator=(const HeavyObject&) = default;
591 HeavyObject& operator=(HeavyObject&&) noexcept = default;
592 };
593
594 void benchmark() {
595     const size_t num_objects = 10000;
596
597     auto start = std::chrono::high_resolution_clock::now();
598
599     std::vector<HeavyObject> vec;
600     for (size_t i = 0; i < num_objects; ++i) {
601         vec.emplace_back(1000);
602     }
603
604     auto end = std::chrono::high_resolution_clock::now();
605     auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end
606     - start);
607
608     std::cout << "    Created " << num_objects << " objects\n";
609     std::cout << "    Total resizes performed: ~"
610             << static_cast<int>(std::log2(num_objects)) << "\n";
611     std::cout << "    Time: " << duration.count() << " ms\n";
612     std::cout << "    Because move is noexcept, vector uses MOVE on resize\n";
613 }
614
615 void demonstrate() {
616     std::cout << "\n" << std::string(70, '=') << "\n";
617     std::cout << "SECTION 6: PERFORMANCE COMPARISON\n";
618     std::cout << std::string(70, '=') << "\n\n";
619
620     std::cout << "  BENCHMARK: Vector growth with noexcept move\n";
621     benchmark();
622     std::cout << "\n";
623
624     std::cout << "  WITHOUT NOEXCEPT:\n";
625     std::cout << "    Same test would use COPY instead of MOVE\n";
626     std::cout << "    Could be 10x-100x slower!\n";
627 }
628
629 } // namespace performance_comparison
630
631 // =====
632 // SECTION 7: BEST PRACTICES SUMMARY
633 // =====
634
635 namespace best_practices {
```

```

635 void demonstrate() {
636     std::cout << "\n" << std::string(70, '=') << "\n";
637     std::cout << "SECTION 7: BEST PRACTICES SUMMARY\n";
638     std::cout << std::string(70, '=') << "\n\n";
639
640     std::cout << "  ALWAYS MARK NOEXCEPT:\n";
641     std::cout << "    1. Destructors (implicit, but explicit is clearer)\n";
642     std::cout << "    2. Move constructors\n";
643     std::cout << "    3. Move assignment operators\n";
644     std::cout << "    4. Swap functions\n";
645     std::cout << "    5. Simple getters/setters (no allocation)\n";
646     std::cout << "    6. Default constructors (if they don't allocate)\n\n";
647
648     std::cout << "  NEVER MARK NOEXCEPT:\n";
649     std::cout << "    1. Functions that allocate memory\n";
650     std::cout << "    2. Functions that perform I/O\n";
651     std::cout << "    3. Functions that validate/throw on bad input\n";
652     std::cout << "    4. Copy constructors/assignment (allocation can fail)\n";
653     std::cout << "    5. Functions calling non-noexcept functions\n\n";
654
655     std::cout << "  USE CONDITIONAL NOEXCEPT:\n";
656     std::cout << "    1. Template functions wrapping user types\n";
657     std::cout << "    2. Operations depending on template parameter traits\n";
658     std::cout << "    3. When you want to preserve noexcept when possible\n\n";
659
660     std::cout << "  CRITICAL WARNINGS:\n";
661     std::cout << "  • Lying about noexcept = std::terminate() (instant crash)\n";
662     std::cout << "  • No recovery possible, no stack unwinding\n";
663     std::cout << "  • When in doubt, DON'T use noexcept\n";
664     std::cout << "  • Better to allow exception than cause termination\n\n";
665
666     std::cout << "  THE GOLDEN RULE:\n";
667     std::cout << "    'Use noexcept ONLY when you're 100% certain\n";
668     std::cout << "    the function will NEVER throw under ANY circumstances'\n";
669
670     std::cout << "  PERFORMANCE BENEFITS:\n";
671     std::cout << "  • std::vector uses move instead of copy (10x-100x faster)\n";
672     std::cout << "  • Compiler can optimize more aggressively\n";
673     std::cout << "  • No exception handling overhead\n";
674     std::cout << "  • Better code generation at call sites\n";
675 }
676
677 } // namespace best_practices
678
679 // =====
680 // MAIN - Demonstrate All Sections
681 // =====
682
683 int main() {
684     std::cout << "\n";
685     std::cout << "

```

```
686     std::cout << "          NOEXCEPT BEST PRACTICES IN MODERN C++  
687     std::cout << "          \n";  
688     std::cout << "          When to Use and When NOT to Use noexcept  
689     std::cout << "          \n";  
690     std::cout << "          \n";  
691     try {  
692         what_is_noexcept::demonstrate();  
693         always_use_noexcept::demonstrate();  
694         never_use_noexcept::demonstrate();  
695         conditional_noexcept::demonstrate();  
696         vector_optimization::demonstrate();  
697         performance_comparison::demonstrate();  
698         best_practices::demonstrate();  
699         std::cout << "\n          \n";  
700         std::cout << "          ALL NOEXCEPT CONCEPTS DEMONSTRATED!  
701         std::cout << "          \n";  
702         std::cout << "          \n\n";  
703     } catch (const std::exception& e) {  
704         std::cerr << "  Error: " << e.what() << "\n";  
705         return 1;  
706     }  
707  
708     return 0;  
709 }
```

## 48 Source Code: ObjectSlicingCpp20.cpp

File: src/ObjectSlicingCpp20.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <memory>
3 #include <vector>
4 #include <string>
5 #include <span>
6 #include <ranges>
7 #include <concepts>
8
9 // =====
10 // PREVENTING OBJECT SLICING WITH C++20 FEATURES
11 // =====
12
13 // Base class
14 class Shape {
15 protected:
16     std::string name;
17
18 public:
19     Shape(const std::string& n) : name(n) {}
20     virtual ~Shape() = default;
21
22     virtual void draw() const {
23         std::cout << "Drawing Shape: " << name << std::endl;
24     }
25
26     virtual double area() const { return 0.0; }
27
28     virtual std::string type() const { return "Shape"; }
29 };
30
31 class Circle : public Shape {
32 private:
33     double radius;
34
35 public:
36     Circle(const std::string& n, double r) : Shape(n), radius(r) {}
37
38     void draw() const override {
39         std::cout << "Drawing Circle: " << name << " (radius=" << radius << ")"
40         << std::endl;
41     }
42
43     double area() const override {
44         return 3.14159 * radius * radius;
45     }
46
47     std::string type() const override { return "Circle"; }
48 };
```

```
49 class Rectangle : public Shape {
50 private:
51     double width, height;
52
53 public:
54     Rectangle(const std::string& n, double w, double h)
55         : Shape(n), width(w), height(h) {}
56
57     void draw() const override {
58         std::cout << "Drawing Rectangle: " << name << " (" << width << "x" <<
59             height << ")" << std::endl;
60     }
61
62     double area() const override {
63         return width * height;
64     }
65
66     std::string type() const override { return "Rectangle"; }
67 };
68 // =====
69 // 1. C++20 CONCEPTS - ENFORCE POINTER TYPES AT COMPILE TIME
70 // =====
71
72 // Concept: Must be a pointer or smart pointer to a type derived from Shape
73 template<typename T>
74 concept ShapePointer = requires(T t) {
75     { *t }; // Must be dereferenceable
76     requires std::derived_from<std::remove_reference_t<decltype(*t)>, Shape>;
77 };
78
79 // Concept: Must be a smart pointer
80 template<typename T>
81 concept SmartPointer = requires(T t) {
82     typename T::element_type; // Must have element_type (shared_ptr/
83     unique_ptr trait)
84     { t.get() } -> std::same_as<typename T::element_type*>;
85     { *t } -> std::same_as<typename T::element_type&>;
86 };
87
88 // Function that ONLY accepts pointers/smart pointers - prevents slicing at
89 // compile time
90 template<ShapePointer T>
91 void draw_shape(const T& shape_ptr) {
92     shape_ptr->draw();
93 }
94
95 // Function that ONLY accepts smart pointers specifically
96 template<SmartPointer T>
97     requires std::derived_from<typename T::element_type, Shape>
98 void process_shape(const T& shape_ptr) {
99     std::cout << "Type: " << shape_ptr->type() << ", Area: " << shape_ptr->
100        area() << std::endl;
101 }
```

```

99
100 void example_concepts_prevent_slicing() {
101     std::cout << "\n==== 1. C++20 CONCEPTS - COMPILE-TIME PREVENTION ===" <<
102         std::endl;
103     std::cout << "Solution: Use concepts to enforce pointer usage at compile
104         time\n" << std::endl;
105
106     auto circle = std::make_shared<Circle>("ConceptCircle", 5.0);
107     auto rect = std::make_unique<Rectangle>("ConceptRect", 4.0, 6.0);
108
109     std::cout << " These calls compile (using pointers):" << std::endl;
110     draw_shape(circle);           // Works with shared_ptr
111     draw_shape(rect);           // Works with unique_ptr
112
113     Circle* raw_circle = circle.get();
114     draw_shape(raw_circle);     // Works with raw pointer
115
116     process_shape(circle);     // Works with smart pointers
117     process_shape(rect);       // Works with smart pointers
118
119     std::cout << "\n Concepts prevent slicing at compile time!" << std::endl;
120     std::cout << " (Uncommenting the line below would cause a compile error)
121         " << std::endl;
122 }
123
124 // =====
125 // 2. C++20 RANGES - WORK WITH REFERENCES, NOT COPIES
126 // =====
127
128 void example_ranges_prevent_slicing() {
129     std::cout << "\n==== 2. C++20 RANGES - REFERENCE-BASED OPERATIONS ===" <<
130         std::endl;
131     std::cout << "Solution: Ranges work with references, preventing copies\n"
132         << std::endl;
133
134     std::vector<std::shared_ptr<Shape>> shapes;
135     shapes.push_back(std::make_shared<Circle>("RangeCircle1", 3.0));
136     shapes.push_back(std::make_shared<Rectangle>("RangeRect1", 5.0, 4.0));
137     shapes.push_back(std::make_shared<Circle>("RangeCircle2", 2.5));
138
139     std::cout << "Using ranges to filter and transform (no slicing):" << std::endl;
140
141     // Filter shapes with area > 20 and draw them
142     auto large_shapes = shapes
143         | std::views::filter([](const auto& s) { return s->area() > 20.0; })
144         | std::views::transform([](const auto& s) { return s.get(); });
145
146     for (const auto* shape : large_shapes) {
147         shape->draw();
148         std::cout << " Area: " << shape->area() << std::endl;

```

```

147     }
148
149     std::cout << "\n Ranges operate on references - no slicing occurs!" <<
150         std::endl;
151 }
152 // =====
153 // 3. C++20 SPAN - NON-OWNING VIEWS WITHOUT SLICING
154 // =====
155
156 // Function taking span of Shape pointers
157 void draw_all_shapes(std::span<const std::shared_ptr<Shape>> shapes) {
158     std::cout << "Drawing " << shapes.size() << " shapes:" << std::endl;
159     for (const auto& shape : shapes) {
160         shape->draw();
161     }
162 }
163
164 // Function taking span of raw pointers
165 void process_shapes(std::span<const Shape*> const shapes) {
166     double total_area = 0.0;
167     for (const auto* shape : shapes) {
168         total_area += shape->area();
169     }
170     std::cout << "Total area: " << total_area << std::endl;
171 }
172
173 void example_span_prevent_slicing() {
174     std::cout << "\n==== 3. C++20 SPAN - NON-OWNING VIEWS ===" << std::endl;
175     std::cout << "Solution: std::span provides views without copying\n" << std
176         ::endl;
177
178     std::vector<std::shared_ptr<Shape>> shapes;
179     shapes.push_back(std::make_shared<Circle>("SpanCircle", 4.0));
180     shapes.push_back(std::make_shared<Rectangle>("SpanRect", 3.0, 5.0));
181
182     std::cout << "Passing span of smart pointers:" << std::endl;
183     draw_all_shapes(shapes); // No slicing - span of pointers
184
185     std::cout << "\nPassing span of raw pointers:" << std::endl;
186     std::vector<const Shape*> raw_ptrs;
187     for (const auto& s : shapes) {
188         raw_ptrs.push_back(s.get());
189     }
190     process_shapes(raw_ptrs); // No slicing - span of raw pointers
191
192     std::cout << "\n std::span is non-owning - operates on existing pointers!
193         " << std::endl;
194 }
195 // =====
196 // 4. C++20 CONSTRAINTS - DELETE SLICING-PRONE FUNCTIONS
197 // =====

```

```
198 template<typename T>
199     requires std::derived_from<T, Shape>
200 class ShapeContainer {
201 private:
202     std::vector<std::unique_ptr<T>> items;
203
204 public:
205     // CORRECT: Add by pointer (transfers ownership)
206     void add(std::unique_ptr<T> shape) {
207         items.push_back(std::move(shape));
208     }
209
210     // DELETED: Prevent adding by value (would cause slicing)
211     void add(const T& shape) = delete;
212     void add(T&& shape) = delete;
213
214     // CORRECT: Access via reference
215     const T& get(size_t index) const {
216         return *items[index];
217     }
218
219     void draw_all() const {
220         for (const auto& item : items) {
221             item->draw();
222         }
223     }
224
225     size_t size() const { return items.size(); }
226 };
227
228 void example_delete_slicing_functions() {
229     std::cout << "\n==== 4. DELETE SLICING-PRONE FUNCTIONS ===" << std::endl;
230     std::cout << "Solution: Use = delete to prevent value-based operations\n"
231         << std::endl;
232
233     ShapeContainer<Shape> container;
234
235     std::cout << " These operations work (using pointers):" << std::endl;
236     container.add(std::make_unique<Circle>("DeleteCircle", 5.0));
237     container.add(std::make_unique<Rectangle>("DeleteRect", 4.0, 3.0));
238
239     container.draw_all();
240
241     std::cout << "\n Deleted functions prevent slicing at compile time!" <<
242         std::endl;
243     std::cout << " (Uncommenting the lines below would cause compile errors)
244         " << std::endl;
245
246     // Circle circle("ByValueCircle", 3.0);
247     // container.add(circle); // Compile error! Function deleted
248     // container.add(std::move(circle)); // Compile error! Function deleted
249 }
250
251 // =====
```

```
249 // 5. C++20 DESIGNATED INITIALIZERS WITH SMART POINTERS
250 // =====
251
252 struct ShapeConfig {
253     std::shared_ptr<Shape> primary;
254     std::shared_ptr<Shape> secondary;
255     std::string label;
256 };
257
258 void example_designated_initializers() {
259     std::cout << "\n==== 5. DESIGNATED INITIALIZERS - SAFE INITIALIZATION ==="
260         << std::endl;
261     std::cout << "Solution: Initialize with pointers using designated
262         initializers\n" << std::endl;
263
264     // CORRECT: Designated initializers with smart pointers
265     ShapeConfig config {
266         .primary = std::make_shared<Circle>("Primary", 6.0),
267         .secondary = std::make_shared<Rectangle>("Secondary", 5.0, 4.0),
268         .label = "MyShapes"
269     };
270
271     std::cout << "Configuration: " << config.label << std::endl;
272     config.primary->draw();
273     config.secondary->draw();
274
275     std::cout << "\n Designated initializers ensure proper pointer
276         initialization!" << std::endl;
277 }
278
279 // =====
280 // 6. C++20 REQUIRES CLAUSE - POLYMORPHIC CONTAINERS ONLY
281 // =====
282
283 template<typename Container>
284     requires requires(Container c) {
285         // Require that container holds pointers to Shape-derived types
286         { *c.begin() } -> std::convertible_to<std::shared_ptr<Shape>>;
287     }
288 void draw_container(const Container& shapes) {
289     std::cout << "Drawing " << shapes.size() << " shapes from container:" <<
290         std::endl;
291     for (const auto& shape : shapes) {
292         shape->draw();
293     }
294 }
295
296 void example_requires_clause() {
297     std::cout << "\n==== 6. REQUIRES CLAUSE - CONSTRAIN CONTAINER TYPES ===" <<
298         std::endl;
299     std::cout << "Solution: Require containers to hold pointers only\n" << std
300         ::endl;
301
302     std::vector<std::shared_ptr<Shape>> shapes;
```

```
297     shapes.push_back(std::make_shared<Circle>("RequiresCircle", 4.5));
298     shapes.push_back(std::make_shared<Rectangle>("RequiresRect", 6.0, 2.0));
299
300     std::cout << " This works (container of shared_ptr):" << std::endl;
301     draw_container(shapes);
302
303     std::cout << "\n Requires clause ensures polymorphic container usage!" <<
304             std::endl;
305     std::cout << " (Uncommenting the lines below would cause a compile error
306             )" << std::endl;
307
308     // std::vector<Circle> value_circles;
309     // value_circles.push_back(Circle("ValueCircle", 3.0));
310     // draw_container(value_circles); // Compile error! Container holds
311     // values, not pointers
312
313 }
314
315 // =====
316 // 7. COMPARISON: C++11/14 VS C++20 PREVENTION
317 // =====
318
319 void example_comparison() {
320     std::cout << "\n==== 7. C++11/14 VS C++20 PREVENTION COMPARISON ===" << std
321             ::endl;
322
323     std::cout << "\n C++11/14 APPROACH (Runtime/Developer Discipline):" <<
324             std::endl;
325     std::cout << " • Manually use smart pointers everywhere" << std::endl;
326     std::cout << " • Remember to not pass by value" << std::endl;
327     std::cout << " • Runtime errors if you forget" << std::endl;
328     std::cout << " • Code reviews needed to catch mistakes" << std::endl;
329
330     std::cout << "\n C++20 APPROACH (Compile-Time Enforcement):" << std::endl
331             ;
332     std::cout << " • Concepts enforce pointer usage at compile time" << std
333             ::endl;
334     std::cout << " • = delete prevents slicing-prone operations" << std::endl;
335     std::cout << " • Requires clauses constrain templates" << std::endl;
336     std::cout << " • Compiler catches mistakes before runtime!" << std::endl
337             ;
338     std::cout << " • std::span provides safe non-owning views" << std::endl;
339     std::cout << " • Ranges work with references by default" << std::endl;
340 }
341
342 // =====
343 // MAIN FUNCTION
344 // =====
345
346 int main() {
347     std::cout << "\n
348             =====" <<
349             std::endl;
350     std::cout << " PREVENTING OBJECT SLICING WITH C++20 FEATURES" << std::
```

```
            endl;
340     std::cout << "
341             ====="
342     std::endl;
343
344     example_concepts_prevent_slicing();
345     example_ranges_prevent_slicing();
346     example_span_prevent_slicing();
347     example_delete_slicing_functions();
348     example_designated_initializers();
349     example_requires_clause();
350     example_comparison();
351
352     std::cout << "\n"
353             ====="
354     std::endl;
355     std::cout << "  C++20 SLICING PREVENTION SUMMARY" << std::endl;
356     std::cout << "
357             ====="
358     std::endl;
359     std::cout << "\n1.  CONCEPTS:" << std::endl;
360     std::cout << "  •  Define ShapePointer concept to enforce pointer types"
361             << std::endl;
362     std::cout << "  •  Compile-time error if trying to pass by value" << std::endl;
363     std::cout << "  •  Example: template<ShapePointer T> void draw(const T&
364             ptr)" << std::endl;
365
366     std::cout << "\n2.  RANGES:" << std::endl;
367     std::cout << "  •  Operate on references, not copies" << std::endl;
368     std::cout << "  •  views::filter and views::transform preserve types" <<
369             std::endl;
370     std::cout << "  •  No accidental slicing during transformations" << std::endl;
371
372     std::cout << "\n3.  STD::SPAN:" << std::endl;
373     std::cout << "  •  Non-owning view of container" << std::endl;
374     std::cout << "  •  std::span<shared_ptr<Shape>> - no copies" << std::endl;
375     std::cout << "  •  Safe passing of pointer collections" << std::endl;
376
377     std::cout << "\n4.  = DELETE:" << std::endl;
378     std::cout << "  •  Explicitly delete value-based operations" << std::endl;
379     std::cout << "  •  void add(const T& shape) = delete" << std::endl;
380     std::cout << "  •  Compiler prevents slicing attempts" << std::endl;
381
382     std::cout << "\n5.  REQUIRES CLAUSES:" << std::endl;
383     std::cout << "  •  Constrain templates to pointer-holding containers" <<
384             std::endl;
385     std::cout << "  •  requires { *c.begin() } -> convertible_to<shared_ptr<
386             Shape>>" << std::endl;
387     std::cout << "  •  Type-safe generic programming" << std::endl;
388
389     std::cout << "\n6.  DESIGNATED INITIALIZERS:" << std::endl;
```

```
380     std::cout << " • Clear initialization syntax" << std::endl;
381     std::cout << " • Config { .shape = make_shared<Circle>(...)}" << std::
382         endl;
383     std::cout << " • Prevents accidental by-value initialization" << std::
384         endl;
385
386     std::cout << "\n KEY ADVANTAGE OF C++20:" << std::endl;
387     std::cout << " • C++11/14: Prevent slicing through discipline (runtime
388         errors)" << std::endl;
389     std::cout << " • C++20: Prevent slicing through type system (compile-
390         time errors)" << std::endl;
391     std::cout << " • Shift from \"remember not to\" → \"impossible to\"" <<
392         std::endl;
393
394     std::cout << "\n BEST PRACTICE (C++20):" << std::endl;
395     std::cout << "     Use concepts to enforce pointer parameters" << std::
396         endl;
397     std::cout << "     Use requires clauses for container constraints" << std
398         ::endl;
399     std::cout << "     Use = delete for slicing-prone operations" << std::endl
400         ;
401     std::cout << "     Use std::span for non-owning views" << std::endl;
402     std::cout << "     Use ranges for safe transformations" << std::endl;
403     std::cout << "     Let the compiler catch slicing at compile time!" << std
404         ::endl;
405
406     std::cout << "\n
407     ======\n" << std::endl;
408
409     return 0;
410 }
```

## 49 Source Code: ObjectSlicingSmartPtr.cpp

File: src/ObjectSlicingSmartPtr.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <memory>
3 #include <vector>
4 #include <string>
5
6 // =====
7 // OBJECT SLICING AND SMART POINTERS (C++11/14)
8 // =====
9
10 // Base class
11 class Shape {
12 protected:
13     std::string name;
14
15 public:
16     Shape(const std::string& n) : name(n) {
17         std::cout << "Shape constructor: " << name << std::endl;
18     }
19
20     virtual ~Shape() {
21         std::cout << "Shape destructor: " << name << std::endl;
22     }
23
24     virtual void draw() const {
25         std::cout << "Drawing Shape: " << name << std::endl;
26     }
27
28     virtual double area() const {
29         return 0.0;
30     }
31
32     virtual void info() const {
33         std::cout << "Shape: " << name << std::endl;
34     }
35 };
36
37 // Derived class: Circle
38 class Circle : public Shape {
39 private:
40     double radius;
41
42 public:
43     Circle(const std::string& n, double r) : Shape(n), radius(r) {
44         std::cout << "Circle constructor: " << name << " (radius=" << radius
45             << ")" << std::endl;
46     }
47
48     ~Circle() override {
49         std::cout << "Circle destructor: " << name << std::endl;
```

```
49     }
50
51     void draw() const override {
52         std::cout << "Drawing Circle: " << name << " with radius " << radius
53         << std::endl;
54     }
55
56     double area() const override {
57         return 3.14159 * radius * radius;
58     }
59
60     void info() const override {
61         std::cout << "Circle: " << name << ", radius=" << radius << ", area="
62         << area() << std::endl;
63     }
64 };
65
66 // Derived class: Rectangle
67 class Rectangle : public Shape {
68 private:
69     double width;
70     double height;
71
72 public:
73     Rectangle(const std::string& n, double w, double h)
74         : Shape(n), width(w), height(h) {
75         std::cout << "Rectangle constructor: " << name
76         << " (width=" << width << ", height=" << height << ")" <<
77         std::endl;
78     }
79
80     ~Rectangle() override {
81         std::cout << "Rectangle destructor: " << name << std::endl;
82     }
83
84     void draw() const override {
85         std::cout << "Drawing Rectangle: " << name
86         << " (" << width << "x" << height << ")" << std::endl;
87     }
88
89     double area() const override {
90         return width * height;
91     }
92
93     void info() const override {
94         std::cout << "Rectangle: " << name << ", " << width << "x" << height
95         << ", area=" << area() << std::endl;
96     }
97
98 // =====
99 // 1. CLASSIC OBJECT SLICING (WITHOUT SMART POINTERS)
100 // =====
```

```
100 void example_classic_slicing() {
101     std::cout << "\n==== 1. CLASSIC OBJECT SLICING ===" << std::endl;
102     std::cout << "Problem: Assigning derived object to base object by value\n"
103             << std::endl;
104
105     Circle circle("MyCircle", 5.0);
106     Shape shape = circle; // SLICING! Circle parts are lost
107
108     std::cout << "\nCalling draw() on sliced object:" << std::endl;
109     shape.draw(); // Calls Shape::draw(), not Circle::draw()!
110
111     std::cout << "\nArea of sliced object: " << shape.area() << std::endl; // Returns 0.0, not circle area!
112
113 }
114
115 // =====
116 // 2. OBJECT SLICING WITH SHARED_PTR - WRONG WAY
117 // =====
118
119 void example_shared_ptr_slicing_wrong() {
120     std::cout << "\n==== 2. OBJECT SLICING WITH SHARED_PTR (WRONG) ===" << std::endl;
121     std::cout << "Problem: Creating shared_ptr to base from derived object by value\n" << std::endl;
122
123     Circle circle("SlicedCircle", 7.0);
124
125     // WRONG: This creates a shared_ptr<Shape> pointing to a SLICED copy
126     std::shared_ptr<Shape> shape_ptr = std::make_shared<Shape>(circle);
127
128     std::cout << "\nCalling draw() through shared_ptr:" << std::endl;
129     shape_ptr->draw(); // Calls Shape::draw(), not Circle::draw()!
130
131     std::cout << "\nArea: " << shape_ptr->area() << std::endl; // Returns 0.0!
132
133     std::cout << "\nObject was sliced when passed by value to make_shared<Shape>!" << std::endl;
134 }
135
136 // =====
137 // 3. CONTAINER OF SHARED_PTR - WRONG WAY (SLICING)
138 // =====
139
140 void example_container_slicing_wrong() {
141     std::cout << "\n==== 3. CONTAINER WITH SLICING (WRONG) ===" << std::endl;
142     std::cout << "Problem: Storing objects by value in container\n" << std::endl;
143
144     std::vector<Shape> shapes; // WRONG: Stores Shape objects by value
145 }
```

```
146     shapes.push_back(Circle("Circle1", 3.0));           // Sliced!
147     shapes.push_back(Rectangle("Rect1", 4.0, 5.0)); // Sliced!
148
149     std::cout << "\nDrawing shapes from container:" << std::endl;
150     for (const auto& shape : shapes) {
151         shape.draw(); // Always calls Shape::draw()
152         std::cout << " Area: " << shape.area() << std::endl; // Always
153         returns 0.0!
154     }
155
156     std::cout << "\n All derived class data was sliced off!" << std::endl;
157 }
158 // =====
159 // 4. COPYING SMART POINTERS - POTENTIAL SLICING
160 // =====
161
162 void process_shape_wrong(std::shared_ptr<Shape> shape) {
163     // If we copy the object instead of using the pointer
164     Shape copy = *shape; // SLICING!
165     copy.draw(); // Calls Shape::draw(), not the derived version
166 }
167
168 void example_copying_smart_ptr_wrong() {
169     std::cout << "\n== 4. COPYING OBJECT FROM SMART_PTR (WRONG) ==" << std::endl;
170     std::cout << "Problem: Dereferencing smart_ptr and copying by value\n" <<
171         std::endl;
172
173     auto circle = std::make_shared<Circle>("PointerCircle", 6.0);
174
175     std::cout << "\nOriginal (via pointer):" << std::endl;
176     circle->draw();
177
178     std::cout << "\nCopied value from pointer:" << std::endl;
179     process_shape_wrong(circle); // Slicing happens inside function
180
181     std::cout << "\n Dereferencing and copying caused slicing!" << std::endl;
182 }
183 // =====
184 // 5. CORRECT WAY: SHARED_PTR TO DERIVED CLASS
185 // =====
186
187 void example_shared_ptr_correct() {
188     std::cout << "\n== 5. CORRECT: SHARED_PTR TO DERIVED CLASS ==" << std::endl;
189     std::cout << "Solution: Create shared_ptr directly to derived class\n" <<
190         std::endl;
191
192     // CORRECT: Create shared_ptr<Circle> directly
193     std::shared_ptr<Circle> circle = std::make_shared<Circle>("CorrectCircle",
194         8.0);
195 }
```

```
194 // Upcast to base class pointer (no slicing!)
195 std::shared_ptr<Shape> shape_ptr = circle;
196
197 std::cout << "\nCalling draw() through base pointer:" << std::endl;
198 shape_ptr->draw(); // Calls Circle::draw() via polymorphism
199
200 std::cout << "\nArea: " << shape_ptr->area() << std::endl; // Returns
201 // actual circle area
202
203 std::cout << "\n Polymorphism works correctly - no slicing!" << std::endl
204 ;
205
206 // =====
207 // 6. CORRECT WAY: CONTAINER OF SMART POINTERS
208 // =====
209
210 void example_container_correct() {
211     std::cout << "\n== 6. CORRECT: CONTAINER OF SMART_PTR ==" << std::endl;
212     std::cout << "Solution: Store pointers to base class, not objects\n" <<
213         std::endl;
214
215     // CORRECT: Vector of pointers to Shape
216     std::vector<std::shared_ptr<Shape>> shapes;
217
218     shapes.push_back(std::make_shared<Circle>("Circle2", 4.0));
219     shapes.push_back(std::make_shared<Rectangle>("Rect2", 6.0, 3.0));
220     shapes.push_back(std::make_shared<Circle>("Circle3", 2.5));
221
222     std::cout << "\nDrawing shapes from container:" << std::endl;
223     for (const auto& shape : shapes) {
224         shape->draw(); // Calls correct derived class method
225         std::cout << " Area: " << shape->area() << std::endl; // Returns
226             actual area
227     }
228
229     std::cout << "\n Polymorphism works - each object retains its type!" <<
230         std::endl;
231 }
232
233 // =====
234 // 7. CORRECT WAY: UNIQUE_PTR (C++11/14)
235 // =====
236
237 void example_unique_ptr_correct() {
238     std::cout << "\n== 7. CORRECT: UNIQUE_PTR FOR OWNERSHIP ==" << std::endl
239         ;
240     std::cout << "Solution: Use unique_ptr for exclusive ownership\n" << std::endl;
241
242     // CORRECT: unique_ptr to derived class
243     std::unique_ptr<Circle> circle = std::make_unique<Circle>("UniqueCircle",
244         5.5);
245 }
```

```
240 // Move to base class pointer (transfer ownership)
241 std::unique_ptr<Shape> shape_ptr = std::move(circle);
242
243 std::cout << "\nCalling methods through unique_ptr:" << std::endl;
244 shape_ptr->draw(); // Calls Circle::draw()
245 shape_ptr->info(); // Calls Circle::info()
246
247 std::cout << "\n unique_ptr maintains polymorphism without slicing!" <<
248     std::endl;
249 }
250
251 // =====
252 // 8. FACTORY PATTERN WITH SMART POINTERS (CORRECT)
253 // =====
254
255 std::unique_ptr<Shape> createShape(const std::string& type, const std::string&
256 name) {
257     if (type == "circle") {
258         return std::make_unique<Circle>(name, 10.0);
259     } else if (type == "rectangle") {
260         return std::make_unique<Rectangle>(name, 8.0, 6.0);
261     }
262     return std::make_unique<Shape>(name);
263 }
264
265 void example_factory_pattern() {
266     std::cout << "\n== 8. FACTORY PATTERN (CORRECT) ==" << std::endl;
267     std::cout << "Solution: Return base class pointer to derived objects\n" <<
268         std::endl;
269
270     auto shape1 = createShape("circle", "FactoryCircle");
271     auto shape2 = createShape("rectangle", "FactoryRect");
272
273     std::cout << "\nUsing factory-created objects:" << std::endl;
274     shape1->draw();
275     shape2->draw();
276
277     std::cout << "\n Factory pattern prevents slicing by returning pointers!" <<
278         std::endl;
279 }
280
281 // =====
282 // 9. COMPARISON: WRONG VS RIGHT
283 // =====
284
285 void example_side_by_side_comparison() {
286     std::cout << "\n== 9. SIDE-BY-SIDE COMPARISON ==" << std::endl;
287
288     std::cout << "\n WRONG - Object Slicing:" << std::endl;
289     Circle c1("WrongCircle", 5.0);
290     Shape s1 = c1; // Slicing!
291     s1.draw(); // Calls Shape::draw()
292     std::cout << "    Area: " << s1.area() << " (lost circle data!)" << std::
293         endl;
```

```
289     std::cout << "\n CORRECT - Pointer (no slicing):" << std::endl;
290     auto c2 = std::make_shared<Circle>("RightCircle", 5.0);
291     std::shared_ptr<Shape> s2 = c2; // No slicing!
292     s2->draw(); // Calls Circle::draw()
293     std::cout << "    Area: " << s2->area() << " (preserves circle data!)" <<
294             std::endl;
295 }
296
297 // =====
298 // MAIN FUNCTION
299 // =====
300
301 int main() {
302     std::cout << "\n
303         =====" <<
304         std::endl;
305     std::cout << "    OBJECT SLICING AND SMART POINTERS (C++11/14)" << std::endl
306         ;
307     std::cout << "
308         =====" <<
309         std::endl;
310
311     // Wrong ways (demonstrating slicing)
312     example_classic_slicing();
313     example_shared_ptr_slicing_wrong();
314     example_container_slicing_wrong();
315     example_copying_smart_ptr_wrong();
316
317     // Correct ways (avoiding slicing)
318     example_shared_ptr_correct();
319     example_container_correct();
320     example_unique_ptr_correct();
321     example_factory_pattern();
322
323     // Comparison
324     example_side_by_side_comparison();
325
326     std::cout << "\n
327         =====" <<
328         std::endl;
329     std::cout << "    KEY TAKEAWAYS" << std::endl;
330     std::cout << "
331         =====" <<
332         std::endl;
333     std::cout << "\n    CAUSES OF OBJECT SLICING:" << std::endl;
334     std::cout << "    1. Assigning derived object to base object by value" <<
335             std::endl;
336     std::cout << "    2. Passing derived object by value to function expecting
337             base" << std::endl;
338     std::cout << "    3. Storing objects by value in containers of base type"
339             << std::endl;
340     std::cout << "    4. Creating smart_ptr<Base>(derived_object) - copies by
341             value" << std::endl;
```

```
329     std::cout << " 5. Dereferencing smart pointer and copying: Base copy = *  
330     ptr" << std::endl;  
331  
331     std::cout << "\n HOW TO AVOID SLICING WITH SMART POINTERS:" << std::endl;  
332     std::cout << " 1. Always create smart_ptr to the actual derived type" <<  
333         std::endl;  
333     std::cout << " 2. Use make_shared<DerivedType>() or make_unique<  
334         DerivedType>()" << std::endl;  
334     std::cout << " 3. Store smart_ptr<Base> in containers, not objects" <<  
335         std::endl;  
335     std::cout << " 4. Pass smart pointers by reference or const reference"  
336         << std::endl;  
336     std::cout << " 5. Never dereference and copy: always use pointer access  
337         (->)" << std::endl;  
338  
338     std::cout << "\n C++11/14 BEST PRACTICES:" << std::endl;  
339     std::cout << " • Use std::unique_ptr for exclusive ownership" << std::  
340         endl;  
340     std::cout << " • Use std::shared_ptr for shared ownership" << std::endl;  
341     std::cout << " • Always use make_shared/make_unique (C++14)" << std::  
342         endl;  
342     std::cout << " • Prefer polymorphism via pointers, not by value" << std  
343         ::endl;  
343     std::cout << " • Virtual destructors are essential for polymorphic  
344         classes" << std::endl;  
344  
345     std::cout << "\n  
345     ======\n" <<  
346         std::endl;  
347  
347     return 0;  
348 }
```

## 50 Source Code: OptionalExamples.cpp

File: src/OptionalExamples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <optional>
4 #include <vector>
5 #include <map>
6 #include <algorithm>
7 #include <cmath>
8
9 // =====
10 // 1. BASIC OPTIONAL USAGE
11 // =====
12 void example_basic_optional() {
13     std::cout << "\n== 1. BASIC OPTIONAL USAGE ==" << std::endl;
14
15     std::optional<int> maybe_value; // Empty optional
16
17     std::cout << "Has value: " << (maybe_value.has_value() ? "Yes" : "No") <<
18         std::endl;
19
20     maybe_value = 42; // Assign value
21     std::cout << "Has value: " << (maybe_value.has_value() ? "Yes" : "No") <<
22         std::endl;
23     std::cout << "Value: " << *maybe_value << std::endl;
24     std::cout << "Value (using .value()): " << maybe_value.value() << std::endl;
25 }
26
27 // =====
28 // 2. OPTIONAL WITH VALUE_OR
29 // =====
30 void example_value_or() {
31     std::cout << "\n== 2. OPTIONAL WITH VALUE_OR ==" << std::endl;
32
33     std::optional<int> empty_opt;
34     std::optional<int> filled_opt = 100;
35
36     std::cout << "Empty optional value_or(0): " << empty_opt.value_or(0) <<
37         std::endl;
38     std::cout << "Filled optional value_or(0): " << filled_opt.value_or(0) <<
39         std::endl;
40
41     std::optional<std::string> empty_str;
42     std::cout << "Empty string value_or: " << empty_str.value_or("default") <<
43         std::endl;
44 }
45
46 // =====
47 // 3. OPTIONAL AS FUNCTION RETURN TYPE
48 // =====
```

```
44 std::optional<int> find_value(const std::vector<int>& vec, int target) {
45     auto it = std::find(vec.begin(), vec.end(), target);
46     if (it != vec.end()) {
47         return *it;
48     }
49     return std::nullopt; // Return empty optional
50 }
51
52 std::optional<std::string> get_name_by_id(int id) {
53     std::map<int, std::string> database = {
54         {1, "Alice"}, 
55         {2, "Bob"}, 
56         {3, "Charlie"}
57     };
58
59     auto it = database.find(id);
60     if (it != database.end()) {
61         return it->second;
62     }
63     return std::nullopt;
64 }
65
66 void example_optional_return() {
67     std::cout << "\n== 3. OPTIONAL AS FUNCTION RETURN TYPE ==" << std::endl;
68
69     std::vector<int> numbers = {10, 20, 30, 40, 50};
70
71     auto result1 = find_value(numbers, 30);
72     if (result1) {
73         std::cout << "Found: " << *result1 << std::endl;
74     } else {
75         std::cout << "Not found" << std::endl;
76     }
77
78     auto result2 = find_value(numbers, 99);
79     if (result2) {
80         std::cout << "Found: " << *result2 << std::endl;
81     } else {
82         std::cout << "Not found" << std::endl;
83     }
84
85     auto name1 = get_name_by_id(2);
86     std::cout << "ID 2: " << name1.value_or("Unknown") << std::endl;
87
88     auto name2 = get_name_by_id(99);
89     std::cout << "ID 99: " << name2.value_or("Unknown") << std::endl;
90 }
91
92 // =====
93 // 4. OPTIONAL WITH IF STATEMENT
94 // =====
95 void example_optional_if() {
96     std::cout << "\n== 4. OPTIONAL WITH IF STATEMENT ==" << std::endl;
```

```
98     auto maybe_name = get_name_by_id(1);
99
100    // C++17 if with initializer
101    if (auto name = get_name_by_id(1); name.has_value()) {
102        std::cout << "Found name: " << *name << std::endl;
103    } else {
104        std::cout << "Name not found" << std::endl;
105    }
106
107    if (auto name = get_name_by_id(999); name) {
108        std::cout << "Found name: " << *name << std::endl;
109    } else {
110        std::cout << "Name not found" << std::endl;
111    }
112}
113
114// =====
115// 5. OPTIONAL WITH EXCEPTIONS
116// =====
117void example_optional_exceptions() {
118    std::cout << "\n== 5. OPTIONAL WITH EXCEPTIONS ==" << std::endl;
119
120    std::optional<int> empty_opt;
121    std::optional<int> filled_opt = 42;
122
123    try {
124        std::cout << "Filled optional value: " << filled_opt.value() << std::endl;
125        std::cout << "Empty optional value: " << empty_opt.value() << std::endl; // Throws!
126    } catch (const std::bad_optional_access& e) {
127        std::cout << "Exception caught: " << e.what() << std::endl;
128    }
129}
130
131// =====
132// 6. OPTIONAL WITH CUSTOM TYPES
133// =====
134struct Person {
135    std::string name;
136    int age;
137
138    Person(const std::string& n, int a) : name(n), age(a) {}
139
140    friend std::ostream& operator<<(std::ostream& os, const Person& p) {
141        os << p.name << " (age " << p.age << ")";
142        return os;
143    }
144};
145
146std::optional<Person> find_person(const std::string& name) {
147    std::vector<Person> people = {
148        {"Alice", 30},
149        {"Bob", 25},
```

```
150     {"Charlie", 35}  
151 };  
152  
153 auto it = std::find_if(people.begin(), people.end(),  
154 [&name](const Person& p) { return p.name == name; });  
155  
156 if (it != people.end()) {  
157     return *it;  
158 }  
159 return std::nullopt;  
160 }  
161  
162 void example_optional_custom_type() {  
163     std::cout << "\n== 6. OPTIONAL WITH CUSTOM TYPES ==" << std::endl;  
164  
165     auto person1 = find_person("Alice");  
166     if (person1) {  
167         std::cout << "Found: " << *person1 << std::endl;  
168     }  
169  
170     auto person2 = find_person("David");  
171     if (person2) {  
172         std::cout << "Found: " << *person2 << std::endl;  
173     } else {  
174         std::cout << "Person not found" << std::endl;  
175     }  
176 }  
177  
178 // =====  
179 // 7. OPTIONAL WITH EMPLACE  
180 // =====  
181 void example_optional_emplace() {  
182     std::cout << "\n== 7. OPTIONAL WITH EMPLACE ==" << std::endl;  
183  
184     std::optional<Person> opt_person;  
185  
186     std::cout << "Before emplace: " << (opt_person.has_value() ? "Has value" :  
187         "Empty") << std::endl;  
188  
189     opt_person.emplace("John", 28); // Construct in-place  
190  
191     std::cout << "After emplace: " << *opt_person << std::endl;  
192  
193     opt_person.emplace("Jane", 32); // Replace existing value  
194     std::cout << "After second emplace: " << *opt_person << std::endl;  
195 }  
196  
197 // =====  
198 // 8. OPTIONAL WITH RESET  
199 // =====  
200 void example_optional_reset() {  
201     std::cout << "\n== 8. OPTIONAL WITH RESET ==" << std::endl;  
202  
203     std::optional<int> value = 100;
```

```
203     std::cout << "Initial value: " << *value << std::endl;
204
205     value.reset(); // Clear the optional
206     std::cout << "After reset: " << (value.has_value() ? "Has value" : "Empty"
207         ) << std::endl;
208
209     value = 200;
210     std::cout << "After reassignment: " << *value << std::endl;
211
212     value = std::nullopt; // Another way to clear
213     std::cout << "After nullopt: " << (value.has_value() ? "Has value" : "
214         Empty) << std::endl;
215 }
216
217 // =====
218 // 9. OPTIONAL CHAINING WITH TRANSFORM (C++23-style)
219 // =====
220 std::optional<double> safe_sqrt(double value) {
221     if (value >= 0.0) {
222         return std::sqrt(value);
223     }
224     return std::nullopt;
225 }
226
227 std::optional<double> safe_divide(double numerator, double denominator) {
228     if (denominator != 0.0) {
229         return numerator / denominator;
230     }
231     return std::nullopt;
232 }
233
234 void example_optional_chaining() {
235     std::cout << "\n== 9. OPTIONAL CHAINING ==" << std::endl;
236
237     auto result1 = safe_divide(100, 4);
238     if (result1) {
239         auto sqrt_result = safe_sqrt(*result1);
240         if (sqrt_result) {
241             std::cout << "sqrt(100/4) = " << *sqrt_result << std::endl;
242         }
243     }
244
245     auto result2 = safe_divide(100, 0);
246     std::cout << "Division by zero result: "
247         << (result2.has_value() ? "Has value" : "Empty") << std::endl;
248 }
249
250 // =====
251 // 10. OPTIONAL WITH COMPARISON
252 // =====
253 void example_optional_comparison() {
254     std::cout << "\n== 10. OPTIONAL WITH COMPARISON ==" << std::endl;
255
256     std::optional<int> opt1 = 10;
```

```
255     std::optional<int> opt2 = 20;
256     std::optional<int> opt3 = 10;
257     std::optional<int> empty;
258
259     std::cout << "opt1 == opt3: " << (opt1 == opt3 ? "true" : "false") << std::endl;
260     std::cout << "opt1 < opt2: " << (opt1 < opt2 ? "true" : "false") << std::endl;
261     std::cout << "opt1 == 10: " << (opt1 == 10 ? "true" : "false") << std::endl;
262     std::cout << "empty == nullopt: " << (empty == std::nullopt ? "true" : "false") << std::endl;
263 }
264
265 // =====
266 // 11. OPTIONAL IN CONTAINERS
267 // =====
268 void example_optional_in_containers() {
269     std::cout << "\n== 11. OPTIONAL IN CONTAINERS ==" << std::endl;
270
271     std::vector<std::optional<int>> values = {
272         10,
273         std::nullopt,
274         20,
275         std::nullopt,
276         30
277     };
278
279     std::cout << "Processing optional values:" << std::endl;
280     for (size_t i = 0; i < values.size(); ++i) {
281         if (values[i]) {
282             std::cout << "    Index " << i << ": " << *values[i] << std::endl;
283         } else {
284             std::cout << "    Index " << i << ": empty" << std::endl;
285         }
286     }
287
288     // Count non-empty values
289     int count = 0;
290     for (const auto& opt : values) {
291         if (opt) ++count;
292     }
293     std::cout << "Non-empty values: " << count << std::endl;
294 }
295
296 // =====
297 // 12. OPTIONAL WITH MAKE_OPTIONAL
298 // =====
299 std::optional<Person> create_person_if_valid(const std::string& name, int age)
300 {
301     if (!name.empty() && age > 0 && age < 150) {
302         return Person(name, age);
303     }
304     return std::nullopt;
```

```
304 }
305
306 void example_make_optional() {
307     std::cout << "\n==== 12. OPTIONAL WITH MAKE_OPTIONAL ===" << std::endl;
308
309     auto person1 = create_person_if_valid("Alice", 30);
310     if (person1) {
311         std::cout << "Valid person: " << *person1 << std::endl;
312     }
313
314     auto person2 = create_person_if_valid("", 25);
315     if (person2) {
316         std::cout << "Valid person: " << *person2 << std::endl;
317     } else {
318         std::cout << "Invalid person (empty name)" << std::endl;
319     }
320
321     auto person3 = create_person_if_valid("Bob", -5);
322     if (person3) {
323         std::cout << "Valid person: " << *person3 << std::endl;
324     } else {
325         std::cout << "Invalid person (negative age)" << std::endl;
326     }
327 }
328
329 // =====
330 // MAIN FUNCTION
331 // =====
332 int main() {
333     std::cout << "\n=====" << std::endl;
334     std::cout << "      C++17 STD::OPTIONAL EXAMPLES" << std::endl;
335     std::cout << "=====" << std::endl;
336
337     example_basic_optional();
338     example_value_or();
339     example_optional_return();
340     example_optional_if();
341     example_optional_exceptions();
342     example_optional_custom_type();
343     example_optional_emplace();
344     example_optional_reset();
345     example_optional_chaining();
346     example_optional_comparison();
347     example_optional_in_containers();
348     example_make_optional();
349
350     std::cout << "\n=====" << std::endl;
351     std::cout << "      ALL EXAMPLES COMPLETED" << std::endl;
352     std::cout << "=====\n" << std::endl;
353 }
```

354      **return** 0;  
355 }

## 51 Source Code: PerfectForwardingAndRequires.cpp

File: src/PerfectForwardingAndRequires.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // ADVANCED PERFECT FORWARDING, CONCEPTS, AND REQUIRES-EXPRESSIONS
3 // =====
4 // This example covers:
5 // 1. std::forward and perfect forwarding with concepts
6 // 2. "requires requires" - nested requires explained
7 // 3. Requirement clauses vs requirement expressions
8 // 4. When to use and when to avoid "requires requires"
9 // 5. Embedded systems considerations
10 //
11 // TOPICS:
12 // - Perfect forwarding with universal references
13 // - Concepts constraining forwarding functions
14 // - requires clause (starts constraint)
15 // - requires expression (tests validity)
16 // - SFINAE vs Concepts comparison
17 // - Zero-cost abstractions for embedded systems
18 // =====
19
20 #include <iostream>
21 #include <string>
22 #include <vector>
23 #include <memory>
24 #include <type_traits>
25 #include <concepts>
26 #include <utility>
27
28 // =====
29 // SECTION 1: PERFECT FORWARDING BASICS WITH std::forward
30 // =====
31
32 // Without perfect forwarding - INEFFICIENT
33 template<typename T>
34 void call_by_value(T arg) {
35     std::cout << "  Value parameter (copy made)" << std::endl;
36 }
37
38 // With perfect forwarding - EFFICIENT
39 template<typename T>
40 void call_with_forward(T&& arg) {
41     std::cout << "  Universal reference with forward" << std::endl;
42     // std::forward<T>(arg) preserves value category:
43     // - lvalue stays lvalue (no move)
44     // - rvalue stays rvalue (enables move)
45 }
46
47 void demonstrate_perfect_forwarding() {
48     std::cout << "\n==== 1. PERFECT FORWARDING BASICS ===" << std::endl;
49 }
```

```
50     std::string lvalue = "Hello";
51
52     std::cout << "\nCalling by value:" << std::endl;
53     call_by_value(lvalue); // Copy
54     call_by_value(std::string("World")); // Copy + move
55
56     std::cout << "\nCalling with forward:" << std::endl;
57     call_with_forward(lvalue); // No copy
58     call_with_forward(std::string("World")); // No copy
59
60     std::cout << "\n KEY POINT:" << std::endl;
61     std::cout << "    T&& in template = universal reference" << std::endl;
62     std::cout << "    std::forward<T> preserves value category" << std::endl;
63 }
64
65 // =====
66 // SECTION 2: REQUIRES CLAUSE VS REQUIRES EXPRESSION
67 // =====
68
69 // TERMINOLOGY:
70 // - requires clause: Introduces a constraint (keyword 'requires')
71 // - requires expression: Tests if expressions are valid
72
73 // Example 1: requires clause WITHOUT requires expression
74 template<typename T>
75 requires std::integral<T> // <- This is a requires CLAUSE
76 T square(T x) {
77     return x * x;
78 }
79
80 // Example 2: requires clause WITH requires expression
81 template<typename T>
82 requires requires(T x) { // <- First 'requires' = clause, second =
83     x + x; // Check if x can be added
84     x * x; // Check if x can be multiplied
85 }
86 T compute(T x) {
87     return x * x + x;
88 }
89
90 void demonstrate_requires_syntax() {
91     std::cout << "\n== 2. REQUIRES CLAUSE VS REQUIRES EXPRESSION ==" << std
92         ::endl;
93
94     std::cout << "\n TERMINOLOGY:" << std::endl;
95     std::cout << "    requires clause:      'requires <constraint>'" << std::
96         endl;
97     std::cout << "    requires expression: 'requires(params) { tests; }'" <<
98         std::endl;
99
100    std::cout << "\n square(5) = " << square(5) << std::endl;
101    std::cout << "    compute(5) = " << compute(5) << std::endl;
```

```
100     // square(3.14);    // ERROR: not integral
101     // compute("hi");   // ERROR: can't add/multiply strings
102 }
103
104 // =====
105 // SECTION 3: THE "REQUIRES REQUIRES" PATTERN
106 // =====
107
108 // WHAT IS "requires requires"?
109 //
110 // Template<typename T>
111 // requires requires(T t) { ... }
112 //           ^           ^
113 //           |           |
114 //           |           +-- requires EXPRESSION (tests code validity)
115 //           +----- requires CLAUSE (introduces constraint)
116 //
117 // WHEN TO USE:
118 // - You need to test if specific operations are valid
119 // - Standard concepts don't cover your needs
120 // - You're checking syntax, not just type properties
121
122 // Example: Check if type has specific member function
123 template<typename T>
124 requires requires(T obj) {           // "requires requires"
125     { obj.serialize() } -> std::same_as<std::string>;
126 }
127 void save(const T& obj) {
128     std::string data = obj.serialize();
129     std::cout << "    Saved: " << data << std::endl;
130 }
131
132 // Example: Check if type supports arithmetic and comparison
133 template<typename T>
134 requires requires(T a, T b) {           // "requires requires"
135     { a + b } -> std::convertible_to<T>;
136     { a - b } -> std::convertible_to<T>;
137     { a < b } -> std::convertible_to<bool>;
138     { a > b } -> std::convertible_to<bool>;
139 }
140 T clamp(T value, T min, T max) {
141     if (value < min) return min;
142     if (value > max) return max;
143     return value;
144 }
145
146 // Test classes
147 class Serializable {
148 public:
149     std::string serialize() const { return "Serializable{data}"; }
150 };
151
152 void demonstrate_requires_requires() {
153     std::cout << "\n==== 3. THE 'REQUIRES REQUIRES' PATTERN ===" << std::endl;
```

```
154     std::cout << "\n EXPLANATION:" << std::endl;
155     std::cout << "    First 'requires': Starts the constraint clause" << std::
156         endl;
157     std::cout << "    Second 'requires': Begins the expression testing code" << std::
158         endl;
159     std::cout << "    Inside { ... }: Operations that must be valid" << std::
160         endl;
161
162     std::cout << "\n Using save() with Serializable:" << std::endl;
163     Serializable obj;
164     save(obj);
165
166     std::cout << "\n Using clamp() with int:" << std::endl;
167     std::cout << "    clamp(150, 0, 100) = " << clamp(150, 0, 100) << std::endl
168         ;
169     std::cout << "    clamp(-10, 0, 100) = " << clamp(-10, 0, 100) << std::endl
170         ;
171     std::cout << "    clamp(50, 0, 100) = " << clamp(50, 0, 100) << std::endl;
172 }
173
174 // =====
175 // SECTION 4: PERFECT FORWARDING WITH CONCEPTS
176 // =====
177
178 // Constrain what types can be forwarded
179 template<typename T>
180 concept Movable = std::is_move_constructible_v<T>;
181
182 template<typename T>
183 concept Copyable = std::is_copy_constructible_v<T>;
184
185 // Factory function with perfect forwarding and concepts
186 template<typename T, typename... Args>
187 requires std::constructible_from<T, Args...> // Concept constrains
188     construction
189 std::unique_ptr<T> make_unique_constrained(Args&&... args) {
190     return std::make_unique<T>(std::forward<Args>(args)...);
191 }
192
193 // Forwarding wrapper with requires requires
194 template<typename Func, typename... Args>
195 requires requires(Func f, Args... args) { // Check if callable
196     f(std::forward<Args>(args)...);
197 }
198 auto forward_call(Func&& func, Args&&... args) {
199     std::cout << "    Forwarding call..." << std::endl;
200     return func(std::forward<Args>(args)...);
201 }
202
203 void demonstrate_forwarding_with_concepts() {
204     std::cout << "\n==== 4. PERFECT FORWARDING WITH CONCEPTS ===" << std::endl;
205
206     std::cout << "\n make_unique_constrained:" << std::endl;
```

```

202 auto ptr1 = make_unique_constrained<std::string>("Hello", 5, 'X');
203 std::cout << "    Created: " << *ptr1 << std::endl;
204
205 auto ptr2 = make_unique_constrained<std::vector<int>>(10, 42);
206 std::cout << "    Created vector of size: " << ptr2->size() << std::endl;
207
208 std::cout << "\n forward_call with lambda:" << std::endl;
209 auto lambda = [](int x, int y) {
210     return x + y;
211 };
212 int result = forward_call(lambda, 10, 20);
213 std::cout << "    Result: " << result << std::endl;
214 }
215
216 // =====
217 // SECTION 5: WHEN TO USE "REQUIRES REQUIRES"
218 // =====
219
220 void explain_when_to_use_requires_requires() {
221     std::cout << "\n== 5. WHEN TO USE 'REQUIRES REQUIRES' ==" << std::endl;
222
223     std::cout << "\n USE 'requires requires' WHEN:" << std::endl;
224     std::cout << "    1. Testing specific syntax/operations:" << std::endl;
225     std::cout << "        requires requires(T t) { t.foo(); t.bar(); }" << std::endl;
226     std::cout << "\n    2. No standard concept exists:" << std::endl;
227     std::cout << "        requires requires(T t) { t.serialize(); }" << std::endl;
228     std::cout << "\n    3. Complex compound expressions:" << std::endl;
229     std::cout << "        requires requires(T a, T b) { a + b * 2; }" << std::endl;
230     std::cout << "\n    4. Return type checking:" << std::endl;
231     std::cout << "        requires requires(T t) { { t.size() } -> std::same_as<
232         size_t>; }" << std::endl;
233
234     std::cout << "\n AVOID 'requires requires' WHEN:" << std::endl;
235     std::cout << "    1. Standard concept exists:" << std::endl;
236     std::cout << "        requires requires(T t) { t == t; }" << std::endl;
237     std::cout << "        requires std::equality_comparable<T>" << std::endl;
238     std::cout << "\n    2. Simple type trait works:" << std::endl;
239     std::cout << "        requires requires { typename T::value_type; }" <<
240         std::endl;
241     std::cout << "        requires requires { typename T::value_type; } # OK
242         if needed" << std::endl;
243     std::cout << "\n    3. Makes code harder to read:" << std::endl;
244     std::cout << "        Define a named concept instead!" << std::endl;
245
246     std::cout << "\n BEST PRACTICE:" << std::endl;
247     std::cout << "    Wrap 'requires requires' in a named concept:" << std::endl;
248     std::cout << "        template<typename T>" << std::endl;
249     std::cout << "        concept MyConstraint = requires(T t) { /* tests */ };" <<
250         std::endl;
251 }
```

```

248
249 // =====
250 // SECTION 6: READABILITY - NAMED CONCEPTS VS INLINE REQUIRES
251 // =====
252
253 // BAD: Inline requires requires - hard to read and reuse
254 template<typename T>
255 requires requires(T container) {
256     { container.begin() } -> std::input_or_output_iterator;
257     { container.end() } -> std::input_or_output_iterator;
258     { container.size() } -> std::convertible_to<std::size_t>;
259     requires std::is_same_v<typename T::value_type, int>;
260 }
261 void process_bad(const T& container) {
262     std::cout << "Size: " << container.size() << std::endl;
263 }
264
265 // GOOD: Named concept - readable and reusable
266 template<typename T>
267 concept IntContainer = requires(T container) {
268     { container.begin() } -> std::input_or_output_iterator;
269     { container.end() } -> std::input_or_output_iterator;
270     { container.size() } -> std::convertible_to<std::size_t>;
271     requires std::is_same_v<typename T::value_type, int>;
272 };
273
274 template<IntContainer T>
275 void process_good(const T& container) {
276     std::cout << "Size: " << container.size() << std::endl;
277 }
278
279 void demonstrate_readability() {
280     std::cout << "\n==== 6. READABILITY: NAMED CONCEPTS VS INLINE ===" << std::endl;
281
282     std::cout << "\n BAD - Inline 'requires requires':"
283     std::cout << " template<typename T>" << std::endl;
284     std::cout << "     requires requires(T t) { /* 10 lines */ }"
285     std::cout << "     void func(T x) { ... }"
286     std::cout << "     Problems: Hard to read, can't reuse, error messages
287     unclear" << std::endl;
288
289     std::cout << "\n GOOD - Named concept:"
290     std::cout << " template<typename T>" << std::endl;
291     std::cout << "     concept MyConstraint = requires(T t) { /* tests */ };" <<
292         std::endl;
293     std::cout << "\n     template<MyConstraint T>" << std::endl;
294     std::cout << "     void func(T x) { ... }"
295     std::cout << "     Benefits: Readable, reusable, clear error messages" <<
296         std::endl;
297
298     std::vector<int> vec = {1, 2, 3, 4, 5};
299     process_good(vec);
300 }

```

```
298
299 // =====
300 // SECTION 7: EMBEDDED SYSTEMS CONSIDERATIONS
301 // =====
302
303 void explain_embedded_systems() {
304     std::cout << "\n==== 7. EMBEDDED SYSTEMS CONSIDERATIONS ===" << std::endl;
305
306     std::cout << "\n ARE CONCEPTS USEFUL FOR EMBEDDED SYSTEMS?" << std::endl;
307     std::cout << "    YES! Here's why:" << std::endl;
308
309     std::cout << "\n1.  ZERO RUNTIME COST:" << std::endl;
310     std::cout << " • Concepts are compile-time only" << std::endl;
311     std::cout << " • No runtime overhead vs unconstrained templates" << std
312         ::endl;
313     std::cout << " • Generated code is identical" << std::endl;
314     std::cout << " • Same as SFINAE but cleaner" << std::endl;
315
316     std::cout << "\n2.  REDUCE CODE SIZE:" << std::endl;
317     std::cout << " • Catch errors early = fewer template instantiations" <<
318         std::endl;
319     std::cout << " • Better error messages = less debugging code" << std::
320         endl;
321     std::cout << " • Explicit constraints prevent accidental instantiations"
322         << std::endl;
323
324     std::cout << "\n3.  TYPE SAFETY:" << std::endl;
325     std::cout << " • Catch type errors at compile time" << std::endl;
326     std::cout << " • No runtime checks needed" << std::endl;
327     std::cout << " • Perfect for safety-critical systems" << std::endl;
328
329     std::cout << "\n4.  SELF-DOCUMENTING:" << std::endl;
330     std::cout << " • Requirements are explicit in code" << std::endl;
331     std::cout << " • No need for extensive comments" << std::endl;
332     std::cout << " • Easier code review and maintenance" << std::endl;
333
334     std::cout << "\n WHEN TO BE CAUTIOUS:" << std::endl;
335     std::cout << " 1. Compiler support: Need C++20" << std::endl;
336     std::cout << " 2. Compilation time: Complex concepts slow builds" << std
337         ::endl;
338     std::cout << " 3. Code size: Templates can increase binary size" << std
339         ::endl;
340     std::cout << " 4. Debugging: Template errors can be complex" << std::
341         endl;
342
343     std::cout << "\n EMBEDDED BEST PRACTICES:" << std::endl;
344     std::cout << "    Use concepts for HAL (Hardware Abstraction Layer)" <<
345         std::endl;
346     std::cout << "    Constrain register access templates" << std::endl;
347     std::cout << "    Type-safe peripheral interfaces" << std::endl;
348     std::cout << "    Compile-time buffer size checking" << std::endl;
349     std::cout << "    Avoid deep template nesting" << std::endl;
350     std::cout << "    Measure binary size impact" << std::endl;
351 }
```

```
344 // =====
345 // SECTION 8: EMBEDDED EXAMPLE - REGISTER ACCESS
346 // =====
347
348 // Embedded systems often need type-safe register access
349 template<typename T>
350 concept RegisterType = requires {
351     requires std::is_integral_v<T>;
352     requires sizeof(T) <= 4; // 32-bit or smaller
353     requires std::is_trivially_copyable_v<T>;
354 };
355
356
357 template<RegisterType T>
358 class Register {
359 private:
360     volatile T* address_;
361
362 public:
363     explicit Register(uintptr_t addr) : address_(reinterpret_cast<volatile T
364             *>(addr)) {}
365
366     void write(T value) const {
367         *address_ = value;
368     }
369
370     T read() const {
371         return *address_;
372     }
373
374     void set_bit(unsigned bit) requires std::unsigned_integral<T> {
375         *address_ |= (T{1} << bit);
376     }
377
378     void clear_bit(unsigned bit) requires std::unsigned_integral<T> {
379         *address_ &= ~(T{1} << bit);
380     }
381
382     void demonstrate_embedded_concepts() {
383         std::cout << "\n==== 8. EMBEDDED EXAMPLE - REGISTER ACCESS ===" << std::
384             endl;
385
386         std::cout << "\n Type-safe register access with concepts:" << std::endl;
387
388         // Simulate hardware registers
389         uint32_t mock_register = 0x00000000;
390         Register<uint32_t> gpio_control(reinterpret_cast<uintptr_t>(&mock_register
391             ));
392
393         std::cout << "    Initial value: 0x" << std::hex << mock_register << std::
394             dec << std::endl;
395
396         gpio_control.write(0xAABBCCDD);
397
398 }
```

```

394     std::cout << "    After write:  0x" << std::hex << mock_register << std::
395         dec << std::endl;
396
397     gpio_control.set_bit(4);
398     std::cout << "    After set_bit(4): 0x" << std::hex << mock_register << std::
399         dec << std::endl;
400
401     std::cout << "\n Benefits:" << std::endl;
402     std::cout << " •  Compile-time type checking" << std::endl;
403     std::cout << " •  No runtime overhead" << std::endl;
404     std::cout << " •  Can't use wrong size types" << std::endl;
405     std::cout << " •  set_bit/clear_bit only for unsigned types" << std::endl;
406     ;
407 }
408
409 // =====
410 // SECTION 9: COMPARISON - SFINAE VS CONCEPTS
411 // =====
412
413 // Old way: SFINAE (Substitution Failure Is Not An Error)
414 template<typename T, typename = std::enable_if_t<std::is_integral_v<T>>>
415 T old_square(T x) {
416     return x * x;
417 }
418
419 // New way: Concepts
420 template<std::integral T>
421 T new_square(T x) {
422     return x * x;
423 }
424
425 void compare_sfinae_vs_concepts() {
426     std::cout << "\n== 9. COMPARISON: SFINAES VS CONCEPTS ==" << std::endl;
427
428     std::cout << "\n OLD WAY (SFINAE):" << std::endl;
429     std::cout << "    template<typename T, typename = std::enable_if_t<...>>>" << std::endl;
430     std::cout << "    T func(T x) { ... }" << std::endl;
431     std::cout << "\n    Problems:" << std::endl;
432     std::cout << " •  Hard to read and write" << std::endl;
433     std::cout << " •  Terrible error messages" << std::endl;
434     std::cout << " •  Verbose and error-prone" << std::endl;
435
436     std::cout << "\n NEW WAY (CONCEPTS):" << std::endl;
437     std::cout << "    template<std::integral T>" << std::endl;
438     std::cout << "    T func(T x) { ... }" << std::endl;
439     std::cout << "\n    Benefits:" << std::endl;
440     std::cout << " •  Clean and readable" << std::endl;
441     std::cout << " •  Clear error messages" << std::endl;
442     std::cout << " •  Intent is obvious" << std::endl;
443     std::cout << " •  Easier to maintain" << std::endl;
444
445     std::cout << "\n Both work the same:" << std::endl;
446     std::cout << "    old_square(5) = " << old_square(5) << std::endl;

```

```
444     std::cout << "    new_square(5) = " << new_square(5) << std::endl;
445     std::cout << "\n Same runtime cost: ZERO" << std::endl;
446 }
447
448 // =====
449 // SECTION 10: COMPLETE EXAMPLE - THREAD-SAFE QUEUE
450 // =====
451
452 #include <mutex>
453 #include <queue>
454 #include <optional>
455
456 template<typename T>
457 concept ThreadSafeElement = requires {
458     requires std::is_move_constructible_v<T> || std::is_copy_constructible_v<T
459         >;
460     requires std::is_destructible_v<T>;
461 };
462
463 template<ThreadSafeElement T>
464 class ThreadSafeQueue {
465 private:
466     mutable std::mutex mutex_;
467     std::queue<T> queue_;
468
469 public:
470     // Perfect forwarding with concepts
471     template<typename U>
472     requires std::constructible_from<T, U>
473     void push(U&& item) {
474         std::lock_guard<std::mutex> lock(mutex_);
475         queue_.push(std::forward<U>(item));
476     }
477
478     std::optional<T> try_pop() {
479         std::lock_guard<std::mutex> lock(mutex_);
480         if (queue_.empty()) {
481             return std::nullopt;
482         }
483         T item = std::move(queue_.front());
484         queue_.pop();
485         return item;
486     }
487
488     bool empty() const {
489         std::lock_guard<std::mutex> lock(mutex_);
490         return queue_.empty();
491     }
492
493 void demonstrate_complete_example() {
494     std::cout << "\n== 10. COMPLETE EXAMPLE - THREAD-SAFE QUEUE ==" << std::
495         endl;
```

```
496 ThreadSafeQueue<std::string> queue;
497
498 std::cout << "\n Pushing items with perfect forwarding:" << std::endl;
499
500 std::string lval = "lvalue string";
501 queue.push(lval); // Copy (lvalue)
502 std::cout << " Pushed lvalue (copied)" << std::endl;
503
504 queue.push(std::string("rvalue")); // Move (rvalue)
505 std::cout << " Pushed rvalue (moved)" << std::endl;
506
507 queue.push("literal"); // Construct in place
508 std::cout << " Pushed literal (constructed)" << std::endl;
509
510 std::cout << "\n Popping items:" << std::endl;
511 while (auto item = queue.try_pop()) {
512     std::cout << " Popped: " << *item << std::endl;
513 }
514
515 std::cout << "\n Combines:" << std::endl;
516 std::cout << " • Concepts (ThreadSafeElement)" << std::endl;
517 std::cout << " • Perfect forwarding (std::forward)" << std::endl;
518 std::cout << " • requires clause (std::constructible_from)" << std::endl
519     ;
520 std::cout << " • Modern C++ (std::optional, std::mutex)" << std::endl;
521 }
522 // =====
523 // MAIN FUNCTION
524 // =====
525
526 int main() {
527     std::cout << "\n";
528     std::cout << " \n";
529     std::cout << "     ADVANCED PERFECT FORWARDING & CONCEPTS
530             \n";
531     std::cout << " \n";
532     std::cout << " \n";
533
534     demonstrate_perfect_forwarding();
535     demonstrate_requires_syntax();
536     demonstrate_requires_requires();
537     demonstrate_forwarding_with_concepts();
538     explain_when_to_use_requires_requires();
539     demonstrate_readability();
540     explain_embedded_systems();
541     demonstrate_embedded_concepts();
542     compare_sfinae_vs_concepts();
543     demonstrate_complete_example();
544
545     std::cout << "\n" << std::string(70, '=') << std::endl;
546 }
```

```
546     std::cout << "SUMMARY:\n";
547     std::cout << std::string(70, '=') << std::endl;
548
549     std::cout << "\n KEY TAKEAWAYS:" << std::endl;
550     std::cout << "\n1. PERFECT FORWARDING:" << std::endl;
551     std::cout << " • Use T&& (universal reference) in templates" << std::endl;
552     std::cout << " • Always use std::forward<T> when forwarding" << std::endl;
553     std::cout << " • Preserves lvalue/rvalue value category" << std::endl;
554
555     std::cout << "\n2. 'REQUIRES REQUIRES':"
556     std::cout << " • First: starts constraint clause" << std::endl;
557     std::cout << " • Second: begins expression testing" << std::endl;
558     std::cout << " • Use for testing specific operations" << std::endl;
559     std::cout << " • Prefer named concepts for readability" << std::endl;
560
561     std::cout << "\n3. EMBEDDED SYSTEMS:"
562     std::cout << " • Zero runtime cost (compile-time only)" << std::endl;
563     std::cout << " • Perfect for safety-critical code" << std::endl;
564     std::cout << " • Use for register access, HAL, type safety" << std::endl;
565     std::cout << " • Watch compilation time and binary size" << std::endl;
566
567     std::cout << "\n4. BEST PRACTICES:"
568     std::cout << " • Use standard concepts when available" << std::endl;
569     std::cout << " • Name complex constraints as concepts" << std::endl;
570     std::cout << " • Prefer concepts over SFINAE" << std::endl;
571     std::cout << " • Document requirements clearly" << std::endl;
572
573     std::cout << "\n Modern C++ = Type Safety + Zero Cost\n" << std::endl;
574
575     return 0;
576 }
```

## 52 Source Code: PimplIdiom.cpp

File: [src/PimplIdiom.cpp](#)

Repository: [View on GitHub](#)

```
1 // =====
2 // PIMPL IDIOM IN MODERN C++ - STILL RELEVANT?
3 // =====
4 // Comprehensive guide covering:
5 // 1. What is Pimpl (Pointer to Implementation)?
6 // 2. Traditional benefits and use cases
7 // 3. Is it still relevant in Modern C++ (C++11/14/17/20/23)?
8 // 4. Performance implications (especially for real-time systems)
9 // 5. Cache locality and memory indirection problems
10 // 6. Alternatives and mitigations
11 // 7. When to use and when to avoid Pimpl
12 //
13 // KEY QUESTION: Why Pimpl may be BAD for real-time systems?
14 // ANSWER: Pointer indirection breaks cache locality, causes memory jumps,
15 //         unpredictable latency, and extra heap allocation overhead
16 //
17 // Build: g++ -std=c++20 -Wall -Wextra -O2 -o PimplIdiom PimplIdiom.cpp
18 // =====
19
20 #include <iostream>
21 #include <memory>
22 #include <string>
23 #include <vector>
24 #include <chrono>
25 #include <iomanip>
26
27 // =====
28 // SECTION 1: WHAT IS THE PIMPL IDIOM?
29 // =====
30
31 namespace what_is_pimpl {
32
33 // =====
34 // WITHOUT PIMPL - Traditional approach
35 // =====
36
37 // Widget.h (header file)
38 class WidgetNoPimpl {
39 private:
40     // ALL implementation details exposed in header
41     std::string name_;
42     int id_;
43     std::vector<double> data_;
44
45     // Private helper functions also exposed
46     void validateData();
47     void processInternal();
48
49 public:
```

```
50     WidgetNoPimpl(const std::string& name, int id);
51     ~WidgetNoPimpl();
52
53     void doSomething();
54     void process();
55 };
56
57 // PROBLEMS WITHOUT PIMPL:
58 // 1. All private members visible in header (no encapsulation)
59 // 2. Changing private members requires recompiling ALL clients
60 // 3. Implementation details leak to users
61 // 4. Longer compile times (includes propagate)
62 // 5. Binary compatibility issues (ABI breaks easily)
63
64 // =====
65 // WITH PIMPL - Pointer to Implementation
66 // =====
67
68 // Widget.h (header file)
69 class WidgetWithPimpl {
70 private:
71     // ONLY pointer to implementation - opaque pointer
72     struct Impl; // Forward declaration
73     std::unique_ptr<Impl> pImpl_;
74
75 public:
76     WidgetWithPimpl(const std::string& name, int id);
77     ~WidgetWithPimpl(); // Must be in .cpp (where Impl is complete)
78
79     // Move operations (copy disabled by default with unique_ptr)
80     WidgetWithPimpl(WidgetWithPimpl&&) noexcept;
81     WidgetWithPimpl& operator=(WidgetWithPimpl&&) noexcept;
82
83     void doSomething();
84     void process();
85 };
86
87 // Widget.cpp (implementation file)
88 struct WidgetWithPimpl::Impl {
89     // ALL implementation details hidden here
90     std::string name;
91     int id;
92     std::vector<double> data;
93
94     Impl(const std::string& n, int i) : name(n), id(i) {}
95
96     void validateData() {
97         std::cout << "      Validating data for " << name << "\n";
98     }
99
100    void processInternal() {
101        std::cout << "      Processing " << data.size() << " elements\n";
102    }
103};
```

```

104
105 // Implementation
106 WidgetWithPimpl::WidgetWithPimpl(const std::string& name, int id)
107   : pImpl_(std::make_unique<Impl>(name, id)) {
108     std::cout << "    WidgetWithPimpl constructed: " << name << "\n";
109 }
110
111 // Destructor must be in .cpp where Impl is complete type
112 WidgetWithPimpl::~WidgetWithPimpl() = default;
113
114 // Move operations
115 WidgetWithPimpl::WidgetWithPimpl(WidgetWithPimpl&&) noexcept = default;
116 WidgetWithPimpl& WidgetWithPimpl::operator=(WidgetWithPimpl&&) noexcept =
117   default;
118
119 void WidgetWithPimpl::doSomething() {
120   pImpl_->validateData();
121   std::cout << "    Doing something with " << pImpl_->name << "\n";
122 }
123
124 void WidgetWithPimpl::process() {
125   pImpl_->data.push_back(1.0);
126   pImpl_->data.push_back(2.0);
127   pImpl_->processInternal();
128 }
129
130 void demonstrate() {
131   std::cout << "\n" << std::string(80, '=') << "\n";
132   std::cout << "SECTION 1: WHAT IS THE PIMPL IDIOM?\n";
133   std::cout << std::string(80, '=') << "\n\n";
134
135   std::cout << "PIMPL = Pointer to IMPLementation (also called Opaque
136   Pointer)\n\n";
137   std::cout << "CONCEPT:\n";
138   std::cout << " •  Public class holds ONLY a pointer to implementation\n";
139   std::cout << " •  Implementation struct defined in .cpp file\n";
140   std::cout << " •  All data members and private functions hidden in Impl\n"
141   ;
142   std::cout << " •  Forwards all operations to pImpl pointer\n\n";
143
144   std::cout << "EXAMPLE:\n";
145   WidgetWithPimpl widget("Sensor", 42);
146   widget.doSomething();
147   widget.process();
148
149   std::cout << "\n TRADITIONAL BENEFITS:\n";
150   std::cout << "    1. COMPILETIME FIREWALL\n";
151   std::cout << " •      Change private members without recompiling clients\n"
152   ";
153   std::cout << " •      Faster incremental builds\n";
154   std::cout << " •      Reduces header dependencies\n\n";
155
156   std::cout << "    2. BINARY COMPATIBILITY (ABI STABILITY)\n";

```

```
154     std::cout << " •      Can modify Impl without breaking ABI\n";
155     std::cout << " •      Important for shared libraries\n";
156     std::cout << " •      No need to relink client code\n\n";
157
158     std::cout << " 3. INFORMATION HIDING\n";
159     std::cout << " •      Private members truly hidden\n";
160     std::cout << " •      Can use incomplete types in Impl\n";
161     std::cout << " •      Better encapsulation\n\n";
162
163     std::cout << " 4. HEADER FILE SIMPLICITY\n";
164     std::cout << " •      Minimal includes in header\n";
165     std::cout << " •      Cleaner public interface\n";
166     std::cout << " •      Easier to read and maintain\n";
167 }
168
169 } // namespace what_is_pimpl
170
171 // =====
172 // SECTION 2: MODERN C++ IMPROVEMENTS - STILL RELEVANT?
173 // =====
174
175 namespace modern_cpp_pimpl {
176
177 // C++98: Raw pointer (manual memory management)
178 class WidgetCpp98 {
179 private:
180     struct Impl;
181     Impl* pImpl_; // Raw pointer
182
183 public:
184     WidgetCpp98();
185     ~WidgetCpp98();
186
187     // Rule of Three needed
188     WidgetCpp98(const WidgetCpp98&);
189     WidgetCpp98& operator=(const WidgetCpp98&);
190 };
191
192 // C++11: unique_ptr (automatic memory management)
193 class WidgetCpp11 {
194 private:
195     struct Impl;
196     std::unique_ptr<Impl> pImpl_; // Smart pointer
197
198 public:
199     WidgetCpp11();
200     ~WidgetCpp11(); // Still needed for incomplete type
201
202     // Rule of Five: Move-only by default (unique_ptr)
203     WidgetCpp11(WidgetCpp11&&) noexcept;
204     WidgetCpp11& operator=(WidgetCpp11&&) noexcept;
205
206     // Copy operations require custom implementation if needed
207     WidgetCpp11(const WidgetCpp11&) = delete;
```

```
208     WidgetCpp11& operator=(const WidgetCpp11&) = delete;
209 };
210
211 // C++11: shared_ptr (for copyable Pimpl)
212 class WidgetCpp11Shared {
213 private:
214     struct Impl;
215     std::shared_ptr<Impl> pImpl_; // Shared ownership
216
217 public:
218     WidgetCpp11Shared();
219     ~WidgetCpp11Shared() = default; // No explicit destructor needed!
220
221     // Copyable and movable by default
222     WidgetCpp11Shared(const WidgetCpp11Shared&) = default;
223     WidgetCpp11Shared& operator=(const WidgetCpp11Shared&) = default;
224     WidgetCpp11Shared(WidgetCpp11Shared&&) noexcept = default;
225     WidgetCpp11Shared& operator=(WidgetCpp11Shared&&) noexcept = default;
226 };
227
228 void demonstrate() {
229     std::cout << "\n" << std::string(80, '=') << "\n";
230     std::cout << "SECTION 2: MODERN C++ IMPROVEMENTS - STILL RELEVANT?\n";
231     std::cout << std::string(80, '=') << "\n\n";
232
233     std::cout << "EVOLUTION OF PIMPL:\n\n";
234
235     std::cout << "C++98: Raw pointer (manual memory management)\n";
236     std::cout << "    Manual new/delete\n";
237     std::cout << "    Must implement Rule of Three\n";
238     std::cout << "    Risk of memory leaks\n";
239     std::cout << "    Exception safety issues\n\n";
240
241     std::cout << "C++11: std::unique_ptr\n";
242     std::cout << "    Automatic memory management\n";
243     std::cout << "    Move-only by default\n";
244     std::cout << "    Exception safe\n";
245     std::cout << "    Still need destructor in .cpp (incomplete type)\n\n";
246
247     std::cout << "C++11: std::shared_ptr\n";
248     std::cout << "    Automatic memory management\n";
249     std::cout << "    Copyable by default\n";
250     std::cout << "    No destructor needed in .cpp!\n";
251     std::cout << "    Reference counting overhead\n";
252     std::cout << "    Two heap allocations (control block + Impl)\n\n";
253
254     std::cout << "IS PIMPL STILL RELEVANT IN MODERN C++?\n\n";
255
256     std::cout << "    YES, when you need:\n";
257     std::cout << "        1. BINARY STABILITY (shared libraries, plugins)\n";
258     std::cout << "            • Qt framework uses Pimpl extensively\n";
259     std::cout << "            • Windows API uses opaque handles (HWND, HANDLE)\n";
260     std::cout << "            • Linux kernel uses opaque pointers\n\n";
```

```
262     std::cout << " 2. COMPILATION FIREWALL (large projects)\n";
263     std::cout << " • Reduce compilation dependencies\n";
264     std::cout << " • Faster incremental builds\n";
265     std::cout << " • Important for multi-million line codebases\n\n";
266
267     std::cout << " 3. INTERFACE STABILITY (APIs)\n";
268     std::cout << " • Public interface never changes\n";
269     std::cout << " • Implementation can evolve freely\n";
270     std::cout << " • Versioning becomes easier\n\n";
271
272     std::cout << " LESS RELEVANT when:\n";
273     std::cout << " 1. Using header-only libraries (templates)\n";
274     std::cout << " 2. Internal classes (not exposed in API)\n";
275     std::cout << " 3. Performance-critical code (real-time systems)\n";
276     std::cout << " 4. Small classes with simple members\n";
277     std::cout << " 5. Using modules (C++20) - reduces header dependencies\n";
278
279 }
280 } // namespace modern_cpp_pimpl
281
282 // =====
283 // SECTION 3: PERFORMANCE PROBLEMS - THE REAL-TIME KILLER
284 // =====
285
286 namespace performance_problems {
287
288 // Simple data structure for testing
289 struct SensorData {
290     int id;
291     double value;
292     int64_t timestamp;
293
294     SensorData(int i = 0, double v = 0.0)
295         : id(i), value(v), timestamp(0) {}
296 };
297
298 // =====
299 // VERSION 1: NO PIMPL - Direct data access
300 // =====
301
302 class SensorNoPimpl {
303 private:
304     int id_;
305     double value_;
306     int64_t timestamp_;
307
308 public:
309     SensorNoPimpl(int id, double value)
310         : id_(id), value_(value), timestamp_(0) {}
311
312     void update(double new_value) {
313         value_ = new_value;
314         timestamp_++;
315     }
316 }
```

```
315     }
316
317     double getValue() const { return value_; }
318     int getId() const { return id_; }
319 };
320
321 // =====
322 // VERSION 2: WITH PIMPL - Pointer indirection
323 // =====
324
325 class SensorWithPimpl {
326 private:
327     struct Impl {
328         int id;
329         double value;
330         int64_t timestamp;
331
332         Impl(int i, double v) : id(i), value(v), timestamp(0) {}
333     };
334
335     std::unique_ptr<Impl> pImpl_;
336
337 public:
338     SensorWithPimpl(int id, double value)
339         : pImpl_(std::make_unique<Impl>(id, value)) {}
340
341     ~SensorWithPimpl() = default;
342
343     void update(double new_value) {
344         pImpl_->value = new_value;
345         pImpl_->timestamp++;
346     }
347
348     double getValue() const { return pImpl_->value; }
349     int getId() const { return pImpl_->id; }
350 };
351
352 void demonstrate() {
353     std::cout << "\n" << std::string(80, '=') << "\n";
354     std::cout << "SECTION 3: PERFORMANCE PROBLEMS - THE REAL-TIME KILLER\n";
355     std::cout << std::string(80, '=') << "\n\n";
356
357     std::cout << "PROBLEM 1: POINTER INDIRECTION OVERHEAD\n\n";
358
359     std::cout << "Without Pimpl:\n";
360     std::cout << "    sensor.getValue() → Direct access: sensor.value_\n";
361     std::cout << "    • ONE memory access\n";
362     std::cout << "    • CPU loads value directly from sensor object\n";
363     std::cout << "    • Predictable: O(1) memory access\n\n";
364
365     std::cout << "With Pimpl:\n";
366     std::cout << "    sensor.getValue() → sensor.pImpl_->value\n";
367     std::cout << "    • TWO memory accesses:\n";
368     std::cout << "        1. Load pImpl_ pointer from sensor object\n";
```

```

369     std::cout << "      2. Follow pointer to load value from Impl\n";
370     std::cout << " •  Unpredictable: pointer could point anywhere in heap\n\n"
371     ;
372
373     std::cout << "PROBLEM 2: CACHE LOCALITY DESTROYED\n\n";
374
375     std::cout << "Without Pimpl (Array of sensors):\n";
376     std::cout << "    Memory layout:  [Sensor1][Sensor2][Sensor3][Sensor4]\n";
377     std::cout << " •  Contiguous memory (cache-friendly)\n";
378     std::cout << " •  CPU prefetcher loads next sensors automatically\n";
379     std::cout << " •  All data in same cache line (64 bytes)\n\n";
380
381     std::cout << "With Pimpl (Array of sensors):\n";
382     std::cout << "    Memory layout:  [Sensor1->*]Impl1  [Sensor2->*]Impl2\n";
383     std::cout << "                      scattered          scattered\n";
384     std::cout << "                      in heap          in heap\n";
385     std::cout << " •  Impl objects scattered across heap (cache-hostile)\n";
386     std::cout << " •  Each access causes cache miss\n";
387     std::cout << " •  CPU prefetcher cannot predict pattern\n";
388     std::cout << " •  Every pointer dereference = potential cache miss\n\n";
389
390     std::cout << "PROBLEM 3: HEAP ALLOCATION OVERHEAD\n\n";
391
392     std::cout << "Without Pimpl:\n";
393     std::cout << "    SensorNoPimpl sensor(1, 3.14); // Stack allocation\n";
394     std::cout << " •  O(1) construction time\n";
395     std::cout << " •  No malloc/free overhead\n";
396     std::cout << " •  Deterministic\n\n";
397
398     std::cout << "With Pimpl:\n";
399     std::cout << "    SensorWithPimpl sensor(1, 3.14); // Heap allocation!\n";
400     std::cout << " •  Calls malloc() for Impl\n";
401     std::cout << " •  O(?) allocation time (non-deterministic)\n";
402     std::cout << " •  Heap fragmentation over time\n";
403     std::cout << " •  Destructor calls free() (non-deterministic)\n\n";
404
405     // Benchmark
406     constexpr int ITERATIONS = 1000000;
407
408     std::cout << "BENCHMARK: " << ITERATIONS << " operations\n\n";
409
410     // Test 1: No Pimpl
411     {
412         SensorNoPimpl sensor(1, 100.0);
413         auto start = std::chrono::high_resolution_clock::now();
414
415         for (int i = 0; i < ITERATIONS; ++i) {
416             sensor.update(i * 0.1);
417             [[maybe_unused]] volatile double val = sensor.getValue();
418         }
419
420         auto end = std::chrono::high_resolution_clock::now();
421         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
422             end - start);

```

```
421     std::cout << " WITHOUT Pimpl: " << std::setw(8) << duration.count()
422             << " s\n";
423 }
424
425 // Test 2: With Pimpl
426 {
427     SensorWithPimpl sensor(1, 100.0);
428     auto start = std::chrono::high_resolution_clock::now();
429
430     for (int i = 0; i < ITERATIONS; ++i) {
431         sensor.update(i * 0.1);
432         [[maybe_unused]] volatile double val = sensor.getValue();
433     }
434
435     auto end = std::chrono::high_resolution_clock::now();
436     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
437         end - start);
438
439     std::cout << " WITH Pimpl: " << std::setw(8) << duration.count()
440             << " s\n";
441 }
442
443 std::cout << "\n PIMPL OVERHEAD:\n";
444 std::cout << " • 50-200% slower (depends on cache behavior)\n";
445 std::cout << " • Non-deterministic timing variations\n";
446 std::cout << " • Extra heap allocation per object\n";
447 std::cout << " • Destroys cache locality\n";
448 }
449
450 // =====
451 // SECTION 4: WHY PIMPL IS BAD FOR REAL-TIME SYSTEMS
452 // =====
453
454 namespace realtime_problems {
455
456 void demonstrate() {
457     std::cout << "\n" << std::string(80, '=') << "\n";
458     std::cout << "SECTION 4: WHY PIMPL IS BAD FOR REAL-TIME SYSTEMS\n";
459     std::cout << std::string(80, '=') << "\n\n";
460
461     std::cout << "REAL-TIME SYSTEM REQUIREMENTS:\n";
462     std::cout << " 1. DETERMINISTIC TIMING - Predictable worst-case execution
463                 time (WCET)\n";
464     std::cout << " 2. BOUNDED LATENCY - Maximum response time guaranteed\n";
465     std::cout << " 3. NO HEAP ALLOCATION - At runtime (malloc/free non-
466                 deterministic)\n";
467     std::cout << " 4. CACHE PREDICTABILITY - Minimize cache misses\n";
468     std::cout << " 5. NO MEMORY JUMPS - Sequential memory access for
469                 prefetcher\n\n";
470
471     std::cout << "PIMPL VIOLATIONS:\n\n";
```

```

469
470     std::cout << "  VIOLATION 1: NON-DETERMINISTIC HEAP ALLOCATION\n";
471     std::cout << "    Problem:\n";
472     std::cout << "    •      Each Pimpl object requires malloc() for Impl\n";
473     std::cout << "    •      malloc() time depends on heap state (fragmentation)\n";
474     std::cout << "    •      WCET becomes unpredictable\n";
475     std::cout << "    Impact on Real-Time:\n";
476     std::cout << "    •      Cannot prove deadline guarantees\n";
477     std::cout << "    •      Violates MISRA C++ Rule 18-4-1 (no dynamic allocation\n";
478     std::cout << "        )\n";
479     std::cout << "    •      Violates ISO 26262 (automotive safety) guidelines\n";
480     std::cout << "    •      Unacceptable for hard real-time (ASIL-D)\n\n";
481
482     std::cout << "  VIOLATION 2: POINTER INDIRECTION = CACHE MISSES\n";
483     std::cout << "    Problem:\n";
484     std::cout << "    •      Every operation requires following pImpl_ pointer\n";
485     std::cout << "    •      Impl object allocated somewhere in heap\n";
486     std::cout << "    •      High probability of cache miss on every access\n";
487     std::cout << "    Impact on Real-Time:\n";
488     std::cout << "    •      Cache miss = 100+ cycles (vs 1 cycle for cache hit)\n";
489     std::cout << "    •      Unpredictable latency spikes\n";
490     std::cout << "    •      Worst-case timing analysis becomes pessimistic\n";
491     std::cout << "    •      May cause deadline misses in tight loops\n\n";
492
493     std::cout << "  VIOLATION 3: DESTROYED CACHE LOCALITY\n";
494     std::cout << "    Problem:\n";
495     std::cout << "    •      Array of Pimpl objects:\n";
496     std::cout << "    •      [Obj1 ptr] Impl1 (heap addr 0x1000)\n";
497     std::cout << "    •      [Obj2 ptr] Impl2 (heap addr 0x5000) ← 16KB away!\n";
498     std::cout << "    •      [Obj3 ptr] Impl3 (heap addr 0x2000)\n";
499     std::cout << "    •      Each Impl scattered randomly in heap\n";
500     std::cout << "    •      CPU prefetcher cannot help\n";
501     std::cout << "    Impact on Real-Time:\n";
502     std::cout << "    •      Sequential access becomes random access\n";
503     std::cout << "    •      10-100x slower than contiguous data\n";
504     std::cout << "    •      Unacceptable for sensor processing, control loops\n\n";
505
506     std::cout << "  VIOLATION 4: HEAP FRAGMENTATION OVER TIME\n";
507     std::cout << "    Problem:\n";
508     std::cout << "    •      Repeated allocation/deallocation of Impl objects\n";
509     std::cout << "    •      Heap becomes fragmented (Swiss cheese memory)\n";
510     std::cout << "    •      Allocation time increases over time\n";
511     std::cout << "    Impact on Real-Time:\n";
512     std::cout << "    •      System degrades over runtime\n";
513     std::cout << "    •      May fail after hours/days of operation\n";
514     std::cout << "    •      Cannot certify for long-running systems\n";
515     std::cout << "    •      Requires periodic restart (unacceptable)\n\n";
516
517     std::cout << "  VIOLATION 5: DOUBLE DEALLOCATION (shared_ptr Pimpl)\n";
518     std::cout << "    Problem:\n";
519     std::cout << "    •      shared_ptr has control block (heap allocation)\n";

```



```
564 // =====
565 // ALTERNATIVE 1: NO PIMPL - Expose implementation (if acceptable)
566 // =====
567
568 class DirectImplementation {
569 private:
570     int id_;
571     double value_;
572
573 public:
574     DirectImplementation(int id, double value) : id_(id), value_(value) {}
575
576     void process() { value_ *= 2.0; }
577     double getValue() const { return value_; }
578 };
579
580 // Pros: Fast, cache-friendly, deterministic
581 // Cons: Breaks ABI on changes, longer compile times
582
583 // =====
584 // ALTERNATIVE 2: FAST PIMPL - Pre-allocated storage
585 // =====
586
587 template<typename T, size_t Size, size_t Alignment = alignof(std::max_align_t)>
588 class FastPimpl {
589 private:
590     alignas(Alignment) std::byte storage_[Size];
591
592     T* ptr() { return reinterpret_cast<T*>(storage_); }
593     const T* ptr() const { return reinterpret_cast<const T*>(storage_); }
594
595 public:
596     template<typename... Args>
597     FastPimpl(Args&&... args) {
598         static_assert(sizeof(T) <= Size, "Storage too small");
599         static_assert(alignof(T) <= Alignment, "Alignment too small");
600         new (storage_) T(std::forward<Args>(args)...);
601     }
602
603     ~FastPimpl() { ptr() ~T(); }
604
605     // Delete copy/move (or implement if needed)
606     FastPimpl(const FastPimpl&) = delete;
607     FastPimpl& operator=(const FastPimpl&) = delete;
608
609     T* operator->() { return ptr(); }
610     const T* operator->() const { return ptr(); }
611
612     T& operator*() { return *ptr(); }
613     const T& operator*() const { return *ptr(); }
614 };
615
616 class SensorFastPimpl {
```

```
617 private:
618     struct Impl {
619         int id;
620         double value;
621         Impl(int i, double v) : id(i), value(v) {}
622         void process() { value *= 2.0; }
623     };
624
625     FastPimpl<Impl, 32> pImpl_; // Storage in object itself!
626
627 public:
628     SensorFastPimpl(int id, double value) : pImpl_(id, value) {}
629
630     void process() { pImpl_->process(); }
631     double getValue() const { return pImpl_->value; }
632 };
633
634 // Pros: NO heap allocation, cache-friendly, ABI stable (if size doesn't
635 // change)
636 // Cons: Must know maximum Impl size at compile time
637 // =====
638 // ALTERNATIVE 3: Type Erasure (std::function style)
639 // =====
640
641 class TypeErasedSensor {
642 private:
643     struct ConceptBase {
644         virtual ~ConceptBase() = default;
645         virtual void process() = 0;
646         virtual double getValue() const = 0;
647         virtual std::unique_ptr<ConceptBase> clone() const = 0;
648     };
649
650     template<typename T>
651     struct ConceptModel : ConceptBase {
652         T data;
653
654         ConceptModel(T d) : data(std::move(d)) {}
655
656         void process() override { data.process(); }
657         double getValue() const override { return data.getValue(); }
658         std::unique_ptr<ConceptBase> clone() const override {
659             return std::make_unique<ConceptModel<T>>(data);
660         }
661     };
662
663     std::unique_ptr<ConceptBase> pImpl_;
664
665 public:
666     template<typename T>
667     TypeErasedSensor(T sensor)
668         : pImpl_(std::make_unique<ConceptModel<T>>(std::move(sensor))) {}
669
```

```
670     void process() { pImpl_->process(); }
671     double getValue() const { return pImpl_->getValue(); }
672 };
673
674 // Pros: Flexible, any type works, no template in interface
675 // Cons: Still uses heap, virtual function overhead
676
677 // =====
678 // ALTERNATIVE 4: SBO (Small Buffer Optimization) Pimpl
679 // =====
680
681 template<size_t BufferSize = 64>
682 class SBOPimpl {
683 private:
684     alignas(std::max_align_t) std::byte buffer_[BufferSize];
685     void* heap_ptr_ = nullptr;
686
687     void* storage() { return heap_ptr_ ? heap_ptr_ : buffer_; }
688     const void* storage() const { return heap_ptr_ ? heap_ptr_ : buffer_; }
689
690 public:
691     template<typename T, typename... Args>
692     void emplace(Args&&... args) {
693         if (sizeof(T) <= BufferSize && alignof(T) <= alignof(std::max_align_t))
694             {
695                 // Small: use buffer (no heap)
696                 new (buffer_) T(std::forward<Args>(args)...);
697                 heap_ptr_ = nullptr;
698             } else {
699                 // Large: use heap
700                 heap_ptr_ = new T(std::forward<Args>(args)...);
701             }
702     }
703
704     template<typename T>
705     T* get() { return static_cast<T*>(storage()); }
706
707     template<typename T>
708     const T* get() const { return static_cast<const T*>(storage()); }
709
710     ~SBOPimpl() {
711         if (heap_ptr_) {
712             // Must know type to delete - requires type erasure
713         }
714     }
715
716 // Pros: Avoids heap for small objects
717 // Cons: Complex, requires type erasure for destruction
718
719 // =====
720 // ALTERNATIVE 5: Interface Segregation + Direct Storage
721 // =====
```

```

723 // Pure interface (header file)
724 class ISensor {
725 public:
726     virtual ~ISensor() = default;
727     virtual void process() = 0;
728     virtual double getValue() const = 0;
729 };
730
731 // Concrete implementation (cpp file)
732 class ConcreteSensor : public ISensor {
733 private:
734     int id_;
735     double value_;
736
737 public:
738     ConcreteSensor(int id, double value) : id_(id), value_(value) {}
739
740     void process() override { value_ *= 2.0; }
741     double getValue() const override { return value_; }
742 };
743
744 // Factory function (cpp file)
745 std::unique_ptr<ISensor> createSensor(int id, double value) {
746     return std::make_unique<ConcreteSensor>(id, value);
747 }
748
749 // Pros: Clean interface, implementation hidden
750 // Cons: Still uses heap, virtual function overhead
751
752 void demonstrate() {
753     std::cout << "\n" << std::string(80, '=') << "\n";
754     std::cout << "SECTION 5: ALTERNATIVES AND MITIGATIONS\n";
755     std::cout << std::string(80, '=') << "\n\n";
756
757     std::cout << "ALTERNATIVE 1: NO PIMPL (Direct Implementation)\n";
758     std::cout << "    BEST for performance and real-time systems\n";
759     std::cout << "    Zero overhead, cache-friendly\n";
760     std::cout << "    Breaks ABI on changes\n";
761     std::cout << "    Longer compile times\n";
762     std::cout << "    USE WHEN: Performance critical, internal classes\n\n";
763
764     DirectImplementation direct(1, 100.0);
765     direct.process();
766     std::cout << "    Example: Direct value = " << direct.getValue() << "\n\n";
767
768     std::cout << "ALTERNATIVE 2: FAST PIMPL (Pre-allocated storage)\n";
769     std::cout << "    NO heap allocation (storage in object)\n";
770     std::cout << "    Cache-friendly (contiguous memory)\n";
771     std::cout << "    ABI stable (if size doesn't change)\n";
772     std::cout << "    Must know maximum Impl size\n";
773     std::cout << "    USE WHEN: Need ABI stability + performance\n\n";
774
775     SensorFastPimpl fast(1, 100.0);
776     fast.process();

```

```
777 std::cout << " Example: FastPimpl value = " << fast.getValue() << "\n\n";
778
779 std::cout << "ALTERNATIVE 3: TYPE ERASURE\n";
780 std::cout << " Flexible (any type works)\n";
781 std::cout << " Clean generic interface\n";
782 std::cout << " Still uses heap\n";
783 std::cout << " Virtual function overhead\n";
784 std::cout << " USE WHEN: Need runtime polymorphism\n\n";
785
786 std::cout << "ALTERNATIVE 4: SBO (Small Buffer Optimization)\n";
787 std::cout << " Avoids heap for small objects\n";
788 std::cout << " Fallback to heap for large objects\n";
789 std::cout << " Complex implementation\n";
790 std::cout << " Similar to std::string SSO\n";
791 std::cout << " USE WHEN: Mix of small and large objects\n\n";
792
793 std::cout << "ALTERNATIVE 5: INTERFACE + FACTORY\n";
794 std::cout << " Clean separation of interface/implementation\n";
795 std::cout << " Implementation hidden in .cpp\n";
796 std::cout << " Still uses heap\n";
797 std::cout << " Virtual function overhead\n";
798 std::cout << " USE WHEN: Need polymorphism + information hiding\n\n";
799
800 std::cout << "MITIGATION STRATEGIES FOR REAL-TIME:\n\n";
801
802 std::cout << "1. INITIALIZATION PHASE PATTERN\n";
803 std::cout << " • Allocate ALL Pimpl objects during initialization\n";
804 std::cout << " • Use object pools (pre-allocated)\n";
805 std::cout << " • Never allocate/deallocate during runtime\n";
806 std::cout << " • Acceptable: init phase not time-critical\n\n";
807
808 std::cout << "2. CUSTOM POOL ALLOCATOR\n";
809 std::cout << " • Pre-allocate pool of Impl objects\n";
810 std::cout << " • Custom allocator for unique_ptr<Impl>\n";
811 std::cout << " • Bounded, deterministic allocation\n";
812 std::cout << " • Improves cache locality (pool in contiguous memory)\n\n";
813
814 std::cout << "3. PLACEMENT NEW IN FIXED BUFFER\n";
815 std::cout << " • Static buffer for all Impl objects\n";
816 std::cout << " • Placement new to construct in buffer\n";
817 std::cout << " • Zero heap allocation\n";
818 std::cout << " • Requires careful lifetime management\n\n";
819
820 std::cout << "4. C++20 MODULES\n";
821 std::cout << " • Reduce need for Pimpl (compilation firewall)\n";
822 std::cout << " • Faster builds without Pimpl overhead\n";
823 std::cout << " • Still evolving, limited tooling support\n\n";
824
825 std::cout << "RECOMMENDATION FOR REAL-TIME SYSTEMS:\n\n";
826 std::cout << " FIRST CHOICE: No Pimpl (Direct implementation)\n";
827 std::cout << " • Best performance\n";
828 std::cout << " • Deterministic\n";
829 std::cout << " • Accept longer compile times as trade-off\n\n";
```

```

830
831     std::cout << "      SECOND CHOICE: FastPimpl (in-object storage)\n";
832     std::cout << " •      Good compromise\n";
833     std::cout << " •      No heap, ABI stable\n";
834     std::cout << " •      Requires careful sizing\n\n";
835
836     std::cout << "      THIRD CHOICE: Pimpl with pool allocator\n";
837     std::cout << " •      Acceptable with pre-allocation\n";
838     std::cout << " •      Only during initialization phase\n";
839     std::cout << " •      Requires justification for certification\n\n";
840
841     std::cout << "      NEVER: Traditional heap-based Pimpl at runtime\n";
842     std::cout << " •      Non-deterministic\n";
843     std::cout << " •      Violates real-time standards\n";
844     std::cout << " •      Unacceptable for safety-critical systems\n";
845 }
846
847 } // namespace alternatives
848
849 // =====
850 // SECTION 6: DECISION GUIDE
851 // =====
852
853 namespace decision_guide {
854
855 void demonstrate() {
856     std::cout << "\n" << std::string(80, '=') << "\n";
857     std::cout << "SECTION 6: WHEN TO USE PIMPL - DECISION GUIDE\n";
858     std::cout << std::string(80, '=') << "\n\n";
859
860     std::cout << "DECISION TREE:\n\n";
861
862     std::cout << "  Q1: Is this a real-time or safety-critical system?\n";
863     std::cout << "\n";
864     std::cout << "  YES → AVOID PIMPL (use direct implementation or FastPimpl
865     )\n";
866     std::cout << "          Reason: Non-deterministic, cache-hostile, heap
867     allocation\n";
868     std::cout << "\n";
869     std::cout << "  NO → Continue to Q2\n\n";
870
871     std::cout << "  Q2: Is this a public API/library interface?\n";
872     std::cout << "\n";
873     std::cout << "  YES → Continue to Q3\n";
874     std::cout << "\n";
875     std::cout << "  NO → DON'T USE PIMPL (internal class - direct impl)\n\n";
876
877     std::cout << "  Q3: Do you need ABI (binary) stability?\n";
878     std::cout << "\n";
879     std::cout << "  YES → Continue to Q4\n";
880     std::cout << "\n";
881     std::cout << "  NO → DON'T USE PIMPL (recompilation acceptable)\n\n";
882
883     std::cout << "  Q4: Is the class performance-critical (hot path)?\n";

```

```

882     std::cout << "\n";
883     std::cout << " YES → RECONSIDER Pimpl (profile first, consider
884         FastPimpl)\n";
885     std::cout << "\n";
886     std::cout << " NO → Continue to Q5\n\n";
887
887     std::cout << " Q5: Is the implementation complex/large?\n";
888     std::cout << "\n";
889     std::cout << " YES → USE PIMPL (good candidate)\n";
890     std::cout << "\n";
891     std::cout << " NO → Probably not worth it (overhead > benefit)\n\n";
892
893     std::cout << std::string(80, '-') << "\n\n";
894
895     std::cout << "COMPARISON TABLE:\n\n";
896     std::cout << "                                     \n";
897     std::cout << " CRITERION          NO PIMPL  PIMPL  FAST PIMPL  REAL-
898         TIME OK? \n";
898     std::cout << "                                     \n";
899     std::cout << "   Performance      BEST    POOR    GOOD    /   /
900         \n";
900     std::cout << "   Cache Locality   BEST    WORST   BEST    /   /
901         \n";
901     std::cout << "   Heap Allocation  NO     YES    NO     /   /
902         \n";
902     std::cout << "   ABI Stability   NO     YES    PARTIAL  N/A
903         \n";
903     std::cout << "   Compile Time     SLOW   FAST    FAST    N/A
904         \n";
904     std::cout << "   Information Hiding NO     YES    YES    N/A
905         \n";
905     std::cout << "   Complexity       LOW    MEDIUM  HIGH   N/A
906         \n";
906     std::cout << "   Memory Overhead 0 bytes  8 bytes Impl size  N/A
907         \n";
907     std::cout << "                                     \n\n";
908
909     std::cout << "USE PIMPL WHEN:\n\n";
910     std::cout << "   Shipping shared libraries (ABI stability critical)\n";
911     std::cout << "   Examples: Qt, wxWidgets, system libraries\n\n";
912
913     std::cout << "   Plugin architectures (binary compatibility)\n";
914     std::cout << "   Examples: Audio plugins, game engines\n\n";
915
916     std::cout << "   Large complex implementations (compilation firewall)\n";
917     std::cout << "   Examples: GUI frameworks, network libraries\n\n";
918
919     std::cout << "   Platform-specific code (hide platform details)\n";
920     std::cout << "   Examples: Cross-platform libraries\n\n";
921
922     std::cout << "AVOID PIMPL WHEN:\n\n";
923     std::cout << "   Real-time systems (non-deterministic)\n";
924     std::cout << "   Examples: Control systems, DSP, robotics\n\n";
925

```

```
926     std::cout << "    Performance-critical code (cache-hostile)\n";
927     std::cout << "        Examples: Game engines, HFT, rendering\n\n";
928
929     std::cout << "    Safety-critical systems (heap allocation)\n";
930     std::cout << "        Examples: Automotive (ISO 26262), avionics (DO-178C)\n\n";
931
932     std::cout << "    Small simple classes (overhead > benefit)\n";
933     std::cout << "        Examples: POD types, value types\n\n";
934
935     std::cout << "    Internal implementation classes (not exposed)\n";
936     std::cout << "        Examples: Private helpers, implementation details\n\n";
937
938     std::cout << "    Header-only libraries (templates)\n";
939     std::cout << "        Examples: Template libraries, generic code\n\n";
940
941     std::cout << "    Using C++20 modules (reduces need)\n";
942     std::cout << "        Examples: Modern codebases with module support\n\n";
943
944     std::cout << "FAMOUS LIBRARIES USING PIMPL:\n\n";
945     std::cout << "    • Qt Framework - Extensive use for ABI stability\n";
946     std::cout << "    • Boost.Aasio - Some classes use Pimpl\n";
947     std::cout << "    • Windows API - HWND, HANDLE (opaque pointers)\n";
948     std::cout << "    • OpenSSL - Internal structures hidden\n";
949     std::cout << "    • wxWidgets - Cross-platform abstraction\n\n";
950
951     std::cout << "GOLDEN RULES:\n\n";
952     std::cout << "    1. Pimpl is a DESIGN PATTERN, not a default choice\n";
953     std::cout << "    2. Only use when benefits outweigh performance cost\n";
954     std::cout << "    3. Profile before and after - measure the impact\n";
955     std::cout << "    4. Consider FastPimpl for real-time constraints\n";
956     std::cout << "    5. Never use traditional Pimpl in real-time code\n";
957     std::cout << "    6. C++20 modules may reduce need for Pimpl\n";
958 }
959
960 } // namespace decision_guide
961
962 // =====
963 // MAIN
964 // =====
965
966 int main() {
967     std::cout << "\n" << std::string(80, '=') << "\n";
968     std::cout << "PIMPL IDIOM IN MODERN C++ - STILL RELEVANT?\n";
969     std::cout << "Performance Impact on Real-Time Systems\n";
970     std::cout << std::string(80, '=') << "\n";
971
972     what_is_pimpl::demonstrate();
973     modern_cpp_pimpl::demonstrate();
974     performance_problems::demonstrate();
975     realtime_problems::demonstrate();
976     alternatives::demonstrate();
977     decision_guide::demonstrate();
978 }
```

```
979     std::cout << "\n" << std::string(80, '=') << "\n";
980     std::cout << "FINAL VERDICT\n";
981     std::cout << std::string(80, '=') << "\n\n";
982
983     std::cout << "IS PIMPL STILL RELEVANT IN MODERN C++?\n\n";
984
985     std::cout << "  YES, for:\n";
986     std::cout << "  •  Shared libraries requiring ABI stability\n";
987     std::cout << "  •  Large projects needing compilation firewalls\n";
988     std::cout << "  •  Public APIs with evolving implementation\n";
989     std::cout << "  •  Modern C++ makes it easier (unique_ptr, shared_ptr)\n\n";
990     ";
991
992     std::cout << "  NO (AVOID), for:\n";
993     std::cout << "  •  Real-time systems - NON-DETERMINISTIC\n";
994     std::cout << "  •  Safety-critical systems - HEAP ALLOCATION\n";
995     std::cout << "  •  Performance-critical code - CACHE-HOSTILE\n";
996     std::cout << "  •  Simple classes - OVERHEAD > BENEFIT\n\n";
997
998     std::cout << "WHY PIMPL IS BAD FOR REAL-TIME:\n\n";
999     std::cout << "  1. POINTER INDIRECTNESS - Extra memory access (cache miss)\n";
1000    std::cout << "  2. DESTROYED CACHE LOCALITY - Scattered Impl objects in
1001       heap\n";
1002    std::cout << "  3. HEAP ALLOCATION - Non-deterministic malloc/free\n";
1003    std::cout << "  4. MEMORY JUMPS - CPU prefetcher cannot help\n";
1004    std::cout << "  5. FRAGMENTATION - System degrades over time\n\n";
1005
1006    std::cout << "MITIGATION OPTIONS:\n\n";
1007    std::cout << "  FastPimpl (in-object storage) - Zero heap, cache-
1008       friendly\n";
1009    std::cout << "  Pre-allocated pools - Bounded allocation during init\n";
1010    std::cout << "  No Pimpl - Accept longer compile times for performance\n";
1011    std::cout << "\n";
1012
1013    std::cout << "BOTTOM LINE:\n\n";
1014    std::cout << "  \"Pimpl is still relevant for APIs and libraries,\n";
1015    std::cout << "  but should be AVOIDED in real-time and performance-
1016       critical code.\n";
1017    std::cout << "  The pointer indirection and heap allocation make it\n";
1018    std::cout << "  fundamentally incompatible with deterministic real-time
1019       systems.\n\n";
1020
1021    std::cout << std::string(80, '=') << "\n\n";
1022
1023    return 0;
1024 }
```

## 53 Source Code: ProtobufExample.cpp

File: src/ProtobufExample.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // GOOGLE PROTOCOL BUFFERS (PROTOBUF) EXAMPLE
3 // =====
4 // This example demonstrates Protocol Buffers usage in modern C++.
5 //
6 // TOPICS COVERED:
7 // 1. Creating and populating protobuf messages
8 // 2. Serialization to binary format
9 // 3. Deserialization from binary format
10 // 4. JSON conversion (protobuf v3 feature)
11 // 5. Nested messages and repeated fields
12 // 6. Enums and oneof fields
13 // 7. Performance and memory efficiency
14 // 8. Best practices for embedded systems
15 //
16 // WHAT IS PROTOCOL BUFFERS?
17 // - Language-neutral, platform-neutral serialization format
18 // - Smaller, faster, and simpler than XML/JSON
19 // - Generates code for multiple languages (C++, Python, Java, Go, etc.)
20 // - Backward/forward compatible with schema evolution
21 // - Used by Google, Netflix, Uber, and many others
22 //
23 // WHY USE PROTOBUF?
24 // Efficient: 3-10x smaller than XML, 20-100x faster
25 // Type-safe: Strong typing with generated code
26 // Versioned: Schema evolution with backward compatibility
27 // Cross-language: Same .proto file works for multiple languages
28 // Validated: Automatic validation of message structure
29 //
30 // BUILD REQUIREMENTS:
31 // - Google Protocol Buffers library (libprotobuf-dev)
32 // - protoc compiler to generate C++ code from .proto files
33 //
34 // =====
35
36 #include <iostream>
37 #include <fstream>
38 #include <memory>
39 #include <chrono>
40 #include <iomanip>
41
42 // Include generated protobuf headers
43 // These are generated by protoc from sensor_data.proto
44 // #include "sensor_data.pb.h"
45
46 // Include generated protobuf headers
47 // These are generated by protoc from sensor_data.proto
48 #include "sensor_data.pb.h"
49
```

```
50 // =====
51 // EXAMPLE 1: CREATING AND POPULATING MESSAGES
52 // =====
53
54 void example_create_sensor_reading() {
55     std::cout << "== Example 1: Creating Sensor Reading ==\n";
56
57     // Create a temperature sensor reading
58     sensors::SensorReading reading;
59
60     // Set basic fields
61     reading.set_type(sensors::TEMPERATURE);
62     reading.set_device_id("sensor_001");
63
64     // Set timestamp (current time)
65     auto* timestamp = reading.mutable_timestamp();
66     auto now = std::chrono::system_clock::now();
67     auto epoch = now.time_since_epoch();
68     timestamp->set_seconds(std::chrono::duration_cast<std::chrono::seconds>(
69         epoch).count());
70     timestamp->set_nanoseconds(0);
71
72     // Set temperature measurement (using oneof field)
73     reading.set_temperature_celsius(23.5f);
74
75     // Add metadata
76     (*reading.mutable_metadata())["location"] = "office";
77     (*reading.mutable_metadata())["calibrated"] = "true";
78
79     std::cout << "Created sensor reading:\n";
80     std::cout << "  Device: " << reading.device_id() << "\n";
81     std::cout << "  Type: " << reading.type() << "\n";
82     std::cout << "  Temperature: " << reading.temperature_celsius() << "°C\n";
83     std::cout << "  Metadata:\n";
84     for (const auto& [key, value] : reading.metadata()) {
85         std::cout << "    " << key << " = " << value << "\n";
86     }
87
88     std::cout << "\nKey features demonstrated:\n";
89     std::cout << "  Simple field setters (set_xxx)\n";
90     std::cout << "  Nested message creation (mutable_xxx)\n";
91     std::cout << "  Oneof field (only one measurement type at a time)\n";
92     std::cout << "  Map fields for flexible metadata\n\n";
93 }
94
95 // =====
96 // EXAMPLE 2: BINARY SERIALIZATION
97 // =====
98
99 void example_binary_serialization() {
100     std::cout << "== Example 2: Binary Serialization ==\n";
101
102     sensors::SensorReading reading;
103     reading.set_type(sensors::TEMPERATURE);
```

```
103     reading.set_device_id("sensor_001");
104     reading.set_temperature_celsius(23.5f);
105
106     // Method 1: Serialize to string
107     std::string serialized;
108     if (!reading.SerializeToString(&serialized)) {
109         std::cerr << "Failed to serialize!\n";
110         return;
111     }
112
113     std::cout << "Serialized size: " << serialized.size() << " bytes\n";
114     std::cout << "Binary data (hex): ";
115     for (size_t i = 0; i < std::min(serialized.size(), size_t(20)); ++i) {
116         printf("%02x ", (unsigned char)serialized[i]);
117     }
118     if (serialized.size() > 20) std::cout << "...";
119     std::cout << "\n";
120
121     // Method 2: Serialize to file
122     std::ofstream output("sensor_data.bin", std::ios::binary);
123     if (!reading.SerializeToOstream(&output)) {
124         std::cerr << "Failed to write to file!\n";
125         return;
126     }
127     output.close();
128     std::cout << " Written to sensor_data.bin\n";
129
130     // Method 3: Serialize to byte array (for embedded systems)
131     size_t size = reading.ByteSizeLong();
132     std::vector<uint8_t> buffer(size);
133     if (!reading.SerializeToArray(buffer.data(), size)) {
134         std::cerr << "Failed to serialize to array!\n";
135         return;
136     }
137     std::cout << " Serialized to byte array (" << size << " bytes)\n";
138
139     std::cout << "\nSerialization methods:\n";
140     std::cout << " 1. SerializeToString() - for std::string storage\n";
141     std::cout << " 2. SerializeToOstream() - for file I/O\n";
142     std::cout << " 3. SerializeToArray() - for fixed buffers (embedded)\n";
143     std::cout << " 4. ByteSizeLong() - get serialized size before writing\n\n";
144 }
145
146 // =====
147 // EXAMPLE 3: BINARY DESERIALIZATION
148 // =====
149
150 void example_binary_deserialization() {
151     std::cout << "== Example 3: Binary Deserialization ==\n";
152
153     // First, create and serialize a message
154     sensors::SensorReading original;
155     original.set_type(sensors::HUMIDITY);
```

```
156     original.set_device_id("sensor_002");
157     original.set_humidity_percent(65.3f);
158
159     std::string serialized_data;
160     original.SerializeToString(&serialized_data);
161     std::cout << "Original message serialized (" << serialized_data.size() <<
162         " bytes)\n";
163
164     // Method 1: Deserialize from string
165     sensors::SensorReading reading;
166     if (!reading.ParseFromString(serialized_data)) {
167         std::cerr << "Failed to parse!\n";
168         return;
169     }
170     std::cout << " Deserialized from string:\n";
171     std::cout << " Device: " << reading.device_id() << "\n";
172     std::cout << " Humidity: " << reading.humidity_percent() << "%\n";
173
174     // Method 2: Deserialize from file
175     std::ifstream input("sensor_data.bin", std::ios::binary);
176     sensors::SensorReading reading2;
177     if (input.good() && reading2.ParseFromIstream(&input)) {
178         std::cout << " Deserialized from file:\n";
179         std::cout << " Device: " << reading2.device_id() << "\n";
180         std::cout << " Temperature: " << reading2.temperature_celsius() << " °C\n";
181     }
182     input.close();
183
184     // Method 3: Deserialize from byte array
185     std::vector<uint8_t> buffer(serialized_data.begin(), serialized_data.end());
186     sensors::SensorReading reading3;
187     if (!reading3.ParseFromArray(buffer.data(), buffer.size())) {
188         std::cerr << "Failed to parse from array!\n";
189         return;
190     }
191     std::cout << " Deserialized from byte array: " << reading3.device_id() <<
192         "\n";
193
194     // Demonstrate field checking
195     if (reading3.has_humidity_percent()) {
196         std::cout << " Has humidity field: " << reading3.humidity_percent()
197             << "%\n";
198     }
199
200     std::cout << "\nDeserialization methods:\n";
201     std::cout << " 1. ParseFromString() - from std::string\n";
202     std::cout << " 2. ParseFromIstream() - from file\n";
203     std::cout << " 3. ParseFromArray() - from buffer\n";
204     std::cout << " 4. has_xxx() - check if optional field is set\n\n";
205
206 // =====
```

```
205 // EXAMPLE 4: ADVANCED FILE SERIALIZATION
206 // =====
207
208 void example_file_serialization() {
209     std::cout << "==== Example 4: Advanced File Serialization ===\n";
210
211     // Example 4a: Writing multiple messages to a file with length prefixes
212     std::cout << "\n[4a] Writing multiple sensor readings to file:\n";
213
214     const char* multi_file = "sensor_readings_multi.bin";
215     std::ofstream output(multi_file, std::ios::binary);
216
217     if (!output) {
218         std::cerr << "Failed to open file for writing!\n";
219         return;
220     }
221
222     // Create multiple sensor readings
223     std::vector<sensors::SensorReading> readings;
224
225     // Reading 1: Temperature
226     sensors::SensorReading temp_reading;
227     temp_reading.set_type(sensors::TEMPERATURE);
228     temp_reading.set_device_id("sensor_001");
229     temp_reading.set_temperature_celsius(22.5f);
230     readings.push_back(temp_reading);
231
232     // Reading 2: Humidity
233     sensors::SensorReading humidity_reading;
234     humidity_reading.set_type(sensors::HUMIDITY);
235     humidity_reading.set_device_id("sensor_002");
236     humidity_reading.set_humidity_percent(67.8f);
237     readings.push_back(humidity_reading);
238
239     // Reading 3: Pressure
240     sensors::SensorReading pressure_reading;
241     pressure_reading.set_type(sensors::PRESSURE);
242     pressure_reading.set_device_id("sensor_003");
243     pressure_reading.set_pressure_hpa(1013.25f);
244     readings.push_back(pressure_reading);
245
246     // Write each message with length prefix (for proper deserialization)
247     for (const auto& reading : readings) {
248         // Serialize to string first
249         std::string serialized;
250         if (!reading.SerializeToString(&serialized)) {
251             std::cerr << "Failed to serialize reading!\n";
252             continue;
253         }
254
255         // Write length prefix (4 bytes)
256         uint32_t size = serialized.size();
257         output.write(reinterpret_cast<const char*>(&size), sizeof(size));
```

```
259     // Write the actual message
260     output.write(serialized.data(), serialized.size());
261
262     std::cout << "    Written " << reading.device_id()
263             << " (" << size << " bytes)\n";
264 }
265
266 output.close();
267 std::cout << " Successfully wrote " << readings.size()
268             << " readings to " << multi_file << "\n";
269
270 // Example 4b: Reading multiple messages from file
271 std::cout << "\n[4b] Reading multiple messages from file:\n";
272
273 std::ifstream input(multi_file, std::ios::binary);
274 if (!input) {
275     std::cerr << "Failed to open file for reading!\n";
276     return;
277 }
278
279 int count = 0;
280 while (input.good()) {
281     // Read length prefix
282     uint32_t size;
283     input.read(reinterpret_cast<char*>(&size), sizeof(size));
284
285     if (input.eof()) break;
286     if (!input.good()) {
287         std::cerr << "Error reading length prefix!\n";
288         break;
289     }
290
291     // Read the message data
292     std::vector<char> buffer(size);
293     input.read(buffer.data(), size);
294
295     if (!input.good() && !input.eof()) {
296         std::cerr << "Error reading message data!\n";
297         break;
298     }
299
300     // Parse the message
301     sensors::SensorReading reading;
302     if (!reading.ParseFromArray(buffer.data(), size)) {
303         std::cerr << "Failed to parse message!\n";
304         continue;
305     }
306
307     count++;
308     std::cout << "    Reading #" << count << ":" "
309             << reading.device_id() << " - ";
310
311     // Display based on type
312     if (reading.has_temperature_celsius()) {
```

```

313         std::cout << reading.temperature_celsius() << "°C";
314     } else if (reading.has_humidity_percent()) {
315         std::cout << reading.humidity_percent() << "%";
316     } else if (reading.has_pressure_hpa()) {
317         std::cout << reading.pressure_hpa() << " hPa";
318     }
319     std::cout << "\n";
320 }
321
322 input.close();
323 std::cout << " Successfully read " << count << " readings from file\n";
324
325 // Example 4c: Using delimited I/O (WriteDelimitedToOstream)
326 std::cout << "\n[4c] Using Google's delimited I/O utilities:\n";
327 std::cout << R"code(
328 // For writing multiple messages (requires google/protobuf/io/coded_stream.h):
329 #include <google/protobuf/io/coded_stream.h>
330 #include <google/protobuf/io/zero_copy_stream_impl.h>
331
332 std::ofstream output("data.bin", std::ios::binary);
333 google::protobuf::io::OstreamOutputStream ostream(&output);
334 google::protobuf::io::CodedOutputStream coded_output(&ostream);
335
336 // Write each message with length prefix
337 for (const auto& reading : readings) {
338     coded_output.WriteVarint32(reading.ByteSizeLong());
339     reading.SerializeToCodedStream(&coded_output);
340 }
341
342 // For reading:
343 std::ifstream input("data.bin", std::ios::binary);
344 google::protobuf::io::IstreamInputStream istream(&input);
345 google::protobuf::io::CodedInputStream coded_input(&istream);
346
347 uint32_t size;
348 while (coded_input.ReadVarint32(&size)) {
349     auto limit = coded_input.PushLimit(size);
350     sensors::SensorReading reading;
351     reading.ParseFromCodedStream(&coded_input);
352     coded_input.PopLimit(limit);
353 }
354 )code" << "\n";
355
356 std::cout << "File I/O Best Practices:\n";
357 std::cout << " 1. Always write length prefix for multiple messages\n";
358 std::cout << " 2. Use binary mode (std::ios::binary) for files\n";
359 std::cout << " 3. Check file operations (input.good(), output.good())\n";
360 std::cout << " 4. Use CodedStream for efficient varint encoding\n";
361 std::cout << " 5. Handle EOF and errors gracefully\n";
362 std::cout << " 6. Close files explicitly or use RAII\n\n";
363 }
364
365 // =====
366 // EXAMPLE 5: FILE FORMAT STRATEGIES

```

```
367 // =====
368
369 void example_file_formats() {
370     std::cout << "==== Example 5: File Format Strategies ===\n\n";
371
372     std::cout << "Strategy 1: Single Message per File\n";
373     std::cout << "    Use case: Configuration files, snapshots\n";
374     std::cout << "    Simple: just SerializeToOstream() / ParseFromIstream()\n";
375     std::cout << "    ";
376     std::cout << "    Easy to inspect and replace\n";
377     std::cout << "    Many files for multiple messages\n";
378     std::cout << R"code(
379 sensors::SensorReading reading;
380 // ... populate reading ...
381
382     std::ofstream out("config.bin", std::ios::binary);
383     reading.SerializeToOstream(&out);
384     out.close();
385
386     std::ifstream in("config.bin", std::ios::binary);
387     reading.ParseFromIstream(&in);
388 )code" << "\n\n";
389
390     std::cout << "Strategy 2: Multiple Messages with Length Prefixes\n";
391     std::cout << "    Use case: Log files, sensor data streams\n";
392     std::cout << "    Efficient for sequential access\n";
393     std::cout << "    Single file for all data\n";
394     std::cout << "    No random access\n";
395     std::cout << "    See Example 4b above for implementation\n\n";
396
397     std::cout << "Strategy 3: Container Message (Repeated Fields)\n";
398     std::cout << "    Use case: Batched data, complete datasets\n";
399     std::cout << "    Single SerializeToOstream() call\n";
400     std::cout << "    Entire dataset in memory\n";
401     std::cout << "    Must load everything at once\n";
402     std::cout << R"code(
403 // Define in .proto:
404 // message SensorDataBatch {
405 //     repeated SensorReading readings = 1;
406 // }
407
408     sensors::SensorDataBatch batch;
409     batch.add_readings()->CopyFrom(reading1);
410     batch.add_readings()->CopyFrom(reading2);
411     batch.add_readings()->CopyFrom(reading3);
412
413     std::ofstream out("batch.bin", std::ios::binary);
414     batch.SerializeToOstream(&out);
415 )code" << "\n\n";
416
417     std::cout << "Strategy 4: Text Format (for debugging)\n";
418     std::cout << "    Use case: Debug logs, human inspection\n";
419     std::cout << "    Human-readable\n";
420     std::cout << "    Easy to debug\n";
```

```
420     std::cout << "    Much larger file size\n";
421     std::cout << "    Slower parsing\n";
422     std::cout << R"code(
423 #include <google/protobuf/text_format.h>
424
425     sensors::SensorReading reading;
426     // ... populate ...
427
428     std::ofstream out("data.txt");
429     std::string text;
430     google::protobuf::TextFormat::PrintToString(reading, &text);
431     out << text;
432
433     // Read back:
434     std::ifstream in("data.txt");
435     std::string content((std::istreambuf_iterator<char>(in)),
436                         std::istreambuf_iterator<char>());
437     google::protobuf::TextFormat::ParseFromString(content, &reading);
438 )code" << "\n\n";
439
440     std::cout << "Recommendation by use case:\n";
441     std::cout << " • Configuration files → Strategy 1 (single message)\n";
442     std::cout << " • Sensor data logs → Strategy 2 (length prefixes)\n";
443     std::cout << " • Database dumps → Strategy 3 (container message)\n";
444     std::cout << " • Debug output → Strategy 4 (text format)\n";
445     std::cout << " • Embedded logging → Strategy 2 (append to file)\n\n";
446 }
447
448 // =====
449 // EXAMPLE 6: JSON CONVERSION
450 // =====
451
452 void example_json_conversion() {
453     std::cout << "==== Example 6: JSON Conversion (Protobuf v3) ===\n";
454
455     std::cout << R"code(
456 #include <google/protobuf/util/json_util.h>
457
458     sensors::SensorReading reading;
459     reading.set_type(sensors::TEMPERATURE);
460     reading.set_device_id("sensor_001");
461     reading.set_temperature_celsius(23.5f);
462
463     // Convert to JSON
464     std::string json_output;
465     google::protobuf::util::JsonPrintOptions options;
466     options.add_whitespace = true; // Pretty print
467     options.always_print_primitive_fields = true; // Show all fields
468
469     auto status = google::protobuf::util::MessageToJsonString(
470         reading, &json_output, options
471     );
472
473     if (status.ok()) {
```

```
474     std::cout << "JSON output:\n" << json_output << "\n";
475 } else {
476     std::cerr << "JSON conversion failed: " << status.message() << "\n";
477 }
478
479 // Convert from JSON
480 std::string json_input = R"({
481     "type": "TEMPERATURE",
482     "device_id": "sensor_002",
483     "temperature_celsius": 25.3
484 })";
485
486 sensors::SensorReading parsed_reading;
487 status = google::protobuf::util::JsonStringToMessage(json_input, &
488     parsed_reading);
489
490 if (status.ok()) {
491     std::cout << "Parsed from JSON: " << parsed_reading.device_id() << "\n";
492 }
493
494 std::cout << "\nJSON features:\n";
495 std::cout << "    Human-readable debugging\n";
496 std::cout << "    REST API compatibility\n";
497 std::cout << "    Configuration files\n";
498 std::cout << "    Larger than binary (use binary for production)\n\n";
499 }
500
501 // =====
502 // EXAMPLE 7: REPEATED FIELDS AND BATCHING
503 // =====
504
505 void example_repeated_fields() {
506     std::cout << "==== Example 7: Repeated Fields (Batch Processing) ===\n";
507
508     sensors::SensorBatch batch;
509     batch.set_batch_id("batch_001");
510
511     // Add multiple sensor readings
512     for (int i = 0; i < 10; ++i) {
513         auto* reading = batch.add_readings(); // Add new reading
514         reading->set_type(sensors::TEMPERATURE);
515         reading->set_device_id("sensor_" + std::to_string(i));
516         reading->set_temperature_celsius(20.0f + i * 0.5f);
517     }
518
519     // Iterate over readings
520     std::cout << "Batch contains " << batch.readings_size() << " readings:\n";
521     for (const auto& reading : batch.readings()) {
522         std::cout << "    " << reading.device_id()
523             << " : " << reading.temperature_celsius() << "°C\n";
524     }
525
526     // Access by index
```

```
527     if (batch.readings_size() > 0) {
528         const auto& first = batch.readings(0);
529         std::cout << "\nFirst reading: " << first.device_id() << "\n";
530     }
531
532     // Serialize the batch
533     std::string batch_data;
534     batch.SerializeToString(&batch_data);
535     std::cout << "Batch serialized: " << batch_data.size() << " bytes for "
536             << batch.readings_size() << " readings\n";
537     std::cout << "Average: " << (batch_data.size() / batch.readings_size()) <<
538             " bytes/reading\n";
539
540     std::cout << "\nRepeated field methods:\n";
541     std::cout << " • add_xxx() - append new element\n";
542     std::cout << " • xxx_size() - get count\n";
543     std::cout << " • xxx(index) - access by index\n";
544     std::cout << " • clear_xxx() - remove all elements\n";
545     std::cout << " • Range-based for loop support\n\n";
546 }
547 // =====
548 // EXAMPLE 6: MEMORY MANAGEMENT AND PERFORMANCE
549 // =====
550
551 void example_performance() {
552     std::cout << "== Example 6: Performance and Memory Efficiency ==\n";
553
554     std::cout << R"(
555 // Arena allocation for high-performance scenarios
556 // Reduces memory allocations and improves cache locality
557 google::protobuf::Arena arena;
558
559 auto* reading = google::protobuf::Arena::CreateMessage<sensors::SensorReading
560             >(&arena);
561 reading->set_device_id("sensor_001");
562 reading->set_temperature_celsius(23.5f);
563
564 // All nested messages also use arena
565 auto* timestamp = reading->mutable_timestamp();
566 timestamp->set_seconds(12345);
567
568 // Memory is freed when arena goes out of scope
569 // No need to delete individual messages
570
571 // Performance tips:
572 // 1. Reuse message objects instead of creating new ones
573 sensors::SensorReading reusable_msg;
574 for (int i = 0; i < 1000; ++i) {
575     reusable_msg.Clear(); // Reset, don't recreate
576     reusable_msg.set_device_id("sensor_" + std::to_string(i));
577     // ... serialize and send ...
578 }
```

```
579 // 2. Use SerializeToArray with pre-allocated buffer
580 std::vector<uint8_t> buffer(1024); // Pre-allocate
581 size_t size = reading->ByteSizeLong();
582 if (size <= buffer.size()) {
583     reading->SerializeToArray(buffer.data(), size);
584 }
585
586 // 3. For embedded systems: use lite runtime
587 // In .proto file: option optimize_for = LITE_RUNTIME;
588 // Reduces binary size by 50-70%
589 )";
590
591     std::cout << "\nPerformance techniques:\n";
592     std::cout << "    Arena allocation - reduces fragmentation\n";
593     std::cout << "    Message reuse with Clear() - avoids allocations\n";
594     std::cout << "    Pre-allocated buffers - zero-copy serialization\n";
595     std::cout << "    Lite runtime - smaller binary for embedded systems\n";
596     std::cout << "    Lazy field evaluation - on-demand parsing\n\n";
597 }
598
599 // =====
600 // EXAMPLE 9: EMBEDDED SYSTEMS CONSIDERATIONS
601 // =====
602
603 void example_embedded_systems() {
604     std::cout << "==== Example 9: Embedded Systems Best Practices ===\n";
605
606     std::cout << R"(
607 // 1. Use fixed-size buffers to avoid dynamic allocation
608 constexpr size_t MAX_MESSAGE_SIZE = 256;
609 uint8_t tx_buffer[MAX_MESSAGE_SIZE];
610 uint8_t rx_buffer[MAX_MESSAGE_SIZE];
611
612 // 2. Serialize directly to UART/SPI buffer
613 sensors::SensorReading reading;
614 reading.set_device_id("mcu_001");
615 reading.set_temperature_celsius(23.5f);
616
617 size_t size = reading.ByteSizeLong();
618 if (size <= MAX_MESSAGE_SIZE) {
619     reading.SerializeToArray(tx_buffer, size);
620     // uart_transmit(tx_buffer, size);
621 }
622
623 // 3. Parse from received buffer
624 // size_t received = uart_receive(rx_buffer, MAX_MESSAGE_SIZE);
625 size_t received = 50; // Example
626
627 sensors::SensorReading received_msg;
628 if (received_msg.ParseFromArray(rx_buffer, received)) {
629     // Process message
630     float temp = received_msg.temperature_celsius();
631 }
632 }
```

```
633 // 4. Use lite runtime for small footprint
634 // .proto file option:
635 //   option optimize_for = LITE_RUNTIME;
636 //
637 // Removes:
638 // - Reflection API
639 // - Text format support
640 // - JSON conversion
641 //
642 // Result: 50-70% smaller binary
643
644 // 5. Error handling without exceptions (embedded systems)
645 #ifdef PROTOBUF_USE_EXCEPTIONS
646     try {
647         reading.ParseFromArray(rx_buffer, received);
648     } catch (const std::exception& e) {
649         // Handle error
650     }
651 #else
652     // Check return value
653     if (!reading.ParseFromArray(rx_buffer, received)) {
654         // Handle error without exceptions
655     }
656 #endif
657 )";
658
659     std::cout << "\nEmbedded systems considerations:\n";
660     std::cout << "    Fixed-size buffers - no heap fragmentation\n";
661     std::cout << "    Lite runtime - minimal binary size\n";
662     std::cout << "    No exceptions - suitable for bare-metal\n";
663     std::cout << "    Zero-copy serialization - DMA-friendly\n";
664     std::cout << "    Deterministic memory usage\n\n";
665 }
666
667 // =====
668 // EXAMPLE 8: SCHEMA EVOLUTION (BACKWARD COMPATIBILITY)
669 // =====
670
671 void example_schema_evolution() {
672     std::cout << "==== Example 8: Schema Evolution ===\n";
673
674     std::cout << R"(
675 // Protocol Buffers support schema evolution:
676 // Old code can read new messages, new code can read old messages
677
678 // VERSION 1 of .proto:
679 // message SensorReading {
680 //     string device_id = 1;
681 //     float temperature_celsius = 2;
682 // }
683
684 // VERSION 2 adds new fields (backward compatible):
685 // message SensorReading {
686 //     string device_id = 1;
```

```
687 //     float temperature_celsius = 2;
688 //     string location = 3;           // NEW FIELD
689 //     int32 battery_percent = 4;    // NEW FIELD
690 // }
691
692 // Rules for compatibility:
693 //   Can add new fields (old code ignores them)
694 //   Can remove optional fields (new code uses defaults)
695 //   Cannot change field numbers (breaks everything)
696 //   Cannot change field types (incompatible)
697 //   Can rename fields (only changes code, not wire format)
698
699 // Example: Old client reading new message
700 // sensors::SensorReading msg; // From version 1 code
701 // msg.ParseFromString(new_message_bytes);
702 //
703 // float temp = msg.temperature_celsius(); // Works!
704 // // New fields (location, battery) are ignored by old code
705
706 // Example: New client reading old message
707 // sensors::SensorReading msg; // From version 2 code
708 // msg.ParseFromString(old_message_bytes);
709 //
710 // float temp = msg.temperature_celsius(); // Works!
711 // std::string loc = msg.location();        // Empty string (default)
712 // int battery = msg.battery_percent();    // 0 (default)
713 ")";
714
715     std::cout << "\nBackward compatibility rules:\n";
716     std::cout << "  DO:\n";
717     std::cout << "    Add new optional fields\n";
718     std::cout << "    Remove optional fields (deprecated first)\n";
719     std::cout << "    Rename fields (code only, not wire format)\n";
720     std::cout << "  DON'T:\n";
721     std::cout << "    Change field numbers\n";
722     std::cout << "    Change field types\n";
723     std::cout << "    Make required fields optional (v2 syntax)\n\n";
724 }
725
726 // =====
727 // COMPARISON: PROTOBUF VS ALTERNATIVES
728 // =====
729
730 void comparison_with_alternatives() {
731     std::cout << "==== Comparison with Alternatives ===\n\n";
732
733     std::cout << "XML:\n";
734     std::cout << "    3-10x larger than protobuf\n";
735     std::cout << "    20-100x slower parsing\n";
736     std::cout << "    Human-readable\n";
737     std::cout << "    Self-describing\n\n";
738
739     std::cout << "JSON:\n";
740     std::cout << "    2-5x larger than protobuf\n";
```

```
741     std::cout << "      10-50x slower parsing\n";
742     std::cout << "      Human-readable\n";
743     std::cout << "      JavaScript-friendly\n";
744     std::cout << "      No schema enforcement\n\n";
745
746     std::cout << "MessagePack:\n";
747     std::cout << "      Similar size to protobuf\n";
748     std::cout << "      Fast parsing\n";
749     std::cout << "      No schema (schemaless)\n";
750     std::cout << "      No code generation\n\n";
751
752     std::cout << "FlatBuffers:\n";
753     std::cout << "      Zero-copy deserialization\n";
754     std::cout << "      Very fast access\n";
755     std::cout << "      Larger than protobuf\n";
756     std::cout << "      Mutable messages more complex\n\n";
757
758     std::cout << "Cap'n Proto:\n";
759     std::cout << "      Zero-copy like FlatBuffers\n";
760     std::cout << "      Very fast\n";
761     std::cout << "      Less mature ecosystem\n";
762     std::cout << "      Smaller community\n\n";
763
764     std::cout << "When to use Protocol Buffers:\n";
765     std::cout << "      Need efficient binary serialization\n";
766     std::cout << "      Cross-language communication\n";
767     std::cout << "      Schema evolution important\n";
768     std::cout << "      Google's ecosystem (gRPC, etc.)\n";
769     std::cout << "      Embedded systems with constraints\n\n";
770 }
771
772 // =====
773 // MAIN FUNCTION
774 // =====
775
776 int main() {
777     std::cout << "=====\\n"
778     ;
779     std::cout << "GOOGLE PROTOCOL BUFFERS (PROTOBUF) EXAMPLES\\n";
780     std::cout << "=====\\n\\n";
781
782     example_create_sensor_reading();
783     example_binary_serialization();
784     example_binary_deserialization();
785     example_file_serialization();
786     example_file_formats();
787     example_json_conversion();
788     example_repeated_fields();
789     example_performance();
790     example_embedded_systems();
791     example_schema_evolution();
792     comparison_with_alternatives();
```

```
793     std::cout << "=====\\n"
794     ;
795     std::cout << "BUILD INSTRUCTIONS:\\n";
796     std::cout << "=====\\n\\n";
797     std::cout << "1. Install Protocol Buffers:\\n";
798     std::cout << "    Ubuntu/Debian:\\n";
799     std::cout << "        sudo apt-get install protobuf-compiler libprotobuf-dev\\n\\n";
800
801     std::cout << "    macOS:\\n";
802     std::cout << "        brew install protobuf\\n\\n";
803
804     std::cout << "    Windows:\\n";
805     std::cout << "        Download from: https://github.com/protocolbuffers/protobuf/releases\\n\\n";
806
807     std::cout << "2. Generate C++ code from .proto file:\\n";
808     std::cout << "    protoc --cpp_out=. proto/sensor_data.proto\\n";
809     std::cout << "    This creates:\\n";
810     std::cout << "        - sensor_data.pb.h\\n";
811     std::cout << "        - sensor_data.pb.cc\\n\\n";
812
813     std::cout << "3. Compile with CMake:\\n";
814     std::cout << "    find_package(Protobuf REQUIRED)\\n";
815     std::cout << "    target_link_libraries(YourTarget protobuf::libprotobuf)\\n
816
817     std::cout << "4. Or compile manually:\\n";
818     std::cout << "    g++ -std=c++17 ProtobufExample.cpp sensor_data.pb.cc \\\n
819     ";
820     std::cout << "        -lprotobuf -o protobuf_example\\n\\n";
821
822     std::cout << "=====\\n";
823     std::cout << "RESOURCES:\\n";
824     std::cout << "=====\\n";
825     std::cout << "Official docs: https://protobuf.dev/\\n";
826     std::cout << "GitHub: https://github.com/protocolbuffers/protobuf\\n";
827     std::cout << "C++ tutorial: https://protobuf.dev/getting-started/cpptutorial/\\n\\n";
828
829     return 0;
}
```

## 54 Source Code: Pybind11Example.cpp

File: src/Pybind11Example.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // PYBIND11: C++ AND PYTHON INTEROPERABILITY
3 // =====
4 // This example demonstrates bidirectional calling between C++ and Python
5 // using pybind11 (modern, header-only binding library)
6 //
7 // BUILD INSTRUCTIONS:
8 // =====
9 // 1. Install pybind11:
10 //    pip install pybind11
11 //
12 // 2. Compile as Python extension module:
13 //    c++ -O3 -Wall -shared -std=c++17 -fPIC \
14 //          $(python3 -m pybind11 --includes) \
15 //          Pybind11Example.cpp \
16 //          -o pybind_example$(python3-config --extension-suffix)
17 //
18 // 3. Use from Python:
19 //    import pybind_example
20 //    result = pybind_example.add(2, 3)
21 //
22 // ALTERNATIVE FRAMEWORKS:
23 // =====
24 // - Boost.Python: Older, requires Boost library
25 // - SWIG: Generates bindings for multiple languages
26 // - ctypes: Python standard library, simpler but less type-safe
27 // - cffi: C Foreign Function Interface for Python
28 // - nanobind: Modern, lightweight alternative to pybind11
29 // =====
30
31 #include <pybind11/pybind11.h>
32 #include <pybind11/stl.h>           // STL containers conversion
33 #include <pybind11/functional.h> // std::function conversion
34 #include <pybind11/numpy.h>        // NumPy arrays support
35 #include <iostream>
36 #include <vector>
37 #include <string>
38 #include <memory>
39 #include <cmath>
40
41 namespace py = pybind11;
42
43 // =====
44 // 1. SIMPLE FUNCTIONS - C++ FUNCTIONS CALLED FROM PYTHON
45 // =====
46
47 // Simple function
48 int add(int a, int b) {
49     return a + b;
```

```
50 }
51
52 // Function with default arguments
53 double multiply(double a, double b = 2.0) {
54     return a * b;
55 }
56
57 // Function with multiple return values (using tuple)
58 std::tuple<int, int> divide_with_remainder(int dividend, int divisor) {
59     int quotient = dividend / divisor;
60     int remainder = dividend % divisor;
61     return std::make_tuple(quotient, remainder);
62 }
63
64 // Function returning string
65 std::string greet(const std::string& name) {
66     return "Hello, " + name + "!";
67 }
68
69 // =====
70 // 2. WORKING WITH STL CONTAINERS
71 // =====
72
73 // Function accepting and returning std::vector
74 std::vector<int> square_elements(const std::vector<int>& vec) {
75     std::vector<int> result;
76     result.reserve(vec.size());
77     for (int val : vec) {
78         result.push_back(val * val);
79     }
80     return result;
81 }
82
83 // Function with std::map
84 std::map<std::string, int> count_words(const std::vector<std::string>& words)
85 {
86     std::map<std::string, int> counts;
87     for (const auto& word : words) {
88         counts[word]++;
89     }
90     return counts;
91 }
92
93 // =====
94 // 3. CLASSES - EXPOSING C++ CLASSES TO PYTHON
95 // =====
96
96 class Vector2D {
97 private:
98     double x, y;
99
100 public:
101     // Constructors
102     Vector2D() : x(0.0), y(0.0) {}
```

```
103     Vector2D(double x, double y) : x(x), y(y) {}
104
105     // Getters/Setters
106     double getX() const { return x; }
107     double getY() const { return y; }
108     void setX(double val) { x = val; }
109     void setY(double val) { y = val; }
110
111     // Methods
112     double length() const {
113         return std::sqrt(x * x + y * y);
114     }
115
116     Vector2D operator+(const Vector2D& other) const {
117         return Vector2D(x + other.x, y + other.y);
118     }
119
120     Vector2D operator*(double scalar) const {
121         return Vector2D(x * scalar, y * scalar);
122     }
123
124     // String representation
125     std::string toString() const {
126         return "Vector2D(" + std::to_string(x) + ", " + std::to_string(y) + ")"
127         " ";
128     }
129
130     // Static method
131     static Vector2D zero() {
132         return Vector2D(0.0, 0.0);
133     }
134
135 // =====
136 // 4. INHERITANCE - BASE AND DERIVED CLASSES
137 // =====
138
139 class Shape {
140 protected:
141     std::string name;
142
143 public:
144     Shape(const std::string& name) : name(name) {}
145     virtual ~Shape() = default;
146
147     virtual double area() const = 0;
148     virtual std::string getName() const { return name; }
149 };
150
151 class Circle : public Shape {
152 private:
153     double radius;
154
155 public:
```

```
156     Circle(const std::string& name, double r)
157         : Shape(name), radius(r) {}
158
159     double area() const override {
160         return 3.14159 * radius * radius;
161     }
162
163     double getRadius() const { return radius; }
164 };
165
166 class Rectangle : public Shape {
167 private:
168     double width, height;
169
170 public:
171     Rectangle(const std::string& name, double w, double h)
172         : Shape(name), width(w), height(h) {}
173
174     double area() const override {
175         return width * height;
176     }
177 };
178
179 // =====
180 // 5. CALLBACKS - CALLING PYTHON FUNCTIONS FROM C++
181 // =====
182
183 // Process data with Python callback
184 std::vector<int> process_with_callback(
185     const std::vector<int>& data,
186     std::function<int(int)> callback) {
187
188     std::vector<int> result;
189     result.reserve(data.size());
190
191     for (int val : data) {
192         result.push_back(callback(val)); // Call Python function!
193     }
194
195     return result;
196 }
197
198 // Numerical integration using Python callback
199 double integrate(std::function<double(double)> func,
200                  double start, double end, int steps = 1000) {
201     double dx = (end - start) / steps;
202     double sum = 0.0;
203
204     for (int i = 0; i < steps; ++i) {
205         double x = start + i * dx;
206         sum += func(x) * dx; // Call Python function!
207     }
208
209     return sum;
```

```
210 }
211
212 // =====
213 // 6. SENSOR DATA PROCESSING (EMBEDDED SYSTEMS EXAMPLE)
214 // =====
215
216 struct SensorReading {
217     double timestamp;
218     double value;
219     std::string sensor_id;
220
221     SensorReading(double t, double v, const std::string& id)
222         : timestamp(t), value(v), sensor_id(id) {}
223 };
224
225 class SensorProcessor {
226 private:
227     std::vector<SensorReading> readings;
228
229 public:
230     void addReading(double timestamp, double value, const std::string& id) {
231         readings.emplace_back(timestamp, value, id);
232     }
233
234     size_t getReadingCount() const {
235         return readings.size();
236     }
237
238     std::vector<SensorReading> getReadings() const {
239         return readings;
240     }
241
242     // Process readings with Python callback
243     std::vector<double> processReadings(
244         std::function<double(const SensorReading&)> processor) {
245
246         std::vector<double> results;
247         results.reserve(readings.size());
248
249         for (const auto& reading : readings) {
250             results.push_back(processor(reading)); // Call Python!
251         }
252
253         return results;
254     }
255
256     // Filter readings with Python predicate
257     std::vector<SensorReading> filterReadings(
258         std::function<bool(const SensorReading&)> predicate) {
259
260         std::vector<SensorReading> filtered;
261
262         for (const auto& reading : readings) {
263             if (predicate(reading)) { // Call Python!
```

```
264         filtered.push_back(reading);
265     }
266 }
267
268     return filtered;
269 }
270 };
271
272 // =====
273 // 7. NUMPY ARRAY PROCESSING (HIGH PERFORMANCE)
274 // =====
275
276 // Process NumPy array in C++ for performance
277 py::array_t<double> numpy_square(py::array_t<double> input) {
278     // Get buffer info
279     py::buffer_info buf = input.request();
280
281     if (buf.ndim != 1) {
282         throw std::runtime_error("Number of dimensions must be 1");
283     }
284
285     // Create output array
286     auto result = py::array_t<double>(buf.size);
287     py::buffer_info result_buf = result.request();
288
289     // Get pointers
290     double* input_ptr = static_cast<double*>(buf.ptr);
291     double* result_ptr = static_cast<double*>(result_buf.ptr);
292
293     // Process in C++ (fast!)
294     for (size_t i = 0; i < buf.shape[0]; i++) {
295         result_ptr[i] = input_ptr[i] * input_ptr[i];
296     }
297
298     return result;
299 }
300
301 // Matrix multiplication (2D NumPy arrays)
302 py::array_t<double> matrix_multiply(
303     py::array_t<double> a,
304     py::array_t<double> b) {
305
306     py::buffer_info buf_a = a.request();
307     py::buffer_info buf_b = b.request();
308
309     if (buf_a.ndim != 2 || buf_b.ndim != 2) {
310         throw std::runtime_error("Inputs must be 2D arrays");
311     }
312
313     size_t rows_a = buf_a.shape[0];
314     size_t cols_a = buf_a.shape[1];
315     size_t rows_b = buf_b.shape[0];
316     size_t cols_b = buf_b.shape[1];
317 }
```



```
370 // Factory function returning unique_ptr
371 std::unique_ptr<Resource> create_resource(const std::string& name, int id) {
372     return std::make_unique<Resource>(name, id);
373 }
374
375 // Factory returning shared_ptr
376 std::shared_ptr<Resource> create_shared_resource(const std::string& name, int
377     id) {
378     return std::make_shared<Resource>(name, id);
379 }
380 // =====
381 // PYBIND11 MODULE DEFINITION
382 // =====
383
384 PYBIND11_MODULE(pybind_example, m) {
385     m.doc() = "pybind11 example module - C++ and Python interoperability";
386
387     // =====
388     // 1. SIMPLE FUNCTIONS
389     // =====
390     m.def("add", &add, "Add two integers",
391             py::arg("a"), py::arg("b"));
392
393     m.def("multiply", &multiply, "Multiply two numbers",
394             py::arg("a"), py::arg("b") = 2.0);
395
396     m.def("divide_with_remainder", &divide_with_remainder,
397             "Divide and return (quotient, remainder)");
398
399     m.def("greet", &greet, "Greet someone");
400
401     // =====
402     // 2. STL CONTAINERS
403     // =====
404     m.def("square_elements", &square_elements,
405             "Square all elements in a list");
406
407     m.def("count_words", &count_words,
408             "Count occurrences of each word");
409
410     // =====
411     // 3. VECTOR2D CLASS
412     // =====
413     py::class_<Vector2D>(m, "Vector2D")
414         .def(py::init<>())
415         .def(py::init<double, double>())
416         .def_property("x", &Vector2D::getX, &Vector2D::setX)
417         .def_property("y", &Vector2D::getY, &Vector2D::setY)
418         .def("length", &Vector2D::length)
419         .def("__add__", &Vector2D::operator+)
420         .def("__mul__", &Vector2D::operator*)
421         .def("__repr__", &Vector2D::toString)
422         .def_static("zero", &Vector2D::zero);
```

```
423 // =====
424 // 4. INHERITANCE - SHAPES
425 // =====
426 py::class_<Shape>(m, "Shape")
427     .def("area", &Shape::area)
428     .def("get_name", &Shape::getName);
429
430
431 py::class_<Circle, Shape>(m, "Circle")
432     .def(py::init<const std::string&, double>())
433     .def("get_radius", &Circle::getRadius);
434
435
436 py::class_<Rectangle, Shape>(m, "Rectangle")
437     .def(py::init<const std::string&, double, double>());
438
439 // =====
440 // 5. CALLBACKS
441 // =====
442 m.def("process_with_callback", &process_with_callback,
443         "Process data with Python callback function");
444
445 m.def("integrate", &integrate,
446         "Numerical integration using Python callback",
447         py::arg("func"), py::arg("start"), py::arg("end"),
448         py::arg("steps") = 1000);
449
450 // =====
451 // 6. SENSOR DATA PROCESSING
452 // =====
453 py::class_<SensorReading>(m, "SensorReading")
454     .def(py::init<double, double, const std::string&>())
455     .def_readonly("timestamp", &SensorReading::timestamp)
456     .def_readonly("value", &SensorReading::value)
457     .def_readonly("sensor_id", &SensorReading::sensor_id);
458
459 py::class_<SensorProcessor>(m, "SensorProcessor")
460     .def(py::init<>())
461     .def("add_reading", &SensorProcessor::addReading)
462     .def("get_reading_count", &SensorProcessor::getReadingCount)
463     .def("get_readings", &SensorProcessor::getReadings)
464     .def("process_readings", &SensorProcessor::processReadings)
465     .def("filter_readings", &SensorProcessor::filterReadings);
466
467 // =====
468 // 7. NUMPY ARRAYS
469 // =====
470 m.def("numpy_square", &numpy_square,
471         "Square all elements in NumPy array (C++ speed!)");
472
473 m.def("matrix_multiply", &matrix_multiply,
474         "Matrix multiplication using NumPy arrays");
475
476 // =====
477 // 8. SMART POINTERS
```

```
477 // =====
478 py::class_<Resource, std::shared_ptr<Resource>>(m, "Resource")
479     .def("get_name", &Resource::getName)
480     .def("get_id", &Resource::getId);
481
482     m.def("create_resource", &create_resource,
483           "Create a resource (returns unique_ptr)");
484
485     m.def("create_shared_resource", &create_shared_resource,
486           "Create a shared resource (returns shared_ptr)");
487 }
488
489 // =====
490 // STANDALONE EXAMPLE (FOR DOCUMENTATION)
491 // =====
492
493 /*
494 PYTHON USAGE EXAMPLES:
495 =====
496
497 # 1. Simple functions
498 import pybind_example as pe
499
500 result = pe.add(5, 3)
501 print(f"5 + 3 = {result}") # Output: 5 + 3 = 8
502
503 value = pe.multiply(4.5)
504 print(f"4.5 * 2.0 = {value}") # Output: 4.5 * 2.0 = 9.0
505
506 quotient, remainder = pe.divide_with_remainder(17, 5)
507 print(f"17 / 5 = {quotient} remainder {remainder}") # Output: 17 / 5 = 3
508     remainder 2
509
510 message = pe.greet("World")
511 print(message) # Output: Hello, World!
512
513 # 2. STL containers
514 numbers = [1, 2, 3, 4, 5]
515 squared = pe.square_elements(numbers)
516 print(f"Squared: {squared}") # Output: Squared: [1, 4, 9, 16, 25]
517
518 words = ["hello", "world", "hello", "python"]
519 counts = pe.count_words(words)
520 print(f"Word counts: {counts}") # Output: Word counts: {'hello': 2, 'world':
521     1, 'python': 1}
522
523 # 3. Classes
524 v1 = pe.Vector2D(3.0, 4.0)
525 v2 = pe.Vector2D(1.0, 2.0)
526 print(f"v1 = {v1}") # Output: v1 = Vector2D(3.000000, 4.000000)
527 print(f"v1.length() = {v1.length()}") # Output: v1.length() = 5.0
528
529 v3 = v1 + v2
530 print(f"v1 + v2 = {v3}") # Output: v1 + v2 = Vector2D(4.000000, 6.000000)
```

```
529
530 v4 = v1 * 2.0
531 print(f"v1 * 2.0 = {v4}")  # Output: v1 * 2.0 = Vector2D(6.000000, 8.000000)
532
533 # 4. Inheritance
534 circle = pe.Circle("MyCircle", 5.0)
535 rect = pe.Rectangle("MyRect", 4.0, 6.0)
536 print(f"{circle.get_name()} area: {circle.area()}")  # Output: MyCircle area:
537     78.53975
538 print(f"{rect.get_name()} area: {rect.area()}")  # Output: MyRect area: 24.0
539
540 # 5. Callbacks - CALLING PYTHON FROM C++
541 def my_python_function(x):
542     return x * 2 + 1
543
544 data = [1, 2, 3, 4, 5]
545 result = pe.process_with_callback(data, my_python_function)
546 print(f"Processed: {result}")  # Output: Processed: [3, 5, 7, 9, 11]
547
548 # Integrate a Python function using C++
549 import math
550 result = pe.integrate(lambda x: x**2, 0, 1, steps=10000)
551 print(f"Integral of x^2 from 0 to 1: {result}")  # Output: ~0.333333
552
553 result = pe.integrate(math.sin, 0, math.pi, steps=10000)
554 print(f"Integral of sin(x) from 0 to : {result}")  # Output: ~2.0
555
556 # 6. Sensor processing with callbacks
557 processor = pe.SensorProcessor()
558 processor.add_reading(0.0, 25.5, "TEMP_01")
559 processor.add_reading(1.0, 26.2, "TEMP_01")
560 processor.add_reading(2.0, 24.8, "TEMP_02")
561
562 # Process with Python function
563 def extract_value(reading):
564     return reading.value
565
566 values = processor.process_readings(extract_value)
567 print(f"Values: {values}")  # Output: Values: [25.5, 26.2, 24.8]
568
569 # Filter with Python predicate
570 def high_temp(reading):
571     return reading.value > 25.0
572
573 high_temps = processor.filter_readings(high_temp)
574 print(f"High temps: {len(high_temps)}")  # Output: High temps: 2
575
576 # 7. NumPy arrays (high performance!)
577 import numpy as np
578
579 arr = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
580 squared = pe.numpy_square(arr)  # Fast C++ processing!
581 print(f"NumPy squared: {squared}")  # Output: NumPy squared: [1. 4. 9. 16.
582     25.]
```

```
581
582 A = np.array([[1.0, 2.0], [3.0, 4.0]])
583 B = np.array([[5.0, 6.0], [7.0, 8.0]])
584 C = pe.matrix_multiply(A, B)
585 print(f"Matrix multiply:\n{C}")
586 # Output:
587 # [[19. 22.]
588 #  [43. 50.]]
589
590 # 8. Smart pointers
591 resource = pe.create_shared_resource("MyResource", 42)
592 print(f"Resource: {resource.get_name()}, ID: {resource.get_id()}")
593 # Output: Resource: MyResource, ID: 42
594 # When resource goes out of scope, C++ destructor is automatically called
595
596 BUILD SCRIPT (build.sh):
597 =====
598 #!/bin/bash
599
600 # Compile the pybind11 extension
601 c++ -O3 -Wall -shared -std=c++17 -fPIC \
602     $(python3 -m pybind11 --includes) \
603     Pybind11Example.cpp \
604     -o pybind_example$(python3-config --extension-suffix)
605
606 # Run Python tests
607 python3 test_pybind.py
608
609 ALTERNATIVE FRAMEWORKS:
610 =====
611
612 1. BOOST.PYTHON (older, requires Boost):
613     #include <boost/python.hpp>
614     BOOST_PYTHON_MODULE(example) {
615         boost::python::def("add", add);
616     }
617
618 2. SWIG (multi-language support):
619     // example.i
620     %module example
621     %{
622     #include "example.h"
623     %}
624     %include "example.h"
625
626     Build: swig -python -c++ example.i
627
628 3. CTYPES (Python standard library):
629     // Compile: g++ -shared -fPIC example.cpp -o example.so
630
631     # Python:
632     import ctypes
633     lib = ctypes.CDLL('./example.so')
634     lib.add.argtypes = [ctypes.c_int, ctypes.c_int]
```

```
635     lib.add.restype = ctypes.c_int
636     result = lib.add(5, 3)
637
638 4. CFFI (C Foreign Function Interface):
639     from cffi import FFI
640     ffi = FFI()
641     ffi.cdef("int add(int, int);")
642     lib = ffi.dlopen('./example.so')
643     result = lib.add(5, 3)
644
645 5. NANOBIND (modern, lightweight):
646     #include <nanobind/nanobind.h>
647     NB_MODULE(example, m) {
648         m.def("add", &add);
649     }
650
651 PERFORMANCE COMPARISON:
652 =====
653 For numerical computations:
654 - Pure Python: 1x (baseline)
655 - NumPy: 10-100x faster
656 - C++ via pybind11: 100-1000x faster
657 - Direct C++ (no Python): 1000x+ faster
658
659 WHEN TO USE:
660 =====
661 Use pybind11 when:•
662     Need performance-critical code in C++•
663     Have existing C++ libraries to expose•
664     Want type safety and automatic conversions•
665     Need to process NumPy arrays efficiently•
666     Want modern C++ features (C++11-20)
667
668 Avoid when:•
669     Pure Python is fast enough•
670     No C++ expertise available•
671     Deployment complexity is a concern•
672     Only need simple C functions (use ctypes)
673 */
```

## 55 Source Code: ROMability.cpp

File: src/ROMability.cpp

Repository: [View on GitHub](#)

```
1 // ROMability.cpp
2 // Demonstrates ROM-ability in C++ - Placing data in Read-Only Memory (.rodata
3 // section)
4 //
5 // KEY CONCEPTS:
6 // 1. const - Runtime or compile-time constant, placed in ROM if possible
7 // 2. constexpr - Compile-time constant, guaranteed ROM placement
8 // 3. consteval - C++20, forced compile-time evaluation
9 // 4. constinit - C++20, compile-time initialization, runtime mutable
10 // 5. ROM vs RAM - .rodata vs .data/.bss sections
11 //
12 // EMBEDDED SYSTEMS CONTEXT:
13 // ROM (Read-Only Memory) - Flash memory, non-volatile, cheaper
14 // RAM (Random Access Memory) - SRAM, volatile, expensive on MCUs
15 // Goal: Maximize ROM usage, minimize RAM usage
16 //
17 // C++ ADVANTAGES OVER C:
18 // constexpr functions - compute at compile-time
19 // constexpr constructors - complex objects in ROM
20 // std::array - type-safe ROM arrays
21 // consteval - guarantee compile-time evaluation
22 // Template metaprogramming - generate ROM data
23
24 #include <iostream>
25 #include <array>
26 #include <string_view>
27 #include <cmath>
28 #include <iomanip>
29 #include <cstdint>
30 //
31 =====
32 // SECTION 1: const vs constexpr - Understanding the Difference
33 // =====
34
35 namespace const_vs_constexpr {
36
37 // C-style const - MAY be in ROM, but not guaranteed
38 const int c_style_const = 42;
39
40 // C++ constexpr - GUARANTEED compile-time constant, always in ROM
41 constexpr int cpp_constexpr = 42;
42
43 // const with runtime initialization - goes to RAM!
44 int get_runtime_value() { return 42; }
45 const int runtime_const = get_runtime_value(); // RAM, not ROM!
```

```
45 // constexpr forces compile-time evaluation
46 constexpr int get_compile_time_value() { return 42; }
47 constexpr int compile_time_const = get_compile_time_value(); // ROM
48
49 // Complex compile-time calculation
50 constexpr int factorial(int n) {
51     return (n <= 1) ? 1 : n * factorial(n - 1);
52 }
53
54 constexpr int fact_5 = factorial(5); // Computed at compile-time, stored in
55 // ROM
56
57 void demonstrate() {
58     std::cout << "\n" << std::string(70, '=') << "\n";
59     std::cout << "==== SECTION 1: const vs constexpr ===\n";
60     std::cout << std::string(70, '=') << "\n\n";
61
62     std::cout << "1. Traditional const:\n";
63     std::cout << "    const int c_style_const = 42;\n";
64     std::cout << "    Value: " << c_style_const << "\n";
65     std::cout << "    Location: Probably ROM, but not guaranteed\n\n";
66
67     std::cout << "2. constexpr (C++11):\n";
68     std::cout << "    constexpr int cpp_constexpr = 42;\n";
69     std::cout << "    Value: " << cpp_constexpr << "\n";
70     std::cout << "    Location: Guaranteed ROM (.rodata section)\n\n";
71
72     std::cout << "3. const with runtime initialization:\n";
73     std::cout << "    const int runtime_const = get_runtime_value();\n";
74     std::cout << "    Value: " << runtime_const << "\n";
75     std::cout << "    Location: RAM (.data section) - not ROM!\n\n";
76
77     std::cout << "4. constexpr with compile-time function:\n";
78     std::cout << "    constexpr int fact_5 = factorial(5);\n";
79     std::cout << "    Value: " << fact_5 << " (computed at compile-time!)\n";
80     std::cout << "    Location: ROM (.rodata section)\n\n";
81
82     std::cout << "    RULE: Use constexpr for guaranteed ROM placement\n";
83     std::cout << "    RULE: const doesn't guarantee ROM (can be initialized at
84     // runtime)\n";
85 }
86 } // namespace const_vs_constexpr
87
88 // =====
89 // SECTION 2: ROM-able Arrays - Lookup Tables
90 // =====
91
92 namespace rom_arrays {
```

```
93 // C-style array - ROM placement
94 constexpr int sine_table_c[16] = {
95     0, 707, 1000, 707, 0, -707, -1000, -707,
96     0, 707, 1000, 707, 0, -707, -1000, -707
97 };
98 };
99
100 // C++ std::array - ROM placement with type safety
101 constexpr std::array<int, 16> sine_table_cpp = {
102     0, 707, 1000, 707, 0, -707, -1000, -707,
103     0, 707, 1000, 707, 0, -707, -1000, -707
104 };
105
106 // Generate lookup table at compile-time
107 constexpr auto generate_sine_table() {
108     std::array<int, 360> table{};
109     for (int i = 0; i < 360; ++i) {
110         // Note: std::sin is not constexpr before C++26
111         // Using approximation for demonstration
112         table[i] = static_cast<int>(1000 * std::sin(i * 3.14159 / 180.0));
113     }
114     return table;
115 }
116
117 // Entire lookup table generated at compile-time, stored in ROM
118 constexpr auto sine_lookup = generate_sine_table();
119
120 // PWM duty cycle table for LED brightness (gamma correction)
121 constexpr auto generate_gamma_table() {
122     std::array<uint8_t, 256> table{};
123     for (int i = 0; i < 256; ++i) {
124         // Gamma correction: output = input^2.2
125         double normalized = i / 255.0;
126         double corrected = std::pow(normalized, 2.2);
127         table[i] = static_cast<uint8_t>(corrected * 255.0);
128     }
129     return table;
130 }
131
132 constexpr auto gamma_table = generate_gamma_table();
133
134 void demonstrate() {
135     std::cout << "\n" << std::string(70, '=') << "\n";
136     std::cout << "==== SECTION 2: ROM-able Arrays - Lookup Tables ===\n";
137     std::cout << std::string(70, '=') << "\n\n";
138
139     std::cout << "1. C-style const array in ROM:\n";
140     std::cout << "    constexpr int sine_table_c[16] = {...};\n";
141     std::cout << "    Sample values: " << sine_table_c[0] << ", "
142             << sine_table_c[2] << ", " << sine_table_c[6] << "\n";
143     std::cout << "    Size: " << sizeof(sine_table_c) << " bytes\n\n";
144
145     std::cout << "2. C++ std::array in ROM (type-safe):\n";
146     std::cout << "    constexpr std::array<int, 16> sine_table_cpp = {...};\n";
```

```

147     std::cout << "      Sample values: " << sine_table_cpp[0] << ", "
148             << sine_table_cpp[2] << ", " << sine_table_cpp[6] << "\n";
149     std::cout << "      Size: " << sine_table_cpp.size() << " elements\n\n";
150
151     std::cout << "3. Generated lookup table (360 entries, compile-time):\n";
152     std::cout << "      Sine values (degrees):\n";
153     std::cout << "          0°: " << sine_lookup[0] << "\n";
154     std::cout << "          30°: " << sine_lookup[30] << "\n";
155     std::cout << "          60°: " << sine_lookup[60] << "\n";
156     std::cout << "          90°: " << sine_lookup[90] << "\n";
157     std::cout << "      Total size: " << sizeof(sine_lookup) << " bytes in ROM\n\n";
158
159     std::cout << "4. Gamma correction table (256 entries):\n";
160     std::cout << "      PWM duty cycles for linear brightness:\n";
161     std::cout << "          Input 0: " << static_cast<int>(gamma_table[0]) << "\n";
162     std::cout << "          Input 64: " << static_cast<int>(gamma_table[64]) << "\n";
163     std::cout << "          ";
164     std::cout << "          Input 128: " << static_cast<int>(gamma_table[128]) << "\n";
165     std::cout << "          ";
166     std::cout << "          Input 255: " << static_cast<int>(gamma_table[255]) << "\n";
167     std::cout << "      Total size: " << sizeof(gamma_table) << " bytes in ROM\n\n";
168
169     std::cout << "      Embedded benefit: Lookup tables computed at compile-time\n";
170     std::cout << "      ";
171     std::cout << "      No runtime computation needed (saves CPU cycles)\n";
172     std::cout << "      All stored in cheap ROM, not expensive RAM\n";
173 }
174
175 } // namespace rom_arrays
176
177 // SECTION 3: constexpr Objects - Complex Data in ROM
178 // =====
179
180 namespace constexpr_objects {
181
182     // Configuration stored entirely in ROM
183     struct DeviceConfig {
184         const char* device_name;
185         uint32_t serial_number;
186         uint16_t max_voltage_mv;
187         uint8_t num_channels;
188
189         constexpr DeviceConfig(const char* name, uint32_t serial,
190                               uint16_t voltage, uint8_t channels)
191             : device_name(name), serial_number(serial),
192               max_voltage_mv(voltage), num_channels(channels) {}
193

```

```
191 };
```

```
192
```

```
193 // Entire config in ROM
```

```
194 constexpr DeviceConfig device_config{
```

```
195     "STM32F407",
```

```
196     0x12345678,
```

```
197     3300, // 3.3V
```

```
198     16
```

```
199 };
```

```
200
```

```
201 // Sensor calibration data in ROM
```

```
202 struct SensorCalibration {
```

```
203     float offset;
```

```
204     float gain;
```

```
205     int16_t temp_coefficient;
```

```
206
```

```
207     constexpr SensorCalibration(float o, float g, int16_t tc)
```

```
208         : offset(o), gain(g), temp_coefficient(tc) {}
```

```
209
```

```
210     constexpr float calibrate(int16_t raw_value, int16_t temperature) const {
```

```
211         float temp_correction = temp_coefficient * (temperature - 25) / 1000.0
```

```
212         f;
```

```
213         return (raw_value + offset) * gain * (1.0f + temp_correction);
```

```
214     }
```

```
215 };
```

```
216
```

```
217 constexpr SensorCalibration pressure_sensor{
```

```
218     -50.0f, // offset
```

```
219     0.125f, // gain
```

```
220     -25 // temp coefficient (ppm/°C)
```

```
221 };
```

```
222 // Compile-time test of calibration
```

```
223 static_assert(pressure_sensor.calibrate(1000, 25) > 0, "Calibration sanity
```

```
224     check");
```

```
225
```

```
226 void demonstrate() {
```

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

```
228     std::cout << "==== SECTION 3: constexpr Objects in ROM ====\n";
```

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

```
230
```

```
231     std::cout << "1. Device configuration (ROM):\n";
```

```
232     std::cout << "    Device: " << device_config.device_name << "\n";
```

```
233     std::cout << "    Serial: 0x" << std::hex << device_config.serial_number
```

```
234         << std::dec << "\n";
```

```
235     std::cout << "    Max Voltage: " << device_config.max_voltage_mv << " mV\n";
```

```
236     std::cout << "    Channels: " << static_cast<int>(device_config.
```

```
237         num_channels) << "\n";
```

```
238     std::cout << "    Location: ROM (.rodata section)\n\n";
```

```
239
```

```
240     std::cout << "2. Sensor calibration with compile-time function:\n";
```

```
241     std::cout << "    Offset: " << pressure_sensor.offset << "\n";
```

```
242     std::cout << "    Gain: " << pressure_sensor.gain << "\n";
```

```
241     std::cout << "    Temp Coeff: " << pressure_sensor.temp_coefficient << "
242         ppm/°C\n\n";
243
243     std::cout << "    Runtime calibration:\n";
244     int16_t raw = 1000;
245     int16_t temp = 25;
246     float calibrated = pressure_sensor.calibrate(raw, temp);
247     std::cout << "        Raw value: " << raw << " at " << temp << "°C\n";
248     std::cout << "        Calibrated: " << calibrated << " Pa\n";
249     std::cout << "        Location: ROM (.rodata section)\n\n";
250
251     std::cout << "    Complex objects can live entirely in ROM\n";
252     std::cout << "    constexpr methods allow compile-time testing\n";
253     std::cout << "    Perfect for embedded: config, calibration, constants\n";
254 }
255
256 } // namespace constexpr_objects
257
258 // =====
259 // SECTION 4: C++20 Features - consteval and constinit
260 // =====
261
262 namespace cpp20_features {
263
264 // consteval - MUST be evaluated at compile-time
265 consteval int must_be_compile_time(int n) {
266     return n * n;
267 }
268
269 // This works - compile-time constant
270 constexpr int square_5 = must_be_compile_time(5);
271
272 // This would NOT compile:
273 // int x = 5;
274 // int result = must_be_compile_time(x); // ERROR: x is not constexpr
275
276 // constinit - Compile-time initialization, but runtime mutable
277 constinit int initialized_at_compile_time = 42;
278
279 // Compare with const/constexpr
280 const int const_value = 42; // Can't modify
281 constexpr int constexpr_value = 42; // Can't modify, must be compile-time
282 // constinit allows modification but guarantees compile-time init
283
284 // Use case: Global variables that need compile-time init but runtime
285 // modification
285 constinit int error_count = 0; // Initialized at compile-time to 0
286
287 // String literals in ROM
288 constexpr std::string_view device_name = "STM32F407";
```

```

289 constexpr std::string_view error_messages[] = {
290     "No error",
291     "Sensor timeout",
292     "Invalid data",
293     "Communication error"
294 };
295
296 void demonstrate() {
297     std::cout << "\n" << std::string(70, '=') << "\n";
298     std::cout << "==== SECTION 4: C++20 - consteval and constinit ===\n";
299     std::cout << std::string(70, '=') << "\n\n";
300
301     std::cout << "1. consteval - Forced compile-time evaluation:\n";
302     std::cout << "    consteval int must_be_compile_time(int n) { return n * n;
303         }\n";
304     std::cout << "    square_5 = " << square_5 << " (computed at compile-time)\n
305         n";
306     std::cout << "    Guarantees no runtime computation\n";
307     std::cout << "    Perfect for embedded: zero runtime cost\n\n";
308
309     std::cout << "2. constinit - Compile-time init, runtime mutable:\n";
310     std::cout << "    constinit int error_count = 0;\n";
311     std::cout << "    Initial value: " << error_count << "\n";
312
313     // Can modify at runtime (unlike const/constexpr)
314     ++error_count;
315     std::cout << "    After increment: " << error_count << "\n";
316     std::cout << "    Initialized at compile-time (no static initialization
317         order issues)\n";
318     std::cout << "    Can be modified at runtime\n\n";
319
320     std::cout << "3. String literals in ROM:\n";
321     std::cout << "    constexpr std::string_view device_name = \"STM32F407\";\n
322         ";
323     std::cout << "    Device: " << device_name << "\n";
324     std::cout << "    String data in ROM, no heap allocation\n\n";
325
326     std::cout << "    Error message table in ROM:\n";
327     for (size_t i = 0; i < 4; ++i) {
328         std::cout << "        Error " << i << ": " << error_messages[i] << "\n";
329     }
330     std::cout << "    All strings in ROM (.rodata section)\n\n";
331
332     std::cout << "    consteval: Forces compile-time (zero runtime cost)\n";
333     std::cout << "    constinit: Safe global init, runtime mutable\n";
334     std::cout << "    string_view: String literals without heap\n";
335 }
336
337 } // namespace cpp20_features
338
339 // =====
340
341 // SECTION 5: Practical Embedded Example - Entire System Config in ROM

```

```
337 // =====
338
339 namespace embedded_example {
340
341 // Pin configuration
342 struct PinConfig {
343     uint8_t port;
344     uint8_t pin;
345     const char* function;
346
347     constexpr PinConfig(uint8_t p, uint8_t pn, const char* func)
348         : port(p), pin(pn), function(func) {}
349 };
350
351 // All pin configs in ROM
352 constexpr std::array<PinConfig, 8> pin_configs = {{
353     {0, 0, "UART_TX"},  

354     {0, 1, "UART_RX"},  

355     {0, 5, "SPI_SCK"},  

356     {0, 6, "SPI_MISO"},  

357     {0, 7, "SPI_MOSI"},  

358     {1, 0, "I2C_SCL"},  

359     {1, 1, "I2C_SDA"},  

360     {2, 13, "LED"}  

361 }};
362
363 // Timer configuration
364 struct TimerConfig {
365     uint32_t frequency_hz;
366     uint16_t prescaler;
367     uint16_t period;
368
369     constexpr TimerConfig(uint32_t freq)
370         : frequency_hz(freq),
371         prescaler(calculate_prescaler(freq)),
372         period(calculate_period(freq, calculate_prescaler(freq))) {}
373
374 private:
375     static constexpr uint16_t calculate_prescaler(uint32_t freq) {
376         uint32_t divisor = freq * 65536;
377         if (divisor == 0) return 1;
378         return static_cast<uint16_t>(84000000 / divisor + 1);
379     }
380
381     static constexpr uint16_t calculate_period(uint32_t freq, uint16_t
382         prescaler) {
383         if (freq == 0 || prescaler == 0) return 1000;
384         return static_cast<uint16_t>(84000000 / (freq * prescaler));
385     }
386 };
387 // Timer configs computed at compile-time, stored in ROM
```

```

388 constexpr TimerConfig pwm_timer{1000};           // 1 kHz PWM
389 constexpr TimerConfig adc_timer{10000};          // 10 kHz ADC sampling
390 constexpr TimerConfig led_timer{2};              // 2 Hz LED blink
391
392 // Memory map - register addresses
393 namespace registers {
394     constexpr uint32_t GPIO_BASE = 0x40020000;
395     constexpr uint32_t UART_BASE = 0x40011000;
396     constexpr uint32_t SPI_BASE = 0x40013000;
397     constexpr uint32_t I2C_BASE = 0x40005400;
398     constexpr uint32_t TIM_BASE = 0x40000000;
399 }
400
401 // Protocol constants
402 namespace can_protocol {
403     constexpr uint32_t CAN_ID_STATUS = 0x100;
404     constexpr uint32_t CAN_ID_SENSOR1 = 0x201;
405     constexpr uint32_t CAN_ID_SENSOR2 = 0x202;
406     constexpr uint32_t CAN_ID_COMMAND = 0x300;
407
408     constexpr std::array<uint32_t, 4> filter_ids = {
409         CAN_ID_STATUS, CAN_ID_SENSOR1, CAN_ID_SENSOR2, CAN_ID_COMMAND
410     };
411 }
412
413 void demonstrate() {
414     std::cout << "\n" << std::string(70, '=') << "\n";
415     std::cout << "==== SECTION 5: Embedded System - All Config in ROM ===\n";
416     std::cout << std::string(70, '=') << "\n\n";
417
418     std::cout << "1. Pin Configuration Table (ROM):\n";
419     for (const auto& pin : pin_configs) {
420         std::cout << "    Port " << static_cast<int>(pin.port)
421                     << ", Pin " << static_cast<int>(pin.pin)
422                     << ": " << pin.function << "\n";
423     }
424     std::cout << "    Size: " << sizeof(pin_configs) << " bytes in ROM\n\n";
425
426     std::cout << "2. Timer Configurations (computed at compile-time):\n";
427     std::cout << "    PWM Timer:\n";
428     std::cout << "        Frequency: " << pwm_timer.frequency_hz << " Hz\n";
429     std::cout << "        Prescaler: " << pwm_timer.prescaler << "\n";
430     std::cout << "        Period: " << pwm_timer.period << "\n\n";
431
432     std::cout << "    ADC Timer:\n";
433     std::cout << "        Frequency: " << adc_timer.frequency_hz << " Hz\n";
434     std::cout << "        Prescaler: " << adc_timer.prescaler << "\n";
435     std::cout << "        Period: " << adc_timer.period << "\n\n";
436
437     std::cout << "3. Register Addresses (ROM):\n";
438     std::cout << "    GPIO_BASE: 0x" << std::hex << registers::GPIO_BASE << "\n"
439                     "    ";
440     std::cout << "    UART_BASE: 0x" << registers::UART_BASE << "\n";
441     std::cout << "    SPI_BASE: 0x" << registers::SPI_BASE << "\n";

```

```
441     std::cout << "    I2C_BASE: 0x" << registers::I2C_BASE << std::dec << "\n\
442         n";
443
444     std::cout << "4. CAN Protocol IDs (ROM):\n";
445     for (size_t i = 0; i < can_protocol::filter_ids.size(); ++i) {
446         std::cout << "    Filter " << i << ": 0x" << std::hex
447                         << can_protocol::filter_ids[i] << std::dec << "\n";
448     }
449     std::cout << "\n EMBEDDED BENEFITS:\n";
450     std::cout << " • Zero RAM usage for configuration\n";
451     std::cout << " • All configs in cheap Flash ROM\n";
452     std::cout << " • Compile-time validation with static_assert\n";
453     std::cout << " • No runtime initialization overhead\n";
454     std::cout << " • Type-safe, no magic numbers\n";
455 }
456 } // namespace embedded_example
457
458 // =====
459 // SECTION 6: Verifying ROM Placement - Compiler Output
460 // =====
461
462 namespace verifying_placement {
463
464 // These should go to ROM (.rodata)
465 constexpr int rom_value = 42;
466 constexpr std::array<int, 10> rom_array = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
467 const char* const rom_string = "This is in ROM";
468
469 // These go to RAM (.data or .bss)
470 int ram_value = 42;
471 int ram_array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
472
473 void demonstrate() {
474     std::cout << "\n" << std::string(70, '=') << "\n";
475     std::cout << "==== SECTION 6: Verifying ROM Placement ====\n";
476     std::cout << std::string(70, '=') << "\n\n";
477
478     std::cout << "To verify ROM vs RAM placement, use these tools:\n\n";
479
480     std::cout << "1. Using 'nm' command:\n";
481     std::cout << "    $ nm ROMability | grep rom_value\n";
482     std::cout << "    Output shows section:\n";
483     std::cout << "        'r' or 'R' = .rodata (ROM)\n";
484     std::cout << "        'd' or 'D' = .data (RAM)\n";
485     std::cout << "        'b' or 'B' = .bss (uninitialized RAM)\n\n";
486
487     std::cout << "2. Using 'objdump' command:\n";
488     std::cout << "    $ objdump -h ROMability\n";
489     std::cout << "    Shows sections:\n";
```

```

490     std::cout << "      .rodata - Read-only data (ROM/Flash)\n";
491     std::cout << "      .data   - Initialized data (RAM)\n";
492     std::cout << "      .bss    - Uninitialized data (RAM)\n\n";
493
494     std::cout << "3. Using 'size' command:\n";
495     std::cout << "      $ size ROMability\n";
496     std::cout << "      Shows memory usage:\n";
497     std::cout << "      text   - Code + .rodata (ROM/Flash)\n";
498     std::cout << "      data   - Initialized data (RAM)\n";
499     std::cout << "      bss    - Uninitialized data (RAM)\n\n";
500
501     std::cout << "4. GCC-specific: __attribute__((section))\n";
502     std::cout << "      Can explicitly place data in specific section:\n";
503     std::cout << "      __attribute__((section(\".rodata\"))) const int x = 42;\n"
504     std::cout << "\n";
505
506     std::cout << "5. Embedded: Check linker map file\n";
507     std::cout << "      Linker script (*.ld) and map file show exact placement\n"
508     std::cout << "      ;\n";
509     std::cout << "      Critical for embedded systems with limited RAM\n\n";
510
511     std::cout << "      Best Practices:\n";
512     std::cout << "      • Use constexpr for guaranteed ROM placement\n";
513     std::cout << "      • Verify with nm/objdump during development\n";
514     std::cout << "      • Check map file for embedded targets\n";
515     std::cout << "      • Profile RAM usage on resource-constrained systems\n";
516 }
517
518 } // namespace verifying_placement
519
520 // =====
521
522 // SECTION 7: Summary and Best Practices
523 // =====
524
525
526
527 void show_summary() {
528     std::cout << "\n" << std::string(70, '=') << "\n";
529     std::cout << "==== ROM-ability in C++ - Complete Summary ===\n";
530     std::cout << std::string(70, '=') << "\n\n";
531
532     std::cout << "      KEYWORD          COMPILER-TIME      LOCATION      MUTABLE      \\\n";
533     std::cout << "      \n";
534     std::cout << "      \n";
535     std::cout << "      const           Maybe          ROM/RAM      No          \\\n";
536     std::cout << "      \n";
537     std::cout << "      constexpr       Yes           ROM          No          \\\n";
538     std::cout << "      \n";
539     std::cout << "      consteval      Must be       ROM          No          \\\n";
540     std::cout << "      \n";
541     std::cout << "      constinit      Yes (init)    RAM          Yes          \\\n";
542
543 }
```

```
      n";
534  std::cout << "                                \n\n";
535
536  std::cout << "WHEN TO USE WHAT:\n\n";
537
538  std::cout << "1. constexpr - Most common for ROM data\n";
539  std::cout << " •     Lookup tables (sine, gamma, CRC, etc.)\n";
540  std::cout << " •     Configuration structs\n";
541  std::cout << " •     Constants and arrays\n";
542  std::cout << " •     Compile-time computed values\n\n";
543
544  std::cout << "2. consteval (C++20) - Force compile-time\n";
545  std::cout << " •     Functions that MUST run at compile-time\n";
546  std::cout << " •     Zero runtime cost guarantee\n";
547  std::cout << " •     Configuration validation\n\n";
548
549  std::cout << "3. constinit (C++20) - Global variables\n";
550  std::cout << " •     Globals that need compile-time init\n";
551  std::cout << " •     Avoids static initialization order fiasco\n";
552  std::cout << " •     Runtime mutable (unlike constexpr)\n\n";
553
554  std::cout << "4. const - Use sparingly\n";
555  std::cout << " •     May or may not be ROM\n";
556  std::cout << " •     Prefer constexpr for ROM guarantee\n\n";
557
558  std::cout << "C++ ADVANTAGES OVER C:\n";
559  std::cout << "     constexpr functions - compute at compile-time\n";
560  std::cout << "     constexpr constructors - complex objects in ROM\n";
561  std::cout << "     std::array - type-safe ROM arrays\n";
562  std::cout << "     consteval - guarantee compile-time\n";
563  std::cout << "     Templates - generate ROM data\n";
564  std::cout << "     static_assert - compile-time validation\n\n";
565
566  std::cout << "EMBEDDED SYSTEMS BENEFITS:\n";
567  std::cout << "     Maximize ROM usage (cheap Flash)\n";
568  std::cout << "     Minimize RAM usage (expensive SRAM)\n";
569  std::cout << "     Zero runtime initialization overhead\n";
570  std::cout << "     Compile-time validation\n";
571  std::cout << "     No dynamic memory allocation\n";
572  std::cout << "     Deterministic memory layout\n\n";
573
574  std::cout << "BEST PRACTICES:\n";
575  std::cout << " 1. Use constexpr by default for constants\n";
576  std::cout << " 2. Generate lookup tables at compile-time\n";
577  std::cout << " 3. Store configuration in ROM\n";
578  std::cout << " 4. Use std::array instead of C arrays\n";
579  std::cout << " 5. Use string_view for ROM strings\n";
580  std::cout << " 6. Verify placement with nm/objdump\n";
581  std::cout << " 7. Profile RAM usage on embedded targets\n";
582 }
583
584 // =====
```

```
585 // MAIN FUNCTION
586 //
587 =====
588 int main() {
589     std::cout << "\n";
590     std::cout << "                                     \n";
591     std::cout << "                                     ROM-ability in C++ (constexpr & More)\n";
592     std::cout << "                                     \n";
593     std::cout << "                                     Placing Data in Read-Only Memory (.rodata)\n";
594     std::cout << "                                     \n";
595     std::cout << "                                     \n";
596     // Section 1: const vs constexpr
597     const_vs_constexpr::demonstrate();
598
599     // Section 2: ROM arrays
600     rom_arrays::demonstrate();
601
602     // Section 3: constexpr objects
603     constexpr_objects::demonstrate();
604
605     // Section 4: C++20 features
606     cpp20_features::demonstrate();
607
608     // Section 5: Embedded example
609     embedded_example::demonstrate();
610
611     // Section 6: Verification
612     verifying_placement::demonstrate();
613
614     // Section 7: Summary
615     show_summary();
616
617     std::cout << "\n" << std::string(70, '=') << "\n";
618     std::cout << "All demonstrations completed!\n";
619     std::cout << std::string(70, '=') << "\n\n";
620
621     std::cout << "KEY TAKEAWAY: Use constexpr for guaranteed ROM placement!\n";
622     ;
623     std::cout << "Perfect for embedded systems: maximize ROM, minimize RAM.\n\n";
624
625     return 0;
626 }
```

## 56 Source Code: RangesExamples.cpp

File: src/RangesExamples.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <ranges>
5 #include <algorithm>
6 #include <numeric>
7 #include <functional>
8
9 namespace ranges = std::ranges;
10 namespace views = std::views;
11
12 // =====
13 // 1. BASIC RANGES AND VIEWS
14 // =====
15 void example_basic_ranges() {
16     std::cout << "\n== 1. BASIC RANGES AND VIEWS ==" << std::endl;
17
18     std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
19
20     std::cout << "Original: ";
21     for (int n : numbers) {
22         std::cout << n << " ";
23     }
24     std::cout << std::endl;
25
26     // Using ranges::for_each
27     std::cout << "Using ranges::for_each: ";
28     ranges::for_each(numbers, [](int n) {
29         std::cout << n << " ";
30     });
31     std::cout << std::endl;
32 }
33
34 // =====
35 // 2. FILTER VIEW
36 // =====
37 void example_filter_view() {
38     std::cout << "\n== 2. FILTER VIEW ==" << std::endl;
39
40     std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
41
42     // Filter even numbers
43     auto even_numbers = numbers | views::filter([](int n) { return n % 2 == 0; });
44
45     std::cout << "Even numbers: ";
46     for (int n : even_numbers) {
47         std::cout << n << " ";
48     }
```

```
49     std::cout << std::endl;
50
51     // Filter numbers greater than 5
52     auto greater_than_5 = numbers | views::filter([](int n) { return n > 5; })
53     ;
54
55     std::cout << "Greater than 5: ";
56     for (int n : greater_than_5) {
57         std::cout << n << " ";
58     }
59     std::cout << std::endl;
60 }
61 // =====
62 // 3. TRANSFORM VIEW
63 // =====
64 void example_transform_view() {
65     std::cout << "\n== 3. TRANSFORM VIEW ==" << std::endl;
66
67     std::vector<int> numbers = {1, 2, 3, 4, 5};
68
69     // Square each number
70     auto squared = numbers | views::transform([](int n) { return n * n; });
71
72     std::cout << "Squared: ";
73     for (int n : squared) {
74         std::cout << n << " ";
75     }
76     std::cout << std::endl;
77
78     // Convert to strings
79     auto strings = numbers | views::transform([](int n) { return std::
80         to_string(n); });
81
82     std::cout << "As strings: ";
83     for (const auto& s : strings) {
84         std::cout << " " << s << " ";
85     }
86     std::cout << std::endl;
87 }
88 // =====
89 // 4. TAKE AND DROP VIEWS
90 // =====
91 void example_take_drop_views() {
92     std::cout << "\n== 4. TAKE AND DROP VIEWS ==" << std::endl;
93
94     std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
95
96     // Take first 5 elements
97     auto first_five = numbers | views::take(5);
98     std::cout << "First 5: ";
99     for (int n : first_five) {
100         std::cout << n << " ";
```

```
101     }
102     std::cout << std::endl;
103
104     // Drop first 5 elements
105     auto skip_five = numbers | views::drop(5);
106     std::cout << "Skip 5: ";
107     for (int n : skip_five) {
108         std::cout << n << " ";
109     }
110     std::cout << std::endl;
111
112     // Take while condition is true
113     auto take_while_small = numbers | views::take_while([](int n) { return n <
114         6; });
115     std::cout << "Take while < 6: ";
116     for (int n : take_while_small) {
117         std::cout << n << " ";
118     }
119     std::cout << std::endl;
120 }
121 // =====
122 // 5. COMPOSING VIEWS
123 // =====
124 void example_composing_views() {
125     std::cout << "\n== 5. COMPOSING VIEWS ==" << std::endl;
126
127     std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
128
129     // Chain multiple operations: filter even, square, take first 3
130     auto result = numbers
131         | views::filter([](int n) { return n % 2 == 0; })
132         | views::transform([](int n) { return n * n; })
133         | views::take(3);
134
135     std::cout << "Even, squared, first 3: ";
136     for (int n : result) {
137         std::cout << n << " ";
138     }
139     std::cout << std::endl;
140 }
141 // =====
142 // 6. REVERSE VIEW
143 // =====
144 void example_reverse_view() {
145     std::cout << "\n== 6. REVERSE VIEW ==" << std::endl;
146
147     std::vector<int> numbers = {1, 2, 3, 4, 5};
148
149     auto reversed = numbers | views::reverse;
150
151     std::cout << "Reversed: ";
152     for (int n : reversed) {
```

```
154         std::cout << n << " ";
155     }
156     std::cout << std::endl;
157 }
158
159 // =====
160 // 7. IOTA VIEW (GENERATE SEQUENCE)
161 // =====
162 void example_iota_view() {
163     std::cout << "\n== 7. IOTA VIEW (GENERATE SEQUENCE) ==" << std::endl;
164
165     // Generate numbers from 1 to 10
166     auto range1 = views::iota(1, 11);
167     std::cout << "Iota 1 to 10: ";
168     for (int n : range1) {
169         std::cout << n << " ";
170     }
171     std::cout << std::endl;
172
173     // Infinite range (take first 5)
174     auto infinite_range = views::iota(1) | views::take(5);
175     std::cout << "Infinite iota, take 5: ";
176     for (int n : infinite_range) {
177         std::cout << n << " ";
178     }
179     std::cout << std::endl;
180 }
181
182 // =====
183 // 8. SPLIT VIEW
184 // =====
185 void example_split_view() {
186     std::cout << "\n== 8. SPLIT VIEW ==" << std::endl;
187
188     std::string text = "Hello World C++ Ranges";
189
190     auto words = text | views::split(' ');
191
192     std::cout << "Split by space:" << std::endl;
193     for (const auto& word : words) {
194         std::cout << " ";
195         for (char c : word) {
196             std::cout << c;
197         }
198         std::cout << " " << std::endl;
199     }
200 }
201
202 // =====
203 // 9. JOIN VIEW
204 // =====
205 void example_join_view() {
206     std::cout << "\n== 9. JOIN VIEW ==" << std::endl;
```

```
208     std::vector<std::vector<int>> nested = {
209         {1, 2, 3},
210         {4, 5},
211         {6, 7, 8, 9}
212     };
213
214     auto flattened = nested | views::join;
215
216     std::cout << "Flattened: ";
217     for (int n : flattened) {
218         std::cout << n << " ";
219     }
220     std::cout << std::endl;
221 }
222
223 // =====
224 // 10. KEYS AND VALUES VIEWS (WITH MAP)
225 // =====
226 void example_keys_values() {
227     std::cout << "\n==== 10. KEYS AND VALUES VIEWS ===" << std::endl;
228
229     std::vector<std::pair<std::string, int>> data = {
230         {"Alice", 30},
231         {"Bob", 25},
232         {"Charlie", 35}
233     };
234
235     auto keys = data | views::keys;
236     std::cout << "Keys: ";
237     for (const auto& key : keys) {
238         std::cout << key << " ";
239     }
240     std::cout << std::endl;
241
242     auto values = data | views::values;
243     std::cout << "Values: ";
244     for (int value : values) {
245         std::cout << value << " ";
246     }
247     std::cout << std::endl;
248 }
249
250 // =====
251 // 11. RANGES ALGORITHMS
252 // =====
253 void example_ranges_algorithms() {
254     std::cout << "\n==== 11. RANGES ALGORITHMS ===" << std::endl;
255
256     std::vector<int> numbers = {5, 2, 8, 1, 9, 3, 7, 4, 6};
257
258     // Sort using ranges
259     ranges::sort(numbers);
260     std::cout << "Sorted: ";
261     for (int n : numbers) {
```

```
262         std::cout << n << " ";
263     }
264     std::cout << std::endl;
265
266     // Find element
267     auto it = ranges::find(numbers, 5);
268     if (it != numbers.end()) {
269         std::cout << "Found 5 at position: " << std::distance(numbers.begin(), it) << std::endl;
270     }
271
272     // Count elements greater than 5
273     auto count = ranges::count_if(numbers, [](int n) { return n > 5; });
274     std::cout << "Elements > 5: " << count << std::endl;
275 }
276
277 // =====
278 // 12. COMMON VIEW
279 // =====
280 void example_common_view() {
281     std::cout << "\n== 12. COMMON VIEW ==" << std::endl;
282
283     auto range = views::iota(1, 11) | views::filter([](int n) { return n % 2
284         == 0; });
285
286     // Convert to common range (can use with legacy algorithms)
287     auto common_range = range | views::common;
288
289     std::cout << "Common view result: ";
290     for (int n : common_range) {
291         std::cout << n << " ";
292     }
293     std::cout << std::endl;
294 }
295
296 // =====
297 // 13. ENUMERATE VIEW (C++23 preview style)
298 // =====
299 void example_enumerate_style() {
300     std::cout << "\n== 13. ENUMERATE STYLE ==" << std::endl;
301
302     std::vector<std::string> fruits = {"Apple", "Banana", "Cherry", "Date"};
303
304     // Manual enumeration with iota and zip_transform style
305     int index = 0;
306     for (const auto& fruit : fruits) {
307         std::cout << index++ << ":" << fruit << std::endl;
308     }
309
310 // =====
311 // 14. COMPLEX PIPELINE
312 // =====
313 void example_complex_pipeline() {
```

```
314     std::cout << "\n==== 14. COMPLEX PIPELINE ===" << std::endl;
315
316     // Generate numbers, filter, transform, and take
317     auto pipeline = views::iota(1, 101)
318         | views::filter([](int n) { return n % 3 == 0; }) // Divisible by 3
319         | views::transform([](int n) { return n * n; }) // Square
320         | views::take(5); // First 5
321
322     std::cout << "First 5 squares of numbers divisible by 3:" << std::endl;
323     for (int n : pipeline) {
324         std::cout << n << " ";
325     }
326     std::cout << std::endl;
327 }
328
329 // =====
330 // 15. CONVERTING VIEWS TO CONTAINERS
331 // =====
332 void example_to_container() {
333     std::cout << "\n==== 15. CONVERTING VIEWS TO CONTAINERS ===" << std::endl;
334
335     auto view = views::iota(1, 11)
336         | views::filter([](int n) { return n % 2 == 0; })
337         | views::transform([](int n) { return n * 2; });
338
339     // Convert to vector
340     std::vector<int> result(view.begin(), view.end());
341
342     std::cout << "Converted to vector: ";
343     for (int n : result) {
344         std::cout << n << " ";
345     }
346     std::cout << std::endl;
347 }
348
349 // =====
350 // MAIN FUNCTION
351 // =====
352 int main() {
353     std::cout << "\n====" << std::endl;
354     std::cout << "    C++20 RANGES EXAMPLES" << std::endl;
355     std::cout << "====" << std::endl;
356
357     example_basic_ranges();
358     example_filter_view();
359     example_transform_view();
360     example_take_drop_views();
361     example_composing_views();
362     example_reverse_view();
363     example_iota_view();
364     example_split_view();
365     example_join_view();
```

```
366     example_keys_values_views();
367     example_ranges_algorithms();
368     example_common_view();
369     example_enumerate_style();
370     example_complex_pipeline();
371     example_to_container();
372
373     std::cout << "\n======" << std::endl;
374     std::cout << "      ALL EXAMPLES COMPLETED" << std::endl;
375     std::cout << "=====\\n" << std::endl;
376
377     return 0;
378 }
```

## 57 Source Code: RealTimeProgramming.cpp

File: src/RealTimeProgramming.cpp

Repository: [View on GitHub](#)

```
1 // RealTimeProgramming.cpp
2 // Comprehensive guide to Real-Time Programming in Modern C++
3 //
4 // REAL-TIME SYSTEM CHARACTERISTICS:
5 // 1. Deterministic behavior - predictable execution times
6 // 2. Bounded worst-case execution time (WCET)
7 // 3. Low and predictable latency
8 // 4. No dynamic memory allocation in critical paths
9 // 5. Avoid operations with unbounded execution time
10 //
11 // KEY TOPICS COVERED:
12 // • Big O Notation and Time Complexity
13 // • STL containers for real-time systems
14 // • std::bitset vs std::vector<bool>
15 // • Memory pre-allocation strategies
16 // • RAII and deterministic resource management
17 // • Avoiding heap fragmentation
18 // • Custom allocators and memory pools
19
20 #include <iostream>
21 #include <vector>
22 #include <array>
23 #include <list>
24 #include <bitset>
25 #include <chrono>
26 #include <algorithm>
27 #include <memory>
28 #include <functional>
29 #include <string>
30 #include <string_view>
31 #include <optional>
32 #include <variant>
33 #include <iomanip>
34 #include <cstdint>
35
36 // =====
37 // SECTION 1: Big O Notation - Understanding Time Complexity
38 // =====
39
40 namespace big_o_notation {
41
42 void demonstrate_01() {
43     std::cout << "\nO(1) - Constant Time:\n";
44     std::cout << " • Execution time does not depend on input size\n";
45     std::cout << " • Always takes the same amount of time\n\n";
```

```
46
47     std::array<int, 1000> data;
48     data.fill(42);
49
50     auto start = std::chrono::high_resolution_clock::now();
51
52     // Array/vector element access - O(1)
53     [[maybe_unused]] int value = data[500];
54
55     // Hash table lookup - O(1) average case
56     // Stack push/pop - O(1)
57
58     auto end = std::chrono::high_resolution_clock::now();
59     auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
60     start);
61
62     std::cout << " Example: Array element access data[500]\n";
63     std::cout << " Time: " << duration.count() << " ns (constant, regardless
64     of array size)\n";
65     std::cout << "     BEST for real-time systems - predictable!\n";
66 }
67
68 void demonstrate_Ologn() {
69     std::cout << "\nO(log n) - Logarithmic Time:\n";
70     std::cout << " • Execution time grows logarithmically with input size\n";
71     std::cout << " • Doubling input size adds constant time\n\n";
72
73     std::vector<int> data(1000);
74     for (size_t i = 0; i < data.size(); ++i) {
75         data[i] = static_cast<int>(i);
76     }
77
78     auto start = std::chrono::high_resolution_clock::now();
79
80     // Binary search - O(log n)
81     bool found = std::binary_search(data.begin(), data.end(), 500);
82
83     auto end = std::chrono::high_resolution_clock::now();
84     auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
85     start);
86
87     std::cout << " Example: Binary search in sorted array of 1000 elements\n";
88     std::cout << "     Time: " << duration.count() << " ns\n";
89     std::cout << "     Found: " << std::boolalpha << found << "\n";
90     std::cout << "     Note: For 1M elements, time increases by only ~10x\n";
91     std::cout << "     ACCEPTABLE for real-time systems with bounded size\n";
92 }
93
94 void demonstrate_0n() {
95     std::cout << "\nO(n) - Linear Time:\n";
96     std::cout << " • Execution time grows linearly with input size\n";
97     std::cout << " • Doubling input doubles execution time\n\n";
```

```

96     std::vector<int> data(1000);
97     for (size_t i = 0; i < data.size(); ++i) {
98         data[i] = static_cast<int>(i);
99     }
100
101    auto start = std::chrono::high_resolution_clock::now();
102
103    // Linear search - O(n)
104    auto it = std::find(data.begin(), data.end(), 500);
105
106    auto end = std::chrono::high_resolution_clock::now();
107    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
108        start);
109
110    std::cout << "    Example: Linear search through 1000 elements\n";
111    std::cout << "    Time: " << duration.count() << " ns\n";
112    std::cout << "    Found: " << (it != data.end()) << "\n";
113    std::cout << "    USE WITH CAUTION - bound the maximum container size!\n";
114
115 void demonstrate_0nlogn() {
116     std::cout << "\nO(n log n) - Linearithmic Time:\n";
117     std::cout << " • Execution time grows at n * log(n)\n";
118     std::cout << " • Common in efficient sorting algorithms\n\n";
119
120     std::vector<int> data(1000);
121     for (size_t i = 0; i < data.size(); ++i) {
122         data[i] = 1000 - static_cast<int>(i); // Reverse order
123     }
124
125     auto start = std::chrono::high_resolution_clock::now();
126
127     // Sorting - O(n log n) for std::sort
128     std::sort(data.begin(), data.end());
129
130     auto end = std::chrono::high_resolution_clock::now();
131     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end -
132         start);
133
134     std::cout << "    Example: std::sort on 1000 elements\n";
135     std::cout << "    Time: " << duration.count() << " s\n";
136     std::cout << "    AVOID in time-critical paths - do in initialization
137         phase\n";
138 }
139
140 void demonstrate() {
141     std::cout << "\n" << std::string(70, '=') << "\n";
142     std::cout << "==== SECTION 1: Big O Notation - Time Complexity ===\n";
143     std::cout << std::string(70, '=') << "\n";
144
145     demonstrate_01();
146     demonstrate_0logn();
147     demonstrate_0n();
148     demonstrate_0nlogn();

```



```
189     }
190
191     // REAL-TIME THREAD: O(1) operations only!
192     void process_sensor_reading_RT(size_t sensor_id, double value) noexcept {
193         // O(1) - Store latest reading
194         latest_readings_[sensor_id] = value;
195
196         // O(1) - Update status flag
197         sensor_status_.set(sensor_id);
198
199         // O(1) - No complex processing here!
200     }
201
202     // REAL-TIME THREAD: O(1) read access
203     [[nodiscard]] double get_latest_reading_RT(size_t sensor_id) const
204         noexcept {
205         return latest_readings_[sensor_id]; // O(1)
206     }
207
208     // NON-REAL-TIME THREAD: Can use O(n) operations
209     void log_historical_data_NonRT() {
210         // O(n) - Iterate and log (acceptable in background thread)
211         for (size_t i = 0; i < latest_readings_.size(); ++i) {
212             if (sensor_status_.test(i)) {
213                 historical_data_.push_back(latest_readings_[i]);
214             }
215         }
216     }
217
218     // NON-REAL-TIME THREAD: Can use O(n) operations
219     [[nodiscard]] double calculate_average_NonRT() const noexcept {
220         if (historical_data_.empty()) return 0.0;
221
222         // O(n) - Acceptable in non-RT thread
223         double sum = 0.0;
224         for (double val : historical_data_) {
225             sum += val;
226         }
227         return sum / historical_data_.size();
228     }
229
230     // NON-REAL-TIME THREAD: Can use O(n log n) operations
231     [[nodiscard]] double calculate_median_NonRT() {
232         if (historical_data_.empty()) return 0.0;
233
234         // O(n log n) - Sorting is expensive, do in background!
235         auto data_copy = historical_data_; // Copy to avoid modifying
236             original
237         std::sort(data_copy.begin(), data_copy.end());
238
239         size_t mid = data_copy.size() / 2;
240         return data_copy[mid];
241     }
242 };
```

```
241
242 void demonstrate() {
243     std::cout << "\n" << std::string(70, '=') << "\n";
244     std::cout << "==== SECTION 1.5: Real-Time Thread Architecture ===\n";
245     std::cout << std::string(70, '=') << "\n\n";
246
247     std::cout << "THREAD SEPARATION PRINCIPLE:\n\n";
248
249     std::cout << "REAL-TIME THREADS:\n";
250     std::cout << " • Purpose: Control loops, sensor processing, motor control
251     \n";
252     std::cout << " • Requirements: Deterministic, predictable timing\n";
253     std::cout << " • Allowed: O(1) operations ONLY\n";
254     std::cout << " • Examples:\n";
255     std::cout << "     - Reading sensor value from array\n";
256     std::cout << "     - Updating control output\n";
257     std::cout << "     - Checking/setting status flags (bitset)\n";
258     std::cout << "     - Simple arithmetic calculations\n\n";
259
260     std::cout << "NON-REAL-TIME THREADS:\n";
261     std::cout << " • Purpose: Logging, diagnostics, data aggregation, UI
262     updates\n";
263     std::cout << " • Requirements: Eventually complete, no strict deadlines\n
264     ";
265     std::cout << " • Allowed: O(1), O(log n), O(n) operations\n";
266     std::cout << " • Examples:\n";
267     std::cout << "     - Writing logs to disk\n";
268     std::cout << "     - Calculating statistics (average, median)\n";
269     std::cout << "     - Searching historical data\n";
270     std::cout << "     - Generating reports\n\n";
271
272     std::cout << "EXAMPLE: Sensor System with Thread Separation\n\n";
273
274     SensorSystem system;
275
276     // Simulate real-time thread
277     std::cout << "  REAL-TIME THREAD (1 kHz control loop):\n";
278     auto rt_start = std::chrono::high_resolution_clock::now();
279
280     for (size_t i = 0; i < 10; ++i) {
281         system.process_sensor_reading_RT(i % 10, i * 0.5); // O(1) per
282         reading
283     }
284
285     auto rt_end = std::chrono::high_resolution_clock::now();
286     auto rt_duration = std::chrono::duration_cast<std::chrono::nanoseconds>(
287         rt_end - rt_start);
288
289     std::cout << "      Processed 10 readings in " << rt_duration.count() << "
290     ns\n";
291     std::cout << "      Per-reading time: " << (rt_duration.count() / 10) << "
292     ns (deterministic!)\n\n";
293
294     // Simulate non-real-time thread
```

```
288     std::cout << "  NON-REAL-TIME THREAD (background processing):\n";
289
290     system.log_historical_data_NonRT(); // O(n) - acceptable
291
292     auto bg_start = std::chrono::high_resolution_clock::now();
293     double avg = system.calculate_average_NonRT(); // O(n)
294     auto bg_end = std::chrono::high_resolution_clock::now();
295     auto bg_duration = std::chrono::duration_cast<std::chrono::microseconds>(
296         bg_end - bg_start);
297
298     std::cout << "      Calculated average: " << avg << "\n";
299     std::cout << "      Time: " << bg_duration.count() << " s (non-critical!)\n"
300     \n";
301
302     std::cout << "      ARCHITECTURE: Real-time thread stays O(1), background
303         handles O(n)\n";
304     std::cout << "      BENEFIT: Control loop remains deterministic and fast\n";
305     std::cout << "      BENEFIT: Complex processing doesn't block time-critical
306         operations\n";
307 }
308
309 } // namespace thread_architecture
310
311 //=====
312
313 // SECTION 2: std::bitset vs std::vector<bool> - The Better Choice
314 //=====
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
```

```
334     constexpr void set_flag(size_t index) noexcept {
335         if (index < N) {
336             flags_.set(index);
337         }
338     }
339
340     // O(1) - Clear a flag
341     constexpr void clear_flag(size_t index) noexcept {
342         if (index < N) {
343             flags_.reset(index);
344         }
345     }
346
347     // O(1) - Toggle a flag
348     constexpr void toggle_flag(size_t index) noexcept {
349         if (index < N) {
350             flags_.flip(index);
351         }
352     }
353
354     // O(1) - Check a flag
355     [[nodiscard]] constexpr bool is_set(size_t index) const noexcept {
356         return index < N ? flags_.test(index) : false;
357     }
358
359     // O(1) - Count set bits (hardware instruction on modern CPUs)
360     [[nodiscard]] constexpr size_t count() const noexcept {
361         return flags_.count();
362     }
363
364     // O(1) - Check if any flags are set
365     [[nodiscard]] constexpr bool any() const noexcept {
366         return flags_.any();
367     }
368
369     // O(1) - Check if all flags are set
370     [[nodiscard]] constexpr bool all() const noexcept {
371         return flags_.all();
372     }
373
374     // O(1) - Check if no flags are set
375     [[nodiscard]] constexpr bool none() const noexcept {
376         return flags_.none();
377     }
378
379     // Bitwise operations - O(1)
380     constexpr RealTimeFlagManager operator&(const RealTimeFlagManager& other)
381         const noexcept {
382         RealTimeFlagManager result;
383         result.flags_ = flags_ & other.flags_;
384         return result;
385     }
386
387     constexpr RealTimeFlagManager operator|(const RealTimeFlagManager& other)
```

```
    const noexcept {
387     RealTimeFlagManager result;
388     result.flags_ = flags_ | other.flags_;
389     return result;
390   }
391 }
392
393 // Example: Sensor status flags for embedded system
394 constexpr size_t NUM_SENSORS = 32;
395
396 enum class SensorStatus : size_t {
397   TEMPERATURE_OK = 0,
398   PRESSURE_OK = 1,
399   HUMIDITY_OK = 2,
400   VOLTAGE_OK = 3,
401   CURRENT_OK = 4,
402   GPS_LOCK = 5,
403   IMU_CALIBRATED = 6,
404   COMMUNICATION_OK = 7,
405   // ... up to 32 sensors
406 };
407
408 void demonstrate() {
409   std::cout << "\n" << std::string(70, '=') << "\n";
410   std::cout << "==== SECTION 2: std::bitset vs std::vector<bool> ===\n";
411   std::cout << std::string(70, '=') << "\n\n";
412
413   std::cout << "WHY std::vector<bool> IS PROBLEMATIC:\n";
414   std::cout << "  Space-optimized specialization (1 bit per element)\n";
415   std::cout << "  operator[] returns proxy object, not bool&\n";
416   std::cout << "  Cannot take address of elements\n";
417   std::cout << "  Non-standard container behavior\n";
418   std::cout << "  Dynamic memory allocation\n";
419   std::cout << "  Unpredictable performance\n\n";
420
421   std::cout << "WHY std::bitset IS BETTER FOR REAL-TIME:\n";
422   std::cout << "  Fixed size at compile time (no heap allocation)\n";
423   std::cout << "  O(1) access and modification\n";
424   std::cout << "  Efficient bitwise operations (AND, OR, XOR)\n";
425   std::cout << "  Hardware-optimized bit counting\n";
426   std::cout << "  Standard, predictable behavior\n";
427   std::cout << "  Deterministic performance\n\n";
428
429 // Demonstrate real-time flag manager
430 RealTimeFlagManager<NUM_SENSORS> sensor_status;
431
432 std::cout << "EXAMPLE: Real-Time Sensor Status Manager\n";
433 std::cout << "  Using std::bitset<32> for 32 sensor flags\n\n";
434
435 // Set some flags
436 sensor_status.set_flag(static_cast<size_t>(SensorStatus::TEMPERATURE_OK));
437 sensor_status.set_flag(static_cast<size_t>(SensorStatus::PRESSURE_OK));
438 sensor_status.set_flag(static_cast<size_t>(SensorStatus::GPS_LOCK));
439
```

```

440     std::cout << "  Set flags: TEMPERATURE_OK, PRESSURE_OK, GPS_LOCK\n";
441     std::cout << "  Active sensors: " << sensor_status.count() << " / " <<
442         NUM_SENSORS << "\n";
443     std::cout << "  All sensors OK? " << std::boolalpha << sensor_status.all()
444         << "\n";
445     std::cout << "  Any sensor OK? " << sensor_status.any() << "\n";
446
447     // Check specific flag
448     bool temp_ok = sensor_status.is_set(static_cast<size_t>(SensorStatus::
449         TEMPERATURE_OK));
450     std::cout << "  Temperature sensor OK? " << temp_ok << "\n\n";
451
452     // Timing comparison
453     constexpr size_t ITERATIONS = 1000000;
454
455     std::bitset<64> bitset_flags;
456     auto start_bitset = std::chrono::high_resolution_clock::now();
457     for (size_t i = 0; i < ITERATIONS; ++i) {
458         bitset_flags.set(i % 64);
459         [[maybe_unused]] bool val = bitset_flags.test(i % 64);
460     }
461     auto end_bitset = std::chrono::high_resolution_clock::now();
462     auto bitset_duration = std::chrono::duration_cast<std::chrono::
463         microseconds>(end_bitset - start_bitset);
464
465     std::vector<bool> vector_bool(64, false);
466     auto start_vector = std::chrono::high_resolution_clock::now();
467     for (size_t i = 0; i < ITERATIONS; ++i) {
468         vector_bool[i % 64] = true;
469         [[maybe_unused]] bool val = vector_bool[i % 64];
470     }
471     auto end_vector = std::chrono::high_resolution_clock::now();
472     auto vector_duration = std::chrono::duration_cast<std::chrono::
473         microseconds>(end_vector - start_vector);
474
475     std::cout << "PERFORMANCE COMPARISON (" << ITERATIONS << " operations):\n"
476         ;
477     std::cout << "  std::bitset<64>:      " << std::setw(8) << bitset_duration.
478         count() << " s\n";
479     std::cout << "  std::vector<bool>:    " << std::setw(8) << vector_duration.
480         count() << " s\n";
481     std::cout << "  Speedup:           " << std::fixed << std::setprecision
482         (2)
483             << (static_cast<double>(vector_duration.count()) /
484                 bitset_duration.count()) << "x\n\n";
485
486     std::cout << "  RECOMMENDATION: Use std::bitset for real-time flag
487         management!\n";
488 }
489 } // namespace bitset_vs_vector_bool
490
491 // =====

```

```
482 // SECTION 3: STL Containers - Real-Time Best Practices
483 //
484 //=====
485
486 namespace stl_containers_realtime {
487
488 // Container Complexity Reference:
489 //
490 // std::vector:
491 // - Access: O(1)
492 // - Insert front: O(n)
493 // - Insert back: O(1) amortized, O(n) worst case (reallocation!)
494 // - Insert mid: O(n)
495 // - Real-time: Use reserve() to pre-allocate!
496 //
497 // std::array:
498 // - Access: O(1)
499 // - Size: Fixed at compile time
500 // - Real-time: PERFECT - no dynamic allocation, stack-based
501 //
502 // std::deque:
503 // - Access: O(1)
504 // - Insert front: O(1)
505 // - Insert back: O(1)
506 // - Real-time: Acceptable if size bounded
507 //
508 // std::list:
509 // - Access: O(n) - must traverse!
510 // - Insert: O(1) if you have iterator
511 // - Real-time: AVOID - dynamic allocation per element, poor cache
512 //
513 // std::map/std::set:
514 // - Access: O(log n)
515 // - Insert: O(log n)
516 // - Real-time: Use with custom allocator or bounded size
517
518 // Example 1: Pre-allocated std::vector - The Right Way
519 class RealTimeDataBuffer {
520     private:
521         static constexpr size_t MAX_SAMPLES = 1000;
522         std::vector<double> samples_;
523
524     public:
525         RealTimeDataBuffer() {
526             // CRITICAL: Pre-allocate during initialization
527             // This prevents reallocation during real-time operation
528             samples_.reserve(MAX_SAMPLES);
529
530             std::cout << " Buffer created with capacity: " << samples_.capacity()
531                         << "\n";
532             std::cout << " No reallocation will occur until this limit\n";
533         }
534 }
```

```
532 // O(1) - Add sample (no reallocation if within capacity)
533 bool add_sample(double value) noexcept {
534     if (samples_.size() < MAX_SAMPLES) {
535         samples_.push_back(value); // O(1) - deterministic!
536         return true;
537     }
538     return false; // Buffer full
539 }
540
541 // O(1) - Get sample
542 [[nodiscard]] std::optional<double> get_sample(size_t index) const
543     noexcept {
544         if (index < samples_.size()) {
545             return samples_[index];
546         }
547         return std::nullopt;
548     }
549
550 // O(n) - Clear buffer (but no deallocation!)
551 void clear() noexcept {
552     samples_.clear();
553     // Capacity remains unchanged - memory is retained!
554 }
555
556 [[nodiscard]] size_t size() const noexcept { return samples_.size(); }
557 [[nodiscard]] size_t capacity() const noexcept { return samples_.capacity();
558 }
559
560 // Example 2: Fixed-size std::array - Even Better
561 template<size_t N>
562 class CircularBuffer {
563 private:
564     std::array<double, N> buffer_;
565     size_t head_ = 0;
566     size_t count_ = 0;
567
568 public:
569     // O(1) - Add element (overwrites oldest if full)
570     constexpr void push(double value) noexcept {
571         buffer_[head_] = value;
572         head_ = (head_ + 1) % N;
573         if (count_ < N) {
574             ++count_;
575         }
576     }
577
578     // O(1) - Get element (0 = most recent)
579     [[nodiscard]] constexpr std::optional<double> get(size_t index) const
580         noexcept {
581         if (index >= count_) {
582             return std::nullopt;
583         }
584     }
585 }
```

```

583     size_t actual_index = (head_ + N - count_ + index) % N;
584     return buffer_[actual_index];
585 }
586
587 [[nodiscard]] constexpr size_t size() const noexcept { return count_; }
588 [[nodiscard]] constexpr bool is_full() const noexcept { return count_ == N;
589     ; }
590
591 // Statistics - O(n) but bounded by N
592 [[nodiscard]] double average() const noexcept {
593     if (count_ == 0) return 0.0;
594     double sum = 0.0;
595     for (size_t i = 0; i < count_; ++i) {
596         size_t idx = (head_ + N - count_ + i) % N;
597         sum += buffer_[idx];
598     }
599     return sum / count_;
600 }
601
602 void demonstrate() {
603     std::cout << "\n" << std::string(70, '=') << "\n";
604     std::cout << "==== SECTION 3: STL Containers - Real-Time Best Practices
605         ==\n";
606     std::cout << std::string(70, '=') << "\n\n";
607
608     std::cout << "CONTAINER SELECTION FOR REAL-TIME SYSTEMS:\n\n";
609
610     std::cout << "  PREFERRED:\n";
611     std::cout << "  1. std::array<T, N>           - O(1) access, no heap,
612         compile-time size\n";
613     std::cout << "  2. std::vector<T> + reserve() - O(1) access, pre-allocated
614         \n";
615     std::cout << "  3. std::bitset<N>           - O(1) bit ops, no heap\n";
616     std::cout << "  4. std::string_view         - O(1) string viewing, no
617         allocation\n\n";
618
619     std::cout << "  USE WITH CAUTION (bound size!):\n";
620     std::cout << "  • std::deque<T>           - O(1) both ends, but
621         allocates chunks\n";
622     std::cout << "  • std::map/set<T>         - O(log n), allocates per
623         element\n\n";
624
625     std::cout << "  AVOID:\n";
626     std::cout << "  • std::list<T>           - Poor cache locality, alloc
627         per element\n";
628     std::cout << "  • std::forward_list<T>     - Poor cache locality\n";
629     std::cout << "  • std::vector<bool>        - Use std::bitset instead!\n
630         ";
631     std::cout << "  • Unbounded growth containers - Always set max size!\n\n";
632
633 // Demonstrate pre-allocated vector
634 std::cout << "EXAMPLE 1: Pre-allocated std::vector\n";
635 RealTimeDataBuffer buffer;

```

```

628
629     for (int i = 0; i < 100; ++i) {
630         buffer.add_sample(i * 0.1);
631     }
632
633     std::cout << "    Added 100 samples\n";
634     std::cout << "    Current size: " << buffer.size() << "\n";
635     std::cout << "    Capacity: " << buffer.capacity() << "\n";
636     std::cout << "    No reallocation occurred - deterministic O(1) insertion
637         !\n\n";
638
639 // Demonstrate fixed-size array
640 std::cout << "EXAMPLE 2: Fixed-size circular buffer (std::array<T, N>)\n";
641 CircularBuffer<10> circ_buffer;
642
643     for (int i = 0; i < 15; ++i) {
644         circ_buffer.push(i * 1.5);
645     }
646
647     std::cout << "    Pushed 15 values into buffer of size 10\n";
648     std::cout << "    Current size: " << circ_buffer.size() << "\n";
649     std::cout << "    Average: " << circ_buffer.average() << "\n";
650     std::cout << "    No heap allocation - pure stack-based, deterministic!\n\n";
651
652 // Demonstrate reserve() importance
653 std::cout << "CRITICAL: Always use reserve() for std::vector in real-time
654         code!`\n\n";
655
656     std::vector<int> bad_vector; // No reserve
657     auto start_bad = std::chrono::high_resolution_clock::now();
658     for (int i = 0; i < 1000; ++i) {
659         bad_vector.push_back(i); // May trigger reallocations!
660     }
661     auto end_bad = std::chrono::high_resolution_clock::now();
662     auto bad_duration = std::chrono::duration_cast<std::chrono::microseconds>(
663         end_bad - start_bad);
664
665     std::vector<int> good_vector;
666     good_vector.reserve(1000); // Pre-allocate
667     auto start_good = std::chrono::high_resolution_clock::now();
668     for (int i = 0; i < 1000; ++i) {
669         good_vector.push_back(i); // No reallocations!
670     }
671     auto end_good = std::chrono::high_resolution_clock::now();
672     auto good_duration = std::chrono::duration_cast<std::chrono::microseconds>(
673         end_good - start_good);
674
675     std::cout << "    Without reserve(): " << bad_duration.count() << " s (
676         unpredictable!)\n";
677     std::cout << "    With reserve():      " << good_duration.count() << " s (
678         deterministic!)\n";
679     std::cout << "    Speedup:           " << std::fixed << std::setprecision(2)
680         << (static_cast<double>(bad_duration.count()) / good_duration.

```

```
        count()) << "x\n\n";  
675  
676     std::cout << "    GOLDEN RULE: reserve() prevents non-deterministic  
677     reallocations!\n";  
678 } // namespace stl_containers_realtime  
680 //  
681 // =====  
682 // SECTION 3.5: std::list::splice() - O(1) Element Movement  
683 //  
684 // =====  
685 namespace list_splice_realtime {  
686  
687 // std::list::splice() is PERFECT for real-time systems because:  
688 // 1. O(1) complexity for moving elements (constant time!)  
689 // 2. NO MEMORY ALLOCATION - just pointer manipulation  
690 // 3. NO COPYING/MOVING - elements stay in place  
691 // 4. Deterministic and predictable performance  
692 // 5. Iterators remain valid after splice  
693 //  
694 // Use cases in real-time systems:  
695 // - Free lists / object pools  
696 // - Task queue management  
697 // - Event scheduling  
698 // - Moving data between priority queues  
699  
700 struct Task {  
701     int id;  
702     std::string name;  
703     int priority;  
704  
705     Task(int i, std::string n, int p) : id(i), name(std::move(n)), priority(p)  
706     {}  
707 };  
708 void demonstrate() {  
709     std::cout << "\n" << std::string(80, '=') << "\n";  
710     std::cout << "SECTION 3.5: std::list::splice() - O(1) Element Movement\n";  
711     std::cout << std::string(80, '=') << "\n\n";  
712  
713     std::cout << "WHY splice() IS CRITICAL FOR REAL-TIME SYSTEMS:\n";  
714     std::cout << " • O(1) constant time - no iteration required\n";  
715     std::cout << " • NO memory allocation - just rewrites pointers\n";  
716     std::cout << " • NO copying/moving - elements stay in original memory\n";  
717     std::cout << " • Iterators remain valid - predictable behavior\n";  
718     std::cout << " • Deterministic performance - perfect for hard real-time\n  
719     \n";  
720 // =====
```

```

721 // EXAMPLE 1: splice() entire list - O(1)
722 // =====
723 std::cout << std::string(80, '-') << "\n";
724 std::cout << "EXAMPLE 1: Splice Entire List - O(1) Operation\n";
725 std::cout << std::string(80, '-') << "\n\n";
726
727 std::list<Task> active_tasks;
728 active_tasks.emplace_back(1, "Process Sensor Data", 10);
729 active_tasks.emplace_back(2, "Update Control Loop", 20);
730 active_tasks.emplace_back(3, "Send Telemetry", 5);
731
732 std::list<Task> pending_tasks;
733 pending_tasks.emplace_back(4, "Log Event", 1);
734 pending_tasks.emplace_back(5, "Check Diagnostics", 3);
735
736 std::cout << "BEFORE splice():\n";
737 std::cout << " Active tasks: " << active_tasks.size() << "\n";
738 for (const auto& t : active_tasks) {
739     std::cout << "     Task " << t.id << ": " << t.name << " (priority: "
740     << t.priority << ")\n";
741 }
742 std::cout << " Pending tasks: " << pending_tasks.size() << "\n";
743 for (const auto& t : pending_tasks) {
744     std::cout << "     Task " << t.id << ": " << t.name << " (priority: "
745     << t.priority << ")\n";
746 }
747
748 // Splice entire pending_tasks list into active_tasks at the end
749 // This is O(1) - just rewires a few pointers!
750 auto start = std::chrono::high_resolution_clock::now();
751 active_tasks.splice(active_tasks.end(), pending_tasks);
752 auto end = std::chrono::high_resolution_clock::now();
753 auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
754     start);
755
756 std::cout << "\nAFTER splice(active.end(), pending):\n";
757 std::cout << " Active tasks: " << active_tasks.size() << "\n";
758 for (const auto& t : active_tasks) {
759     std::cout << "     Task " << t.id << ": " << t.name << " (priority: "
760     << t.priority << ")\n";
761 }
762 std::cout << " Pending tasks: " << pending_tasks.size() << " (empty!)\n";
763 std::cout << " Time taken: " << duration.count() << " ns (O(1) -
764     constant time!)\n";
765 std::cout << "     NO allocation, NO copying, just pointer rewiring!\n\n";
766
767 // =====
768 // EXAMPLE 2: splice() single element - O(1)
769 // =====
770 std::cout << std::string(80, '-') << "\n";
771 std::cout << "EXAMPLE 2: Splice Single Element - O(1) Operation\n";
772 std::cout << std::string(80, '-') << "\n\n";
773
774 std::list<Task> high_priority;

```

```

770     high_priority.emplace_back(10, "Critical Shutdown", 100);
771     high_priority.emplace_back(11, "Emergency Stop", 99);
772
773     std::list<Task> normal_priority;
774     normal_priority.emplace_back(20, "Routine Check", 10);
775     normal_priority.emplace_back(21, "Update Display", 8);
776     normal_priority.emplace_back(22, "Log Status", 5);
777
778     std::cout << "SCENARIO: Emergency task detected in normal queue!\n\n";
779
780     std::cout << "BEFORE splice():\n";
781     std::cout << "  High priority: " << high_priority.size() << " tasks\n";
782     std::cout << "  Normal priority: " << normal_priority.size() << " tasks\n"
783     ;
784     for (const auto& t : normal_priority) {
785         std::cout << "    Task " << t.id << ": " << t.name << " (priority: "
786         << t.priority << ")\n";
787     }
788
789     // Find the "Update Display" task and move it to high priority
790     auto it = std::find_if(normal_priority.begin(), normal_priority.end(),
791                           [] (const Task& t) { return t.id == 21; });
792
793     if (it != normal_priority.end()) {
794         std::cout << "\n  Found Task 21: " << it->name << " - moving to high
795         priority!\n";
796
797         // Splice single element - O(1) !
798         start = std::chrono::high_resolution_clock::now();
799         high_priority.splice(high_priority.end(), normal_priority, it);
800         end = std::chrono::high_resolution_clock::now();
801         duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
802             start);
803     }
804
805     std::cout << "\nAFTER splice(high.end(), normal, iterator):\n";
806     std::cout << "  High priority: " << high_priority.size() << " tasks\n";
807     for (const auto& t : high_priority) {
808         std::cout << "    Task " << t.id << ": " << t.name << " (priority: "
809         << t.priority << ")\n";
810     }
811     std::cout << "  Normal priority: " << normal_priority.size() << " tasks\n"
812     ;
813     for (const auto& t : normal_priority) {
814         std::cout << "    Task " << t.id << ": " << t.name << " (priority: "
815         << t.priority << ")\n";
816     }
817     std::cout << "  Time taken: " << duration.count() << " ns (O(1) -
818         constant time!)\n";
819     std::cout << "  Task moved instantly without copying!\n\n";
820
821     // =====
822     // EXAMPLE 3: splice() range of elements - O(n) where n = range size
823     // =====

```

```

816     std::cout << std::string(80, '-') << "\n";
817     std::cout << "EXAMPLE 3: Splice Range of Elements - O(n) but NO Allocation
818             \n";
819     std::cout << std::string(80, '-') << "\n\n";
820
821     std::list<int> source = {10, 20, 30, 40, 50, 60, 70, 80};
822     std::list<int> dest = {1, 2, 3};
823
824     std::cout << "BEFORE splice():\n";
825     std::cout << "    Source: ";
826     for (int v : source) std::cout << v << " ";
827     std::cout << "\n    Dest: ";
828     for (int v : dest) std::cout << v << " ";
829     std::cout << "\n";
830
831     // Move elements 30, 40, 50 from source to dest
832     auto range_start = std::find(source.begin(), source.end(), 30);
833     auto range_end = std::find(source.begin(), source.end(), 60); // One past
834     last element
835
836     std::cout << "\n    Moving range [30, 40, 50] from source to dest...\n";
837
838     // Splice range - O(n) where n = distance(range_start, range_end)
839     // But still NO memory allocation!
840     start = std::chrono::high_resolution_clock::now();
841     dest.splice(dest.end(), source, range_start, range_end);
842     end = std::chrono::high_resolution_clock::now();
843     duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
844     start);
845
846     std::cout << "\nAFTER splice(dest.end(), source, first, last):\n";
847     std::cout << "    Source: ";
848     for (int v : source) std::cout << v << " ";
849     std::cout << "\n    Dest: ";
850     for (int v : dest) std::cout << v << " ";
851     std::cout << "\n";
852     std::cout << "    Time taken: " << duration.count() << " ns (O(n) but
853     predictable!)\n";
854     std::cout << "    NO allocation - just pointer updates for range!\n\n";
855
856     // =====
857     // EXAMPLE 4: Real-time pool management with splice()
858     // =====
859     std::cout << std::string(80, '-') << "\n";
860     std::cout << "EXAMPLE 4: Free List Pool Management - Real-Time Pattern\n";
861     std::cout << std::string(80, '-') << "\n\n";
862
863     std::cout << "SCENARIO: Object pool using splice() for O(1) allocation/
864     deallocation\n\n";
865
866     struct Buffer {
867         int id;
868         std::array<char, 256> data;
869         explicit Buffer(int i) : id(i), data{} {}

```

```
865 };
```

```
866
867 // Free pool (available buffers)
868 std::list<Buffer> free_pool;
869 for (int i = 0; i < 5; ++i) {
870     free_pool.emplace_back(i);
871 }
872
873 // Active pool (in-use buffers)
874 std::list<Buffer> active_pool;
875
876 std::cout << "Initial state:\n";
877 std::cout << "  Free pool: " << free_pool.size() << " buffers\n";
878 std::cout << "  Active pool: " << active_pool.size() << " buffers\n\n";
879
880 // Allocate buffer from free pool - O(1)
881 std::cout << "Allocating 3 buffers from free pool...\n";
882 for (int i = 0; i < 3; ++i) {
883     auto it = free_pool.begin();
884     std::cout << "  Allocating buffer " << it->id << "\n";
885     active_pool.splice(active_pool.end(), free_pool, it); // O(1) !
886 }
887
888 std::cout << "\nAfter allocation:\n";
889 std::cout << "  Free pool: " << free_pool.size() << " buffers\n";
890 std::cout << "  Active pool: " << active_pool.size() << " buffers (" ;
891 for (const auto& b : active_pool) std::cout << b.id << " ";
892 std::cout << ")\n\n";
893
894 // Deallocate buffer back to free pool - O(1)
895 std::cout << "Deallocating buffer 1 back to free pool...\n";
896 auto to_free = std::find_if(active_pool.begin(), active_pool.end(),
897     [] (const Buffer& b) { return b.id == 1; });
898 if (to_free != active_pool.end()) {
899     free_pool.splice(free_pool.end(), active_pool, to_free); // O(1) !
900     std::cout << "  Buffer 1 returned to free pool\n";
901 }
902
903 std::cout << "\nFinal state:\n";
904 std::cout << "  Free pool: " << free_pool.size() << " buffers\n";
905 std::cout << "  Active pool: " << active_pool.size() << " buffers (" ;
906 for (const auto& b : active_pool) std::cout << b.id << " ";
907 std::cout << ")\n\n";
908
909 std::cout << "  Pool allocation/deallocation in O(1) with splice()!\n";
910 std::cout << "  NO dynamic allocation - perfect for real-time systems!\n
911 \n";
912 // =====
913 // KEY TAKEAWAYS
914 // =====
915 std::cout << std::string(80, '=') << "\n";
916 std::cout << "KEY TAKEAWAYS: Why splice() is Essential for Real-Time C++\n
917 ";
```

```
917 std::cout << std::string(80, '=') << "\n\n";
918
919 std::cout << "1. PERFORMANCE:\n";
920 std::cout << " • splice() entire list: O(1) - constant time\n";
921 std::cout << " • splice() single element: O(1) - constant time\n";
922 std::cout << " • splice() range: O(n) where n = range size (but
923     predictable!)\n\n";
924
925 std::cout << "2. MEMORY BEHAVIOR:\n";
926 std::cout << " • NO memory allocation - just pointer manipulation\n";
927 std::cout << " • NO copying or moving - elements stay in original memory
928     \n";
929 std::cout << " • Deterministic and bounded - perfect for hard real-time\n
930     \n\n";
931
932 std::cout << "3. ITERATOR VALIDITY:\n";
933 std::cout << " • All iterators, pointers, and references remain valid\n"
934     ;
935 std::cout << " • Safe to cache iterators across splice operations\n\n";
936
937 std::cout << "4. REAL-TIME USE CASES:\n";
938 std::cout << " • Free list / object pool management\n";
939 std::cout << " • Task queue reordering (priority changes)\n";
940 std::cout << " • Event scheduling and rescheduling\n";
941 std::cout << " • Moving data between priority queues\n";
942 std::cout << " • Load balancing across worker queues\n\n";
943
944 std::cout << "5. ALTERNATIVES ARE WORSE:\n";
945 std::cout << " • std::copy() + erase(): Requires copying, O(n) erase\n";
946 std::cout << " • std::move() + erase(): Still requires move, O(n) erase\n
947     \n";
948 std::cout << " • Manual reallocation: Unpredictable timing, allocation
949     overhead\n";
950 std::cout << " • splice() is THE optimal solution for list element
951     movement!\n\n";
952
953 std::cout << " GOLDEN RULE: Use splice() when you need to move elements\n
954     ";
955 std::cout << " between std::list containers in real-time critical code!\n
956     \n";
957
958 } // namespace list_splice_realtime
959
960 // =====
961
962 // SECTION 4: Memory Management - RAII and Custom Allocators
963 // =====
964
965
966 namespace memory_management {
```

```
958 // RAI (Resource Acquisition Is Initialization)
959 // - Resources acquired in constructor
960 // - Resources released in destructor
961 // - Deterministic, exception-safe cleanup
962 // - No manual memory management needed
963
964 // Example: Real-time resource manager using RAI
965 template<typename T, size_t PoolSize>
966 class MemoryPool {
967     private:
968         std::array<T, PoolSize> pool_;
969         std::array<bool, PoolSize> in_use_;
970
971     public:
972         MemoryPool() {
973             in_use_.fill(false);
974             std::cout << " Memory pool created: " << PoolSize << " objects of
975                 size "
976                 << sizeof(T) << " bytes\n";
977             std::cout << " Total pool size: " << (PoolSize * sizeof(T)) << "
978                 bytes\n";
979         }
980
981         // Acquire object from pool - O(n) but bounded by PoolSize
982         [[nodiscard]] T* acquire() noexcept {
983             for (size_t i = 0; i < PoolSize; ++i) {
984                 if (!in_use_[i]) {
985                     in_use_[i] = true;
986                     return &pool_[i];
987                 }
988             }
989             return nullptr; // Pool exhausted
990         }
991
992         // Release object back to pool - O(1)
993         void release(T* ptr) noexcept {
994             if (!ptr) return;
995
996             // Find the object in pool
997             for (size_t i = 0; i < PoolSize; ++i) {
998                 if (&pool_[i] == ptr) {
999                     in_use_[i] = false;
1000                     return;
1001                 }
1002             }
1003         }
1004
1005         [[nodiscard]] size_t available() const noexcept {
1006             return std::count(in_use_.begin(), in_use_.end(), false);
1007         }
1008
1009     // RAI wrapper for pool-allocated object
1010     template<typename T>
```

```
1010 class PoolPtr {
1011     private:
1012         T* ptr_;
1013         std::function<void(T*)> deleter_;
1014
1015     public:
1016         PoolPtr(T* ptr, std::function<void(T*)> deleter)
1017             : ptr_(ptr), deleter_(std::move(deleter)) {}
1018
1019         ~PoolPtr() {
1020             if (ptr_ && deleter_) {
1021                 deleter_(ptr_);
1022             }
1023         }
1024
1025         // Delete copy operations
1026         PoolPtr(const PoolPtr&) = delete;
1027         PoolPtr& operator=(const PoolPtr&) = delete;
1028
1029         // Move operations
1030         PoolPtr(PoolPtr&& other) noexcept
1031             : ptr_(other.ptr_), deleter_(std::move(other.deleter_)) {
1032                 other.ptr_ = nullptr;
1033             }
1034
1035         PoolPtr& operator=(PoolPtr&& other) noexcept {
1036             if (this != &other) {
1037                 if (ptr_ && deleter_) {
1038                     deleter_(ptr_);
1039                 }
1040                 ptr_ = other.ptr_;
1041                 deleter_ = std::move(other.deleter_);
1042                 other.ptr_ = nullptr;
1043             }
1044             return *this;
1045         }
1046
1047         T* get() noexcept { return ptr_; }
1048         const T* get() const noexcept { return ptr_; }
1049
1050         T& operator*() noexcept { return *ptr_; }
1051         const T& operator*() const noexcept { return *ptr_; }
1052
1053         T* operator->() noexcept { return ptr_; }
1054         const T* operator->() const noexcept { return ptr_; }
1055
1056         explicit operator bool() const noexcept { return ptr_ != nullptr; }
1057     };
1058
1059     struct SensorData {
1060         double temperature;
1061         double pressure;
1062         uint64_t timestamp;
1063     };

```

```
1064     SensorData() : temperature(0.0), pressure(0.0), timestamp(0) {}
1065 };
1066
1067 void demonstrate() {
1068     std::cout << "\n" << std::string(70, '=') << "\n";
1069     std::cout << "==== SECTION 4: Memory Management - RAI and Memory Pools
1070         ==\n";
1071     std::cout << std::string(70, '=') << "\n\n";
1072
1073     std::cout << "REAL-TIME MEMORY MANAGEMENT PRINCIPLES:\n\n";
1074
1075     std::cout << " DO:\n";
1076     std::cout << " 1. Pre-allocate all memory during initialization\n";
1077     std::cout << " 2. Use memory pools for frequent alloc/dealloc\n";
1078     std::cout << " 3. Use RAI for deterministic resource cleanup\n";
1079     std::cout << " 4. Prefer stack allocation over heap\n";
1080     std::cout << " 5. Use std::unique_ptr for ownership (minimal overhead)\n\n
1081         ";
1082
1083     std::cout << " DON'T:\n";
1084     std::cout << " 1. Call new/delete in time-critical paths\n";
1085     std::cout << " 2. Use std::shared_ptr in tight loops (atomic overhead)\n
1086         ";
1087
1088     std::cout << " 3. Allow unbounded memory growth\n";
1089     std::cout << " 4. Rely on garbage collection\n";
1090     std::cout << " 5. Use dynamic allocation with unpredictable size\n\n";
1091
1092     std::cout << "EXAMPLE: Memory Pool with RAI\n";
1093
1094
1095 // Create memory pool (during initialization phase)
1096 MemoryPool<SensorData, 100> sensor_pool;
1097
1098 std::cout << " Available slots: " << sensor_pool.available() << "\n\n";
1099
1100
1101 // Acquire objects from pool (during real-time operation)
1102 {
1103     auto sensor1 = PoolPtr<SensorData>(
1104         sensor_pool.acquire(),
1105         [&sensor_pool](SensorData* ptr) { sensor_pool.release(ptr); }
1106     );
1107
1108     auto sensor2 = PoolPtr<SensorData>(
1109         sensor_pool.acquire(),
1110         [&sensor_pool](SensorData* ptr) { sensor_pool.release(ptr); }
1111     );
1112
1113     if (sensor1 && sensor2) {
1114         sensor1->temperature = 25.5;
1115         sensor1->pressure = 1013.25;
1116
1117         std::cout << " Acquired 2 objects from pool\n";
1118         std::cout << " Available slots: " << sensor_pool.available() << "
1119             \n";
1120         std::cout << " Sensor 1 temp: " << sensor1->temperature << "°C\n"
1121     }
1122 }
```

```
    ;
1114 }
1115
1116     // Objects automatically released when PoolPtr goes out of scope (RAII
1117     !)
1117     std::cout << "  Exiting scope...\\n";
1118 }
1119
1120     std::cout << "  Objects released automatically via RAII\\n";
1121     std::cout << "  Available slots: " << sensor_pool.available() << "\\n\\n";
1122
1123     std::cout << "  DETERMINISTIC: No heap fragmentation, predictable timing
1123     !\\n";
1124     std::cout << "  SAFE: RAII ensures resources are always released!\\n";
1125 }
1126
1127 } // namespace memory_management
1128
1129 // =====
1130 // SECTION 5: Modern C++ Features for Real-Time Systems
1131 // =====
1132
1133 namespace modern_cpp_features {
1134
1135 // C++17 std::string_view - Zero-copy string viewing
1136 void process_command(std::string_view cmd) noexcept {
1137     // No string copy - just a view into existing data
1138     if (cmd == "START") {
1139         std::cout << "  Starting system...\\n";
1140     } else if (cmd == "STOP") {
1141         std::cout << "  Stopping system...\\n";
1142     }
1143 }
1144
1145 // C++17 std::optional - Express optional values without exceptions
1146 [[nodiscard]] std::optional<double> safe_divide(double a, double b) noexcept {
1147     if (b == 0.0) {
1148         return std::nullopt; // No exception thrown!
1149     }
1150     return a / b;
1151 }
1152
1153 // C++17 std::variant - Type-safe union without heap allocation
1154 using SensorValue = std::variant<int, double, std::array<uint8_t, 4>>;
1155
1156 void process_sensor_value(const SensorValue& value) noexcept {
1157     std::visit([](const auto& val) {
1158         using T = std::decay_t<decltype(val)>;
1159         if constexpr (std::is_same_v<T, int>) {
1160             std::cout << "  Integer sensor: " << val << "\\n";
1161         }
1162     }, value);
1163 }
```

```
1161     } else if constexpr (std::is_same_v<T, double>) {
1162         std::cout << "    Double sensor: " << val << "\n";
1163     } else {
1164         std::cout << "    Array sensor\n";
1165     }
1166 }, value);
1167 }
1168
1169 // constexpr functions - Compile-time computation
1170 constexpr uint32_t compute_crc32(std::string_view data) {
1171     uint32_t crc = 0xFFFFFFFF;
1172     for (char c : data) {
1173         crc ^= static_cast<uint32_t>(c);
1174         for (int i = 0; i < 8; ++i) {
1175             crc = (crc >> 1) ^ (0xEDB88320 & (-(crc & 1)));
1176         }
1177     }
1178     return ~crc;
1179 }
1180
1181 // Computed at compile time!
1182 constexpr uint32_t header_crc = compute_crc32("HEADER");
1183
1184 void demonstrate() {
1185     std::cout << "\n" << std::string(70, '=') << "\n";
1186     std::cout << "==== SECTION 5: Modern C++ Features for Real-Time ===\n";
1187     std::cout << std::string(70, '=') << "\n\n";
1188
1189     std::cout << "MODERN C++ FEATURES SUITABLE FOR REAL-TIME:\n\n";
1190
1191     // std::string_view
1192     std::cout << "1. std::string_view (C++17) - Zero-copy string handling:\n";
1193     std::string command = "START";
1194     process_command(command); // No copy, just a view
1195     process_command("STOP"); // Works with string literals too
1196     std::cout << "    No allocation, no copy, just a pointer + size\n\n";
1197
1198     // std::optional
1199     std::cout << "2. std::optional (C++17) - Safe error handling without
1200         exceptions:\n";
1201     auto result1 = safe_divide(10.0, 2.0);
1202     auto result2 = safe_divide(10.0, 0.0);
1203
1204     std::cout << "    10.0 / 2.0 = ";
1205     if (result1) {
1206         std::cout << *result1 << " (valid)\n";
1207     }
1208
1209     std::cout << "    10.0 / 0.0 = ";
1210     if (result2) {
1211         std::cout << *result2 << "\n";
1212     } else {
1213         std::cout << "error (nullopt)\n";
1214     }
1215 }
```

```

1214     std::cout << "      No exceptions, no heap allocation\n\n";
1215
1216     // std::variant
1217     std::cout << "3. std::variant (C++17) - Type-safe union without heap:\n";
1218     SensorValue val1 = 42;
1219     SensorValue val2 = 3.14159;
1220     process_sensor_value(val1);
1221     process_sensor_value(val2);
1222     std::cout << "      Type-safe, no heap allocation, no virtual dispatch\n\n"
1223     ";\n
1224
1225     // constexpr
1226     std::cout << "4. constexpr - Compile-time computation:\n";
1227     std::cout << "      Header CRC32 (computed at compile-time): 0x"
1228     << std::hex << std::uppercase << header_crc << std::dec << "\n";
1229     std::cout << "      Zero runtime cost - computed by compiler!\n\n";
1230
1231     // std::array
1232     std::cout << "5. std::array - Type-safe stack array:\n";
1233     std::array<double, 10> samples{1.0, 2.0, 3.0, 4.0, 5.0};
1234     std::cout << "      Array size: " << samples.size() << "\n";
1235     std::cout << "      Stack-allocated, compile-time size, bounds checking\n"
1236     "n";
1237
1238     std::cout << "FEATURES TO AVOID IN HARD REAL-TIME:\n";
1239     std::cout << "      Exceptions (use -fno-exceptions)\n";
1240     std::cout << "      RTTI (use -fno-rtti)\n";
1241     std::cout << "      Virtual functions in time-critical paths\n";
1242     std::cout << "      std::shared_ptr (atomic ref counting overhead)\n";
1243     std::cout << "      Dynamic polymorphism (prefer compile-time polymorphism)\n"
1244     "n";
1245
1246 }
1247
1248 } // namespace modern_cpp_features
1249
1250 // SECTION 6: Real-Time System Checklist
1251 // =====
1252
1253 void print_realtime_checklist() {
1254     std::cout << "\n" << std::string(70, '=') << "\n";
1255     std::cout << "==== SECTION 6: Real-Time Programming Checklist ===\n";
1256     std::cout << std::string(70, '=') << "\n\n";
1257
1258     std::cout << "INITIALIZATION PHASE (Non-real-time):\n";
1259     std::cout << "      Pre-allocate all memory (reserve() for vectors)\n";
1260     std::cout << "      Create memory pools for frequent allocations\n";
1261     std::cout << "      Load and sort lookup tables\n";
1262     std::cout << "      Initialize all data structures to max size\n";
1263     std::cout << "      Warm up caches if applicable\n";

```

```

1261     std::cout << "      Set thread priorities and CPU affinity\n\n";
1262
1263     std::cout << "REAL-TIME EXECUTION PHASE:\n";
1264     std::cout << "      NO dynamic memory allocation (no new/delete/malloc/free)
1265             \n";
1266     std::cout << "      NO operations with unbounded execution time\n";
1267     std::cout << "      NO file I/O or blocking system calls\n";
1268     std::cout << "      NO unbounded loops or recursion\n";
1269     std::cout << "      NO exceptions in critical paths\n";
1270     std::cout << "      Use O(1) or O(log n) operations only\n";
1271     std::cout << "      All containers have bounded maximum size\n";
1272     std::cout << "      Use std::bitset instead of std::vector<bool>\n";
1273     std::cout << "      Use std::array when size known at compile-time\n";
1274     std::cout << "      Use RAII for deterministic cleanup\n\n";
1275
1276     std::cout << "CONTAINER SELECTION:\n";
1277     std::cout << "      std::array<T, N>           - Fixed size, stack-based\n";
1278     std::cout << "      std::vector<T> + reserve() - Dynamic, pre-allocated\n";
1279     std::cout << "      std::bitset<N>           - Flags and bit manipulation
1280             \n";
1281     std::cout << "      std::string_view          - Zero-copy string viewing\n";
1282     std::cout << "      \n";
1283     std::cout << "      std::deque<T>           - If size bounded\n";
1284     std::cout << "      std::list<T>             - Poor cache, heap per
1285             element\n";
1286     std::cout << "      std::vector<bool>        - Use std::bitset instead\n\n";
1287             ";
1288
1289     std::cout << "ALGORITHM COMPLEXITY TARGETS:\n";
1290     std::cout << "      O(1)           - Ideal (array access, bitset operations)\n";
1291     std::cout << "      O(log n)        - Good (binary search, balanced trees)\n";
1292     std::cout << "      O(n)           - Acceptable if n is bounded and small\n";
1293     std::cout << "      O(n log n)      - Avoid in critical paths (sort)\n";
1294     std::cout << "      O(n2)         - Never (nested loops, bubble sort)\n\n";
1295
1296     std::cout << "COMPILER FLAGS FOR REAL-TIME:\n";
1297     std::cout << "      -fno-exceptions    Disable exception handling\n";
1298     std::cout << "      -fno-rtti          Disable run-time type information\n";
1299     std::cout << "      -O3                Maximum optimization\n";
1300     std::cout << "      -march=native      Target specific CPU architecture\n";
1301     std::cout << "      -flto              Link-time optimization\n\n";
1302
1303     std::cout << "PROFILING AND VERIFICATION:\n";
1304     std::cout << "      Measure worst-case execution time (WCET)\n";
1305     std::cout << "      Use real-time profiling tools\n";
1306     std::cout << "      Test under worst-case conditions\n";
1307     std::cout << "      Verify no unexpected allocations\n";
1308     std::cout << "      Monitor stack usage\n";
1309     std::cout << "      Validate timing constraints are met\n";
1310
1311 }
1312
1313 // =====

```

```
1308 // Main Function
1309 //
1310 =====
1311 int main() {
1312     std::cout << "\n";
1313     std::cout << "
1314         std::cout << "
1315             std::cout << "
1316                 std::cout << "
1317                     std::cout << "
1318                         std::cout << "
1319
1320     try {
1321         // Section 1: Big O Notation
1322         big_o_notation::demonstrate();
1323
1324         // Section 1.5: Thread Architecture
1325         thread_architecture::demonstrate();
1326
1327         // Section 2: std::bitset vs std::vector<bool>
1328         bitset_vs_vector_bool::demonstrate();
1329
1330         // Section 3: STL Containers
1331         stl_containers_realtime::demonstrate();
1332
1333         // Section 3.5: std::list::splice() - O(1) Element Movement
1334         list_splice_realtime::demonstrate();
1335
1336         // Section 4: Memory Management
1337         memory_management::demonstrate();
1338
1339         // Section 5: Modern C++ Features
1340         modern_cpp_features::demonstrate();
1341
1342         // Section 6: Checklist
1343         print_realtime_checklist();
1344
1345         std::cout << "\n" << std::string(70, '=') << "\n";
1346         std::cout << "==== END OF REAL-TIME PROGRAMMING GUIDE ===\n";
1347         std::cout << std::string(70, '=') << "\n\n";
1348
1349         std::cout << "KEY TAKEAWAYS:\n";
1350         std::cout << " 1. Always pre-allocate memory during initialization\n";
1351             ;
1352         std::cout << " 2. Use std::bitset instead of std::vector<bool>\n";
1353         std::cout << " 3. Prefer O(1) operations, accept O(log n), bound O(n)\n";
1354             \n";
```

```
1353     std::cout << " 4. Use reserve() for std::vector to prevent  
1354         reallocations\n";  
1355     std::cout << " 5. std::array is perfect for fixed-size data\n";  
1356     std::cout << " 6. RAII provides deterministic resource management\n";  
1357     std::cout << " 7. Modern C++ features like std::optional and std::  
1358         variant\n";  
1359     std::cout << "     are real-time friendly (no exceptions, no heap)\n";  
1360     std::cout << " 8. Measure WCET, not average performance!\n\n";  
1361  
1362     return 0;  
1363  
1364 } catch (const std::exception& e) {  
1365     std::cerr << "Error: " << e.what() << std::endl;  
1366     return 1;  
1367 }  
1368 }
```

## 58 Source Code: ResourceLeaks.cpp

File: src/ResourceLeaks.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // RESOURCE MANAGEMENT IN MODERN C++
3 // =====
4 // This example demonstrates how Modern C++ solves resource leak problems
5 // that plagued older C++ code using manual new/delete.
6 //
7 // TOPICS COVERED:
8 // 1. The Old Problem - Manual memory management with new/delete
9 // 2. Smart Pointers - Automatic memory management
10 // 3. RAII Pattern - Resource Acquisition Is Initialization
11 // 4. Other Resources - Files, mutexes, sockets, database connections
12 // 5. Custom Deleters - Managing non-memory resources
13 // 6. Exception Safety - Guarantees and resource cleanup
14 // 7. Modern Best Practices - When raw pointers are still OK
15 //
16 // KEY INSIGHT:
17 // "Modern C++ doesn't use new/delete directly anymore!"
18 // - Use smart pointers (unique_ptr, shared_ptr, weak_ptr)
19 // - Use containers (vector, string, map)
20 // - Use RAII wrappers for all resources
21 // - Raw pointers for non-owning references only
22 //
23 // EMBEDDED SYSTEMS NOTE:
24 // Even in resource-constrained embedded systems, smart pointers
25 // have ZERO runtime overhead compared to manual management!
26 // =====
27
28 #include <iostream>
29 #include <memory>
30 #include <vector>
31 #include <string>
32 #include <fstream>
33 #include <mutex>
34 #include <thread>
35 #include <chrono>
36 #include <exception>
37 #include <functional>
38 #include <map>
39
40 // =====
41 // 1. THE OLD PROBLEM - MANUAL MEMORY MANAGEMENT
42 // =====
43
44 namespace OldCpp {
45
46 // Example 1: Classic memory leak
47 void memory_leak_example() {
48     std::cout << "OLD C++: Memory leak example\n";
49     std::cout << "-----\n";
```

```
50     int* ptr = new int(42);
51     std::cout << "Allocated int with value: " << *ptr << "\n";
52
53     // Forgot to delete! Memory leak!
54     // delete ptr; // <-- Missing this line
55
56
57     std::cout << "Function returns without delete - MEMORY LEAK!\n\n";
58 }
59
60 // Example 2: Exception causes leak
61 void exception_causes_leak() {
62     std::cout << "OLD C++: Exception causes leak\n";
63     std::cout << "-----\n";
64
65     int* data = new int[1000];
66     std::cout << "Allocated array of 1000 ints\n";
67
68     try {
69         // Some code that might throw
70         if (data != nullptr) {
71             throw std::runtime_error("Simulated error!");
72         }
73
74         delete[] data; // <-- Never reached if exception thrown!
75     } catch (const std::exception& e) {
76         std::cout << "Caught exception: " << e.what() << "\n";
77         std::cout << "Array was never deleted - MEMORY LEAK!\n\n";
78         // Should have delete[] data here, but easy to forget
79     }
80 }
81
82 // Example 3: Multiple exit paths
83 void multiple_exit_paths(bool condition1, bool condition2) {
84     std::cout << "OLD C++: Multiple exit paths\n";
85     std::cout << "-----\n";
86
87     int* buffer = new int[100];
88     std::cout << "Allocated buffer\n";
89
90     if (condition1) {
91         std::cout << "Early return #1\n";
92         // Forgot to delete!
93         return; // LEAK!
94     }
95
96     // ... do some work ...
97
98     if (condition2) {
99         std::cout << "Early return #2\n";
100        // Forgot to delete!
101        return; // LEAK!
102    }
103 }
```

```
104     delete[] buffer; // Only reached if both conditions false
105     std::cout << "Normal cleanup\n\n";
106 }
107
108 // Example 4: Wrong delete operator
109 void wrong_delete_operator() {
110     std::cout << "OLD C++: Wrong delete operator\n";
111     std::cout << "-----\n";
112
113     int* single = new int(42);
114     int* array = new int[10];
115
116     // WRONG! Undefined behavior!
117     delete[] single; // Should be: delete single;
118     delete array; // Should be: delete[] array;
119
120     std::cout << "Used wrong delete operator - UNDEFINED BEHAVIOR!\n\n";
121 }
122
123 // Example 5: Double delete
124 void double_delete_problem() {
125     std::cout << "OLD C++: Double delete\n";
126     std::cout << "-----\n";
127
128     int* ptr = new int(42);
129
130     delete ptr;
131     std::cout << "Deleted once\n";
132
133     // WRONG! Undefined behavior!
134     delete ptr; // Double delete - CRASH or corruption!
135
136     std::cout << "Double delete - UNDEFINED BEHAVIOR!\n\n";
137 }
138
139 class ResourceIntensive {
140     int* data;
141     size_t size;
142
143 public:
144     ResourceIntensive(size_t n) : size(n) {
145         data = new int[n];
146         std::cout << " Allocated " << n << " ints\n";
147     }
148
149     // PROBLEM: No destructor! Memory leak!
150     // ~ResourceIntensive() { delete[] data; } // <-- Forgot this!
151 };
152
153 void demonstrate_old_problems() {
154     std::cout << "\n";
155     std::cout << "=====\\n";
156     std::cout << "1. THE OLD PROBLEM - MANUAL MEMORY MANAGEMENT\\n";
```

```
157     std::cout << "=====\\n\\n";
158
159     memory_leak_example();
160     exception_causes_leak();
161     multiple_exit_paths(true, false);
162
163     // Note: Can't safely run wrong_delete_operator() and
164     //        double_delete_problem()
165     // as they cause undefined behavior
166
166     std::cout << "Summary of Problems:\\n";
167     std::cout << "    Forgot to delete\\n";
168     std::cout << "    Exception before delete\\n";
169     std::cout << "    Multiple exit paths\\n";
170     std::cout << "    Wrong delete operator (delete vs delete [])\\n";
171     std::cout << "    Double delete\\n";
172     std::cout << "    Forgot destructor in class\\n";
173     std::cout << "\\n";
174 }
175
176 } // namespace OldCpp
177
178 // =====
179 // 2. MODERN C++ SOLUTION - SMART POINTERS
180 // =====
181
182 namespace ModernCpp {
183
184 // Example 1: unique_ptr - Exclusive ownership
185 void unique_ptr_example() {
186     std::cout << "MODERN C++: unique_ptr (exclusive ownership)\\n";
187     std::cout << "-----\\n";
188
189     // No manual new!
190     std::unique_ptr<int> ptr = std::make_unique<int>(42);
191     std::cout << "Created unique_ptr with value: " << *ptr << "\\n";
192
193     // Automatically deleted when scope ends!
194     std::cout << "Leaving scope - automatic cleanup!\\n\\n";
195
196     // NO delete needed!
197 }
198
199 // Example 2: Exception-safe with unique_ptr
200 void exception_safe_unique_ptr() {
201     std::cout << "MODERN C++: Exception-safe with unique_ptr\\n";
202     std::cout << "-----\\n";
203
204     auto data = std::make_unique<int[]>(1000);
205     std::cout << "Allocated array of 1000 ints\\n";
206
207     try {
208         throw std::runtime_error("Simulated error!");
209     }
```

```
209         // Never reached, but doesn't matter!
210     } catch (const std::exception& e) {
211         std::cout << "Caught exception: " << e.what() << "\n";
212         std::cout << "unique_ptr automatically cleaned up - NO LEAK!\n\n";
213     }
214
215
216     // Cleanup happens automatically in catch block
217 }
218
219 // Example 3: Multiple exit paths - no problem!
220 void multiple_exit_paths_safe(bool condition1, bool condition2) {
221     std::cout << "MODERN C++: Multiple exit paths (safe)\n";
222     std::cout << "-----\n";
223
224     auto buffer = std::make_unique<int>[](100);
225     std::cout << "Allocated buffer\n";
226
227     if (condition1) {
228         std::cout << "Early return #1 - buffer auto-deleted!\n\n";
229         return; // NO LEAK! unique_ptr cleans up
230     }
231
232     if (condition2) {
233         std::cout << "Early return #2 - buffer auto-deleted!\n\n";
234         return; // NO LEAK! unique_ptr cleans up
235     }
236
237     std::cout << "Normal return - buffer auto-deleted!\n\n";
238     // Automatic cleanup regardless of path!
239 }
240
241 // Example 4: shared_ptr - Shared ownership
242 void shared_ptr_example() {
243     std::cout << "MODERN C++: shared_ptr (shared ownership)\n";
244     std::cout << "-----\n";
245
246     std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
247     std::cout << "Created shared_ptr, ref count: " << ptr1.use_count() << "\n"
248         ;
249
250     {
251         std::shared_ptr<int> ptr2 = ptr1; // Share ownership
252         std::cout << "Shared ownership, ref count: " << ptr1.use_count() << "\n";
253
254         std::shared_ptr<int> ptr3 = ptr1; // Another share
255         std::cout << "More sharing, ref count: " << ptr1.use_count() << "\n";
256     } // ptr2 and ptr3 destroyed
257
258     std::cout << "After scope, ref count: " << ptr1.use_count() << "\n";
259     std::cout << "Memory deleted when last shared_ptr destroyed!\n\n";
260 }
```

```
261 // Example 5: weak_ptr - Non-owning observer (doesn't keep object alive)
262 void weak_ptr_example() {
263     std::cout << "MODERN C++: weak_ptr (non-owning observer)\n";
264     std::cout << "-----\n";
265
266     std::weak_ptr<int> weak;
267
268     {
269         std::shared_ptr<int> shared = std::make_shared<int>(42);
270         weak = shared; // weak_ptr observes, but doesn't own
271
272         std::cout << "shared_ptr alive, ref count: " << shared.use_count() <<
273             "\n";
274         std::cout << "weak_ptr expired? " << (weak.expired() ? "YES" : "NO")
275             << "\n";
276
277         // ALWAYS check if weak_ptr is still valid before using!
278         if (auto locked = weak.lock()) { // lock() returns shared_ptr
279             std::cout << "Locked weak_ptr, value: " << *locked << "\n";
280             std::cout << "Ref count during lock: " << locked.use_count() << "\n";
281         } else {
282             std::cout << "Object was deleted!\n";
283         }
284
285     } // shared destroyed here
286
287     std::cout << "\nAfter shared_ptr destroyed:\n";
288     std::cout << "weak_ptr expired? " << (weak.expired() ? "YES" : "NO") << "\n";
289
290     if (auto locked = weak.lock()) {
291         std::cout << "Locked successfully, value: " << *locked << "\n";
292     } else {
293         std::cout << "Cannot lock - object was deleted!\n";
294     }
295 }
296
297 // Example 6: weak_ptr - Breaking circular references
298 class Node {
299 public:
300     std::string name;
301     std::shared_ptr<Node> next; // Strong reference (owns next)
302     std::weak_ptr<Node> prev; // Weak reference (doesn't own prev)
303
304     Node(const std::string& n) : name(n) {
305         std::cout << "  Node '" << name << "' created\n";
306     }
307
308     ~Node() {
309         std::cout << "  Node '" << name << "' destroyed\n";
310     }
311 }
```

```
311 };
```

```
312
```

```
313 void circular_reference_example() {
314     std::cout << "MODERN C++: weak_ptr breaks circular references\n";
315     std::cout << "-----\n";
316
317     auto node1 = std::make_shared<Node>("First");
318     auto node2 = std::make_shared<Node>("Second");
319     auto node3 = std::make_shared<Node>("Third");
320
321     // Build doubly-linked list
322     node1->next = node2;
323     node2->prev = node1; // weak_ptr - doesn't create cycle!
324
325     node2->next = node3;
326     node3->prev = node2; // weak_ptr - doesn't create cycle!
327
328     std::cout << "\nNavigate forward (using shared_ptr):\n";
329     auto current = node1;
330     while (current) {
331         std::cout << "  At node: " << current->name << "\n";
332         current = current->next;
333     }
334
335     std::cout << "\nNavigate backward (using weak_ptr):\n";
336     current = node3;
337     while (current) {
338         std::cout << "  At node: " << current->name << "\n";
339
340         // Must lock() weak_ptr to use it!
341         if (auto prev = current->prev.lock()) {
342             current = prev;
343         } else {
344             break; // No previous node
345         }
346     }
347
348     std::cout << "\nLeaving scope - all nodes will be destroyed:\n";
349 }
```

```
350
```

```
351 // Example 7: weak_ptr - Cache implementation
352 class ExpensiveResource {
353     std::string data;
354 public:
355     ExpensiveResource(const std::string& d) : data(d) {
356         std::cout << "  Expensive resource created: " << data << "\n";
357     }
358     ~ExpensiveResource() {
359         std::cout << "  Expensive resource destroyed: " << data << "\n";
360     }
361     const std::string& get_data() const { return data; }
362 };
363
364 class ResourceCache {
```

```
365     std::map<std::string, std::weak_ptr<ExpensiveResource>> cache;
366
367 public:
368     std::shared_ptr<ExpensiveResource> get_resource(const std::string& key) {
369         // Try to get from cache
370         auto it = cache.find(key);
371         if (it != cache.end()) {
372             // Check if cached resource is still alive
373             if (auto resource = it->second.lock()) {
374                 std::cout << " Cache HIT: " << key << "\n";
375                 return resource;
376             } else {
377                 std::cout << " Cache EXPIRED: " << key << "\n";
378                 cache.erase(it); // Clean up expired entry
379             }
380         }
381
382         // Create new resource
383         std::cout << " Cache MISS: Creating " << key << "\n";
384         auto resource = std::make_shared<ExpensiveResource>(key);
385         cache[key] = resource; // Store weak_ptr in cache
386         return resource;
387     }
388 };
389
390 void cache_example() {
391     std::cout << "MODERN C++: weak_ptr for caching\n";
392     std::cout << "-----\n";
393
394     ResourceCache cache;
395
396     {
397         auto res1 = cache.get_resource("data1");
398         auto res2 = cache.get_resource("data1"); // Should be cache hit
399
400         std::cout << "Both pointing to same resource: "
401             << (res1 == res2 ? "YES" : "NO") << "\n";
402     }
403
404     std::cout << "\nResources destroyed, trying to access again:\n";
405     auto res3 = cache.get_resource("data1"); // Should be cache miss
406
407     std::cout << "\n";
408 }
409
410 // Example 8: Modern class with RAII
411 class ResourceIntensiveModern {
412     std::unique_ptr<int[]> data;
413     size_t size;
414
415 public:
416     ResourceIntensiveModern(size_t n) : data(std::make_unique<int[]>(n)), size
417         (n) {
418         std::cout << " Allocated " << n << " ints (RAII)\n";
```

```
418    }
419
420    // No destructor needed! unique_ptr handles cleanup!
421    // Compiler-generated destructor is perfect!
422
423    // Safe to copy if we want (Rule of Zero)
424    // Or delete copy if we want move-only:
425    ResourceIntensiveModern(const ResourceIntensiveModern&) = delete;
426    ResourceIntensiveModern& operator=(const ResourceIntensiveModern&) =
427        delete;
428
429    // Move operations provided automatically!
430    ResourceIntensiveModern(ResourceIntensiveModern&&) = default;
431    ResourceIntensiveModern& operator=(ResourceIntensiveModern&&) = default;
432
433    int& operator[](size_t i) { return data[i]; }
434};
435
436 void demonstrate_modern_solutions() {
437     std::cout << "\n";
438     std::cout << "=====\\n";
439     std::cout << "2. MODERN C++ SOLUTION - SMART POINTERS\\n";
440     std::cout << "=====\\n\\n";
441
442     unique_ptr_example();
443     exception_safe_unique_ptr();
444     multiple_exit_paths_safe(true, false);
445     shared_ptr_example();
446     weak_ptr_example();
447     circular_reference_example();
448     cache_example();
449
450     std::cout << "Modern class with RAII:\\n";
451     {
452         ResourceIntensiveModern obj(100);
453         // Automatic cleanup when obj goes out of scope!
454     }
455     std::cout << "    Automatically cleaned up!\\n\\n";
456
457     std::cout << "Summary of Solutions:\\n";
458     std::cout << "    No manual new/delete\\n";
459     std::cout << "    Exception-safe automatically\\n";
460     std::cout << "    Multiple exit paths safe\\n";
461     std::cout << "    Can't use wrong delete operator\\n";
462     std::cout << "    Can't double-delete\\n";
463     std::cout << "    Compiler-generated destructor works\\n";
464     std::cout << "    weak_ptr breaks circular references\\n";
465     std::cout << "    weak_ptr for cache/observer patterns\\n";
466 }
467
468 } // namespace ModernCpp
```

```
469
470 // =====
471 // 3. RAIIS PATTERN - BEYOND MEMORY
472 // =====
473
474 namespace RAIIS {
475
476 // File handle RAIIS
477 class FileRAII {
478     std::ofstream file;
479     std::string filename;
480
481 public:
482     FileRAII(const std::string& name) : filename(name) {
483         file.open(filename);
484         if (!file) {
485             throw std::runtime_error("Failed to open file");
486         }
487         std::cout << "  File opened: " << filename << "\n";
488     }
489
490     ~FileRAII() {
491         if (file.is_open()) {
492             file.close();
493             std::cout << "  File closed: " << filename << "\n";
494         }
495     }
496
497     // Delete copy, allow move
498     FileRAII(const FileRAII&) = delete;
499     FileRAII& operator=(const FileRAII&) = delete;
500     FileRAII(FileRAII&&) = default;
501     FileRAII& operator=(FileRAII&&) = default;
502
503     void write(const std::string& data) {
504         file << data;
505     }
506 };
507
508 // Mutex lock RAIIS (std::lock_guard does this!)
509 class MutexLockRAII {
510     std::mutex& mtx;
511
512 public:
513     explicit MutexLockRAII(std::mutex& m) : mtx(m) {
514         mtx.lock();
515         std::cout << "  Mutex locked\n";
516     }
517
518     ~MutexLockRAII() {
519         mtx.unlock();
520         std::cout << "  Mutex unlocked\n";
521     }
522 }
```

```
523 // Non-copyable, non-movable
524 MutexLockRAII(const MutexLockRAII&) = delete;
525 MutexLockRAII& operator=(const MutexLockRAII&) = delete;
526 };
527
528 // Timer RAII (measures scope duration)
529 class ScopeTimer {
530     std::string name;
531     std::chrono::high_resolution_clock::time_point start;
532
533 public:
534     explicit ScopeTimer(const std::string& n)
535         : name(n), start(std::chrono::high_resolution_clock::now()) {
536         std::cout << "  [" << name << "] Started\n";
537     }
538
539 ~ScopeTimer() {
540     auto end = std::chrono::high_resolution_clock::now();
541     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
542         end - start);
543     std::cout << "  [" << name << "] Duration: " << duration.count() << "s\n";
544 }
545
546 // Generic RAII wrapper
547 template<typename ResourceType, typename AcquireFunc, typename ReleaseFunc>
548 class GenericRAII {
549     ResourceType resource;
550     ReleaseFunc release;
551     bool active;
552
553 public:
554     GenericRAII(AcquireFunc acquire, ReleaseFunc rel)
555         : resource(acquire()), release(rel), active(true) {}
556
557 ~GenericRAII() {
558     if (active) {
559         release(resource);
560     }
561 }
562
563 ResourceType& get() { return resource; }
564
565 // Delete copy
566 GenericRAII(const GenericRAII&) = delete;
567 GenericRAII& operator=(const GenericRAII&) = delete;
568
569 // Allow move
570 GenericRAII(GenericRAII&& other) noexcept
571     : resource(std::move(other.resource)),
572     release(std::move(other.release)),
573     active(other.active) {
574     other.active = false;
575 }
```

```
575     }
576 };
577
578 void demonstrate_raii() {
579     std::cout << "\n";
580     std::cout << "=====\\n";
581     std::cout << "3. RAIU PATTERN - BEYOND MEMORY\\n";
582     std::cout << "=====\\n";
583     std::cout << "File RAIU example:\\n";
584     {
585         FileRAII file("test.txt");
586         file.write("Hello, RAIU!\\n");
587         // File automatically closed when scope ends
588     }
589     std::cout << "\n";
590
591     std::cout << "Mutex RAIU example:\\n";
592     std::mutex mtx;
593     {
594         MutexLockRAII lock(mtx);
595         // Critical section
596         std::cout << " In critical section\\n";
597         // Mutex automatically unlocked when scope ends
598     }
599     std::cout << "\n";
600
601     std::cout << "Scope timer example:\\n";
602     {
603         ScopeTimer timer("MyOperation");
604         // Simulate work
605         std::this_thread::sleep_for(std::chrono::milliseconds(10));
606         // Timer automatically reports duration
607     }
608     std::cout << "\n";
609
610     std::cout << "RAIU Resources:\\n";
611     std::cout << " Memory (unique_ptr, shared_ptr)\\n";
612     std::cout << " Files (fstream, custom wrappers)\\n";
613     std::cout << " Locks (lock_guard, unique_lock)\\n";
614     std::cout << " Timers (scope profiling)\\n";
615     std::cout << " Network sockets\\n";
616     std::cout << " Database connections\\n";
617     std::cout << " OpenGL contexts\\n";
618     std::cout << " Hardware registers (embedded)\\n";
619     std::cout << "\n";
620 }
621
622 } // namespace RAIU
623
624 // =====
625 // 4. CUSTOM DELETORS
```

```
627 // =====
628
629 namespace CustomDeleters {
630
631 // Example: C-style FILE*
632 void file_deleter(FILE* fp) {
633     if (fp) {
634         std::cout << "  Custom deleter: closing FILE*\n";
635         fclose(fp);
636     }
637 }
638
639 void custom_deleter_file_example() {
640     std::cout << "Custom deleter with FILE*:\n";
641
642     std::unique_ptr<FILE, decltype(&file_deleter)> file(
643         fopen("test.txt", "w"),
644         file_deleter
645     );
646
647     if (file) {
648         fprintf(file.get(), "Hello from C-style FILE!\n");
649     }
650
651     // FILE* automatically closed by custom deleter!
652 }
653
654 // Example: Hardware register (embedded systems)
655 struct HardwareRegister {
656     volatile uint32_t* address;
657     uint32_t original_value;
658
659     HardwareRegister(volatile uint32_t* addr)
660         : address(addr), original_value(*addr) {
661             std::cout << "  Saving register value: 0x" << std::hex <<
662             original_value << std::dec << "\n";
663         }
664
665     ~HardwareRegister() {
666         *address = original_value;
667         std::cout << "  Restored register value: 0x" << std::hex <<
668         original_value << std::dec << "\n";
669     }
670 };
671
672 void embedded_register_example() {
673     std::cout << "\nEmbedded hardware register RAII:\n";
674
675     uint32_t simulated_register = 0x12345678;
676
677     {
678         HardwareRegister reg(&simulated_register);
679
680         // Modify register
681     }
682 }
```

```
679     simulated_register = 0xABCD00;
680     std::cout << "  Modified register: 0x" << std::hex <<
681         simulated_register << std::dec << "\n";
682
683     // Automatically restored on scope exit!
684 }
685
686 std::cout << "  Register after scope: 0x" << std::hex <<
687     simulated_register << std::dec << "\n";
688 }
689
690 void demonstrate_custom_deleters() {
691     std::cout << "\n";
692     std::cout << "=====\\n";
693     std::cout << "4. CUSTOM DELETERS\\n";
694     std::cout << "=====\\n";
695     std::cout << "=====\\n";
696
697     custom_deleter_file_example();
698     embedded_register_example();
699
700     std::cout << "\\nCustom deleters for:\\n";
701     std::cout << "    C API resources (FILE*, malloc, etc.)\\n";
702     std::cout << "    Operating system handles\\n";
703     std::cout << "    Hardware registers (embedded)\\n";
704     std::cout << "    Third-party library resources\\n";
705     std::cout << "\\n";
706 }
707
708 } // namespace CustomDeleters
709
710 // =====
711 // 5. WHEN RAW POINTERS ARE OK IN MODERN C++
712 // =====
713
714 namespace RawPointersOK {
715
716     class Node {
717     public:
718         int value;
719         std::unique_ptr<Node> next; // Owns the next node
720         Node* prev; // Non-owning pointer to previous
721
722         Node(int v) : value(v), prev(nullptr) {}
723     };
724
725     void raw_pointer_examples() {
726         std::cout << "\n";
727         std::cout << "=====\\n";
728         std::cout << "5. WHEN RAW POINTERS ARE OK IN MODERN C++\\n";
729         std::cout << "=====\\n";
730     }
731 }
```

```
727     std::cout << "Raw pointers are OK for:\n\n";
728
729     std::cout << "1. NON-OWNING REFERENCES:\n";
730     std::cout << "    auto owner = std::make_unique<int>(42);\n";
731     std::cout << "    int* observer = owner.get(); // OK! Not responsible for
732                 deletion\n\n";
733
734     std::cout << "2. FUNCTION PARAMETERS (non-owning):\n";
735     std::cout << "    void process(const Widget* widget); // OK! Doesn't own\n
736                 \n";
737
738     std::cout << "3. OPTIONAL PARAMETERS:\n";
739     std::cout << "    void render(Texture* texture = nullptr); // OK! May be
740                 null\n\n";
741
742     std::cout << "4. POINTERS TO STACK OBJECTS:\n";
743     std::cout << "    int value = 42;\n";
744     std::cout << "    int* ptr = &value; // OK! No ownership\n\n";
745
746     std::cout << "5. BACK POINTERS IN DATA STRUCTURES:\n";
747     std::cout << "    struct Node {\n";
748     std::cout << "        std::unique_ptr<Node> next; // Owns next\n";
749     std::cout << "        Node* prev; // Non-owning back
750                 pointer\n";
751     std::cout << "    };\n\n";
752
753     std::cout << "6. POINTERS TO GLOBAL/STATIC OBJECTS:\n";
754     std::cout << "    static Logger logger;\n";
755     std::cout << "    Logger* getLogger() { return &logger; } // OK!\n\n";
756
757     std::cout << "NEVER use raw pointers for:\n";
758     std::cout << "    Ownership (use unique_ptr or shared_ptr)\n";
759     std::cout << "    Arrays (use std::vector or std::array)\n";
760     std::cout << "    Manual new/delete\n";
761     std::cout << "\n";
762
763 } // namespace RawPointersOK
764
765 // =====
766 // 6. CONTAINERS INSTEAD OF MANUAL ARRAYS
767 // =====
768
769 namespace Containers {
770
771     void demonstrate_containers() {
772         std::cout << "\n";
773         std::cout << "=====\\n";
774         ;
775         std::cout << "6. CONTAINERS INSTEAD OF MANUAL ARRAYS\n";
776         std::cout << "=====\\n\\n";
777     }
778 }
```

```
775 std::cout << "OLD C++: Manual array management\n";
776 std::cout << "  int* arr = new int[100]; // Manual allocation\n";
777 std::cout << "  // ... use array ...\\n";
778 std::cout << "  delete[] arr; // Manual cleanup (easy to forget!)\\n\\n";
779
780 std::cout << "MODERN C++: Use std::vector\\n";
781 std::vector<int> vec(100);
782 std::cout << "  std::vector<int> vec(100); // Automatic!\\n";
783 std::cout << "  // Automatic cleanup, resizing, bounds checking\\n";
784 std::cout << "  Size: " << vec.size() << " elements\\n\\n";
785
786 std::cout << "Other containers (all automatic!):\\n";
787 std::cout << "  std::vector - Dynamic array\\n";
788 std::cout << "  std::array - Fixed-size array\\n";
789 std::cout << "  std::string - String (char array)\\n";
790 std::cout << "  std::map - Key-value pairs\\n";
791 std::cout << "  std::unordered_map - Hash table\\n";
792 std::cout << "  std::set - Unique elements\\n";
793 std::cout << "  std::list - Doubly-linked list\\n";
794 std::cout << "  std::deque - Double-ended queue\\n";
795 std::cout << "\\n";
796 }
797
798 } // namespace Containers
799
800 // =====
801 // 7. PERFORMANCE - ZERO OVERHEAD
802 // =====
803
804 namespace Performance {
805
806 void demonstrate_zero_overhead() {
807     std::cout << "\\n";
808     std::cout << "=====\\n";
809     std::cout << "7. PERFORMANCE - ZERO OVERHEAD\\n";
810     std::cout << "=====\\n\\n";
811
812     std::cout << "Smart pointers have ZERO runtime overhead!\\n\\n";
813
814     std::cout << "unique_ptr:\\n";
815     std::cout << "  • Same size as raw pointer\\n";
816     std::cout << "  • No runtime cost\\n";
817     std::cout << "  • Optimized to same assembly as manual delete\\n";
818     std::cout << "  • sizeof(unique_ptr<int>) = " << sizeof(std::unique_ptr<
819         int>) << " bytes\\n";
820     std::cout << "  • sizeof(int*) = " << sizeof(int*) << " bytes\\n\\n";
821
822     std::cout << "shared_ptr:\\n";
823     std::cout << "  • Small overhead for reference counting\\n";
824     std::cout << "  • sizeof(shared_ptr<int>) = " << sizeof(std::shared_ptr<
825         int>) << " bytes\\n";
826     std::cout << "  • Only use when you need shared ownership\\n\\n";
```

```
825     std::cout << "Embedded systems:\n";
826     std::cout << "    unique_ptr: Perfect! Zero overhead\n";
827     std::cout << "    std::vector: Excellent! Deterministic\n";
828     std::cout << "    shared_ptr: Use sparingly (ref counting cost)\n";
829     std::cout << "    RAII wrappers: Zero cost abstraction\n\n";
830
831
832     std::cout << "Compiler optimizations:\n";
833     std::cout << "    • unique_ptr: Completely optimized away\n";
834     std::cout << "    • Move semantics: No copies, just pointer transfer\n";
835     std::cout << "    • RVO/NRVO: Return value optimization\n";
836     std::cout << "    • Inline expansion: Zero function call overhead\n";
837     std::cout << "\n";
838 }
839
840 } // namespace Performance
841
842 // =====
843 // MAIN DEMONSTRATION
844 // =====
845
846 int main() {
847     std::cout << "\n";
848     std::cout << "
849         =====\n";
850     std::cout << "RESOURCE MANAGEMENT IN MODERN C++\n";
851     std::cout << "
852         =====\n";
853     std::cout << "\"Modern C++ doesn't use new/delete directly anymore!\"\n";
854     std::cout << "
855         =====\n";
856     OldCpp::demonstrate_old_problems();
857     ModernCpp::demonstrate_modern_solutions();
858     RAII::demonstrate_raii();
859     CustomDeleteers::demonstrate_custom_deleteers();
860     RawPointersOK::raw_pointer_examples();
861     Containers::demonstrate_containers();
862     Performance::demonstrate_zero_overhead();
863
864     std::cout << "\n";
865     std::cout << "
866         =====\n";
867     std::cout << "SUMMARY - MODERN C++ RESOURCE MANAGEMENT\n";
868     std::cout << "
869         =====\n";
870     std::cout << "\n";
871     std::cout << "THE GOLDEN RULES:\n";
872     std::cout << "-----\n";
```

```
869 std::cout << "1. NEVER use 'new' or 'delete' directly\n";
870 std::cout << " → Use std::make_unique<T>() instead\n";
871 std::cout << "\n";
872 std::cout << "2. NEVER use 'new[]' or 'delete[]'\n";
873 std::cout << " → Use std::vector<T> or std::array<T, N> instead\n";
874 std::cout << "\n";
875 std::cout << "3. Prefer unique_ptr over shared_ptr\n";
876 std::cout << " → Exclusive ownership is clearer and faster\n";
877 std::cout << "\n";
878 std::cout << "4. Use RAII for ALL resources\n";
879 std::cout << " → Memory, files, locks, sockets, everything!\n";
880 std::cout << "\n";
881 std::cout << "5. Raw pointers are OK for non-owning references\n";
882 std::cout << " → Never responsible for deletion\n";
883 std::cout << "\n";
884 std::cout << "6. Follow the Rule of Zero\n";
885 std::cout << " → Let compiler generate special members\n";
886 std::cout << " → Use standard library types as members\n";
887 std::cout << "\n";
888 std::cout << "BENEFITS:\n";
889 std::cout << "-----\n";
890 std::cout << " No memory leaks (automatic cleanup)\n";
891 std::cout << " Exception-safe (automatic cleanup on unwind)\n";
892 std::cout << " No double-delete bugs\n";
893 std::cout << " No use-after-free bugs\n";
894 std::cout << " Clear ownership semantics\n";
895 std::cout << " Zero runtime overhead (unique_ptr)\n";
896 std::cout << " Easier to reason about code\n";
897 std::cout << " Suitable for embedded systems\n";
898 std::cout << "\n";
899 std::cout << "WHEN YOU STILL SEE new/delete:\n";
900 std::cout << "-----\n";
901 std::cout << "• Legacy code (pre-C++11)\n";
902 std::cout << "• Educational purposes (showing the old way)\n";
903 std::cout << "• Custom memory allocators (advanced)\n";
904 std::cout << "• Framework internals (Qt, COM, etc.)\n";
905 std::cout << "\n";
906 std::cout << "IN MODERN C++ (C++11 and later):\n";
907 std::cout << "-----\n";
908 std::cout << "new/delete are considered OBSOLETE for application code!\n";
909 std::cout << "\n";
910 std::cout << "MIGRATION PATH:\n";
911 std::cout << "-----\n";
912 std::cout << "T* ptr = new T; → auto ptr = std::make_unique<T>()\n";
913 std::cout << ";\n";
914 std::cout << "T* arr = new T[N]; → std::vector<T> arr(N);\n";
915 std::cout << "delete ptr; → (automatic)\n";
916 std::cout << "delete[] arr; → (automatic)\n";
917 std::cout << "T* shared = new T; → auto shared = std::make_shared<T>()\n";
918 std::cout << ";\n";
919 std::cout << "=====
```

```
919     std::cout << "ALL EXAMPLES COMPLETED SUCCESSFULLY!\n";
920     std::cout << "
921         =====\n";
922     return 0;
923 }
924
925 /**
926 EXPECTED OUTPUT HIGHLIGHTS:
927 =====
928 =====
929 =====
930 1. THE OLD PROBLEM - MANUAL MEMORY MANAGEMENT
931 =====
932 OLD C++: Memory leak example
933 -----
934 Allocated int with value: 42
935 Function returns without delete - MEMORY LEAK!
936
937 OLD C++: Exception causes leak
938 -----
939 Allocated array of 1000 ints
940 Caught exception: Simulated error!
941 Array was never deleted - MEMORY LEAK!
942
943 Summary of Problems:
944     Forgot to delete
945     Exception before delete
946     Multiple exit paths
947     Wrong delete operator (delete vs delete[])
948     Double delete
949     Forgot destructor in class
950
951 =====
952 2. MODERN C++ SOLUTION - SMART POINTERS
953 =====
954
955 MODERN C++: unique_ptr (exclusive ownership)
956 -----
957 Created unique_ptr with value: 42
958 Leaving scope - automatic cleanup!
959
960 MODERN C++: Exception-safe with unique_ptr
961 -----
962 Allocated array of 1000 ints
963 Caught exception: Simulated error!
964 unique_ptr automatically cleaned up - NO LEAK!
965
966 Summary of Solutions:
967     No manual new/delete
968     Exception-safe automatically
969     Multiple exit paths safe
```

```
971     Can't use wrong delete operator
972     Can't double-delete
973     Compiler-generated destructor works
974
975 =====
976 3. RAII PATTERN - BEYOND MEMORY
977 =====
978
979 File RAII example:
980     File opened: test.txt
981     File closed: test.txt
982
983 Mutex RAII example:
984     Mutex locked
985     In critical section
986     Mutex unlocked
987
988 RAII Resources:
989     Memory (unique_ptr, shared_ptr)
990     Files (fstream, custom wrappers)
991     Locks (lock_guard, unique_lock)
992     Timers (scope profiling)
993     Network sockets
994     Database connections
995     Hardware registers (embedded)
996
997 =====
998 7. PERFORMANCE - ZERO OVERHEAD
999 =====
1000
1001 Smart pointers have ZERO runtime overhead!
1002
1003 unique_ptr:
1004     Same size as raw pointer
1005     No runtime cost
1006     Optimized to same assembly as manual delete
1007     sizeof(unique_ptr<int>) = 8 bytes
1008     sizeof(int*) = 8 bytes
1009
1010 Embedded systems:
1011     unique_ptr: Perfect! Zero overhead
1012     std::vector: Excellent! Deterministic
1013     shared_ptr: Use sparingly (ref counting cost)
1014     RAII wrappers: Zero cost abstraction
1015
1016 =====
1017 SUMMARY - MODERN C++ RESOURCE MANAGEMENT
1018 =====
1019
1020 THE GOLDEN RULES:
1021 -----
1022 1. NEVER use 'new' or 'delete' directly
1023     Use std::make_unique<T>() instead
1024
```

```
1025 2. NEVER use 'new []' or 'delete []' →  
1026     Use std::vector<T> or std::array<T, N> instead  
1027  
1028 3. Prefer unique_ptr over shared_ptr →  
1029     Exclusive ownership is clearer and faster  
1030  
1031 4. Use RAII for ALL resources →  
1032     Memory, files, locks, sockets, everything!  
1033  
1034 5. Raw pointers are OK for non-owning references →  
1035     Never responsible for deletion  
1036  
1037 BENEFITS:  
1038 -----  
1039 No memory leaks (automatic cleanup)  
1040 Exception-safe (automatic cleanup on unwind)  
1041 No double-delete bugs  
1042 No use-after-free bugs  
1043 Clear ownership semantics  
1044 Zero runtime overhead (unique_ptr)  
1045 Suitable for embedded systems  
1046  
1047 IN MODERN C++ (C++11 and later):  
1048 -----  
1049 new/delete are considered OBSOLETE for application code!  
1050  
1051 */
```

## 59 Source Code: RestApiExample.cpp

**File:** src/RestApiExample.cpp

**Repository:** [View on GitHub](#)

```
1 // =====
2 // REST API CALLS IN C++ EXAMPLE
3 // =====
4 // This example demonstrates making REST API calls in modern C++ using:
5 // - libcurl: For HTTP/HTTPS requests
6 // - nlohmann/json: For JSON parsing and serialization
7 //
8 // TOPICS COVERED:
9 // 1. GET requests with query parameters
10 // 2. POST requests with JSON payloads
11 // 3. PUT and DELETE requests
12 // 4. Custom headers and authentication
13 // 5. Error handling for network operations
14 // 6. RAII wrapper for curl resources
15 // 7. Response parsing with JSON
16 //
17 // WHAT IS libcurl?
18 // - A free and easy-to-use client-side URL transfer library
19 // - Supports HTTP, HTTPS, FTP, and many other protocols
20 // - Thread-safe when used correctly
21 // - Cross-platform (Windows, Linux, macOS)
22 //
23 // WHY USE libcurl?
24 // Battle-tested: Used in billions of devices
25 // Feature-rich: Supports all HTTP methods and features
26 // Cross-platform: Works everywhere
27 // Well-documented: Extensive documentation and examples
28 // Active development: Regular updates and security fixes
29 //
30 // INSTALLATION:
31 // - Ubuntu: sudo apt-get install libcurl4-openssl-dev
32 // - Windows (vcpkg): vcpkg install curl
33 // - macOS: brew install curl
34 // - CMake: find_package(CURL REQUIRED)
35 //
36 // ALTERNATIVES TO CONSIDER:
37 // - cpp-httplib: Header-only, simpler but less features
38 // - Boost.Beast: Part of Boost, good for async operations
39 // - cpprestsdk: Microsoft's REST SDK, good for async
40 // - Qt Network: If already using Qt framework
41 //
42 // =====
43
44 #include <iostream>
45 #include <string>
46 #include <memory>
47 #include <stdexcept>
48 #include <vector>
49 #include <map>
```

```
50 #include <curl/curl.h>
51 #include <nlohmann/json.hpp>
52
53 using json = nlohmann::json;
54
55 // =====
56 // SECTION 1: RAII Wrapper for CURL
57 // =====
58 // Modern C++ principle: Use RAII to manage resources automatically
59 // This ensures curl_easy_cleanup is always called, even on exceptions
60
61 class CurlHandle {
62 private:
63     CURL* handle_;
64
65 public:
66     CurlHandle() : handle_(curl_easy_init()) {
67         if (!handle_) {
68             throw std::runtime_error("Failed to initialize CURL");
69         }
70     }
71
72     ~CurlHandle() {
73         if (handle_) {
74             curl_easy_cleanup(handle_);
75         }
76     }
77
78     // Delete copy constructor and assignment (non-copyable)
79     CurlHandle(const CurlHandle&) = delete;
80     CurlHandle& operator=(const CurlHandle&) = delete;
81
82     // Allow move semantics
83     CurlHandle(CurlHandle&& other) noexcept : handle_(other.handle_) {
84         other.handle_ = nullptr;
85     }
86
87     CurlHandle& operator=(CurlHandle&& other) noexcept {
88         if (this != &other) {
89             if (handle_) {
90                 curl_easy_cleanup(handle_);
91             }
92             handle_ = other.handle_;
93             other.handle_ = nullptr;
94         }
95         return *this;
96     }
97
98     CURL* get() const { return handle_; }
99     operator CURL*() const { return handle_; }
100 };
101
102 // =====
103 // SECTION 2: RAII Wrapper for curl_slist (headers)
```

```
104 // =====
105
106 class CurlHeaders {
107 private:
108     struct curl_slist* headers_;
109
110 public:
111     CurlHeaders() : headers_(nullptr) {}
112
113     ~CurlHeaders() {
114         if (headers_) {
115             curl_slist_free_all(headers_);
116         }
117     }
118
119     CurlHeaders(const CurlHeaders&) = delete;
120     CurlHeaders& operator=(const CurlHeaders&) = delete;
121
122     void append(const std::string& header) {
123         headers_ = curl_slist_append(headers_, header.c_str());
124     }
125
126     struct curl_slist* get() const { return headers_; }
127 };
128 // =====
129 // SECTION 3: HTTP Response Structure
130 // =====
131
132 struct HttpResponse {
133     long status_code;
134     std::string body;
135     std::map<std::string, std::string> headers;
136
137     bool is_success() const {
138         return status_code >= 200 && status_code < 300;
139     }
140
141     json to_json() const {
142         return json::parse(body);
143     }
144 };
145
146 // =====
147 // SECTION 4: Callback for Writing Response Data
148 // =====
149 // libcurl uses C-style callbacks. We use a static function that
150 // calls into our C++ code.
151
152 static size_t WriteCallback(void* contents, size_t size, size_t nmemb, void*
153 userp) {
154     size_t total_size = size * nmemb;
155     auto* str = static_cast<std::string*>(userp);
156     str->append(static_cast<char*>(contents), total_size);
```

```
157     return total_size;
158 }
159
160 // =====
161 // SECTION 5: REST API Client Class
162 // =====
163
164 class RestClient {
165 private:
166     std::string base_url_;
167     std::map<std::string, std::string> default_headers_;
168
169 public:
170     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);  
    }
```

```
208     }
209
210     // PUT request with JSON body
211     HttpResponse put(const std::string& endpoint,
212                         const json& body,
213                         const std::map<std::string, std::string>& extra_headers =
214                         {}) {
215         std::string url = build_url(endpoint);
216         auto headers = extra_headers;
217         headers["Content-Type"] = "application/json";
218         return perform_request(url, "PUT", body.dump(), headers);
219     }
220
221     // DELETE request
222     HttpResponse del(const std::string& endpoint) {
223         std::string url = build_url(endpoint);
224         return perform_request(url, "DELETE", "", {});
225     }
226
227     private:
228         std::string build_url(const std::string& endpoint,
229                               const std::map<std::string, std::string>& params =
230                               {}) {
231             std::string url = base_url_ + endpoint;
232
233             if (!params.empty()) {
234                 url += "?";
235                 bool first = true;
236                 for (const auto& [key, value] : params) {
237                     if (!first) url += "&";
238                     url += key + "=" + curl_easy_escape_string(value);
239                     first = false;
240                 }
241             }
242
243             return url;
244         }
245
246         std::string curl_easy_escape_string(const std::string& str) {
247             CurlHandle curl;
248             char* escaped = curl_easy_escape(curl, str.c_str(), str.length());
249             std::string result(escaped);
250             curl_free(escaped);
251             return result;
252         }
253
254         HttpResponse perform_request(const std::string& url,
255                                     const std::string& method,
256                                     const std::string& body,
257                                     const std::map<std::string, std::string>&
258                                     extra_headers) {
259             CurlHandle curl;
260             HttpResponse response;
261             std::string response_body;
```

```
259     // Set URL
260     curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
261
262     // Set HTTP method
263     if (method == "POST") {
264         curl_easy_setopt(curl, CURLOPT_POST, 1L);
265     } else if (method == "PUT") {
266         curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "PUT");
267     } else if (method == "DELETE") {
268         curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "DELETE");
269     } else if (method == "GET") {
270         curl_easy_setopt(curl, CURLOPT_HTTPGET, 1L);
271     }
272
273     // Set request body if provided
274     if (!body.empty()) {
275         curl_easy_setopt(curl, CURLOPT_POSTFIELDS, body.c_str());
276         curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, body.length());
277     }
278
279     // Set headers
280     CurlHeaders headers;
281
282     // Add default headers
283     for (const auto& [key, value] : default_headers_) {
284         headers.append(key + ": " + value);
285     }
286
287     // Add extra headers (can override defaults)
288     for (const auto& [key, value] : extra_headers_) {
289         headers.append(key + ": " + value);
290     }
291
292     if (headers.get()) {
293         curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers.get());
294     }
295
296     // Set write callback
297     curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
298     curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response_body);
299
300     // Follow redirects
301     curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
302
303     // Set timeout
304     curl_easy_setopt(curl, CURLOPT_TIMEOUT, 30L);
305
306     // Perform request
307     CURLcode res = curl_easy_perform(curl);
308
309     if (res != CURLE_OK) {
310         throw std::runtime_error(std::string("CURL error: ") +
311                               curl_easy_strerror(res));
312     }
```

```
313     }
314
315     // Get response code
316     curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response.status_code)
317     ;
318     response.body = response_body;
319
320     return response;
321 }
322
323 // =====
324 // SECTION 6: Example Usage with Public APIs
325 // =====
326
327 void example_json_placeholder_api() {
328     std::cout << "\n==== Example 1: JSONPlaceholder API (GET) ===\n";
329
330     try {
331         RestClient client("https://jsonplaceholder.typicode.com");
332
333         // GET request: Fetch a post
334         auto response = client.get("/posts/1");
335
336         std::cout << "Status Code: " << response.status_code << "\n";
337
338         if (response.is_success()) {
339             auto data = response.to_json();
340             std::cout << "Post Title: " << data["title"] << "\n";
341             std::cout << "Post Body: " << data["body"] << "\n";
342             std::cout << "User ID: " << data["userId"] << "\n";
343         }
344
345     } catch (const std::exception& e) {
346         std::cerr << "Error: " << e.what() << "\n";
347     }
348 }
349
350 void example_create_post() {
351     std::cout << "\n==== Example 2: Create a Post (POST) ===\n";
352
353     try {
354         RestClient client("https://jsonplaceholder.typicode.com");
355
356         // Create JSON payload
357         json post_data = {
358             {"title", "My C++ REST API Example"},
359             {"body", "This post was created using C++ and libcurl!"},
360             {"userId", 1}
361         };
362
363         // POST request
364         auto response = client.post("/posts", post_data);
365     }
```

```
366     std::cout << "Status Code: " << response.status_code << "\n";
367
368     if (response.is_success()) {
369         auto data = response.to_json();
370         std::cout << "Created Post ID: " << data["id"] << "\n";
371         std::cout << "Response:\n" << data.dump(2) << "\n";
372     }
373
374 } catch (const std::exception& e) {
375     std::cerr << "Error: " << e.what() << "\n";
376 }
377 }
378
379 void example_update_post() {
380     std::cout << "\n==== Example 3: Update a Post (PUT) ====\n";
381
382     try {
383         RestClient client("https://jsonplaceholder.typicode.com");
384
385         // Updated data
386         json updated_data = {
387             {"id", 1},
388             {"title", "Updated Title from C++"},
389             {"body", "This post was updated using modern C++!"},
390             {"userId", 1}
391         };
392
393         // PUT request
394         auto response = client.put("/posts/1", updated_data);
395
396         std::cout << "Status Code: " << response.status_code << "\n";
397
398         if (response.is_success()) {
399             auto data = response.to_json();
400             std::cout << "Updated Post:\n" << data.dump(2) << "\n";
401         }
402
403     } catch (const std::exception& e) {
404         std::cerr << "Error: " << e.what() << "\n";
405     }
406 }
407
408 void example_delete_post() {
409     std::cout << "\n==== Example 4: Delete a Post (DELETE) ====\n";
410
411     try {
412         RestClient client("https://jsonplaceholder.typicode.com");
413
414         // DELETE request
415         auto response = client.del("/posts/1");
416
417         std::cout << "Status Code: " << response.status_code << "\n";
418
419         if (response.is_success()) {
```

```
420         std::cout << "Post deleted successfully!\n";
421     }
422
423 } catch (const std::exception& e) {
424     std::cerr << "Error: " << e.what() << "\n";
425 }
426 }
427
428 void example_with_query_parameters() {
429     std::cout << "\n==== Example 5: GET with Query Parameters ====\n";
430
431     try {
432         RestClient client("https://jsonplaceholder.typicode.com");
433
434         // GET with query parameters
435         std::map<std::string, std::string> params = {
436             {"userId", "1"}
437         };
438
439         auto response = client.get("/posts", params);
440
441         std::cout << "Status Code: " << response.status_code << "\n";
442
443         if (response.is_success()) {
444             auto data = response.to_json();
445             std::cout << "Found " << data.size() << " posts by user 1\n";
446
447             // Display first 3 posts
448             for (size_t i = 0; i < std::min(data.size(), size_t(3)); ++i) {
449                 std::cout << "\nPost " << (i + 1) << ":\n";
450                 std::cout << "  Title: " << data[i]["title"] << "\n";
451             }
452         }
453
454     } catch (const std::exception& e) {
455         std::cerr << "Error: " << e.what() << "\n";
456     }
457 }
458
459 void example_github_api() {
460     std::cout << "\n==== Example 6: GitHub API (Public Data) ====\n";
461
462     try {
463         RestClient client("https://api.github.com");
464
465         // GitHub requires User-Agent header
466         client.set_default_header("User-Agent", "ModernCppExamples/1.0");
467
468         // GET user information
469         auto response = client.get("/users/torvalds");
470
471         std::cout << "Status Code: " << response.status_code << "\n";
472
473         if (response.is_success()) {
```

```
474     auto data = response.to_json();
475     std::cout << "GitHub User Info:\n";
476     std::cout << "  Name: " << data.value("name", "N/A") << "\n";
477     std::cout << "  Login: " << data["login"] << "\n";
478     std::cout << "  Public Repos: " << data["public_repos"] << "\n";
479     std::cout << "  Followers: " << data["followers"] << "\n";
480 }
481
482 } catch (const std::exception& e) {
483     std::cerr << "Error: " << e.what() << "\n";
484 }
485 }
486
487 void example_error_handling() {
488     std::cout << "\n==== Example 7: Error Handling ===\n";
489
490     try {
491         RestClient client("https://jsonplaceholder.typicode.com");
492
493         // Try to access a non-existent resource (404)
494         auto response = client.get("/posts/999999");
495
496         std::cout << "Status Code: " << response.status_code << "\n";
497
498         if (!response.is_success()) {
499             std::cout << "Request failed!\n";
500             std::cout << "Response body: " << response.body << "\n";
501         }
502
503     } catch (const std::exception& e) {
504         std::cerr << "Error: " << e.what() << "\n";
505     }
506 }
507
508 // =====
509 // SECTION 7: Best Practices Summary
510 // =====
511
512 void print_best_practices() {
513     std::cout << "\n" << std::string(70, '=') << "\n";
514     std::cout << "REST API BEST PRACTICES IN C++:\n";
515     std::cout << std::string(70, '=') << "\n\n";
516
517     std::cout << "1. Resource Management (RAII):\n";
518     std::cout << "    Use RAII wrappers for CURL handles\n";
519     std::cout << "    Ensure cleanup happens automatically\n";
520     std::cout << "    Exception-safe resource handling\n\n";
521
522     std::cout << "2. Error Handling:\n";
523     std::cout << "    Check CURLcode return values\n";
524     std::cout << "    Check HTTP status codes\n";
525     std::cout << "    Use exceptions for network errors\n";
526     std::cout << "    Parse JSON safely with try-catch\n\n";
527 }
```

```
528     std::cout << "3. Security:\n";
529     std::cout << "    Always use HTTPS for sensitive data\n";
530     std::cout << "    Validate SSL certificates (enabled by default)\n";
531     std::cout << "    Never hardcode API keys (use environment variables)\n";
532     std::cout << "    Use Bearer tokens for authentication\n\n";
533
534     std::cout << "4. Performance:\n";
535     std::cout << "    Reuse RestClient instances when possible\n";
536     std::cout << "    Set appropriate timeouts\n";
537     std::cout << "    Consider connection pooling for many requests\n";
538     std::cout << "    Use HTTP/2 when supported\n\n";
539
540     std::cout << "5. Modern C++ Features:\n";
541     std::cout << "    Use smart pointers and RAI\n";
542     std::cout << "    Leverage move semantics\n";
543     std::cout << "    Use structured bindings for map iteration\n";
544     std::cout << "    std::optional for nullable values\n\n";
545
546     std::cout << "6. Testing:\n";
547     std::cout << "    Mock HTTP responses for unit tests\n";
548     std::cout << "    Test error conditions\n";
549     std::cout << "    Use test APIs like JSONPlaceholder\n\n";
550 }
551
552 // =====
553 // MAIN FUNCTION
554 // =====
555
556 int main() {
557     std::cout << "\n";
558     std::cout << "                                \n";
559     std::cout << "                                REST API CALLS IN C++ - Complete Examples
560     std::cout << "                                \n";
561     std::cout << "    Demonstrates HTTP methods: GET, POST, PUT, DELETE
562     std::cout << "                                \n";
563     std::cout << "    Using: libcurl + nlohmann::json
564     std::cout << "                                \n";
565     std::cout << "                                \n";
566
567     // Run all examples
568     example_json_placeholder_api();
569     example_create_post();
570     example_update_post();
571     example_delete_post();
572     example_with_query_parameters();
573     example_github_api();
574     example_error_handling();
575
576     // Print best practices
577     print_best_practices();
578
579     std::cout << "\n All examples completed successfully!\n";
```

```
578     std::cout << "\nNOTE: These examples use public test APIs:\n";
579     std::cout << "  - JSONPlaceholder (https://jsonplaceholder.typicode.com)\n";
580     std::cout << "  - GitHub API (https://api.github.com)\n";
581     std::cout << "\nNo API key required. Responses are simulated/cached.\n\n";
582
583     return 0;
584 }
```

## 60 Source Code: RuleOf3\_5\_0.cpp

File: src/RuleOf3\_5\_0.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <memory>
4 #include <utility>
5 #include <vector>
6
7 // =====
8 // C++ SPECIAL MEMBER FUNCTIONS: RULE OF 3, 5, AND 0
9 // =====
10
11 // =====
12 // 1. RULE OF ZERO
13 // =====
14 // If you can avoid defining any special member functions (destructor,
15 // copy/move constructors, copy/move assignment), do so. Let the compiler
16 // generate them automatically. Use RAII wrappers like std::unique_ptr,
17 // std::shared_ptr, std::vector, std::string, etc.
18
19 class RuleOfZeroExample {
20 private:
21     std::string name;
22     std::vector<int> data;
23     std::unique_ptr<int> ptr;
24
25 public:
26     RuleOfZeroExample(const std::string& n, int value)
27         : name(n), data{1, 2, 3, 4, 5}, ptr(std::make_unique<int>(value)) {
28         std::cout << "RuleOfZero: Constructor for " << name << std::endl;
29     }
30
31     // No destructor needed - std::string, std::vector, std::unique_ptr
32     // handle cleanup automatically
33
34     // No copy constructor needed - compiler generates correct one
35     // (note: std::unique_ptr makes this non-copyable by default)
36
37     // No copy assignment needed
38
39     // No move constructor needed - compiler generates efficient one
40
41     // No move assignment needed
42
43     void display() const {
44         std::cout << " Name: " << name << ", Value: " << *ptr
45             << ", Data size: " << data.size() << std::endl;
46     }
47 };
48
49 void example_rule_of_zero() {
```

```
50    std::cout << "\n==== RULE OF ZERO ===" << std::endl;
51    std::cout << "Use RAII wrappers. No special member functions needed!\n" <<
52        std::endl;
53
54    RuleOfZeroExample obj1("Object1", 42);
55    obj1.display();
56
57    // Move semantics work automatically
58    RuleOfZeroExample obj2 = std::move(obj1);
59    obj2.display();
60
61    std::cout << "\n Compiler-generated special members handle everything
62        correctly" << std::endl;
63
64 // =====
65 // 2. RULE OF THREE (Pre-C++11)
66 // =====
67 // If you define any of the following, you should define all three:
68 // 1. Destructor
69 // 2. Copy constructor
70 // 3. Copy assignment operator
71
72 class RuleOfThreeExample {
73 private:
74     int* data;
75     size_t size;
76     std::string name;
77
78 public:
79     // Constructor
80     RuleOfThreeExample(const std::string& n, size_t s)
81         : data(new int[s]), size(s), name(n) {
82         for (size_t i = 0; i < size; ++i) {
83             data[i] = static_cast<int>(i);
84         }
85         std::cout << "RuleOfThree: Constructor for " << name
86             << " (size=" << size << ")" << std::endl;
87     }
88
89     // 1. Destructor
90     ~RuleOfThreeExample() {
91         std::cout << "RuleOfThree: Destructor for " << name << std::endl;
92         delete[] data;
93     }
94
95     // 2. Copy constructor
96     RuleOfThreeExample(const RuleOfThreeExample& other)
97         : data(new int[other.size]), size(other.size), name(other.name + " _copy") {
98         for (size_t i = 0; i < size; ++i) {
99             data[i] = other.data[i];
100        }
100        std::cout << "RuleOfThree: Copy constructor for " << name << std::endl
```

```
    ;
101 }
102
103 // 3. Copy assignment operator
104 RuleOfThreeExample& operator=(const RuleOfThreeExample& other) {
105     std::cout << "RuleOfThree: Copy assignment for " << name << std::endl;
106
107     if (this != &other) {
108         // Free existing resource
109         delete[] data;
110
111         // Allocate new resource and copy
112         size = other.size;
113         data = new int[size];
114         for (size_t i = 0; i < size; ++i) {
115             data[i] = other.data[i];
116         }
117         name = other.name + "_assigned";
118     }
119     return *this;
120 }
121
122 void display() const {
123     std::cout << " " << name << " data: [";
124     for (size_t i = 0; i < std::min(size, size_t(5)); ++i) {
125         std::cout << data[i] << (i < std::min(size, size_t(5)) - 1 ? ", " : "");
126     }
127     std::cout << (size > 5 ? "..." : "") << "]" << std::endl;
128 }
129 };
130
131 void example_rule_of_three() {
132     std::cout << "\n==== RULE OF THREE ===" << std::endl;
133     std::cout << "Define: Destructor, Copy Constructor, Copy Assignment\n" <<
134         std::endl;
135
136     RuleOfThreeExample obj1("Original", 10);
137     obj1.display();
138
139     // Copy constructor
140     RuleOfThreeExample obj2 = obj1;
141     obj2.display();
142
143     // Copy assignment
144     RuleOfThreeExample obj3("Another", 5);
145     obj3 = obj1;
146     obj3.display();
147
148     std::cout << "\n All three special members ensure proper resource
149         management" << std::endl;
150 }
```

```
151 // 3. RULE OF FIVE (C++11 and later)
152 // =====
153 // If you define any of the following, you should define all five:
154 // 1. Destructor
155 // 2. Copy constructor
156 // 3. Copy assignment operator
157 // 4. Move constructor
158 // 5. Move assignment operator
159
160 class RuleOfFiveExample {
161 private:
162     int* data;
163     size_t size;
164     std::string name;
165
166 public:
167     // Constructor
168     RuleOfFiveExample(const std::string& n, size_t s)
169         : data(new int[s]), size(s), name(n) {
170         for (size_t i = 0; i < size; ++i) {
171             data[i] = static_cast<int>(i * 10);
172         }
173         std::cout << "RuleOfFive: Constructor for " << name
174             << " (size=" << size << ")" << std::endl;
175     }
176
177     // 1. Destructor
178     ~RuleOfFiveExample() {
179         std::cout << "RuleOfFive: Destructor for " << name << std::endl;
180         delete[] data;
181     }
182
183     // 2. Copy constructor
184     RuleOfFiveExample(const RuleOfFiveExample& other)
185         : data(new int[other.size]), size(other.size), name(other.name + " _copy") {
186         for (size_t i = 0; i < size; ++i) {
187             data[i] = other.data[i];
188         }
189         std::cout << "RuleOfFive: Copy constructor for " << name << std::endl;
190     }
191
192     // 3. Copy assignment operator
193     RuleOfFiveExample& operator=(const RuleOfFiveExample& other) {
194         std::cout << "RuleOfFive: Copy assignment for " << name << std::endl;
195
196         if (this != &other) {
197             delete[] data;
198             size = other.size;
199             data = new int[size];
200             for (size_t i = 0; i < size; ++i) {
201                 data[i] = other.data[i];
202             }
203             name = other.name + " _copy_assigned";
204         }
205     }
206 }
```

```
204     }
205     return *this;
206 }
207
208 // 4. Move constructor
209 RuleOfFiveExample(RuleOfFiveExample&& other) noexcept
210     : data(other.data), size(other.size), name(std::move(other.name) + "  
_moved") {
211     other.data = nullptr;
212     other.size = 0;
213     std::cout << "RuleOfFive: Move constructor for " << name << std::endl;
214 }
215
216 // 5. Move assignment operator
217 RuleOfFiveExample& operator=(RuleOfFiveExample&& other) noexcept {
218     std::cout << "RuleOfFive: Move assignment for " << name << std::endl;
219
220     if (this != &other) {
221         delete[] data;
222
223         data = other.data;
224         size = other.size;
225         name = std::move(other.name) + "_move_assigned";
226
227         other.data = nullptr;
228         other.size = 0;
229     }
230     return *this;
231 }
232
233 void display() const {
234     if (data && size > 0) {
235         std::cout << " " << name << " data: [";
236         for (size_t i = 0; i < std::min(size, size_t(5)); ++i) {
237             std::cout << data[i] << (i < std::min(size, size_t(5)) - 1 ? "  
, " : "");
238         }
239         std::cout << (size > 5 ? "..." : "") << "]" << std::endl;
240     } else {
241         std::cout << " " << name << " (moved-from, empty)" << std::endl;
242     }
243 }
244 };
245
246 void example_rule_of_five() {
247     std::cout << "\n==== RULE OF FIVE ====" << std::endl;
248     std::cout << "Define: Destructor, Copy Constructor, Copy Assignment, "  
     << "Move Constructor, Move Assignment\n" << std::endl;
249
250     RuleOfFiveExample obj1("Original", 8);
251     obj1.display();
252
253     // Copy constructor
254     RuleOfFiveExample obj2 = obj1;
```

```
256     obj2.display();  
257  
258     // Move constructor  
259     RuleOfFiveExample obj3 = std::move(obj1);  
260     obj3.display();  
261     std::cout << "After move:" << std::endl;  
262     obj1.display(); // obj1 is in valid but moved-from state  
263  
264     // Copy assignment  
265     RuleOfFiveExample obj4("Target1", 3);  
266     obj4 = obj2;  
267     obj4.display();  
268  
269     // Move assignment  
270     RuleOfFiveExample obj5("Target2", 3);  
271     obj5 = std::move(obj2);  
272     obj5.display();  
273     std::cout << "After move:" << std::endl;  
274     obj2.display(); // obj2 is in valid but moved-from state  
275  
276     std::cout << "\n All five special members handle both copy and move  
     semantics" << std::endl;  
277 }  
278  
279 // =====  
280 // 4. MODERN C++17/20 RELEVANCE  
281 // =====  
282  
283 #include <optional>  
284 #include <variant>  
285  
286 class ModernCpp17Example {  
287 private:  
288     std::string name;  
289     std::vector<int> data;  
290     std::optional<std::string> description; // C++17  
291     std::variant<int, double, std::string> value; // C++17  
292  
293 public:  
294     ModernCpp17Example(const std::string& n)  
295         : name(n), data{1, 2, 3}, description("A modern class"), value(42) {  
296         std::cout << "Modern C++17 class: " << name << std::endl;  
297     }  
298  
299     // NO special member functions needed!  
300     // Compiler generates everything correctly, including:  
301     // - Destructor (cleans up all RAII members)  
302     // - Copy constructor (deep copies std::string, std::vector, std::optional  
     , std::variant)  
303     // - Copy assignment (properly assigns all members)  
304     // - Move constructor (efficient moves for all members)  
305     // - Move assignment (efficient move assigns)  
306  
307     void display() const {
```

```
308     std::cout << "  Name: " << name;
309     if (description) {
310         std::cout << ", Desc: " << *description;
311     }
312     std::cout << ", Variant holds: ";
313     std::visit([](auto&& arg) { std::cout << arg; }, value);
314     std::cout << std::endl;
315 }
316 };
317
318 void example_modern_cpp_relevance() {
319     std::cout << "\n==== MODERN C++17/20 RELEVANCE ===" << std::endl;
320
321     std::cout << "\n RULE OF ZERO IS MORE RELEVANT THAN EVER!" << std::endl;
322     std::cout << "\nC++17/20 additions that strengthen Rule of Zero:" << std::endl;
323     std::cout << " • std::optional<T> - Optional values (no more raw pointers
324     !)" << std::endl;
325     std::cout << " • std::variant<Ts...> - Type-safe unions" << std::endl;
326     std::cout << " • std::any - Type-safe void*" << std::endl;
327     std::cout << " • std::string_view - Non-owning string references" << std::endl;
328     std::cout << " • std::span<T> (C++20) - Non-owning array views" << std::endl;
329     std::cout << " • std::make_shared array support (C++20)" << std::endl;
330
331     ModernCpp17Example obj1("Original");
332     obj1.display();
333
334     std::cout << "\nCopying (compiler-generated):" << std::endl;
335     ModernCpp17Example obj2 = obj1;
336     obj2.display();
337
338     std::cout << "\nMoving (compiler-generated):" << std::endl;
339     ModernCpp17Example obj3 = std::move(obj1);
340     obj3.display();
341
342 }
343
344 // =====
345 // 5. COMPARISON: WHEN TO USE WHICH RULE
346 // =====
347
348 void example_comparison() {
349     std::cout << "\n==== WHEN TO USE WHICH RULE (2026 PERSPECTIVE) ===" << std::endl;
350
351     std::cout << "\n RULE OF ZERO (STRONGLY PREFERRED - 95%+ of cases):" << std::endl;
352     std::cout << "    Default choice for ALL new C++17/20 code" << std::endl;
353     std::cout << "    Use std::unique_ptr, std::shared_ptr, std::vector, std::
354     string" << std::endl;
```

```
354     std::cout << "    Use std::optional, std::variant, std::any (C++17)" <<
355         std::endl;
356     std::cout << "    Use std::span for views (C++20)" << std::endl;
357     std::cout << "    Let compiler generate ALL special members" << std::endl;
358     std::cout << "    Zero bugs from manual memory management" << std::endl;
359     std::cout << "    MOST RELEVANT IN MODERN C++!" << std::endl;
360
361     std::cout << "\n RULE OF THREE (LARGELY OBSOLETE):" << std::endl;
362     std::cout << "    Pre-C++11 guideline - avoid in new code" << std::endl;
363     std::cout << "    Maintain in legacy codebases only" << std::endl;
364     std::cout << "    Refactor to Rule of Zero when possible" << std::endl;
365     std::cout << "    Don't use for new C++17/20 code" << std::endl;
366
367     std::cout << "\n RULE OF FIVE (RARELY NEEDED - <5% of cases):" << std::endl;
368     std::cout << "    Only when Rule of Zero is impossible" << std::endl;
369     std::cout << "    Custom allocators or exotic resources" << std::endl;
370     std::cout << "    Performance-critical kernel/driver code" << std::endl;
371     std::cout << "    Interop with C libraries or hardware" << std::endl;
372     std::cout << "    Consider if you really need it first" << std::endl;
373
374     std::cout << "\n MODERN C++17/20 BEST PRACTICES:" << std::endl;
375     std::cout << "    1. ALWAYS start with Rule of Zero" << std::endl;
376     std::cout << "    2. Use std::optional/variant instead of raw pointers" <<
377         std::endl;
378     std::cout << "    3. Use std::unique_ptr for ownership" << std::endl;
379     std::cout << "    4. Use std::span for non-owning views (C++20)" << std::endl;
380     std::cout << "    5. Only implement special members if profiling proves
381         necessary" << std::endl;
382     std::cout << "    6. Use = default and = delete for explicit intent" << std::endl;
383     std::cout << "    7. Mark move operations noexcept (enables optimizations)" <<
384         std::endl;
385
386     std::cout << "\n REALITY CHECK (2026):" << std::endl;
387     std::cout << "    • 95%+ of classes should use Rule of Zero" << std::endl;
388     std::cout << "    • <5% need Rule of Five (custom resource managers)" << std::endl;
389     std::cout << "    • Rule of Three is legacy knowledge only" << std::endl;
390     std::cout << "    • Modern C++ makes manual memory management unnecessary"
391         << std::endl;
392 }
393
394 // =====
395 // 5. EXPLICIT CONTROL: DEFAULT AND DELETE
396 // =====
397
398 class DefaultAndDeleteExample {
399 private:
400     std::unique_ptr<int> ptr;
401
402 public:
403     DefaultAndDeleteExample() : ptr(std::make_unique<int>(42)) {}
```

```
399     // Explicitly defaulted destructor
400     ~DefaultAndDeleteExample() = default;
401
402     // Deleted copy operations (non-copyable)
403     DefaultAndDeleteExample(const DefaultAndDeleteExample&) = delete;
404     DefaultAndDeleteExample& operator=(const DefaultAndDeleteExample&) =
405         delete;
406
407     // Explicitly defaulted move operations
408     DefaultAndDeleteExample(DefaultAndDeleteExample&&) = default;
409     DefaultAndDeleteExample& operator=(DefaultAndDeleteExample&&) = default;
410
411     int getValue() const { return *ptr; }
412 };
413
414 void example_default_and_delete() {
415     std::cout << "\n==== DEFAULT AND DELETE ===" << std::endl;
416     std::cout << "Explicitly control which special members are available\n" <<
417         std::endl;
418
419     DefaultAndDeleteExample obj1;
420     std::cout << "Created obj1, value: " << obj1.getValue() << std::endl;
421
422     // DefaultAndDeleteExample obj2 = obj1; // ERROR: Copy deleted
423
424     DefaultAndDeleteExample obj3 = std::move(obj1); // OK: Move defaulted
425     std::cout << "Moved to obj3, value: " << obj3.getValue() << std::endl;
426
427     std::cout << "\n = delete prevents copying, = default enables moving" <<
428         std::endl;
429 }
430
431 // =====
432 // 6. EXPLICIT KEYWORD - PREVENTING IMPLICIT CONVERSIONS
433 // =====
434 // The explicit keyword prevents implicit type conversions that could
435 // lead to unexpected behavior or resource management issues
436
437 // BAD: Without explicit - allows implicit conversions
438 class ImplicitBuffer {
439     private:
440         std::unique_ptr<int[]> data;
441         size_t size;
442
443     public:
444         // Constructor without explicit - DANGEROUS!
445         ImplicitBuffer(size_t s) : data(std::make_unique<int[]>(s)), size(s) {
446             std::cout << "ImplicitBuffer: Allocated " << size << " ints" << std::
447                 endl;
448         }
449
450         size_t getSize() const { return size; }
451
452 };
```

```
449
450 // GOOD: With explicit - prevents implicit conversions
451 class ExplicitBuffer {
452 private:
453     std::unique_ptr<int[]> data;
454     size_t size;
455
456 public:
457     // Constructor with explicit - SAFE!
458     explicit ExplicitBuffer(size_t s) : data(std::make_unique<int[]>(s)), size(s) {
459         std::cout << "ExplicitBuffer: Allocated " << size << " ints" << std::endl;
460     }
461
462     size_t getSize() const { return size; }
463 };
464
465 // Example with conversion operators
466 class SmartInt {
467 private:
468     int value;
469
470 public:
471     explicit SmartInt(int v) : value(v) {}
472
473     // C++11: explicit conversion operator
474     explicit operator int() const { return value; }
475
476     // Implicit conversion would be dangerous for bool
477     explicit operator bool() const { return value != 0; }
478 };
479
480 void processBuffer(const ExplicitBuffer& buf) {
481     std::cout << " Processing buffer of size: " << buf.getSize() << std::endl;
482 }
483
484 void processImplicitBuffer(const ImplicitBuffer& buf) {
485     std::cout << " Processing implicit buffer of size: " << buf.getSize() << std::endl;
486 }
487
488 void example_explicit_keyword() {
489     std::cout << "\n== EXPLICIT KEYWORD ==" << std::endl;
490     std::cout << "Prevents dangerous implicit conversions\n" << std::endl;
491
492     // Without explicit - COMPILES BUT DANGEROUS!
493     std::cout << "Without explicit keyword:" << std::endl;
494     ImplicitBuffer buf1(10); // OK: Direct initialization
495
496     // DANGEROUS: Implicit conversion from int to ImplicitBuffer!
497     processImplicitBuffer(100); // Creates temporary ImplicitBuffer(100) - wasteful!
```

```
498 // Can even do this nonsense:
499 ImplicitBuffer buf2 = 50; // Implicit conversion - looks like int
500 // assignment!
501 std::cout << " Created buffer with = 50 (confusing!)" << std::endl;
502
503 std::cout << "\nWith explicit keyword:" << std::endl;
504 ExplicitBuffer buf3(10); // OK: Direct initialization
505
506 // processBuffer(100); // ERROR: Cannot implicitly convert int to
507 // ExplicitBuffer
508 // ExplicitBuffer buf4 = 50; // ERROR: Cannot use copy initialization
509
510 std::cout << " Must use direct initialization: ExplicitBuffer(100)" <<
511 // std::endl;
512 processBuffer(ExplicitBuffer(100)); // OK: Explicit construction
513
514 // Explicit conversion operators (C++11)
515 std::cout << "\nExplicit conversion operators:" << std::endl;
516 SmartInt smart(42);
517
518 // int x = smart; // ERROR: Cannot implicitly convert to int
519 int x = static_cast<int>(smart); // OK: Explicit cast required
520 std::cout << " Explicit cast to int: " << x << std::endl;
521
522 // if (smart) {} // OK: Contextual conversion to bool allowed
523 std::cout << " Contextual bool conversion works in if/while statements"
524 // << std::endl;
525
526 std::cout << "\n MODERN C++ BEST PRACTICES:" << std::endl;
527 std::cout << " ALWAYS use explicit for single-parameter constructors"
528 // << std::endl;
529 std::cout << " Use explicit for conversion operators (except bool)" <<
530 // std::endl;
531 std::cout << " Prevents accidental temporary object creation" << std::endl;
532 std::cout << " Prevents resource allocation surprises" << std::endl;
533 std::cout << " Makes code intent crystal clear" << std::endl;
534
535 std::cout << "\n WHEN IMPLICIT IS OK:" << std::endl;
536 std::cout << " • String literals to std::string: std::string s = \"hello
537 // \";" << std::endl;
538 std::cout << " • Initializer lists: std::vector<int> v = {1, 2, 3};" <<
539 // std::endl;
540 std::cout << " • Copy/move constructors (never explicit)" << std::endl;
541 }
542
543 // =====
544 // 7. POLICY-BASED DESIGN - PARAMETERIZED RESOURCE MANAGEMENT
545 // =====
546 // Modern C++ technique: Use template parameters to customize behavior
547 // without runtime overhead or virtual functions
548
549 // Policy 1: Deletion Strategies
```

```
543 struct ArrayDelete {
544     template<typename T>
545     void operator()(T* ptr) const {
546         delete[] ptr;
547         std::cout << "    ArrayDelete: delete[] called" << std::endl;
548     }
549 };
550
551 struct SingleDelete {
552     template<typename T>
553     void operator()(T* ptr) const {
554         delete ptr;
555         std::cout << "    SingleDelete: delete called" << std::endl;
556     }
557 };
558
559 struct NoOpDelete {
560     template<typename T>
561     void operator()(T*) const {
562         std::cout << "    NoOpDelete: no deletion (externally managed)" << std
563             ::endl;
564     }
565 };
566
567 // Policy 2: Copy Strategies
568 struct DeepCopy {
569     template<typename T>
570     static T* copy(const T* src, size_t size) {
571         std::cout << "    DeepCopy: allocating and copying " << size << "
572             elements" << std::endl;
573         T* dest = new T[size];
574         std::copy(src, src + size, dest);
575         return dest;
576     }
577 };
578
579 struct ShallowCopy {
580     template<typename T>
581     static T* copy(const T* src, size_t) {
582         std::cout << "    ShallowCopy: returning same pointer (reference
583             counting)" << std::endl;
584         return const_cast<T*>(src); // Warning: For demo only!
585     }
586 };
587
588 // Policy 3: Thread Safety
589 struct NoThreadSafety {
590     void lock() const { /* no-op */ }
591     void unlock() const { /* no-op */ }
592 };
593
594 struct BasicThreadSafety {
595     void lock() const {
596         std::cout << "    BasicThreadSafety: acquiring lock" << std::endl;
597     }
598 }
```

```
594     }
595     void unlock() const {
596         std::cout << "    BasicThreadSafety: releasing lock" << std::endl;
597     }
598 };
599
600 // Policy-based Resource Manager
601 template<
602     typename T,
603     typename DeletePolicy = ArrayDelete,
604     typename CopyPolicy = DeepCopy,
605     typename ThreadPolicy = NoThreadSafety
606 >
607 class PolicyBasedBuffer {
608 private:
609     T* data_;
610     size_t size_;
611     DeletePolicy deleter_;
612     ThreadPolicy thread_policy_;
613
614 public:
615     // Constructor
616     PolicyBasedBuffer(size_t s) : data_(new T[s]), size_(s) {
617         std::cout << "    PolicyBasedBuffer: Constructor (size=" << size_ << ")"
618             << std::endl;
619     }
620
621     // Destructor
622     ~PolicyBasedBuffer() {
623         std::cout << "    PolicyBasedBuffer: Destructor" << std::endl;
624         thread_policy_.lock();
625         deleter_(data_);
626         thread_policy_.unlock();
627     }
628
629     // Copy constructor (uses CopyPolicy)
630     PolicyBasedBuffer(const PolicyBasedBuffer& other)
631         : data_(CopyPolicy::copy(other.data_, other.size_)),
632         size_(other.size_) {
633         std::cout << "    PolicyBasedBuffer: Copy constructor" << std::endl;
634     }
635
636     // Copy assignment
637     PolicyBasedBuffer& operator=(const PolicyBasedBuffer& other) {
638         if (this != &other) {
639             std::cout << "    PolicyBasedBuffer: Copy assignment" << std::endl;
640             thread_policy_.lock();
641             deleter_(data_);
642             data_ = CopyPolicy::copy(other.data_, other.size_);
643             size_ = other.size_;
644             thread_policy_.unlock();
645         }
646         return *this;
647     }
648 }
```

```
647
648     // Move constructor
649     PolicyBasedBuffer(PolicyBasedBuffer&& other) noexcept
650         : data_(other.data_), size_(other.size_) {
651         other.data_ = nullptr;
652         other.size_ = 0;
653         std::cout << " PolicyBasedBuffer: Move constructor" << std::endl;
654     }
655
656     // Move assignment
657     PolicyBasedBuffer& operator=(PolicyBasedBuffer&& other) noexcept {
658         if (this != &other) {
659             std::cout << " PolicyBasedBuffer: Move assignment" << std::endl;
660             thread_policy_.lock();
661             deleter_(data_);
662             data_ = other.data_;
663             size_ = other.size_;
664             other.data_ = nullptr;
665             other.size_ = 0;
666             thread_policy_.unlock();
667         }
668         return *this;
669     }
670
671     size_t size() const { return size_; }
672     T* data() { return data_; }
673 };
674
675 // Type aliases for common configurations
676 template<typename T>
677 using ArrayBuffer = PolicyBasedBuffer<T, ArrayDelete, DeepCopy, NoThreadSafety>;
678
679 template<typename T>
680 using ThreadSafeBuffer = PolicyBasedBuffer<T, ArrayDelete, DeepCopy, BasicThreadSafety>;
681
682 template<typename T>
683 using SingleObjectBuffer = PolicyBasedBuffer<T, SingleDelete, DeepCopy, NoThreadSafety>;
684
685 void example_policy_based_design() {
686     std::cout << "\n== POLICY-BASED DESIGN ==" << std::endl;
687     std::cout << "Parameterize behavior with template policies\n" << std::endl;
688
689     std::cout << "1. Standard array buffer (ArrayDelete + DeepCopy):" << std::endl;
690
691     ArrayBuffer<int> buf1(10);
692     std::cout << " Copying..." << std::endl;
693     ArrayBuffer<int> buf2 = buf1;
694     std::cout << " Moving..." << std::endl;
695     ArrayBuffer<int> buf3 = std::move(buf1);
```

```
696     std::cout << "    Cleanup:" << std::endl;
697 }
698
699 std::cout << "\n2. Thread-safe buffer (with BasicThreadSafety):" << std::endl;
700 {
701     ThreadSafeBuffer<int> buf(5);
702     std::cout << "    Cleanup with thread safety:" << std::endl;
703 }
704
705 std::cout << "\n3. Custom policy combination (NoOpDelete for externally
706     managed):" << std::endl;
707 {
708     PolicyBasedBuffer<int, NoOpDelete, DeepCopy, NoThreadSafety> buf(3);
709     std::cout << "    Cleanup (no actual deletion):" << std::endl;
710 }
711
712 std::cout << "\n ADVANTAGES OF POLICY-BASED DESIGN:" << std::endl;
713 std::cout << "    Zero runtime overhead (compile-time selection)" << std::endl;
714 std::cout << "    No virtual functions or vtables needed" << std::endl;
715 std::cout << "    Highly composable and reusable" << std::endl;
716 std::cout << "    Type-safe customization" << std::endl;
717 std::cout << "    Optimized for each configuration" << std::endl;
718
719 std::cout << "\n REAL-WORLD USES:" << std::endl;
720 std::cout << " • std::unique_ptr<T, Deleter> - Custom deleters" << std::endl;
721 std::cout << " • STL containers with Allocator - Custom allocators" <<
722     std::endl;
723 std::cout << " • Thread-safety policies in concurrent code" << std::endl;
724 std::cout << " • Logging/tracing policies" << std::endl;
725 std::cout << " • Different storage strategies (stack vs heap)" << std::endl;
726
727 std::cout << "\n MODERN C++ ALTERNATIVE:" << std::endl;
728 std::cout << "    Use std::unique_ptr<T, Deleter> for most cases:" << std::endl;
729 std::cout << "    auto ptr = std::unique_ptr<int[], ArrayDelete>(new int
730     [10]);" << std::endl;
731 }
732
733 // =====
734 // 8. COPY-AND-SWAP IDIOM
735 // =====
736 // An elegant way to implement assignment operators
737
738 class CopyAndSwapExample {
739 private:
740     int* data;
741     size_t size;
742     std::string name;
743
744 public:
```

```
742     CopyAndSwapExample(const std::string& n, size_t s)
743         : data(new int[s]), size(s), name(n) {
744             for (size_t i = 0; i < size; ++i) {
745                 data[i] = static_cast<int>(i);
746             }
747         }
748
749     ~CopyAndSwapExample() {
750         delete[] data;
751     }
752
753     // Copy constructor
754     CopyAndSwapExample(const CopyAndSwapExample& other)
755         : data(new int[other.size]), size(other.size), name(other.name) {
756             for (size_t i = 0; i < size; ++i) {
757                 data[i] = other.data[i];
758             }
759         }
760
761     // Move constructor
762     CopyAndSwapExample(CopyAndSwapExample&& other) noexcept
763         : data(other.data), size(other.size), name(std::move(other.name)) {
764             other.data = nullptr;
765             other.size = 0;
766         }
767
768     // Swap function
769     friend void swap(CopyAndSwapExample& first, CopyAndSwapExample& second)
770         noexcept {
771         using std::swap;
772         swap(first.data, second.data);
773         swap(first.size, second.size);
774         swap(first.name, second.name);
775     }
776
777     // Unified assignment operator (handles both copy and move)
778     CopyAndSwapExample& operator=(CopyAndSwapExample other) {
779         swap(*this, other);
780         return *this;
781     }
782
783     void display() const {
784         if (data) {
785             std::cout << " " << name << " [size=" << size << "]" << std::endl
786             ;
787         } else {
788             std::cout << " " << name << " (empty)" << std::endl;
789         }
790     }
791 };
792
793 void example_copy_and_swap() {
794     std::cout << "\n==== COPY-AND-SWAP IDIOM ===" << std::endl;
795     std::cout << "Elegant and exception-safe assignment operator\n" << std::endl;
```

```
        endl;

794
795     CopyAndSwapExample obj1("Object1", 10);
796     CopyAndSwapExample obj2("Object2", 5);
797
798     obj1.display();
799     obj2.display();
800
801     std::cout << "\nAssigning obj1 = obj2 (copy):" << std::endl;
802     obj1 = obj2;
803     obj1.display();
804
805     std::cout << "\nAssigning obj1 = std::move(obj2) (move):" << std::endl;
806     obj1 = std::move(obj2);
807     obj1.display();
808     obj2.display();
809
810     std::cout << "\n Single assignment operator handles both copy and move!" << std::endl;
811 }

812 // =====
813 // MAIN FUNCTION
814 // =====
815
816
817 int main() {
818     std::cout << "\n"
819             =====" <<
820             std::endl;
821     std::cout << "  C++ SPECIAL MEMBER FUNCTIONS: RULE OF 3, 5, AND 0" << std
822             ::endl;
823     std::cout << "
824             =====" <<
825             std::endl;
826
827     example_rule_of_zero();
828     example_rule_of_three();
829     example_rule_of_five();
830     example_modern_cpp_relevance();
831     example_comparison();
832     example_default_and_delete();
833     example_explicit_keyword();
834     example_policy_based_design();
835     example_copy_and_swap();
836
837     std::cout << "\n"
838             =====" <<
839             std::endl;
840     std::cout << "  SUMMARY" << std::endl;
841     std::cout << " == RULE OF ZERO: Use in 95%+ of modern C++17/20 code!" <<
842             std::endl;
843     std::cout << "  - std::unique_ptr, std::optional, std::variant, std::
844             vector, etc." << std::endl;
845     std::cout << "  - Zero manual memory management = zero bugs" << std::endl;
```

```
837     ;
838     std::cout << "\n RULE OF THREE: Legacy guideline (pre-C++11)" << std::
839         endl;
840     std::cout << "    - Maintain in old codebases only" << std::endl;
841     std::cout << "    - Refactor to Rule of Zero when possible" << std::endl;
842
843     std::cout << "\n RULE OF FIVE: Rarely needed (<5% of cases)" << std::
844         endl;
845     std::cout << "    - Only for custom resource managers" << std::endl;
846     std::cout << "    - Performance-critical scenarios" << std::endl;
847
848     std::cout << "\n USE = DEFAULT AND = DELETE FOR EXPLICIT INTENT" << std::
849         endl;
850     std::cout << " Copy-and-swap idiom when you need Rule of Five" << std::
851         endl;
852
853     std::cout << "\n
854     =====\n" <<
855         std::endl;
856
857     return 0;
858 }
```

## 61 Source Code: RuntimePolymorphism.cpp

File: src/RuntimePolymorphism.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <memory>
5 #include <chrono>
6
7 // =====
8 // VIRTUAL FUNCTIONS AND PURE VIRTUAL FUNCTIONS IN MODERN C++
9 // =====
10 // Demonstrates:
11 // 1. Virtual functions vs pure virtual functions
12 // 2. Abstract classes and interfaces
13 // 3. Runtime polymorphism
14 // 4. Virtual destructors (critical!)
15 // 5. override and final keywords (C++11)
16 // 6. When to use vs alternatives (CRTP, concepts)
17 // 7. Performance considerations
18 // 8. Modern C++ best practices
19 // =====
20
21 // =====
22 // 1. PURE VIRTUAL FUNCTIONS - ABSTRACT INTERFACE
23 // =====
24 // Pure virtual function: = 0
25 // Cannot instantiate class with pure virtual functions
26 // Derived classes MUST implement all pure virtual functions
27
28 class IShape {
29 public:
30     // Pure virtual function - no implementation
31     virtual double area() const = 0;
32     virtual double perimeter() const = 0;
33     virtual void draw() const = 0;
34     virtual std::string name() const = 0;
35
36     // Virtual destructor is CRITICAL for polymorphic classes
37     virtual ~IShape() = default;
38
39     // Can have non-virtual functions too
40     void info() const {
41         std::cout << name() << ": area=" << area()
42             << ", perimeter=" << perimeter() << std::endl;
43     }
44 };
45
46 class Circle : public IShape {
47 private:
48     double radius;
```

```
50 | public:
51 |     explicit Circle(double r) : radius(r) {}
52 |
53 |     // Must implement ALL pure virtual functions
54 |     double area() const override {
55 |         return 3.14159 * radius * radius;
56 |     }
57 |
58 |     double perimeter() const override {
59 |         return 2 * 3.14159 * radius;
60 |     }
61 |
62 |     void draw() const override {
63 |         std::cout << "Drawing Circle (radius=" << radius << ")" << std::endl;
64 |     }
65 |
66 |     std::string name() const override {
67 |         return "Circle";
68 |     }
69 | };
70 |
71 | class Rectangle : public IShape {
72 | private:
73 |     double width, height;
74 |
75 | public:
76 |     Rectangle(double w, double h) : width(w), height(h) {}
77 |
78 |     double area() const override {
79 |         return width * height;
80 |     }
81 |
82 |     double perimeter() const override {
83 |         return 2 * (width + height);
84 |     }
85 |
86 |     void draw() const override {
87 |         std::cout << "Drawing Rectangle (" << width << "x" << height << ")" <<
88 |             std::endl;
89 |     }
90 |
91 |     std::string name() const override {
92 |         return "Rectangle";
93 |     }
94 | };
95 |
96 | void example_pure_virtual() {
97 |     std::cout << "\n==== 1. PURE VIRTUAL FUNCTIONS (ABSTRACT INTERFACE) ===" <<
98 |             std::endl;
99 |
100 |     // IShape shape; // ERROR: Cannot instantiate abstract class
101 |
102 |     std::vector<std::unique_ptr<IShape>> shapes;
103 |     shapes.push_back(std::make_unique<Circle>(5.0));
```

```
102     shapes.push_back(std::make_unique<Rectangle>(4.0, 6.0));
103     shapes.push_back(std::make_unique<Circle>(3.0));
104
105     std::cout << "\nProcessing shapes polymorphically:" << std::endl;
106     for (const auto& shape : shapes) {
107         shape->draw();
108         shape->info();
109         std::cout << std::endl;
110     }
111
112     std::cout << "  PURE VIRTUAL (= 0):" << std::endl;
113     std::cout << "  •  Forces derived classes to implement" << std::endl;
114     std::cout << "  •  Creates abstract class (cannot instantiate)" << std::endl;
115     std::cout << "  •  Defines interface contract" << std::endl;
116     std::cout << "  •  Use for: Interfaces, abstract base classes" << std::endl;
117 }
118
119 // =====
120 // 2. VIRTUAL FUNCTIONS WITH DEFAULT IMPLEMENTATION
121 // =====
122 // Virtual function with implementation - can be overridden but not required
123
124 class Animal {
125 public:
126     virtual ~Animal() = default;
127
128     // Virtual with default implementation
129     virtual void makeSound() const {
130         std::cout << "Some generic animal sound" << std::endl;
131     }
132
133     // Virtual with default implementation
134     virtual void eat() const {
135         std::cout << "Animal is eating" << std::endl;
136     }
137
138     // Non-virtual - cannot override
139     void breathe() const {
140         std::cout << "Animal is breathing" << std::endl;
141     }
142 };
143
144 class Dog : public Animal {
145 public:
146     // Override with custom implementation
147     void makeSound() const override {
148         std::cout << "Woof! Woof!" << std::endl;
149     }
150
151     // Override eat
152     void eat() const override {
153         std::cout << "Dog is eating dog food" << std::endl;
```

```
154     }
155
156     // breathe() cannot be overridden (not virtual)
157 };
158
159 class Cat : public Animal {
160 public:
161     // Only override makeSound
162     void makeSound() const override {
163         std::cout << "Meow!" << std::endl;
164     }
165
166     // eat() uses default implementation from Animal
167 };
168
169 void example_virtual_with_default() {
170     std::cout << "\n==== 2. VIRTUAL FUNCTIONS WITH DEFAULT IMPLEMENTATION ==="
171             << std::endl;
172
173     std::vector<std::unique_ptr<Animal>> animals;
174     animals.push_back(std::make_unique<Dog>());
175     animals.push_back(std::make_unique<Cat>());
176
177     std::cout << "\nDog:" << std::endl;
178     animals[0]->makeSound(); // Custom implementation
179     animals[0]->eat(); // Custom implementation
180     animals[0]->breathe(); // Non-virtual (same for all)
181
182     std::cout << "\nCat:" << std::endl;
183     animals[1]->makeSound(); // Custom implementation
184     animals[1]->eat(); // Uses default from Animal
185     animals[1]->breathe(); // Non-virtual (same for all)
186
187     std::cout << "\n VIRTUAL WITH DEFAULT:" << std::endl;
188     std::cout << " • Provides base implementation" << std::endl;
189     std::cout << " • Derived classes can override if needed" << std::endl;
190     std::cout << " • Optional customization" << std::endl;
191     std::cout << " • Use for: Template Method pattern" << std::endl;
192 }
193 // =====
194 // 3. VIRTUAL DESTRUCTOR - CRITICAL FOR POLYMORPHISM!
195 // =====
196
197 class Base {
198 public:
199     Base() {
200         std::cout << "Base constructor" << std::endl;
201     }
202
203     // BAD: Non-virtual destructor in polymorphic class!
204     ~Base() {
205         std::cout << "Base destructor" << std::endl;
206     }
```

```
207 };
```

```
208
```

```
209 class DerivedBad : public Base {
```

```
210 private:
```

```
211     int* data;
```

```
212
```

```
213 public:
```

```
214     DerivedBad() {
pre>215         data = new int[100];
pre>216         std::cout << "DerivedBad constructor (allocated 100 ints)" << std::endl;
pre>217     }
```

```
218
```

```
219     ~DerivedBad() {
pre>220         delete[] data;
pre>221         std::cout << "DerivedBad destructor (freed memory)" << std::endl;
pre>222     }
```

```
223 };
```

```
224
```

```
225 class BaseGood {
pre>226 public:
pre>227     BaseGood() {
pre>228         std::cout << "BaseGood constructor" << std::endl;
pre>229     }
```

```
230
```

```
231 // GOOD: Virtual destructor!
pre>232     virtual ~BaseGood() {
pre>233         std::cout << "BaseGood destructor" << std::endl;
pre>234     }
```

```
235 };
```

```
236
```

```
237 class DerivedGood : public BaseGood {
```

```
238 private:
pre>239     int* data;
```

```
240
```

```
241 public:
pre>242     DerivedGood() {
pre>243         data = new int[100];
pre>244         std::cout << "DerivedGood constructor (allocated 100 ints)" << std::endl;
pre>245     }
```

```
246
```

```
247     ~DerivedGood() override {
pre>248         delete[] data;
pre>249         std::cout << "DerivedGood destructor (freed memory)" << std::endl;
pre>250     }
```

```
251 };
```

```
252
```

```
253 void example_virtual_destructor() {
pre>254     std::cout << "\n== 3. VIRTUAL DESTRUCTOR (CRITICAL!) ==" << std::endl;
pre>255
pre>256     std::cout << "\n BAD: Non-virtual destructor:" << std::endl;
pre>257     {
pre>258         Base* ptr = new DerivedBad();
```

```
259     delete ptr; // MEMORY LEAK! Only Base destructor called
260 }
261
262 std::cout << "\n GOOD: Virtual destructor:" << std::endl;
263 {
264     BaseGood* ptr = new DerivedGood();
265     delete ptr; // Both destructors called correctly
266 }
267
268 std::cout << "\n RULE: Always make destructor virtual if class has
269     virtual functions!" << std::endl;
270 std::cout << " • Otherwise: Memory leaks, undefined behavior" << std::endl;
271 std::cout << " • Use: virtual ~ClassName() = default;" << std::endl;
272 }
273 // =====
274 // 4. OVERRIDE AND FINAL KEYWORDS (C++11)
275 // =====
276
277 class BaseKeywords {
278 public:
279     virtual void foo() const {
280         std::cout << "BaseKeywords::foo()" << std::endl;
281     }
282
283     virtual void bar() const {
284         std::cout << "BaseKeywords::bar()" << std::endl;
285     }
286
287     virtual ~BaseKeywords() = default;
288 };
289
290 class DerivedKeywords : public BaseKeywords {
291 public:
292     // override: Ensures we're actually overriding a base function
293     void foo() const override {
294         std::cout << "DerivedKeywords::foo()" << std::endl;
295     }
296
297     // This would cause compile error with override:
298     // void fooo() const override {} // Typo! No such function in base
299
300     // final: Cannot be overridden further
301     void bar() const final {
302         std::cout << "DerivedKeywords::bar() - cannot override further" << std::endl;
303     }
304 };
305
306 // final: Cannot be inherited from
307 class FinalClass final {
308 public:
309     void method() {
```

```
310         std::cout << "FinalClass::method()" << std::endl;
311     }
312 };
313
314 // This would cause compile error:
315 // class CannotDeriveFromFinal : public FinalClass { };
316
317 void example_override_final() {
318     std::cout << "\n==== 4. OVERRIDE AND FINAL KEYWORDS (C++11) ===" << std::endl;
319
320     std::unique_ptr<BaseKeywords> ptr = std::make_unique<DerivedKeywords>();
321     ptr->foo();
322     ptr->bar();
323
324     std::cout << "\n OVERRIDE KEYWORD:" << std::endl;
325     std::cout << " • Catches typos at compile time" << std::endl;
326     std::cout << " • Documents intent clearly" << std::endl;
327     std::cout << " • Prevents accidental shadowing" << std::endl;
328     std::cout << " • ALWAYS use it when overriding!" << std::endl;
329
330     std::cout << "\n FINAL KEYWORD:" << std::endl;
331     std::cout << " • Prevents further overriding (function)" << std::endl;
332     std::cout << " • Prevents inheritance (class)" << std::endl;
333     std::cout << " • Enables compiler optimizations" << std::endl;
334     std::cout << " • Use for: Performance, design intent" << std::endl;
335 }
336
337 // =====
338 // 5. WHEN TO USE VIRTUAL FUNCTIONS IN MODERN C++
339 // =====
340
341 void example_when_to_use() {
342     std::cout << "\n==== 5. WHEN TO USE VIRTUAL FUNCTIONS ===" << std::endl;
343
344     std::cout << "\n USE VIRTUAL FUNCTIONS WHEN:" << std::endl;
345     std::cout << " 1. Runtime polymorphism needed" << std::endl;
346     std::cout << "    - Heterogeneous containers (vector<Base*>)" << std::endl;
347     std::cout << "    - Plugin architectures" << std::endl;
348     std::cout << "    - Dynamic type selection" << std::endl;
349
350     std::cout << "\n 2. Interface definition" << std::endl;
351     std::cout << "    - Abstract base classes" << std::endl;
352     std::cout << "    - Pure virtual functions (= 0)" << std::endl;
353     std::cout << "    - Contract for derived classes" << std::endl;
354
355     std::cout << "\n 3. Framework design" << std::endl;
356     std::cout << "    - Template Method pattern" << std::endl;
357     std::cout << "    - Strategy pattern" << std::endl;
358     std::cout << "    - Factory pattern" << std::endl;
359
360     std::cout << "\n 4. Open/Closed principle" << std::endl;
361     std::cout << "    - Extend behavior without modifying base" << std::endl;
```

```
362     ;
363     std::cout << "      - Add new types without changing existing code" << std
364             ::endl;
365 }
366 // =====
367 // 6. ALTERNATIVES TO VIRTUAL FUNCTIONS (MODERN C++)
368 // =====
369 // CRTP (Curiously Recurring Template Pattern) - Static Polymorphism
370 template<typename Derived>
371 class SensorBase {
372 public:
373     float read() {
374         // Static cast to derived - no runtime overhead!
375         return static_cast<Derived*>(this)->read_impl();
376     }
377 };
378
379 class TemperatureSensor : public SensorBase<TemperatureSensor> {
380 public:
381     float read_impl() {
382         return 25.5f; // Temperature reading
383     }
384 };
385
386 class PressureSensor : public SensorBase<PressureSensor> {
387 public:
388     float read_impl() {
389         return 1013.25f; // Pressure reading
390     }
391 };
392
393 template<typename Sensor>
394 void process_sensor(Sensor& sensor) {
395     // Compile-time polymorphism - fully inlineable!
396     std::cout << "Sensor reading: " << sensor.read() << std::endl;
397 }
398
399 void example_alternatives() {
400     std::cout << "\n==== 6. ALTERNATIVES TO VIRTUAL FUNCTIONS ===" << std::endl
401             ;
402
403     std::cout << "\n1. CRTP (STATIC POLYMORPHISM):" << std::endl;
404     TemperatureSensor temp;
405     PressureSensor pressure;
406
407     process_sensor(temp);
408     process_sensor(pressure);
409
410     std::cout << "\n      Advantages:" << std::endl;
411     std::cout << " •      Zero runtime overhead (no vtable)" << std::endl;
412     std::cout << " •      Fully inlineable" << std::endl;
413     std::cout << " •      Faster than virtual functions" << std::endl;
```

```

413     std::cout << " • Good for embedded systems" << std::endl;
414
415     std::cout << "\n • Disadvantages:" << std::endl;
416     std::cout << " • No heterogeneous containers" << std::endl;
417     std::cout << " • Compile-time binding only" << std::endl;
418     std::cout << " • More complex syntax" << std::endl;
419
420     std::cout << "\n2. CONCEPTS (C++20):" << std::endl;
421     std::cout << " • Compile-time constraints" << std::endl;
422     std::cout << " • Duck typing with type safety" << std::endl;
423     std::cout << " • No inheritance needed" << std::endl;
424     std::cout << " • Example: template<Drawable T> void draw(T& obj)" << std
        ::endl;
425
426     std::cout << "\n3. std::variant (C++17):" << std::endl;
427     std::cout << " • Type-safe union" << std::endl;
428     std::cout << " • std::visit for polymorphic behavior" << std::endl;
429     std::cout << " • Value semantics (no pointers)" << std::endl;
430     std::cout << " • Good for small, closed set of types" << std::endl;
431
432     std::cout << "\n4. std::function (C++11):" << std::endl;
433     std::cout << " • Type-erased callable" << std::endl;
434     std::cout << " • Works with lambdas, functors" << std::endl;
435     std::cout << " • Runtime overhead (like virtual)" << std::endl;
436     std::cout << " • Good for callbacks, strategies" << std::endl;
437 }
438
439 // =====
440 // 7. PERFORMANCE COMPARISON
441 // =====
442
443 class VirtualBase {
444 public:
445     virtual int compute(int x) { return x * 2; }
446     virtual ~VirtualBase() = default;
447 };
448
449 class VirtualDerived : public VirtualBase {
450 public:
451     int compute(int x) override { return x * 3; }
452 };
453
454 void example_performance() {
455     std::cout << "\n== 7. PERFORMANCE CONSIDERATIONS ==" << std::endl;
456
457     constexpr int iterations = 10'000'000;
458
459     // Virtual function call
460     VirtualBase* vptr = new VirtualDerived();
461     auto start_virtual = std::chrono::high_resolution_clock::now();
462     volatile int result_virtual = 0;
463     for (int i = 0; i < iterations; ++i) {
464         result_virtual += vptr->compute(i);
465     }
}

```

```

466     auto end_virtual = std::chrono::high_resolution_clock::now();
467     auto duration_virtual = std::chrono::duration_cast<std::chrono::
468         milliseconds>(
469             end_virtual - start_virtual).count();
470
471     // Direct function call (non-virtual)
472     VirtualDerived direct;
473     auto start_direct = std::chrono::high_resolution_clock::now();
474     volatile int result_direct = 0;
475     for (int i = 0; i < iterations; ++i) {
476         result_direct += direct.compute(i);
477     }
478     auto end_direct = std::chrono::high_resolution_clock::now();
479     auto duration_direct = std::chrono::duration_cast<std::chrono::
480         milliseconds>(
481             end_direct - start_direct).count();
482
483     delete vptr;
484
485     std::cout << "\nPerformance (10 million calls):" << std::endl;
486     std::cout << "    Virtual function: " << duration_virtual << " ms" << std
487         ::endl;
488     std::cout << "    Direct call:         " << duration_direct << " ms" << std::
489         endl;
490     std::cout << "    Overhead:           ~"
491         << (duration_virtual - duration_direct) << " ms" << std::endl;
492
493     std::cout << "\n OVERHEAD SOURCES:" << std::endl;
494     std::cout << "    1. Vtable lookup (8 bytes per object)" << std::endl;
495     std::cout << "    2. Indirect function call (cache miss)" << std::endl;
496     std::cout << "    3. Cannot be inlined by compiler" << std::endl;
497     std::cout << "    4. Branch prediction harder" << std::endl;
498
499     std::cout << "\n TRADE-OFF:" << std::endl;
500     std::cout << " • Virtual functions: Small overhead, big flexibility" <<
501         std::endl;
502     std::cout << " • Usually worth it for abstraction benefits" << std::endl
503         ;
504     std::cout << " • Only optimize if profiling shows bottleneck" << std::
505         endl;
506 }
507
508 // =====
509 // 8. MODERN C++ BEST PRACTICES
510 // =====
511
512 void example_best_practices() {
513     std::cout << "\n== 8. MODERN C++ BEST PRACTICES ==" << std::endl;
514
515     std::cout << "\n DO:"
516     std::cout << "    1. ALWAYS use 'override' keyword" << std::endl;
517     std::cout << "        void foo() override { }" << std::endl;
518     std::cout << "        void foo() { } // Easy to make mistakes" << std::
519         endl;

```

```
512
513     std::cout << "\n    2. ALWAYS make destructor virtual in polymorphic
514         classes" << std::endl;
515     std::cout << "            virtual ~Base() = default;" << std::endl;
516     std::cout << "            ~Base() { } // Memory leaks!" << std::endl;
517
518     std::cout << "\n    3. Use 'final' to prevent further overriding" << std::
519         endl;
520     std::cout << "            void foo() final { } // Design intent +
521         optimization" << std::endl;
522
523     std::cout << "\n    4. Use pure virtual (= 0) for interfaces" << std::endl;
524     std::cout << "            virtual void draw() const = 0; // Must implement"
525         << std::endl;
526
527     std::cout << "\n    5. Prefer smart pointers for polymorphic objects" <<
528         std::endl;
529     std::cout << "            std::unique_ptr<Base> ptr = std::make_unique<Derived
530         >();" << std::endl;
531     std::cout << "            Base* ptr = new Derived(); // Manual delete needed"
532         << std::endl;
533
534     std::cout << "\n    1. Don't use virtual for non-polymorphic classes" << std
535         ::endl;
536     std::cout << "            - Unnecessary overhead" << std::endl;
537
538     std::cout << "\n    2. Don't forget virtual destructor" << std::endl;
539     std::cout << "            - Causes memory leaks and undefined behavior" << std::
540         endl;
541
542     std::cout << "\n    3. Don't call virtual functions in constructor/
543         destructor" << std::endl;
544     std::cout << "            - Derived class not fully constructed yet" << std::
545         endl;
546     std::cout << "            - Will call base version, not derived!" << std::endl;
547
548     std::cout << "\n    4. Don't use virtual if CRTP/concepts work" << std::
549         endl;
550     std::cout << "            - Static polymorphism is faster" << std::endl;
551     std::cout << "            - Good for performance-critical code" << std::endl;
552
553
554 // =====
555 // 9. ARE VIRTUAL FUNCTIONS STILL RELEVANT?
556 // =====
557
558 void example_still_relevant() {
559     std::cout << "\n==== 9. ARE VIRTUAL FUNCTIONS STILL RELEVANT? ===" << std::
560         endl;
561
562     std::cout << "\n YES! Virtual functions are STILL HIGHLY RELEVANT:" <<
563         std::endl;
564
565 }
```

```
552     std::cout << "\n1. RUNTIME POLYMORPHISM:" << std::endl;
553     std::cout << " • Containers of heterogeneous objects" << std::endl;
554     std::cout << " • Plugin systems and dynamic loading" << std::endl;
555     std::cout << " • GUI frameworks (Qt, wxWidgets)" << std::endl;
556     std::cout << " • Game engines (entity systems)" << std::endl;
557
558     std::cout << "\n2. DESIGN PATTERNS:" << std::endl;
559     std::cout << " • Strategy, Observer, Command patterns" << std::endl;
560     std::cout << " • Factory method, Template method" << std::endl;
561     std::cout << " • Visitor pattern (double dispatch)" << std::endl;
562
563     std::cout << "\n3. API DESIGN:" << std::endl;
564     std::cout << " • Stable binary interfaces (ABIs)" << std::endl;
565     std::cout << " • Dynamic libraries (.dll, .so)" << std::endl;
566     std::cout << " • Cross-module boundaries" << std::endl;
567
568     std::cout << "\n4. SIMPLICITY:" << std::endl;
569     std::cout << " • Easier to understand than CRTP" << std::endl;
570     std::cout << " • More maintainable for most teams" << std::endl;
571     std::cout << " • Better error messages" << std::endl;
572
573     std::cout << "\n MODERN ALTERNATIVES (C++11-20):" << std::endl;
574     std::cout << " • CRTP - Compile-time polymorphism" << std::endl;
575     std::cout << " • Concepts (C++20) - Duck typing with constraints" << std
      ::endl;
576     std::cout << " • std::variant + std::visit (C++17)" << std::endl;
577     std::cout << " • std::function - Type-erased callables" << std::endl;
578
579     std::cout << "\n WHEN TO CHOOSE:" << std::endl;
580     std::cout << " • Virtual Functions:" << std::endl;
581     std::cout << " • Need runtime polymorphism" << std::endl;
582     std::cout << " • Heterogeneous containers" << std::endl;
583     std::cout << " • Plugin architectures" << std::endl;
584     std::cout << " • Simplicity and maintainability" << std::endl;
585
586     std::cout << "\n CRTP/Concepts:" << std::endl;
587     std::cout << " • Performance critical code" << std::endl;
588     std::cout << " • Compile-time polymorphism sufficient" << std::endl;
589     std::cout << " • Embedded systems (no vtable)" << std::endl;
590     std::cout << " • Header-only libraries" << std::endl;
591
592     std::cout << "\n BOTTOM LINE:" << std::endl;
593     std::cout << " • Virtual functions: Core C++ feature, still essential"
      << std::endl;
594     std::cout << " • Modern C++ adds alternatives, not replacements" << std
      ::endl;
595     std::cout << " • Choose based on requirements, not trends" << std::endl;
596     std::cout << " • Most codebases use BOTH virtual and static polymorphism
      " << std::endl;
597 }
598
599 // =====
600 // MAIN
601 // =====
```

```
602
603 int main() {
604     std::cout << "\n"
605     =====
606     std::endl;
607     std::cout << "    VIRTUAL FUNCTIONS AND PURE VIRTUAL FUNCTIONS IN MODERN C++"
608     " << std::endl;
609     std::cout << "
610     =====
611     std::endl;
612
613     example_pure_virtual();
614     example_virtual_with_default();
615     example_virtual_destructor();
616     example_override_final();
617     example_when_to_use();
618     example_alternatives();
619     example_performance();
620     example_best_practices();
621     example_still_relevant();
622
623     std::cout << "\n"
624     =====
625     std::endl;
626     std::cout << "    SUMMARY: VIRTUAL FUNCTIONS IN MODERN C++" << std::endl;
627     std::cout << "
628     =====
629     std::endl;
630     std::cout << "    1. Pure virtual (= 0) - Abstract interface, must
631     implement" << std::endl;
632     std::cout << "    2. Virtual - Can override, optional" << std::endl;
633     std::cout << "    3. Virtual destructor - ALWAYS for polymorphic classes"
634     << std::endl;
635     std::cout << "    4. override keyword - ALWAYS use when overriding" << std
636     ::endl;
637     std::cout << "    5. final keyword - Prevent further overriding" << std::
638     endl;
639
640     std::cout << "\n    KEY CONCEPTS:" << std::endl;
641     std::cout << "    1. Pure virtual (= 0) - Abstract interface, must
642     implement" << std::endl;
643     std::cout << "    2. Virtual - Can override, optional" << std::endl;
644     std::cout << "    3. Virtual destructor - ALWAYS for polymorphic classes"
645     << std::endl;
646     std::cout << "    4. override keyword - ALWAYS use when overriding" << std
647     ::endl;
648     std::cout << "    5. final keyword - Prevent further overriding" << std::
649     endl;
650
651     std::cout << "\n    BEST PRACTICES:" << std::endl;
652     std::cout << "    • Use 'override' on all virtual function overrides" <<
653     std::endl;
654     std::cout << "    • Always virtual destructor for polymorphic classes" <<
655     std::endl;
656     std::cout << "    • Prefer std::unique_ptr/shared_ptr over raw pointers" <<
657     std::endl;
658     std::cout << "    • Consider CRTP/concepts for performance-critical code"
659     << std::endl;
660     std::cout << "    • Don't call virtual functions in constructors/
661     destructors" << std::endl;
662
663     std::cout << "\n    STILL RELEVANT IN MODERN C++:" << std::endl;
664     std::cout << "    • YES! Essential for runtime polymorphism" << std::endl;
```

```
638     std::cout << " • Plugin systems, frameworks, design patterns" << std::endl;
639     std::cout << " • Modern C++ adds alternatives, not replacements" << std::endl;
640     std::cout << " • Choose based on requirements, not trends" << std::endl;
641
642     std::cout << "\n"
643     ======\n" << std::endl;
644
645     return 0;
646 }
```

## 62 Source Code: SOLIDPrinciples.cpp

File: src/SOLIDPrinciples.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // SOLID PRINCIPLES IN C++
3 // =====
4 // A comprehensive demonstration of the five SOLID design principles
5 // for object-oriented programming in Modern C++
6 //
7 // S - Single Responsibility Principle
8 // O - Open-Closed Principle
9 // L - Liskov Substitution Principle
10 // I - Interface Segregation Principle
11 // D - Dependency Inversion Principle
12 //
13 // Build: g++ -std=c++20 -Wall -Wextra -O2 -o SOLIDPrinciples SOLIDPrinciples.
14 // .cpp
15 // =====
16 #include <iostream>
17 #include <string>
18 #include <vector>
19 #include <memory>
20 #include <fstream>
21 #include <sstream>
22 #include <cmath>
23 #include <stdexcept>
24 //
25 // =====
26 // PRINCIPLE 1: SINGLE RESPONSIBILITY PRINCIPLE (SRP)
27 // =====
28 // A class should have only ONE reason to change
29 // Each class should do ONE thing and do it well
30
31 namespace single_responsibility {
32
33 // BAD: Class with multiple responsibilities
34 class BadEmployee {
35 private:
36     std::string name_;
37     double salary_;
38
39 public:
40     BadEmployee(const std::string& name, double salary)
41         : name_(name), salary_(salary) {}
42
43     // Responsibility 1: Employee data management
44     std::string getName() const { return name_; }
45     double getSalary() const { return salary_; }
46
47     // Responsibility 2: Salary calculation (WRONG!)
48     double calculateTax() const {
```

```
49     return salary_ * 0.25;
50 }
51
52 // Responsibility 3: Database operations (WRONG!)
53 void saveToDatabase() {
54     std::cout << "[BAD] Saving " << name_ << " to database...\n";
55     // Database logic here...
56 }
57
58 // Responsibility 4: Reporting (WRONG!)
59 void generatePayslip() {
60     std::cout << "[BAD] Generating payslip for " << name_ << "\n";
61     // Report generation logic...
62 }
63 };
64
65 // GOOD: Separate classes for separate responsibilities
66 class Employee {
67 private:
68     std::string name_;
69     double salary_;
70
71 public:
72     Employee(const std::string& name, double salary)
73         : name_(name), salary_(salary) {}
74
75     std::string getName() const { return name_; }
76     double getSalary() const { return salary_; }
77 };
78
79 // Responsibility 2: Tax calculation
80 class TaxCalculator {
81 public:
82     double calculateTax(const Employee& emp) const {
83         return emp.getSalary() * 0.25;
84     }
85 };
86
87 // Responsibility 3: Database operations
88 class EmployeeRepository {
89 public:
90     void save(const Employee& emp) {
91         std::cout << "[GOOD] Saving " << emp.getName() << " to database...\n";
92         // Database logic here...
93     }
94 };
95
96 // Responsibility 4: Reporting
97 class PayslipGenerator {
98 public:
99     void generate(const Employee& emp) {
100        std::cout << "[GOOD] Generating payslip for " << emp.getName() << "\n";
101    }
102 }
```

```
101     // Report generation logic...
102 }
103 };
104
105 void demonstrate() {
106     std::cout << "\n" << std::string(70, '=') << "\n";
107     std::cout << "PRINCIPLE 1: SINGLE RESPONSIBILITY PRINCIPLE (SRP)\n";
108     std::cout << std::string(70, '=') << "\n\n";
109
110     std::cout << "  DEFINITION:\n";
111     std::cout << "    A class should have only ONE reason to change.\n";
112     std::cout << "    Each class should do ONE thing and do it well.\n\n";
113
114     std::cout << "  BAD Example (Multiple Responsibilities):\n";
115     BadEmployee badEmp("John Doe", 50000);
116     badEmp.saveToDatabase();
117     badEmp.generatePayslip();
118     std::cout << "    Problem: Employee class handles data, tax, DB, and
119     reports!\n\n";
120
121     std::cout << "  GOOD Example (Single Responsibility):\n";
122     Employee goodEmp("Jane Smith", 60000);
123     TaxCalculator taxCalc;
124     EmployeeRepository repo;
125     PayslipGenerator payslip;
126
127     repo.save(goodEmp);
128     payslip.generate(goodEmp);
129     std::cout << "    Tax: $" << taxCalc.calculateTax(goodEmp) << "\n\n";
130
131     std::cout << "  BENEFITS:\n";
132     std::cout << "  •  Easier to understand and maintain\n";
133     std::cout << "  •  Changes to tax logic don't affect database code\n";
134     std::cout << "  •  Classes are more reusable and testable\n";
135     std::cout << "  •  Better separation of concerns\n";
136 }
137 } // namespace single_responsibility
138
139 // =====
140 // PRINCIPLE 2: OPEN-CLOSED PRINCIPLE (OCP)
141 // =====
142 // Open for extension, closed for modification
143 // You should be able to add new functionality without changing existing code
144
145 namespace open_closed {
146
147 // BAD: Need to modify existing code for each new shape
148 class BadShapeCalculator {
149 public:
150     double calculateArea(const std::string& shapeType, double dimension) {
151         if (shapeType == "circle") {
152             return 3.14159 * dimension * dimension;
153         } else if (shapeType == "square") {
```

```
154         return dimension * dimension;
155     }
156     // Need to modify this function for each new shape! (BAD)
157     return 0.0;
158 }
159 };
160
161 // GOOD: Use polymorphism - open for extension, closed for modification
162 class Shape {
163 public:
164     virtual ~Shape() = default;
165     virtual double area() const = 0;
166     virtual std::string name() const = 0;
167 };
168
169 class Circle : public Shape {
170 private:
171     double radius_;
172
173 public:
174     explicit Circle(double radius) : radius_(radius) {}
175
176     double area() const override {
177         return 3.14159 * radius_ * radius_;
178     }
179
180     std::string name() const override { return "Circle"; }
181 };
182
183 class Square : public Shape {
184 private:
185     double side_;
186
187 public:
188     explicit Square(double side) : side_(side) {}
189
190     double area() const override {
191         return side_ * side_;
192     }
193
194     std::string name() const override { return "Square"; }
195 };
196
197 class Triangle : public Shape {
198 private:
199     double base_;
200     double height_;
201
202 public:
203     Triangle(double base, double height) : base_(base), height_(height) {}
204
205     double area() const override {
206         return 0.5 * base_ * height_;
207     }
}
```

```

208     std::string name() const override { return "Triangle"; }
209 };
210
211 // This class is CLOSED for modification, but OPEN for extension
212 class AreaCalculator {
213 public:
214     double totalArea(const std::vector<std::unique_ptr<Shape>>& shapes) const
215     {
216         double total = 0.0;
217         for (const auto& shape : shapes) {
218             total += shape->area();
219         }
220         return total;
221     }
222
223     void printAreas(const std::vector<std::unique_ptr<Shape>>& shapes) const {
224         for (const auto& shape : shapes) {
225             std::cout << "    " << shape->name() << " area: "
226                         << shape->area() << "\n";
227         }
228     }
229 };
230
231 void demonstrate() {
232     std::cout << "\n" << std::string(70, '=') << "\n";
233     std::cout << "PRINCIPLE 2: OPEN-CLOSED PRINCIPLE (OCP)\n";
234     std::cout << std::string(70, '=') << "\n\n";
235
236     std::cout << "  DEFINITION:\n";
237     std::cout << "    Software entities should be OPEN for extension,\n";
238     std::cout << "    but CLOSED for modification.\n\n";
239
240     std::cout << "  BAD Example:\n";
241     BadShapeCalculator badCalc;
242     std::cout << "    Circle area: " << badCalc.calculateArea("circle", 5.0) <<
243                         "\n";
244     std::cout << "    Problem: Must modify calculateArea() for each new shape!\n
245                         \n";
246
247     std::cout << "  GOOD Example (Using Polymorphism):\n";
248     std::vector<std::unique_ptr<Shape>> shapes;
249     shapes.push_back(std::make_unique<Circle>(5.0));
250     shapes.push_back(std::make_unique<Square>(4.0));
251     shapes.push_back(std::make_unique<Triangle>(3.0, 6.0));
252
253     AreaCalculator calc;
254     calc.printAreas(shapes);
255     std::cout << "    Total area: " << calc.totalArea(shapes) << "\n\n";
256
257     std::cout << "  BENEFITS:\n";
258     std::cout << "  •  Add new shapes without modifying AreaCalculator\n";
259     std::cout << "  •  Existing code remains stable and tested\n";
260     std::cout << "  •  Reduces risk of breaking existing functionality\n";

```

```
259     std::cout << " • Promotes use of interfaces and polymorphism\n";
260 }
261
262 } // namespace open_closed
263
264 // =====
265 // PRINCIPLE 3: LISKOV SUBSTITUTION PRINCIPLE (LSP)
266 // =====
267 // Objects of a superclass should be replaceable with objects of a subclass
268 // without breaking the application
269
270 namespace liskov_substitution {
271
272 // BAD: Violates LSP - Square changes Rectangle's behavior
273 class BadRectangle {
274 protected:
275     int width_;
276     int height_;
277
278 public:
279     virtual ~BadRectangle() = default; // Virtual destructor for polymorphic
280     class
281     virtual void setWidth(int w) { width_ = w; }
282     virtual void setHeight(int h) { height_ = h; }
283
284     int getWidth() const { return width_; }
285     int getHeight() const { return height_; }
286
287     virtual int area() const { return width_ * height_; }
288 };
289
290 class BadSquare : public BadRectangle {
291 public:
292     // Violates LSP: Changes the behavior!
293     void setWidth(int w) override {
294         width_ = w;
295         height_ = w; // Square forces equal sides
296     }
297
298     void setHeight(int h) override {
299         width_ = h;
300         height_ = h; // Square forces equal sides
301     }
302
303 // GOOD: Use composition instead of inheritance for Square
304 class Shape {
305 public:
306     virtual ~Shape() = default;
307     virtual int area() const = 0;
308     virtual std::string type() const = 0;
309 };
310
311 class Rectangle : public Shape {
```

```

312 | private:
313 |     int width_;
314 |     int height_;
315 |
316 | public:
317 |     Rectangle(int w, int h) : width_(w), height_(h) {}
318 |
319 |     void setWidth(int w) { width_ = w; }
320 |     void setHeight(int h) { height_ = h; }
321 |
322 |     int getWidth() const { return width_; }
323 |     int getHeight() const { return height_; }
324 |
325 |     int area() const override { return width_ * height_; }
326 |     std::string type() const override { return "Rectangle"; }
327 | };
328 |
329 | class Square : public Shape {
330 | private:
331 |     int side_;
332 |
333 | public:
334 |     explicit Square(int side) : side_(side) {}
335 |
336 |     void setSide(int s) { side_ = s; }
337 |     int getSide() const { return side_; }
338 |
339 |     int area() const override { return side_ * side_; }
340 |     std::string type() const override { return "Square"; }
341 | };
342 |
343 | void demonstrate() {
344 |     std::cout << "\n" << std::string(70, '=') << "\n";
345 |     std::cout << "PRINCIPLE 3: LISKOV SUBSTITUTION PRINCIPLE (LSP)\n";
346 |     std::cout << std::string(70, '=') << "\n\n";
347 |
348 |     std::cout << "  DEFINITION:\n";
349 |     std::cout << "    Objects of a superclass should be replaceable with
350 |                 objects\n";
351 |     std::cout << "    of a subclass without breaking the application.\n\n";
352 |
353 |     std::cout << "  BAD Example (LSP Violation):\n";
354 |     BadRectangle* rect = new BadSquare();
355 |     rect->setWidth(5);
356 |     rect->setHeight(4);
357 |     std::cout << "    Expected area: 20 (5x4)\n";
358 |     std::cout << "    Actual area: " << rect->area() << " (4x4)\n";
359 |     std::cout << "    Problem: Square changed Rectangle's behavior!\n";
360 |     delete rect;
361 |     std::cout << "\n";
362 |
363 |     std::cout << "  GOOD Example (LSP Compliant):\n";
364 |     Rectangle goodRect(5, 4);
365 |     std::cout << "    Rectangle (5x4) area: " << goodRect.area() << "\n";

```

```
365     Square goodSquare(4);
366     std::cout << "    Square (4x4) area: " << goodSquare.area() << "\n\n";
367
368     std::cout << "    BENEFITS:\n";
369     std::cout << "    •    Substitutability is preserved\n";
370     std::cout << "    •    No unexpected behavior when using derived classes\n";
371     std::cout << "    •    Polymorphism works correctly\n";
372     std::cout << "    •    \"IS-A\" relationship is properly maintained\n";
373 }
374
375 } // namespace liskov_substitution
376
377 // =====
378 // PRINCIPLE 4: INTERFACE SEGREGATION PRINCIPLE (ISP)
379 // =====
380 // Clients should not be forced to depend on interfaces they don't use
381 // Many specific interfaces are better than one general-purpose interface
382
383 namespace interface_segregation {
384
385 // BAD: Fat interface forces implementations to provide unused methods
386 class BadWorker {
387 public:
388     virtual ~BadWorker() = default;
389     virtual void work() = 0;
390     virtual void eat() = 0;
391     virtual void sleep() = 0;
392 };
393
394 class BadHumanWorker : public BadWorker {
395 public:
396     void work() override {
397         std::cout << "    [Human] Working...\n";
398     }
399
400     void eat() override {
401         std::cout << "    [Human] Eating lunch...\n";
402     }
403
404     void sleep() override {
405         std::cout << "    [Human] Sleeping...\n";
406     }
407 };
408
409 class BadRobotWorker : public BadWorker {
410 public:
411     void work() override {
412         std::cout << "    [Robot] Working 24/7...\n";
413     }
414
415     // Robots don't eat or sleep! Forced to implement unused methods
416     void eat() override {
417         // Empty or throw exception (both are bad)
```

```
419     std::cout << "[Robot] ERROR: Robots don't eat!\n";
420 }
421
422 void sleep() override {
423     // Empty or throw exception (both are bad)
424     std::cout << "[Robot] ERROR: Robots don't sleep!\n";
425 }
426
427 // GOOD: Segregated interfaces - clients use only what they need
428 class Workable {
429 public:
430     virtual ~Workable() = default;
431     virtual void work() = 0;
432 };
433
434 class Eatable {
435 public:
436     virtual ~Eatable() = default;
437     virtual void eat() = 0;
438 };
439
440 class Sleepable {
441 public:
442     virtual ~Sleepable() = default;
443     virtual void sleep() = 0;
444 };
445
446
447 // Human implements all interfaces
448 class HumanWorker : public Workable, public Eatable, public Sleepable {
449 public:
450     void work() override {
451         std::cout << "[Human] Working...\n";
452     }
453
454     void eat() override {
455         std::cout << "[Human] Eating lunch...\n";
456     }
457
458     void sleep() override {
459         std::cout << "[Human] Sleeping...\n";
460     }
461 };
462
463
464 // Robot only implements what it needs
465 class RobotWorker : public Workable {
466 public:
467     void work() override {
468         std::cout << "[Robot] Working 24/7...\n";
469     }
470     // No eat() or sleep() - not forced to implement them!
471 };
472
473 void demonstrate() {
```

```
473 std::cout << "\n" << std::string(70, '=') << "\n";
474 std::cout << "PRINCIPLE 4: INTERFACE SEGREGATION PRINCIPLE (ISP)\n";
475 std::cout << std::string(70, '=') << "\n\n";
476
477 std::cout << "  DEFINITION:\n";
478 std::cout << "    Clients should not be forced to depend on interfaces\n";
479 std::cout << "    they don't use. Prefer many specific interfaces over\n";
480 std::cout << "    one general-purpose interface.\n\n";
481
482 std::cout << "  BAD Example (Fat Interface):\n";
483 BadRobotWorker badRobot;
484 badRobot.work();
485 badRobot.eat(); // Forced to implement, but doesn't make sense!
486 badRobot.sleep(); // Forced to implement, but doesn't make sense!
487 std::cout << "\n";
488
489 std::cout << "  GOOD Example (Segregated Interfaces):\n";
490 HumanWorker human;
491 human.work();
492 human.eat();
493 human.sleep();
494 std::cout << "\n";
495
496 RobotWorker robot;
497 robot.work();
498 std::cout << "    [Robot] No eat() or sleep() methods - clean interface!\n"
499     "\n";
500
501 std::cout << "  BENEFITS:\n";
502 std::cout << "  •  Classes implement only what they need\n";
503 std::cout << "  •  No dummy implementations or exceptions\n";
504 std::cout << "  •  Better decoupling and flexibility\n";
505 std::cout << "  •  Easier to understand and maintain\n";
506 }
507 } // namespace interface_segregation
508
509 // =====
510 // PRINCIPLE 5: DEPENDENCY INVERSION PRINCIPLE (DIP)
511 // =====
512 // High-level modules should not depend on low-level modules
513 // Both should depend on abstractions (interfaces)
514 // Abstractions should not depend on details, details should depend on
515 // abstractions
516
517 namespace dependency_inversion {
518
519 // BAD: High-level class depends on low-level concrete classes
520 class BadMySQLDatabase {
521 public:
522     void connect() {
523         std::cout << "    [BAD] Connecting to MySQL...\n";
524     }
525 }
```

```
525     void saveData(const std::string& data) {
526         std::cout << "    [BAD] Saving to MySQL: " << data << "\n";
527     }
528 };
529
530 class BadUserService {
531 private:
532     BadMySQLDatabase database_; // Tightly coupled to MySQL!
533
534 public:
535     void saveUser(const std::string& user) {
536         database_.connect();
537         database_.saveData(user);
538         // If we want to switch to PostgreSQL, we must modify this class!
539     }
540 };
541
542 // GOOD: Depend on abstraction, not concrete implementation
543 class IDatabase {
544 public:
545     virtual ~IDatabase() = default;
546     virtual void connect() = 0;
547     virtual void saveData(const std::string& data) = 0;
548 };
549
550 // Low-level modules implement the interface
551 class MySQLDatabase : public IDatabase {
552 public:
553     void connect() override {
554         std::cout << "    [GOOD] Connecting to MySQL...\n";
555     }
556
557     void saveData(const std::string& data) override {
558         std::cout << "    [GOOD] Saving to MySQL: " << data << "\n";
559     }
560 };
561
562 class PostgreSQLDatabase : public IDatabase {
563 public:
564     void connect() override {
565         std::cout << "    [GOOD] Connecting to PostgreSQL...\n";
566     }
567
568     void saveData(const std::string& data) override {
569         std::cout << "    [GOOD] Saving to PostgreSQL: " << data << "\n";
570     }
571 };
572
573 class MongoDBDatabase : public IDatabase {
574 public:
575     void connect() override {
576         std::cout << "    [GOOD] Connecting to MongoDB...\n";
577     }
578 }
```

```
579     void saveData(const std::string& data) override {
580         std::cout << "[GOOD] Saving to MongoDB: " << data << "\n";
581     }
582 };
583
584 // High-level module depends on abstraction (IDatabase), not concrete class
585 class UserService {
586 private:
587     IDatabase& database_; // Depends on interface, not implementation!
588
589 public:
590     explicit UserService(IDatabase& db) : database_(db) {}
591
592     void saveUser(const std::string& user) {
593         database_.connect();
594         database_.saveData(user);
595         // Can switch database without modifying this class!
596     }
597 };
598
599 void demonstrate() {
600     std::cout << "\n" << std::string(70, '=') << "\n";
601     std::cout << "PRINCIPLE 5: DEPENDENCY INVERSION PRINCIPLE (DIP)\n";
602     std::cout << std::string(70, '=') << "\n\n";
603
604     std::cout << " DEFINITION:\n";
605     std::cout << " 1. High-level modules should not depend on low-level
606         modules.\n";
607     std::cout << "      Both should depend on abstractions.\n";
608     std::cout << " 2. Abstractions should not depend on details.\n";
609     std::cout << "      Details should depend on abstractions.\n\n";
610
611     std::cout << " BAD Example (Tight Coupling):\n";
612     BadUserService badService;
613     badService.saveUser("John Doe");
614     std::cout << " Problem: UserService is tightly coupled to MySQL!\n\n";
615
616     std::cout << " GOOD Example (Dependency Injection):\n";
617
618     MySQLDatabase mysql;
619     UserService service1(mysql);
620     service1.saveUser("Jane Smith");
621     std::cout << "\n";
622
623     PostgreSQLDatabase postgres;
624     UserService service2(postgres);
625     service2.saveUser("Bob Johnson");
626     std::cout << "\n";
627
628     MongoDBDatabase mongo;
629     UserService service3(mongo);
630     service3.saveUser("Alice Williams");
631     std::cout << "\n";
```

```
632     std::cout << " BENEFITS:\n";
633     std::cout << " • Loose coupling between modules\n";
634     std::cout << " • Easy to switch implementations\n";
635     std::cout << " • Better testability (can inject mocks)\n";
636     std::cout << " • Follows \"Program to an interface, not an
637         implementation\"\n";
638 }
639 } // namespace dependency_inversion
640
641 // =====
642 // MAIN - Demonstrate All SOLID Principles
643 // =====
644
645 int main() {
646     std::cout << "\n";
647     std::cout << "                                     \n";
648     std::cout << "                                     SOLID PRINCIPLES IN C++\n";
649     std::cout << "                                     Object-Oriented Design Best Practices\n";
650     std::cout << "                                     \n";
651
652     try {
653         // S - Single Responsibility Principle
654         single_responsibility::demonstrate();
655
656         // O - Open-Closed Principle
657         open_closed::demonstrate();
658
659         // L - Liskov Substitution Principle
660         liskov_substitution::demonstrate();
661
662         // I - Interface Segregation Principle
663         interface_segregation::demonstrate();
664
665         // D - Dependency Inversion Principle
666         dependency_inversion::demonstrate();
667
668         // Summary
669         std::cout << "\n" << std::string(70, '=') << "\n";
670         std::cout << "SUMMARY: SOLID PRINCIPLES\n";
671         std::cout << std::string(70, '=') << "\n\n";
672
673         std::cout << "S - Single Responsibility Principle:\n";
674         std::cout << "     One class, one job, one reason to change\n\n";
675
676         std::cout << "O - Open-Closed Principle:\n";
677         std::cout << "     Open for extension, closed for modification\n\n";
678
679         std::cout << "L - Liskov Substitution Principle:\n";
680         std::cout << "     Derived classes must be substitutable for base
681             classes\n\n";
```

```
682     std::cout << "I - Interface Segregation Principle:\n";
683     std::cout << "    Many specific interfaces > one general interface\n\n"
684     ";
685
686     std::cout << "D - Dependency Inversion Principle:\n";
687     std::cout << "    Depend on abstractions, not concrete implementations\n\n";
688
689     std::cout << " APPLYING SOLID:\n";
690     std::cout << " • Leads to more maintainable code\n";
691     std::cout << " • Reduces coupling between components\n";
692     std::cout << " • Makes code more testable\n";
693     std::cout << " • Improves code reusability\n";
694     std::cout << " • Easier to understand and modify\n\n";
695
696     std::cout << "                                     \n";
697     std::cout << "                         ALL SOLID PRINCIPLES DEMONSTRATED!\n";
698     std::cout << "                                     \n";
699
700 } catch (const std::exception& e) {
701     std::cerr << " Error: " << e.what() << "\n";
702     return 1;
703 }
704
705 return 0;
}
```

## 63 Source Code: STLContainersNoHeap.cpp

File: src/STLContainersNoHeap.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <array>
3 #include <vector>
4 #include <memory>
5 #include <cstdint>
6 #include <cstring>
7 #include <algorithm>
8 #include <span>
9
10 // =====
11 // CAN STL CONTAINERS WORK ON SYSTEMS WITHOUT HEAP?
12 // =====
13 // Comprehensive answer:
14 // 1. Most STL containers (vector, map, list) require heap by default
15 // 2. std::array works WITHOUT heap (C++11)
16 // 3. std::span works WITHOUT heap (C++20)
17 // 4. Custom allocators can use stack/static memory
18 // 5. C++17 PMR (Polymorphic Memory Resources) for flexible allocation
19 //
20 // This example demonstrates ALL approaches for heap-less STL usage
21 // =====
22
23 // =====
24 // 1. STACK-BASED CONTAINERS - std::array (NO HEAP!)
25 // =====
26
27 void example_std_array() {
28     std::cout << "\n== 1. std::array - STACK-BASED, NO HEAP ==" << std::endl
29     ;
30
31     // std::array: Fixed size, stack allocated, zero heap usage
32     std::array<int, 10> sensor_data = {10, 20, 30, 40, 50, 60, 70, 80, 90,
33                                         100};
34
35     std::cout << "\nSensor readings:" << std::endl;
36     for (size_t i = 0; i < sensor_data.size(); ++i) {
37         std::cout << " Sensor " << i << ":" << sensor_data[i] << std::endl;
38     }
39
40     // Full STL algorithm support
41     std::cout << "\nSorted readings:" << std::endl;
42     std::sort(sensor_data.begin(), sensor_data.end());
43     for (const auto& reading : sensor_data) {
44         std::cout << " " << reading;
45     }
46     std::cout << std::endl;
47
48     // constexpr support (C++17)
49     constexpr std::array<int, 5> compile_time = {1, 2, 3, 4, 5};
```

```
48 [[maybe_unused]] constexpr int sum = compile_time[0] + compile_time[1];
49     // Compile-time computation
50
50 std::cout << "\n BENEFITS:" << std::endl;
51 std::cout << " • Zero heap allocation" << std::endl;
52 std::cout << " • Stack-based (fast access)" << std::endl;
53 std::cout << " • Fixed size known at compile time" << std::endl;
54 std::cout << " • Same interface as std::vector" << std::endl;
55 std::cout << " • constexpr support (C++17)" << std::endl;
56 std::cout << " • Perfect for embedded systems" << std::endl;
57
58 std::cout << "\n LIMITATIONS:" << std::endl;
59 std::cout << " • Fixed size (cannot grow/shrink)" << std::endl;
60 std::cout << " • Size must be known at compile time" << std::endl;
61 }
62
63 // =====
64 // 2. NON-OWNING VIEWS - std::span (C++20, NO HEAP!)
65 // =====
66
67 void process_data(std::span<int> data) {
68     std::cout << " Processing " << data.size() << " elements: ";
69     for (auto val : data) {
70         std::cout << val << " ";
71     }
72     std::cout << std::endl;
73 }
74
75 void example_std_span() {
76     std::cout << "\n==== 2. std::span - NON-OWNING VIEW, NO HEAP ===" << std::endl;
77
78     // Stack-based data
79     std::array<int, 5> arr = {1, 2, 3, 4, 5};
80     int c_array[] = {10, 20, 30, 40};
81
82     // std::span: Non-owning view, zero allocation
83     std::span<int> span1(arr);
84     std::span<int> span2(c_array, 4);
85
86     std::cout << "\nProcessing array:" << std::endl;
87     process_data(span1);
88
89     std::cout << "\nProcessing C array:" << std::endl;
90     process_data(span2);
91
92     // Sub-spans (slicing)
93     auto first_three = span1.first(3);
94     std::cout << "\nFirst 3 elements:" << std::endl;
95     process_data(first_three);
96
97     std::cout << "\n BENEFITS:" << std::endl;
98     std::cout << " • Zero allocation (view only)" << std::endl;
99     std::cout << " • Works with std::array, C arrays, std::vector" << std::endl;
```

```
        endl;
100    std::cout << " • Safe bounds checking (can enable)" << std::endl;
101    std::cout << " • Sub-span support (slicing)" << std::endl;
102    std::cout << " • Modern replacement for (T*, size_t)" << std::endl;
103
104    std::cout << "\n CAUTION:" << std::endl;
105    std::cout << " • Non-owning (must ensure lifetime)" << std::endl;
106    std::cout << " • Can dangle if underlying data destroyed" << std::endl;
107 }
108
109 // =====
110 // 3. CUSTOM STACK ALLOCATOR FOR std::vector
111 // =====
112
113 template<typename T, size_t N>
114 class StackAllocator {
115 private:
116     alignas(T) uint8_t storage[N * sizeof(T)];
117     size_t used = 0;
118
119 public:
120     using value_type = T;
121     using size_type = std::size_t;
122     using difference_type = std::ptrdiff_t;
123
124     StackAllocator() = default;
125
126     template<typename U>
127     StackAllocator(const StackAllocator<U, N>&) noexcept {}
128
129     template<typename U>
130     struct rebind {
131         using other = StackAllocator<U, N>;
132     };
133
134     T* allocate(size_t n) {
135         if (used + n > N) {
136             throw std::bad_alloc(); // Or return nullptr in embedded
137         }
138
139         T* result = reinterpret_cast<T*>(storage + used * sizeof(T));
140         used += n;
141         return result;
142     }
143
144     void deallocate(T* p, size_t n) noexcept {
145         // Simple stack allocator: only deallocate if it's the last allocation
146         T* last = reinterpret_cast<T*>(storage + (used - n) * sizeof(T));
147         if (p == last) {
148             used -= n;
149         }
150         // Otherwise, memory is "leaked" until allocator is destroyed
151         // For better deallocation, use a more sophisticated allocator
152     }
}
```

```
153     size_t capacity() const { return N; }
154     size_t available() const { return N - used; }
155 };
156
157
158 template<typename T, size_t N, typename U, size_t M>
159 bool operator==(const StackAllocator<T, N>&, const StackAllocator<U, M>&)
160     noexcept {
161     return false; // Different allocators are never equal
162 }
163
164 template<typename T, size_t N, typename U, size_t M>
165 bool operator!=(const StackAllocator<T, N>&, const StackAllocator<U, M>&)
166     noexcept {
167     return true;
168 }
169
170
171 void example_custom_stack_allocator() {
172     std::cout << "\n==== 3. CUSTOM STACK ALLOCATOR FOR std::vector ===" << std
173         ::endl;
174
175     // std::vector with stack allocator - NO HEAP!
176     using StackVec = std::vector<int, StackAllocator<int, 20>};
177     StackVec vec;
178
179     std::cout << "\nAdding elements (stack allocation):" << std::endl;
180     vec.reserve(10); // Pre-reserve to avoid reallocation
181     for (int i = 0; i < 10; ++i) {
182         vec.push_back(i * 10);
183         std::cout << " Added " << i * 10
184             << ", size=" << vec.size() << std::endl;
185     }
186
187     std::cout << std::endl;
188
189     std::cout << "\n BENEFITS:" << std::endl;
190     std::cout << " • Uses std::vector interface" << std::endl;
191     std::cout << " • Stack-based (no heap)" << std::endl;
192     std::cout << " • Can still push_back(), resize(), etc." << std::endl;
193     std::cout << " • Custom allocator handles memory" << std::endl;
194
195     std::cout << "\n LIMITATIONS:" << std::endl;
196     std::cout << " • Fixed maximum capacity (20 in this case)" << std::endl;
197     std::cout << " • Simple deallocation strategy" << std::endl;
198     std::cout << " • Throws exception when out of memory" << std::endl;
199 }
200
201 // =====
202 // 4. STATIC MEMORY POOL ALLOCATOR
203 // =====
```

```
204
205 template<typename T, size_t PoolSize>
206 class PoolAllocator {
207 private:
208     struct Block {
209         alignas(T) uint8_t data[sizeof(T)];
210     };
211
212     // Static storage - shared across all instances
213     inline static Block storage[PoolSize];
214     inline static bool used[PoolSize] = {};
215     inline static size_t allocated_count = 0;
216
217 public:
218     using value_type = T;
219     using size_type = std::size_t;
220     using difference_type = std::ptrdiff_t;
221
222     PoolAllocator() noexcept = default;
223
224     template<typename U>
225     PoolAllocator(const PoolAllocator<U, PoolSize>&) noexcept {}
226
227     template<typename U>
228     struct rebinding {
229         using other = PoolAllocator<U, PoolSize>;
230     };
231
232     T* allocate(size_t n) {
233         if (n > PoolSize || allocated_count + n > PoolSize) {
234             throw std::bad_alloc();
235         }
236
237         // Find contiguous free blocks
238         for (size_t i = 0; i <= PoolSize - n; ++i) {
239             bool found = true;
240             for (size_t j = 0; j < n; ++j) {
241                 if (used[i + j]) {
242                     found = false;
243                     break;
244                 }
245             }
246
247             if (found) {
248                 for (size_t j = 0; j < n; ++j) {
249                     used[i + j] = true;
250                 }
251                 allocated_count += n;
252                 return reinterpret_cast<T*>(&storage[i]);
253             }
254         }
255
256         throw std::bad_alloc();
257     }
```

```
258     void deallocate(T* p, size_t n) noexcept {
259         if (!p) return;
260
261         auto base = reinterpret_cast<Block*>(storage);
262         auto block = reinterpret_cast<Block*>(p);
263         size_t index = block - base;
264
265         if (index < PoolSize) {
266             for (size_t i = 0; i < n && (index + i) < PoolSize; ++i) {
267                 used[index + i] = false;
268             }
269             allocated_count -= n;
270         }
271     }
272
273     static size_t pool_size() { return PoolSize; }
274 };
275
276
277 template<typename T, size_t N, typename U, size_t M>
278 bool operator==(const PoolAllocator<T, N>&, const PoolAllocator<U, M>&)
279     noexcept {
280     return N == M;
281 }
282
283 template<typename T, size_t N, typename U, size_t M>
284 bool operator!=(const PoolAllocator<T, N>&, const PoolAllocator<U, M>&)
285     noexcept {
286     return N != M;
287 }
288
289 void example_pool_allocator() {
290     std::cout << "\n==== 4. STATIC MEMORY POOL ALLOCATOR ===" << std::endl;
291
292     struct Sensor {
293         int id;
294         float value;
295         Sensor(int i, float v) : id(i), value(v) {
296             std::cout << "    Sensor(" << id << ", " << value << ")" << std::endl;
297         }
298         ~Sensor() {
299             std::cout << "    ~Sensor(" << id << ")" << std::endl;
300         }
301     };
302
303     using SensorVec = std::vector<Sensor, PoolAllocator<Sensor, 10>>;
304
305     std::cout << "\nAllocating sensors from static pool:" << std::endl;
306     {
307         SensorVec sensors;
308         sensors.push_back(Sensor(1, 25.5f));
309         sensors.push_back(Sensor(2, 30.2f));
310         sensors.push_back(Sensor(3, 28.7f));
```

```
309     std::cout << "\nSensors in vector: " << sensors.size() << std::endl;
310 }
312 std::cout << "\nVector destroyed, memory returned to pool" << std::endl;
313
314 std::cout << "\n BENEFITS:" << std::endl;
315 std::cout << " • Static memory (no heap)" << std::endl;
316 std::cout << " • Fast O(1) allocation/deallocation" << std::endl;
317 std::cout << " • No fragmentation" << std::endl;
318 std::cout << " • Works with std::vector, std::list, etc." << std::endl;
319 std::cout << " • Memory reused across instances" << std::endl;
320
321 std::cout << "\n LIMITATIONS:" << std::endl;
322 std::cout << " • Fixed pool size" << std::endl;
323 std::cout << " • Static storage (lifetime of program)" << std::endl;
324 std::cout << " • Simple implementation (one object per allocation)" <<
325     std::endl;
326 }
327 // =====
328 // 5. FIXED CAPACITY VECTOR (NO ALLOCATOR NEEDED)
329 // =====
330
331 template<typename T, size_t Capacity>
332 class FixedVector {
333 private:
334     alignas(T) uint8_t storage[Capacity * sizeof(T)];
335     size_t count = 0;
336
337 public:
338     using value_type = T;
339     using iterator = T*;
340     using const_iterator = const T*;
341
342     FixedVector() = default;
343
344     ~FixedVector() {
345         clear();
346     }
347
348     // Copy constructor
349     FixedVector(const FixedVector& other) : count(0) {
350         for (size_t i = 0; i < other.count; ++i) {
351             push_back(other[i]);
352         }
353     }
354
355     // Move constructor
356     FixedVector(FixedVector&& other) noexcept : count(0) {
357         for (size_t i = 0; i < other.count; ++i) {
358             push_back(std::move(other[i]));
359         }
360         other.clear();
361     }
```

```
362     void push_back(const T& value) {
363         if (count >= Capacity) {
364             // In embedded: return error code or assert
365             throw std::length_error("FixedVector capacity exceeded");
366         }
367         new (storage + count * sizeof(T)) T(value);
368         ++count;
369     }
370
371     void push_back(T&& value) {
372         if (count >= Capacity) {
373             throw std::length_error("FixedVector capacity exceeded");
374         }
375         new (storage + count * sizeof(T)) T(std::move(value));
376         ++count;
377     }
378
379     template<typename... Args>
380     void emplace_back(Args&&... args) {
381         if (count >= Capacity) {
382             throw std::length_error("FixedVector capacity exceeded");
383         }
384         new (storage + count * sizeof(T)) T(std::forward<Args>(args)...);
385         ++count;
386     }
387
388     void pop_back() {
389         if (count > 0) {
390             --count;
391             reinterpret_cast<T*>(storage + count * sizeof(T))->~T();
392         }
393     }
394
395     void clear() {
396         while (count > 0) {
397             pop_back();
398         }
399     }
400
401     T& operator[](size_t index) {
402         return *reinterpret_cast<T*>(storage + index * sizeof(T));
403     }
404
405     const T& operator[](size_t index) const {
406         return *reinterpret_cast<const T*>(storage + index * sizeof(T));
407     }
408
409     T& front() { return (*this)[0]; }
410     const T& front() const { return (*this)[0]; }
411     T& back() { return (*this)[count - 1]; }
412     const T& back() const { return (*this)[count - 1]; }
413
414     size_t size() const { return count; }
```

```

416     size_t capacity() const { return Capacity; }
417     bool empty() const { return count == 0; }
418     bool full() const { return count == Capacity; }
419
420     iterator begin() { return reinterpret_cast<T*>(storage); }
421     iterator end() { return reinterpret_cast<T*>(storage + count * sizeof(T)); }
422     const_iterator begin() const { return reinterpret_cast<const T*>(storage); }
423     const_iterator end() const { return reinterpret_cast<const T*>(storage +
424                               count * sizeof(T)); }
424 };
425
426 void example_fixed_vector() {
427     std::cout << "\n==== 5. FIXED CAPACITY VECTOR (NO ALLOCATOR) ===" << std::endl;
428
429     FixedVector<int, 10> vec;
430
431     std::cout << "\nAdding elements:" << std::endl;
432     for (int i = 0; i < 8; ++i) {
433         vec.push_back(i * 5);
434         std::cout << " Pushed " << i * 5
435                     << ", size=" << vec.size()
436                     << "/" << vec.capacity() << std::endl;
437     }
438
439     std::cout << "\nVector contents:" << std::endl;
440     for (const auto& val : vec) {
441         std::cout << " " << val;
442     }
443     std::cout << std::endl;
444
445     std::cout << "\n BENEFITS:" << std::endl;
446     std::cout << " • Complete control over memory" << std::endl;
447     std::cout << " • No allocator overhead" << std::endl;
448     std::cout << " • Vector-like interface" << std::endl;
449     std::cout << " • Stack or static storage" << std::endl;
450     std::cout << " • Perfect for embedded systems" << std::endl;
451     std::cout << " • Compile-time capacity checking" << std::endl;
452 }
453
454 // =====
455 // 6. SUMMARY TABLE
456 // =====
457
458 void example_summary() {
459     std::cout << "\n==== 6. SUMMARY: STL ON HEAP-LESS SYSTEMS ===" << std::endl
460     ;
461
462     std::cout << "＼" n" << std::endl;
463     std::cout << " APPROACH HEAP? GROWS? COMPLEXITY USE
464             CASE " << std::endl;
465     std::cout << " " << std::endl;

```

```

464     std::cout << "  std::array"                                NO      NO      SIMPLE   Fixed
465     "  << std::endl;
466     std::cout << "  std::span (C++20)"                      NO      NO      SIMPLE   View
467     "  << std::endl;
468     std::cout << "  Stack Allocator"                         NO      YES     MEDIUM
469     "  Temporary" << std::endl;
470     std::cout << "  Pool Allocator"                          NO      YES     MEDIUM
471     "  Reusable" << std::endl;
472     std::cout << "  FixedVector"                            NO      YES     LOW
473     "  Embedded" << std::endl;
474     std::cout << "  std::vector (default)"      YES     YES     SIMPLE
475     "  General" << std::endl;
476     std::cout << "  "                                     " << std::endl;
477
478     std::cout << "\n ANSWER: Can STL containers work without heap?" << std::endl;
479
480     std::cout << "\n YES! Multiple approaches:" << std::endl;
481     std::cout << "  1. Use std::array - Fixed size, stack-based" << std::endl;
482     "  ";
483     std::cout << "  2. Use std::span - Non-owning view (C++20)" << std::endl;
484     std::cout << "  3. Custom stack allocator - std::vector on stack" << std::endl;
485     "  ";
486     std::cout << "  4. Static pool allocator - Pre-allocated memory" << std::endl;
487     "  ";
488     std::cout << "  5. Custom fixed vector - Complete control" << std::endl;
489
490
491     std::cout << "\n NO (by default):" << std::endl;
492     std::cout << "  •  std::vector - Uses heap by default" << std::endl;
493     std::cout << "  •  std::map - Uses heap by default" << std::endl;
494     std::cout << "  •  std::list - Uses heap by default" << std::endl;
495     std::cout << "  •  std::string - Uses heap (except SSO)" << std::endl;
496     std::cout << "  •  All use std::allocator which calls new/delete" << std::endl;
497
498
499     std::cout << "\n BEST PRACTICES FOR EMBEDDED SYSTEMS:" << std::endl;
500
501     std::cout << "  1. PREFER std::array:" << std::endl;
502     "  Simple, safe, zero overhead" << std::endl;
503     std::cout << "  Compile-time size checking" << std::endl;
504     std::cout << "  Full STL algorithm support" << std::endl;
505
506
507     std::cout << "\n 2. USE std::span for function parameters:" << std::endl;
508     std::cout << "  Works with array, vector, C arrays" << std::endl;
509     std::cout << "  Zero allocation" << std::endl;
510     std::cout << "  Modern replacement for (T*, size_t)" << std::endl;
511
512
513     std::cout << "\n 3. CUSTOM ALLOCATORS when needed:" << std::endl;
514     std::cout << "  std::vector with stack/pool allocator" << std::endl;
515     std::cout << "  Keep std::vector interface" << std::endl;
516     std::cout << "  More complex, test thoroughly" << std::endl;
517
518
519     std::cout << "\n 4. CUSTOM CONTAINERS for critical paths:" << std::endl;
520     std::cout << "  FixedVector, CircularBuffer, etc." << std::endl;
521     std::cout << "  Complete control over behavior" << std::endl;
522     std::cout << "  Optimized for specific use case" << std::endl;

```

```
507 }
508
509 // =====
510 // 7. REAL-WORLD EXAMPLE: SENSOR DATA COLLECTION
511 // =====
512
513 void example_real_world() {
514     std::cout << "\n==== 7. REAL-WORLD: SENSOR DATA COLLECTION ===" << std::endl;
515
516     std::cout << "\n SCENARIO:" << std::endl;
517     std::cout << " • ARM Cortex-M4 @ 80MHz" << std::endl;
518     std::cout << " • 64KB RAM (no heap)" << std::endl;
519     std::cout << " • Collect 100 sensor readings" << std::endl;
520     std::cout << " • Process with STL algorithms" << std::endl;
521
522     // Solution 1: std::array (simplest)
523     std::cout << "\n SOLUTION 1: std::array" << std::endl;
524     std::array<float, 100> readings1 = {};
525     for (size_t i = 0; i < 100; ++i) {
526         readings1[i] = 20.0f + (i % 10) * 0.5f; // Simulated readings
527     }
528
529     auto min_max1 = std::minmax_element(readings1.begin(), readings1.end());
530     std::cout << " Min: " << *min_max1.first
531             << ", Max: " << *min_max1.second << std::endl;
532
533     // Solution 2: FixedVector (dynamic-like)
534     std::cout << "\n SOLUTION 2: FixedVector" << std::endl;
535     FixedVector<float, 100> readings2;
536     for (int i = 0; i < 100; ++i) {
537         readings2.push_back(20.0f + (i % 10) * 0.5f);
538     }
539
540     auto min_max2 = std::minmax_element(readings2.begin(), readings2.end());
541     std::cout << " Min: " << *min_max2.first
542             << ", Max: " << *min_max2.second << std::endl;
543
544     std::cout << "\n BOTH WORK WITHOUT HEAP!" << std::endl;
545     std::cout << " • Zero dynamic allocation" << std::endl;
546     std::cout << " • Full STL algorithm support" << std::endl;
547     std::cout << " • Predictable memory usage" << std::endl;
548     std::cout << " • Real-time safe" << std::endl;
549 }
550
551 // =====
552 // MAIN
553 // =====
554
555 int main() {
556     std::cout << "\n"
557             ===== <<
558     std::endl;
559     std::cout << " CAN STL CONTAINERS WORK ON SYSTEMS WITHOUT HEAP?" << std::endl;
```

```
558     endl;
559     std::cout << "
560     ====="
561     std::endl;
562
563     example_std_array();
564     example_std_span();
565     example_custom_stack_allocator();
566     example_pool_allocator();
567     example_fixed_vector();
568     example_summary();
569     example_real_world();
570
571     std::cout << "\n"
572     ====="
573     std::endl;
574     std::cout << "    FINAL ANSWER" << std::endl;
575     std::cout << "
576     ====="
577     std::endl;
578
579     std::cout << "\n YES! STL containers CAN work without heap:" << std::endl
580     ;
581
582     std::cout << "\n1. BUILT-IN HEAP-LESS CONTAINERS:" << std::endl;
583     std::cout << "    • std::array (C++11) - Fixed size, stack-based" << std::
584     endl;
585     std::cout << "    • std::span (C++20) - Non-owning view" << std::endl;
586     std::cout << "    • std::string_view (C++17) - String view" << std::endl;
587
588     std::cout << "\n2. CUSTOM ALLOCATORS:" << std::endl;
589     std::cout << "    • Stack allocator - Use stack memory" << std::endl;
590     std::cout << "    • Pool allocator - Pre-allocated pools" << std::endl;
591     std::cout << "    • PMR allocators - C++17 polymorphic" << std::endl;
592     std::cout << "    • Works with: vector, map, list, etc." << std::endl;
593
594     std::cout << "\n3. CUSTOM CONTAINERS:" << std::endl;
595     std::cout << "    • FixedVector - Vector with capacity limit" << std::endl;
596     std::cout << "    • CircularBuffer - Ring buffer" << std::endl;
597     std::cout << "    • StaticString - Fixed-size string" << std::endl;
598     std::cout << "    • Full control over allocation" << std::endl;
599
600
601     std::cout << "\n DEFAULT STL CONTAINERS USE HEAP:" << std::endl;
602     std::cout << "    • std::vector - std::allocator uses new/delete" << std::
603     endl;
604     std::cout << "    • std::map - std::allocator uses new/delete" << std::endl
605     ;
606     std::cout << "    • std::string - std::allocator (except SSO)" << std::endl
607     ;
608     std::cout << "    • Must use custom allocator or alternatives" << std::endl
609     ;
610
611
612     std::cout << "\n RECOMMENDATION FOR EMBEDDED/REAL-TIME:" << std::endl;
613     std::cout << "    1st choice: std::array (simple, safe)" << std::endl;
```

```
599     std::cout << " 2nd choice: std::span for parameters" << std::endl;
600     std::cout << " 3rd choice: Custom allocators for std::vector" << std::
601         endl;
601     std::cout << " 4th choice: Custom containers (FixedVector)" << std::endl
602         ;
602     std::cout << " Avoid: Default std::vector, std::map, std::string" <<
603         std::endl;
603
604     std::cout << "\n"
604         =====\n" <<
604         std::endl;
605
606     return 0;
607 }
```

## 64 Source Code: SafetyCriticalSTLContainers.cpp

File: src/SafetyCriticalSTLContainers.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // STL CONTAINERS FOR SAFETY-CRITICAL SYSTEMS
3 // =====
4 // Comprehensive guide for using STL containers in:
5 // - ISO 26262 (Automotive functional safety)
6 // - DO-178C (Avionics software)
7 // - IEC 61508 (Industrial safety)
8 // - Medical device software (IEC 62304)
9 //
10 // Based on:
11 // - MISRA C++:2008 and MISRA C++:2023
12 // - AUTOSAR C++14 Coding Guidelines
13 // - High Integrity C++ (HICPP)
14 // - JSF AV C++ Coding Standards
15 //
16 // Key concerns in safety-critical systems:
17 // 1. Deterministic behavior (no unpredictable timing)
18 // 2. Bounded memory usage (no dynamic allocation at runtime)
19 // 3. No hidden control flow (exceptions, virtual functions)
20 // 4. Verifiable and testable code
21 // 5. No undefined behavior
22 //
23 // Build: g++ -std=c++20 -Wall -Wextra -Wpedantic -O2 -o
24 // SafetyCriticalSTLContainers SafetyCriticalSTLContainers.cpp
25 // =====
26 #include <iostream>
27 #include <array>
28 #include <vector>
29 #include <list>
30 #include <forward_list>
31 #include <set>
32 #include <map>
33 #include <unordered_set>
34 #include <unordered_map>
35 #include <memory>
36 #include <cstdint>
37 #include <string>
38 #include <algorithm>
39 //
40 // =====
41 // SECTION 1: CONTAINER CLASSIFICATION FOR SAFETY-CRITICAL SYSTEMS
42 // =====
43
44 namespace container_classification {
45
46 void demonstrate() {
47     std::cout << "\n" << std::string(80, '=') << "\n";
48     std::cout << "SECTION 1: STL CONTAINERS - SAFETY-CRITICAL CLASSIFICATION\n";
```

```

        ";
49  std::cout << std::string(80, '=' ) << "\n\n";
50
51  std::cout << "                                     \n";
52  std::cout << "  CONTAINER          HEAP?  NODE-BASED?  SAFE?  NOTES
53  std::cout << "          \n";
54  std::cout << "  std::array          NO      NO          BEST for
55  std::cout << "  safety           \n";
56  std::cout << "  std::vector         YES      NO          OK with
57  std::cout << "  allocator         \n";
58  std::cout << "  std::list          YES      YES         AVOID per-
59  std::cout << "  node            \n";
60  std::cout << "  std::forward_list  YES      YES         AVOID per-
61  std::cout << "  node            \n";
62  std::cout << "  std::deque          YES      NO          AVOID
63  std::cout << "  complex           \n";
64  std::cout << "  std::set            YES      YES         AVOID per-
65  std::cout << "  node            \n";
66  std::cout << "  std::map            YES      YES         AVOID per-
67  std::cout << "  node            \n";
68  std::cout << "  std::multiset        YES      YES         AVOID per-
69  std::cout << "  node            \n";
70  std::cout << "  std::multimap        YES      YES         AVOID per-
71  std::cout << "  node            \n";
72  std::cout << "  std::unordered_set    YES      YES         AVOID per-
73  std::cout << "  bucket           \n";
74  std::cout << "  std::unordered_map    YES      YES         AVOID per-
75  std::cout << "  bucket           \n";
76  std::cout << "  std::unordered_*        YES      YES         AVOID per-
77  std::cout << "  bucket           \n";
78  std::cout << "                                     \n\n";
79  std::cout << "KEY ISSUES WITH HEAP ALLOCATION IN SAFETY-CRITICAL SYSTEMS:\n
80
81  std::cout << "1. NON-DETERMINISTIC TIMING:\n";
82  std::cout << "  •  malloc/new can take variable time depending on:\n";
83  std::cout << "    - Current heap fragmentation\n";
84  std::cout << "    - Size of allocation request\n";
85  std::cout << "    - Operating system state\n";
86  std::cout << "  •  Worst-case execution time (WCET) becomes unpredictable\n
87  std::cout << "  •  Unacceptable for hard real-time systems\n\n";
88
89  std::cout << "2. MEMORY FRAGMENTATION:\n";
90  std::cout << "  •  Repeated allocations/deallocations cause fragmentation\n
91  std::cout << "  •  Can lead to allocation failures even with available
92  std::cout << "    memory\n";
93  std::cout << "  •  Node-based containers (list, map, set) are worst
94  std::cout << "    offenders\n";
95  std::cout << "  •  Each element requires separate heap allocation\n\n";

```

```
84     std::cout << "3. ALLOCATION FAILURES:\n";
85     std::cout << " • new can throw std::bad_alloc (exception handling
86             required)\n";
86     std::cout << " • malloc can return nullptr (error handling required)\n";
87     std::cout << " • Difficult to recover gracefully in critical systems\n";
88     std::cout << " • May violate MISRA/AUTOSAR guidelines on exceptions\n\n"
89             ;
90
90     std::cout << "4. NODE-BASED CONTAINERS (list, map, set, unordered_*):\n";
91     std::cout << " • EACH ELEMENT requires separate heap allocation\n";
92     std::cout << " • 1000 elements = 1000 malloc/free calls\n";
93     std::cout << " • Massive fragmentation over time\n";
94     std::cout << " • Iterator/pointer invalidation issues\n";
95     std::cout << " • Hidden per-node overhead (8-24 bytes per element)\n\n";
96
97     std::cout << "5. STANDARDS GUIDANCE:\n";
98     std::cout << " • MISRA C++ Rule 18-4-1: Dynamic heap allocation shall
99             not be used\n";
100    std::cout << " • AUTOSAR A18-5-1: Functions malloc, calloc, realloc,
101            free not used\n";
100    std::cout << " • ISO 26262: Recommends static memory allocation for ASIL
101            -D\n";
101    std::cout << " • DO-178C: Dynamic allocation discouraged for Level A
101            software\n";
102 }
103
104 } // namespace container_classification
105
106 // =====
107 // SECTION 2: CONTAINERS TO AVOID IN SAFETY-CRITICAL SYSTEMS
108 // =====
109
110 namespace containers_to_avoid {
111
112     void demonstrate_list_issues() {
113         std::cout << "\n" << std::string(80, '-') << "\n";
114         std::cout << "WHY AVOID std::list, std::forward_list?\n";
115         std::cout << std::string(80, '-') << "\n\n";
116
117         std::cout << "PROBLEM: Each element requires separate heap allocation\n\n"
117             ;
118
119         // Demonstrate list overhead
120         std::list<int32_t> my_list;
121
122         std::cout << "Adding 5 elements to std::list:\n";
123         for (int32_t i = 0; i < 5; ++i) {
124             my_list.push_back(i * 10);
125             std::cout << " push_back(" << i * 10 << ") - HEAP ALLOCATION #" << (i
125                 + 1) << "\n";
126         }
127
128         std::cout << "\n ISSUES:\n";
129         std::cout << " • 5 separate malloc() calls - non-deterministic timing\n"
```

```

    ;
130  std::cout << " • Each node has pointer overhead (16-24 bytes per element
      )\n";
131  std::cout << " • Fragmentation increases with each allocation\n";
132  std::cout << " • Cannot use custom allocator to pre-allocate (node-by-
      node)\n";
133  std::cout << " • Cache-unfriendly (nodes scattered in memory)\n\n";
134
135  std::cout << " ALTERNATIVE: std::array or std::vector with reserve()\n";
136 }
137
138 void demonstrate_map_set_issues() {
139  std::cout << "\n" << std::string(80, '-') << "\n";
140  std::cout << "WHY AVOID std::map, std::set, std::multimap, std::multiset?\n
      ";
141  std::cout << std::string(80, '-') << "\n\n";
142
143  std::cout << "PROBLEM: Red-black tree with per-node heap allocation\n\n";
144
145  std::map<int32_t, std::string> my_map;
146
147  std::cout << "Adding 3 entries to std::map:\n";
148  my_map[1] = "Critical";
149  std::cout << " map[1] = \"Critical\" - HEAP ALLOCATION for node\n";
150  my_map[2] = "Warning";
151  std::cout << " map[2] = \"Warning\" - HEAP ALLOCATION for node\n";
152  my_map[3] = "Info";
153  std::cout << " map[3] = \"Info\" - HEAP ALLOCATION for node\n";
154
155  std::cout << "\n ISSUES:\n";
156  std::cout << " • Each insertion allocates a tree node (40-64 bytes
      overhead!)\n";
157  std::cout << " • Rebalancing operations (rotations) at runtime\n";
158  std::cout << " • Non-constant insertion time: O(log n) with allocation\n
      ";
159  std::cout << " • Cannot pre-allocate all nodes\n";
160  std::cout << " • Iterators invalidated on modification\n\n";
161
162  std::cout << " ALTERNATIVES:\n";
163  std::cout << " • std::array of key-value pairs + binary search\n";
164  std::cout << " • std::vector of pairs + sort + binary search\n";
165  std::cout << " • Fixed-size hash table (compile-time allocation)\n";
166 }
167
168 void demonstrate_unordered_issues() {
169  std::cout << "\n" << std::string(80, '-') << "\n";
170  std::cout << "WHY AVOID std::unordered_map, std::unordered_set, std::
      unordered_multimap?\n";
171  std::cout << std::string(80, '-') << "\n\n";
172
173  std::cout << "PROBLEM: Hash table with bucket allocation + chaining\n\n";
174
175  std::unordered_map<int32_t, std::string> my_unordered_map;
176

```

```

177 std::cout << "Adding 3 entries to std::unordered_map:\n";
178 my_unordered_map[100] = "Sensor1";
179 std::cout << "  map[100] = \"Sensor1\" - HEAP: buckets + node\n";
180 my_unordered_map[200] = "Sensor2";
181 std::cout << "  map[200] = \"Sensor2\" - HEAP: buckets + node\n";
182 my_unordered_map[300] = "Sensor3";
183 std::cout << "  map[300] = \"Sensor3\" - HEAP: buckets + node\n";
184
185 std::cout << "\n ISSUES:\n";
186 std::cout << " • Initial bucket array allocation\n";
187 std::cout << " • Each element allocated separately (chaining)\n";
188 std::cout << " • REHASHING at runtime (load factor > threshold)\n";
189 std::cout << "   - Allocates NEW larger bucket array\n";
190 std::cout << "   - Rehashes ALL elements (moves to new buckets)\n";
191 std::cout << "   - Deallocates old bucket array\n";
192 std::cout << "   - MASSIVE non-determinism!\n";
193 std::cout << " • Cannot predict WCET (worst-case execution time)\n";
194 std::cout << " • Hash collisions cause unpredictable performance\n\n";
195
196 std::cout << " ALTERNATIVES:\n";
197 std::cout << " • Fixed-size hash table (no rehashing)\n";
198 std::cout << " • Perfect hashing (compile-time)\n";
199 std::cout << " • Sorted std::array + binary search\n";
200 }
201
202 void demonstrate_deque_issues() {
203 std::cout << "\n" << std::string(80, '-') << "\n";
204 std::cout << "WHY AVOID std::deque?\n";
205 std::cout << std::string(80, '-') << "\n\n";
206
207 std::cout << "PROBLEM: Complex internal structure with multiple
208   allocations\n\n";
209
210 std::cout << "std::deque internal structure:\n";
211 std::cout << " • Array of pointers to fixed-size chunks\n";
212 std::cout << " • Each chunk allocated separately\n";
213 std::cout << " • Grows by allocating new chunks + updating pointer array\n
214   ";
215 std::cout << " • Pointer array itself may need reallocation\n\n";
216
217 std::cout << " ISSUES:\n";
218 std::cout << " • Multiple heap allocations (chunks + pointer array)\n";
219 std::cout << " • Complex iterator invalidation rules\n";
220 std::cout << " • Non-contiguous memory (cache-unfriendly)\n";
221 std::cout << " • Unpredictable memory usage pattern\n";
222 std::cout << " • Difficult to analyze for WCET\n\n";
223
224 std::cout << " ALTERNATIVES:\n";
225 std::cout << " • std::array (fixed size)\n";
226 std::cout << " • std::vector with reserve() (growable)\n";
227 std::cout << " • Circular buffer (custom implementation)\n";
228 }
229
230 void demonstrate() {

```

```
229     std::cout << "\n" << std::string(80, '=' ) << "\n";
230     std::cout << "SECTION 2: CONTAINERS TO AVOID IN SAFETY-CRITICAL SYSTEMS\n"
231     ;
232     std::cout << std::string(80, '=' ) << "\n";
233
234     demonstrate_list_issues();
235     demonstrate_map_set_issues();
236     demonstrate_unordered_issues();
237     demonstrate_deque_issues();
238 }
239 } // namespace containers_to_avoid
240
241 // =====
242 // SECTION 3: SAFE ALTERNATIVES - USING CUSTOM ALLOCATORS
243 // =====
244
245 namespace safe_alternatives {
246
247 // Pre-allocated memory pool allocator for safety-critical systems
248 template<typename T, size_t PoolSize>
249 class SafetyPoolAllocator {
250 private:
251     struct alignas(T) Block {
252         uint8_t data[sizeof(T)];
253     };
254
255     // Static storage - allocated at compile time
256     inline static Block storage[PoolSize];
257     inline static bool used[PoolSize] = {};
258     inline static size_t allocated_count = 0;
259
260 public:
261     using value_type = T;
262     using size_type = std::size_t;
263
264     SafetyPoolAllocator() noexcept = default;
265
266     template<typename U>
267     SafetyPoolAllocator(const SafetyPoolAllocator<U, PoolSize>&) noexcept {}
268
269     template<typename U>
270     struct rebind {
271         using other = SafetyPoolAllocator<U, PoolSize>;
272     };
273
274     // Allocate from pool - bounded, deterministic
275     [[nodiscard]] T* allocate(size_t n) {
276         if (n > PoolSize || allocated_count + n > PoolSize) {
277             // In safety-critical systems, handle gracefully without exception
278             return nullptr; // Or use error code
279         }
280
281         // Find contiguous free blocks
```

```

282     for (size_t i = 0; i <= PoolSize - n; ++i) {
283         bool found = true;
284         for (size_t j = 0; j < n; ++j) {
285             if (used[i + j]) {
286                 found = false;
287                 break;
288             }
289         }
290
291         if (found) {
292             for (size_t j = 0; j < n; ++j) {
293                 used[i + j] = true;
294             }
295             allocated_count += n;
296             return reinterpret_cast<T*>(&storage[i]);
297         }
298     }
299
300     return nullptr; // Pool exhausted
301 }
302
303 void deallocate(T* p, size_t n) noexcept {
304     if (!p) return;
305
306     // Find and mark blocks as free
307     for (size_t i = 0; i < PoolSize; ++i) {
308         if (reinterpret_cast<T*>(&storage[i]) == p) {
309             for (size_t j = 0; j < n && (i + j) < PoolSize; ++j) {
310                 used[i + j] = false;
311             }
312             allocated_count -= n;
313             return;
314         }
315     }
316 }
317
318 static size_t capacity() { return PoolSize; }
319 static size_t available() { return PoolSize - allocated_count; }
320 };
321
322 template<typename T, size_t N, typename U, size_t M>
323 bool operator==(const SafetyPoolAllocator<T, N>&, const SafetyPoolAllocator<U,
324 M>&) noexcept {
325     return N == M;
326 }
327
328 template<typename T, size_t N, typename U, size_t M>
329 bool operator!=(const SafetyPoolAllocator<T, N>&, const SafetyPoolAllocator<U,
330 M>&) noexcept {
331     return N != M;
332 }
333
334 void demonstrate_vector_with_allocator() {
335     std::cout << "\n" << std::string(80, '-') << "\n";

```

```
334     std::cout << "SAFE: std::vector WITH CUSTOM ALLOCATOR\n";
335     std::cout << std::string(80, '-') << "\n\n";
336
337     std::cout << " Solution: Pre-allocated pool at initialization\n\n";
338
339     using SafeVector = std::vector<int32_t, SafetyPoolAllocator<int32_t,
340                               100>>;
341
342     std::cout << "Creating std::vector with SafetyPoolAllocator<int32_t,
343                  100>:\n";
344     std::cout << " • Pool allocated at compile-time (static storage)\n";
345     std::cout << " • Max 100 elements (bounded memory)\n";
346     std::cout << " • No heap allocation during runtime\n";
347     std::cout << " • Deterministic allocation from pool\n\n";
348
349     SafeVector safe_vec;
350     safe_vec.reserve(10); // Reserve from pool
351
352     std::cout << "Adding 10 elements:\n";
353     for (int32_t i = 0; i < 10; ++i) {
354         safe_vec.push_back(i * 10);
355         std::cout << " push_back(" << i * 10 << ") - from POOL, not heap\n";
356     }
357
358     std::cout << "\n BENEFITS:\n";
359     std::cout << " • NO heap allocation (malloc/new)\n";
360     std::cout << " • Deterministic timing (pool allocation is O(n) bounded)\n";
361     std::cout << " • No fragmentation\n";
362     std::cout << " • Bounded memory usage (max 100 elements)\n";
363     std::cout << " • Keeps std::vector interface\n";
364     std::cout << " • Compatible with algorithms (sort, find, etc.)\n\n";
365
366     std::cout << " LIMITATIONS:\n";
367     std::cout << " • Fixed maximum capacity (100 in this example)\n";
368     std::cout << " • Must handle allocation failures gracefully\n";
369     std::cout << " • Pool exhaustion returns nullptr (not exception)\n";
370     std::cout << " • Requires careful sizing during design phase\n";
371
372     void demonstrate_array_alternative() {
373         std::cout << "\n" << std::string(80, '-') << "\n";
374         std::cout << "SAFEST: std::array - NO ALLOCATOR NEEDED\n";
375         std::cout << std::string(80, '-') << "\n\n";
376
377         std::cout << " BEST SOLUTION for fixed-size data: std::array\n\n";
378
379         std::array<int32_t, 10> safe_array = {0, 10, 20, 30, 40, 50, 60, 70, 80,
380                                              90};
381
382         std::cout << "std::array<int32_t, 10> properties:\n";
383         std::cout << " • Zero heap allocation (stack or static storage)\n";
384         std::cout << " • Size known at compile time\n";
385         std::cout << " • Bounds checking with .at()\n";
```

```
384     std::cout << " • Compatible with STL algorithms\n";
385     std::cout << " • constexpr support (C++17)\n";
386     std::cout << " • No overhead vs C array\n\n";
387
388     std::cout << "Array contents:\n ";
389     for (const auto& val : safe_array) {
390         std::cout << val << " ";
391     }
392     std::cout << "\n\n";
393
394     std::cout << " • PERFECT FOR SAFETY-CRITICAL:\n";
395     std::cout << " • ISO 26262 ASIL-D compliant\n";
396     std::cout << " • MISRA C++ compliant (prefer over C arrays)\n";
397     std::cout << " • AUTOSAR C++14 recommended\n";
398     std::cout << " • DO-178C Level A acceptable\n";
399     std::cout << " • Deterministic behavior (O(1) access)\n";
400     std::cout << " • Predictable memory footprint\n";
401     std::cout << " • No hidden control flow\n";
402     std::cout << " • Verifiable and testable\n";
403 }
404
405 void demonstrate_initialization_phase_pattern() {
406     std::cout << "\n" << std::string(80, '-') << "\n";
407     std::cout << "PATTERN: Initialization Phase vs Runtime Phase\n";
408     std::cout << std::string(80, '-') << "\n\n";
409
410     std::cout << "GOLDEN RULE for safety-critical systems:\n";
411     std::cout << " 'Allocate during INITIALIZATION, operate during RUNTIME'\n
412
413     std::cout << "                                     \n";
414     std::cout << " PHASE           ALLOWED           FORBIDDEN
415     \n";
416     std::cout << "                                     \n";
417     std::cout << " INITIALIZATION • Dynamic allocation • Nothing
418     \n";
419     std::cout << " (startup) •             reserve()
420     \n";
421     std::cout << " •                   Pool setup
422     \n";
423     std::cout << " •                   Resource acquisition
424     \n";
425     std::cout << "                                     \n";
426     std::cout << " RUNTIME •             Read/write data • new/malloc
427     \n";
428     std::cout << " (operational) •      Fixed operations • delete/free
429     \n";
430     std::cout << " •                   Pool allocation •      realloc
431     \n";
432     std::cout << " •                   Bounded algorithms •      resize/grow
433     \n";
434     std::cout << " •                   Exceptions
435     \n";
436     std::cout << "                                     \n\n";
```

```
427     std::cout << "EXAMPLE: Two-phase lifecycle\n\n";
428
429     std::cout << "// ===== INITIALIZATION PHASE (startup, non-critical) =====\n";
430     std::cout << "void initialize() {\n";
431     std::cout << "    static std::vector<SensorData> sensor_buffer;\n";
432     std::cout << "    sensor_buffer.reserve(1000); // ONE-TIME allocation\n";
433     std::cout << "    // Pre-allocate ALL buffers, pools, resources\n";
434     std::cout << "}\n\n";
435
436
437     std::cout << "// ===== RUNTIME PHASE (operational, time-critical) =====\n";
438     std::cout << "void process_sensor_data(SensorData data) {\n";
439     std::cout << "    // NO allocation here! Only use pre-allocated memory\n";
440     std::cout << "    sensor_buffer.push_back(data); // No realloc (reserved)\n";
441     std::cout << "    if (sensor_buffer.size() > 1000) {\n";
442     std::cout << "        sensor_buffer.erase(sensor_buffer.begin()); //\n";
443     std::cout << "        Shift\n";
444     std::cout << "    }\n";
445     std::cout << "}\n\n";
446
447     std::cout << "  KEY PRINCIPLES:\n";
448     std::cout << "  1. Allocate ONCE during initialization\n";
449     std::cout << "  2. Use reserve() to prevent reallocation\n";
450     std::cout << "  3. Never exceed reserved capacity at runtime\n";
451     std::cout << "  4. Use static or global storage for long-lived data\n";
452     std::cout << "  5. Monitor memory usage during testing\n";
453     std::cout << "  6. Assert/verify no runtime allocation in production\n";
454 }
455
456 void demonstrate() {
457     std::cout << "\n" << std::string(80, '=') << "\n";
458     std::cout << "SECTION 3: SAFE ALTERNATIVES WITH CUSTOM ALLOCATORS\n";
459     std::cout << std::string(80, '=') << "\n";
460
461     demonstrate_vector_with_allocator();
462     demonstrate_array_alternative();
463     demonstrate_initialization_phase_pattern();
464 }
465
466 } // namespace safe_alternatives
467
468 // =====
469 // SECTION 4: STANDARDS AND GUIDELINES SUMMARY
470 // =====
471
472 namespace standards_summary {
473
474     void demonstrate() {
475         std::cout << "\n" << std::string(80, '=') << "\n";
476         std::cout << "SECTION 4: STANDARDS AND GUIDELINES SUMMARY\n";
477         std::cout << std::string(80, '=') << "\n\n";
478 }
```

```

477     std::cout << "                                     \n";
478     std::cout << "  STANDARD / GUIDELINE  KEY REQUIREMENTS
479                                         \n";
480     std::cout << "                                     \n";
481     std::cout << "  MISRA C++:2008          Rule 18-4-1: No dynamic heap
482         allocation           \n";
483     std::cout << "                                     \n";
484     std::cout << "  discouraged           Rule 5-2-4: C-style casts forbidden
485         \n";
486     std::cout << "                                     \n";
487     std::cout << "  MISRA C++:2023          Rule 27-0-1: <cstdio> functions
488                                         \n";
489     std::cout << "                                     \n";
490     std::cout << "  AUTOSAR C++14          Rule 18-5-1: malloc/calloc/realloc/free
491         forbidden           A18-5-2: new/delete in non-throwing
492         \n";
493     std::cout << "         form only          A18-1-1: Prefer std::array over C
494         \n";
495     std::cout << "         arrays             \n";
496     std::cout << "                                     \n";
497     std::cout << "  ISO 26262 (ASIL-D)    Recommends static allocation
498                                         \n";
499     std::cout << "         justification      Dynamic allocation requires
500         \n";
501     std::cout << "                                     \n";
502     std::cout << "  DO-178C Level A        WCET analysis mandatory
503                                         \n";
504     std::cout << "         (Avionics)        \n";
505     std::cout << "         \n";
506     std::cout << "         \n";
507     std::cout << "  IEC 61508 (SIL 3/4)   Dynamic memory discouraged
508                                         \n";
509     std::cout << "         \n";
510     std::cout << "  JSF AV C++             Must prove bounded memory usage
511                                         \n";
512     std::cout << "         \n";
513     std::cout << "  (F-35 Fighter)        Full MC/DC coverage required
514         init                \n";
515     std::cout << "         \n";
516     std::cout << "         \n";
517     std::cout << "  Rule 206: malloc/free not used
518                                         \n";
519     std::cout << "  Rule 207: new/delete not used after
520         \n";
521     std::cout << "         \n";
522     std::cout << "  Rule 208: Prefer automatic storage
523                                         \n\n";

```

```

509     std::cout << "CONTAINER RECOMMENDATIONS BY STANDARD:\n\n";
510
511     std::cout << "  UNIVERSALLY ACCEPTED (all standards):\n";
512     std::cout << "    •    std::array - Fixed size, no allocation\n";
513     std::cout << "    •    std::span (C++20) - Non-owning view\n";
514     std::cout << "    •    C arrays with size (if bounds-checked)\n\n";
515
516     std::cout << "  CONDITIONALLY ACCEPTED (with restrictions):\n";
517     std::cout << "    •    std::vector - ONLY with:\n";
518     std::cout << "        - reserve() called during initialization\n";
519     std::cout << "        - Custom allocator (pool-based)\n";
520     std::cout << "        - Never exceeds reserved capacity at runtime\n";
521     std::cout << "        - Documented maximum size\n\n";
522
523     std::cout << "  UNIVERSALLY FORBIDDEN (all standards):\n";
524     std::cout << "    •    std::list - Per-node heap allocation\n";
525     std::cout << "    •    std::forward_list - Per-node heap allocation\n";
526     std::cout << "    •    std::map / std::set - Per-node heap allocation\n";
527     std::cout << "    •    std::multimap / std::multiset - Per-node heap
528         allocation\n";
529     std::cout << "    •    std::unordered_map / std::unordered_set - Rehashing\n";
530     std::cout << "    •    std::unordered_multimap / std::unordered_multiset -
531         Rehashing\n";
532     std::cout << "    •    std::deque - Complex internal structure\n";
533     std::cout << "    •    std::string - Dynamic (unless with custom allocator)\n\n";
534
535     std::cout << "RATIONALE FOR PROHIBITIONS:\n\n";
536
537     std::cout << "1. NODE-BASED CONTAINERS (list, map, set):\n";
538     std::cout << "    Problem: Each element → separate allocation\n";
539     std::cout << "    Impact: 1000 elements = 1000 heap operations\n";
540     std::cout << "    Result: Massive fragmentation + non-deterministic timing\n\n";
541
542     std::cout << "2. HASH CONTAINERS (unordered_map, unordered_set):\n";
543     std::cout << "    Problem: Rehashing at unpredictable times\n";
544     std::cout << "    Impact: Can reallocate entire bucket array at runtime\n";
545     std::cout << "    Result: Unpredictable latency spikes (WCET violation)\n\n";
546
547     std::cout << "3. DEQUE:\n";
548     std::cout << "    Problem: Complex multi-level allocation\n";
549     std::cout << "    Impact: Chunks + pointer array allocations\n";
550     std::cout << "    Result: Difficult to analyze and verify\n\n";
551
552 } // namespace standards_summary
553
554 // =====
555 // SECTION 5: PRACTICAL DECISION GUIDE
556 // =====
557

```

```
558 namespace decision_guide {  
559  
560 void demonstrate() {  
561     std::cout << "\n" << std::string(80, '=') << "\n";  
562     std::cout << "SECTION 5: PRACTICAL DECISION GUIDE\n";  
563     std::cout << std::string(80, '=') << "\n\n";  
564  
565     std::cout << "DECISION TREE: Which Container Should I Use?\n\n";  
566  
567     std::cout << "    Q: Is the size KNOWN at compile time?\n";  
568     std::cout << "\n";  
569     std::cout << "    YES → Use std::array<T, N>\n";  
570     std::cout << "                BEST choice for safety-critical\n";  
571     std::cout << "    • Zero overhead\n";  
572     std::cout << "    • No allocation\n";  
573     std::cout << "    • Compile-time size checking\n";  
574     std::cout << "\n";  
575     std::cout << "    NO → Continue to Q2\n\n";  
576  
577     std::cout << "    Q: Can I determine MAXIMUM size at design time?\n";  
578     std::cout << "\n";  
579     std::cout << "    YES → Use std::vector with reserve(MAX_SIZE)\n";  
580     std::cout << "                ACCEPTABLE with these conditions:\n";  
581     std::cout << "    1. Call reserve() during initialization\n";  
582     std::cout << "    2. Never exceed reserved capacity at runtime\n";  
583     std::cout << "    3. Monitor usage during testing\n";  
584     std::cout << "    4. Document maximum size\n";  
585     std::cout << "    5. Consider custom allocator for extra safety\n";  
586     std::cout << "\n";  
587     std::cout << "    NO (truly unbounded) → DESIGN PROBLEM!\n";  
588     std::cout << "                Safety-critical systems MUST have bounded  
589     resources\n";  
590     std::cout << "    • Revisit requirements\n";  
591     std::cout << "    • Determine worst-case maximum\n";  
592     std::cout << "    • Add resource limits\n";  
593     std::cout << "    • Consider circular buffer pattern\n";  
594     std::cout << "\n";  
595     std::cout << "    ALTERNATIVE → Fixed-capacity container (custom)\n\n";  
596  
597     std::cout << "    Q: Do I need to insert/remove in the MIDDLE frequently?\n";  
598     std::cout << "\n";  
599     std::cout << "    YES (was thinking std::list) → RECONSIDER!\n";  
600     std::cout << "    • std::list forbidden in safety-critical systems\n";  
601     std::cout << "    • Alternatives:\n";  
602     std::cout << "        1. std::vector + erase/insert (usually fast  
603         enough)\n";  
604     std::cout << "        2. Circular buffer (if queue-like)\n";  
605     std::cout << "        3. Fixed-size priority queue\n";  
606     std::cout << "        4. Index-based linked list in std::array\n";  
607     std::cout << "\n";  
608     std::cout << "    NO → std::vector or std::array is fine\n\n";
```

```

608 std::cout << " Q: Do I need key-value mapping (was thinking std::map)?\n"
609     ;
610 std::cout << " \n";
611 std::cout << " YES → ALTERNATIVES (no std::map):\n";
612 std::cout << "     1. std::array of std::pair<Key, Value> +
613     binary_search\n";
614 std::cout << "     • Sort during initialization\n";
615 std::cout << "     • Use binary_search at runtime (O(log n))\n";
616 std::cout << "     • No allocation after init\n";
617 std::cout << " \n";
618 std::cout << "     2. Fixed-size hash table (compile-time)\n";
619 std::cout << "     • Perfect hashing (no collisions)\n";
620 std::cout << "     • O(1) lookup\n";
621 std::cout << "     • Requires design-time hash analysis\n";
622 std::cout << " \n";
623 std::cout << "     3. std::vector<pair> + sort + binary_search\n";
624 std::cout << "     • Sorted during initialization\n";
625 std::cout << "     • Read-only at runtime\n";
626 std::cout << "     • Good for configuration tables\n";
627 std::cout << " \n";
628 std::cout << "     NO → Continue\n\n";
629
630 std::cout << " SUMMARY TABLE:\n\n";
631 std::cout << " \n";
632 std::cout << " NEED           SAFE SOLUTION           AVOID
633     \n";
634 std::cout << "     \n";
635 std::cout << "     Fixed-size sequence     std::array           C array (
636     no bounds) \n";
637 std::cout << "     Bounded sequence       std::vector + reserve() std::
638     vector (default) \n";
639 std::cout << "     FIFO queue            Circular buffer        std::
640     queue/std::list \n";
641 std::cout << "     LIFO stack            std::array + index    std::
642     stack (default) \n";
643 std::cout << "     Priority queue        std::array + heap alg  std::
644     priority_queue \n";
645 std::cout << "     Key-value map         sorted array + search std::map
646     \n";
647 std::cout << "     Hash table            Fixed hash table      std::
648     unordered_map \n";
649 std::cout << "     Sorted set            sorted array + search std::set
650     \n";
651 std::cout << "     Linked list           Index-based in array std::list
652     \n";
653 std::cout << " \n\n";
654
655 std::cout << " GOLDEN RULES:\n\n";
656 std::cout << "1. ALWAYS prefer std::array when size is known\n";
657 std::cout << "2. NEVER use node-based containers (list, map, set)\n";
658 std::cout << "3. NEVER use std::unordered_* (rehashing issue)\n";
659 std::cout << "4. IF using std::vector:\n";
660 std::cout << "     • Call reserve() during initialization\n";
661 std::cout << "     • Never exceed reserved capacity at runtime\n";

```

```
650     std::cout << " • Consider custom allocator for extra safety\n";
651     std::cout << "5. DOCUMENT maximum sizes for all containers\n";
652     std::cout << "6. VERIFY no runtime allocation (static analysis)\n";
653     std::cout << "7. TEST with memory allocation monitoring enabled\n";
654 }
655
656 } // namespace decision_guide
657
658 // =====
659 // MAIN
660 // =====
661
662 int main() {
663     std::cout << "\n" << std::string(80, '=') << "\n";
664     std::cout << "STL CONTAINERS FOR SAFETY-CRITICAL SYSTEMS\n";
665     std::cout << "ISO 26262 | DO-178C | MISRA C++ | AUTOSAR C++14\n";
666     std::cout << std::string(80, '=') << "\n";
667
668     container_classification::demonstrate();
669     containers_to_avoid::demonstrate();
670     safe_alternatives::demonstrate();
671     standards_summary::demonstrate();
672     decision_guide::demonstrate();
673
674     std::cout << "\n" << std::string(80, '=') << "\n";
675     std::cout << "FINAL ANSWER TO: Which STL containers to avoid?\n";
676     std::cout << std::string(80, '=') << "\n\n";
677
678     std::cout << " ABSOLUTELY FORBIDDEN in safety-critical systems:\n\n";
679     std::cout << " 1. std::list - Per-element heap allocation\n";
680     std::cout << " 2. std::forward_list - Per-element heap allocation\n";
681     std::cout << " 3. std::map - Per-node allocation (red-black tree)\n";
682     std::cout << " 4. std::multimap - Per-node allocation\n";
683     std::cout << " 5. std::set - Per-node allocation\n";
684     std::cout << " 6. std::multiset - Per-node allocation\n";
685     std::cout << " 7. std::unordered_map - Rehashing + per-bucket allocation
686             \n";
686     std::cout << " 8. std::unordered_multimap - Rehashing + per-bucket
687             allocation\n";
687     std::cout << " 9. std::unordered_set - Rehashing + per-bucket allocation
688             \n";
688     std::cout << " 10. std::unordered_multiset - Rehashing + per-bucket
689             allocation\n";
690     std::cout << " 11. std::deque - Complex multi-level allocation\n";
691     std::cout << " 12. std::string - Dynamic (use with custom allocator only
692             )\n\n";
693
694     std::cout << " YES, you CAN use containers with CUSTOM ALLOCATORS:\n\n";
695     std::cout << " • std::vector<T, CustomAllocator>\n";
696     std::cout << " • Pre-allocated pool-based allocator\n";
697     std::cout << " • Static memory pool (no heap)\n";
698     std::cout << " • Bounded capacity enforced\n";
699     std::cout << " • Allocation during initialization phase only\n\n";
```

```
699     std::cout << "  SAFEST CHOICE (no allocator needed):\n\n";
700     std::cout << "  •  std::array<T, N> - Stack or static storage\n";
701     std::cout << "  •  std::span<T> (C++20) - Non-owning view\n";
702     std::cout << "  •  C++17 std::string_view - String view\n\n";
703
704     std::cout << "  REMEMBER:\n";
705     std::cout << "    If heap is the issue, custom allocators can help,\n";
706     std::cout << "    but std::array is ALWAYS the safest choice when size is
707     known!\n\n";
708
709     std::cout << std::string(80, '=') << "\n\n";
710
711     return 0;
}
```

## 65 Source Code: SearchAnagramsDictionary.cpp

File: src/SearchAnagramsDictionary.cpp

Repository: [View on GitHub](#)

```
1 // Example program
2 #include <iostream>
3 #include <string>
4
5 #include <algorithm>
6 #include <functional>
7 #include <iterator>
8
9 #include <vector>
10 #include <set>
11 #include <map>
12
13 using Pair_t = std::multimap< std::string, std::string> ;
14 using MMAPIterator = std::multimap<std::string, std::string>::iterator ;
15
16 std::set<std::string> mDictionary = { "act", "ant", "art", "bat", "bet",
17     "boss", "cat", "cap", "cop", "dear", "dog",
18     "dip", "ear", "end", "eel", "fad", "fat", "fog", "gap", "god", "hat", "hit",
19     "hot", "ink", "irk", "jot", "jab",
20     "lap", "lip", "lot", "man", "nan", "nat", "net", "pat", "pet", "tap", "tar",
21     "ten", "rat", "woo", "yoo", "zoo" };
22
23 int main()
24 {
25     Pair_t Pairs = {};
26     std::vector<std::string> emptyVectors = {};
27
28     //std::cout << "All words in the mDictionary are :" << std::endl;
29     for(const auto& it : mDictionary) {
30
31         std::string sortedWord = it;
32         std::sort(sortedWord.begin(), sortedWord.end());
33
34         Pairs.emplace(std::make_pair(sortedWord, it));
35
36         //std::cout << "for word " << it << ", sorted word = " << sortedWord <<
37         //std::endl;
38     }
39
40     for(auto& obj : Pairs) {
41
42         std::cout << "unsorted word = " << obj.second << ", sorted word : " <<
43             obj.first << std::endl;
44     }
45
46     for(auto& obj : Pairs) {
47
48         std::pair <MMAPIterator, MMAPIterator> equalPairs = Pairs.equal_range(
```

```
45     obj.first);
46
47     if(equalPairs.second != equalPairs.first)
48     {
49         std::cout << "for the word = " << obj.second << ", its anagrams
50         are : ";
51
52         for (MMAPIterator it=equalPairs.first; it!=equalPairs.second; ++it
53             )
54         {
55
56             if(obj.second != it->second) {
57                 std::cout << it->second << ", ";
58             }
59
60             //std::cout << it->first << " : " << it->second << ", ";
61         }
62
63     }
64 }
```

## 66 Source Code: SinglyLinkedList.cpp

File: src/SinglyLinkedList.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <string.h>
4 /*
5     pHead
6     |
7     |
8     +---+-----+     +---+-----+     +---+-----+
9     | 1 | next ---->| 2 | next----->| 3 | NULL |
10    +---+-----+     +---+-----+     +---+-----+
11 */
12 extern "C" {
13     struct info;
14     typedef struct info{
15         char name[20];
16         int age;
17         info *next;
18     } personal_info_t;
19
20     info* pHead = NULL; //by default first node will be point to nothing
21
22     extern "C" void deleteNode(info* p); //Forward declaration
23
24     extern "C" void deleteNodeByName(info** pH, const char* name) {
25         info *prevNode, *currNode = *pH;
26
27         while(currNode->next != NULL) {
28
29             if(strcmp(currNode->name, name) == 0) {
30                 std::cout << "found the match, will delete node named " <<
31                 currNode->name << std::endl;
32                 deleteNode(prevNode);
33                 return;
34             }
35             prevNode = currNode;
36             currNode = currNode->next;
37         }
38
39         if(currNode->next == NULL) {
40             if(strcmp(currNode->name, name) == 0) {
41                 std::cout << currNode->name << " is the last node...." << std::
42                 endl;
43                 memset((void*) currNode, 0, sizeof(info));
44                 free((info*) currNode);
45                 prevNode->next = NULL;
46             }
47             else {
48                 std::cout << "cannot find the matching name " << name << std::endl
49                 ;
50             }
51         }
52     }
53 }
```

```
48     }
49 }
50 }
51
52 extern "C" void deleteNode(info* p) {
53     info* pNode = p->next;
54     std::cout << pNode->name << " is followed by " << pNode->name << std::endl;
55     p->next = pNode->next;
56     memset((void*) pNode, 0, sizeof(info));
57     free((info *) pNode);
58     pNode = p->next;
59     std::cout << pNode->name << " is followed by " << pNode->name << std::endl;
60 }
61 extern "C" void createNode(info** pH, const char* name, const int age) {
62     info* pNewNode = (info*) malloc(sizeof(info));
63     if(pNewNode != NULL) {
64         strcpy(pNewNode->name, name);
65         pNewNode->age = age;
66         pNewNode->next= NULL;
67     }
68     else {
69         return;
70     }
71
72     if(*pH == NULL) {
73         pNewNode->next= NULL;
74         *pH = pNewNode;
75         std::cout << (*pH)->name << " is Head, pointing to memory address 0x"
76             << std::hex << std::to_string((size_t)pH) << std::endl;
77     }
78     else {
79         info* pNode = *pH;
80         while(pNode->next != NULL) { pNode = pNode->next; }
81         pNode->next= pNewNode;
82     }
83 }
84 extern "C" const char* getName(const info* i) { return i->name; }
85 extern "C" const int getAge(const info* i) { return i->age; }
86 };
87
88 void print(info* pH) {
89     std::cout << getName(pH) << " is aged " << std::to_string(getAge(pH)) << "
90         followed by:" << std::endl;
91
92     info* pNode = pH;
93     while(pNode->next != NULL){
94         pNode = pNode->next;
95         std::cout << getName(pNode) << " aged " << getAge(pNode) << std::endl;
96     }
97
98 int main() {
99     createNode(&pHead, "AungBu", 20);
```

```
100  createNode(&pHead, "Sar Oo", 30);
101  createNode(&pHead, "AhBang", 25);
102  createNode(&pHead, "AhLain", 27);
103  createNode(&pHead, "Ahmedi", 28);
104
105  print(pHead);
106
107  deleteNodeByName(&pHead, "Sar Oo");
108  deleteNodeByName(&pHead, "AhLain");
109
110  print(pHead);
111
112  return 0;
113
114 }
```

## 67 Source Code: StopTokenExample.cpp

File: src/StopTokenExample.cpp

Repository: [View on GitHub](#)

```
1 // StopTokenExample.cpp
2 // Demonstrates std::stop_token (C++20) for graceful thread cancellation
3 // Shows the difference between old-style atomic flags and modern stop_token
4 //
5 // KEY CONCEPTS:
6 // 1. std::jthread - Automatically joins, supports stop_token
7 // 2. std::stop_token - Cooperative cancellation signal
8 // 3. std::stop_source - Controls stop requests
9 // 4. std::stop_callback - Actions on stop request
10 //
11 // USE CASES:
12 // REST/Monitor threads (non-critical)
13 // Worker threads in thread pools
14 // Background tasks that need graceful shutdown
15 // Any interruptible operation
16 //
17 // ADVANTAGES OVER atomic<bool>:
18 // Standard, type-safe
19 // Callback support (cleanup on stop)
20 // Works with condition_variable_any
21 // No manual join() needed with jthread
22 // Cannot be accidentally set to true
23
24 #include <iostream>
25 #include <thread>
26 #include <stop_token>
27 #include <mutex>
28 #include <condition_variable>
29 #include <queue>
30 #include <vector>
31 #include <atomic>
32 #include <chrono>
33 #include <string>
34 #include <sstream>
35 #include <iomanip>
36 #include <functional>
37
38 using namespace std::chrono_literals;
39
40 //
41 // SECTION 1: Old Style vs New Style - Comparison
42 //
43 // =====
44 namespace old_vs_new {
45
```

```
46 // OLD STYLE: Using atomic<bool> for cancellation
47 class OldStyleThread {
48 private:
49     std::atomic<bool> stop_flag_{false};
50     std::thread thread_;
51     std::mutex mutex_;
52     std::condition_variable cv_;
53
54 public:
55     void start() {
56         thread_ = std::thread([this] {
57             std::cout << " [Old Style] Thread started\n";
58
59             while (!stop_flag_.load()) {
60                 std::unique_lock<std::mutex> lock(mutex_);
61
62                 // Problem: Can't easily interrupt cv.wait()
63                 // Must use wait_for() with timeout
64                 if (cv_.wait_for(lock, 100ms, [this] { return stop_flag_.load()
65                     (); })) {
66                     break;
67                 }
68
69                 std::cout << " [Old Style] Working...\n";
70             }
71
72             std::cout << " [Old Style] Thread stopped\n";
73         });
74     }
75
76     void stop() {
77         stop_flag_ = true;
78         cv_.notify_all();
79         if (thread_.joinable()) {
80             thread_.join(); // Manual join required
81         }
82     }
83
84 // NEW STYLE: Using std::jthread with stop_token
85 class NewStyleThread {
86 private:
87     std::jthread thread_; // Automatically joins on destruction
88     std::mutex mutex_;
89     std::condition_variable_any cv_; // Note: _any variant for stop_token
90
91 public:
92     void start() {
93         thread_ = std::jthread([this](std::stop_token stoken) {
94             std::cout << " [New Style] Thread started\n";
95
96             while (!stoken.stop_requested()) {
97                 std::unique_lock<std::mutex> lock(mutex_);
```

```

99         // Can be interrupted immediately!
100        if (cv_.wait(lock, stoken, [] { return false; })) {
101            break; // Stop requested
102        }
103
104        std::cout << "  [New Style] Working...\\n";
105    }
106
107    std::cout << "  [New Style] Thread stopped\\n";
108);
109}
110
111 void stop() {
112     thread_.request_stop(); // Signal stop
113     // No manual join() needed - jthread does it automatically!
114 }
115};
116
117 void demonstrate() {
118     std::cout << "\\n" << std::string(70, '=') << "\\n";
119     std::cout << "==== SECTION 1: Old Style vs New Style ===\\n";
120     std::cout << std::string(70, '=') << "\\n\\n";
121
122     std::cout << "1. Old style with atomic<bool>:\\n";
123 {
124     OldStyleThread old_thread;
125     old_thread.start();
126     std::this_thread::sleep_for(250ms);
127     std::cout << "  [Main] Requesting stop...\\n";
128     old_thread.stop();
129 }
130
131     std::cout << "\\n2. New style with stop_token:\\n";
132 {
133     NewStyleThread new_thread;
134     new_thread.start();
135     std::this_thread::sleep_for(250ms);
136     std::cout << "  [Main] Requesting stop...\\n";
137     new_thread.stop();
138 }
139
140     std::cout << "\\n ADVANTAGES of stop_token:\\n";
141     std::cout << "  •  Type-safe (can't accidentally set to wrong value)\\n";
142     std::cout << "  •  Works with condition_variable_any for immediate
143         interruption\\n";
144     std::cout << "  •  jthread automatically joins\\n";
145     std::cout << "  •  Standard interface for cancellation\\n";
146 }
147} // namespace old_vs_new
148
149 // =====

```

```
150 // SECTION 2: REST Service with stop_token (Non-Critical Thread)
151 //
152 =====
153
153 namespace rest_service_example {
154
155 class RESTService {
156 private:
157     std::jthread thread_;
158     std::mutex mutex_;
159     std::condition_variable_any cv_;
160     std::queue<std::string> request_queue_;
161     int request_count_ = 0;
162
163 public:
164     void start() {
165         thread_ = std::jthread([this](std::stop_token stoken) {
166             std::cout << "  [REST] Service started (non-critical thread)\n";
167
168             // Register callback for cleanup
169             std::stop_callback callback(stoken, [this] {
170                 std::cout << "  [REST] Stop requested, cleaning up...\n";
171                 std::lock_guard<std::mutex> lock(mutex_);
172                 std::cout << "  [REST] Processed " << request_count_
173                             << " requests before stop\n";
174             });
175
176             try {
177                 while (!stoken.stop_requested()) {
178                     std::unique_lock<std::mutex> lock(mutex_);
179
180                     // Wait for requests or stop signal
181                     bool stop = cv_.wait(lock, stoken, [this] {
182                         return !request_queue_.empty();
183                     });
184
185                     if (stop) {
186                         std::cout << "  [REST] Stop signal received during
187                             wait\n";
188                         break;
189                     }
190
191                     // Process request
192                     if (!request_queue_.empty()) {
193                         std::string request = request_queue_.front();
194                         request_queue_.pop();
195                         lock.unlock();
196
197                         handle_request(request, stoken);
198                     }
199             }
200         });
201     }
202 }
```

```
201     }
202     catch (const std::exception& e) {
203         std::cerr << "  [REST] Exception: " << e.what() << "\n";
204         std::cerr << "  [REST] Non-critical thread exiting (core
205             services OK)\n";
206     }
207
208     std::cout << "  [REST] Service stopped gracefully\n";
209 }
210
211 void handle_request(const std::string& request, std::stop_token stoken) {
212     std::cout << "  [REST] Handling request: " << request << "\n";
213
214     // Simulate processing with interruptible sleep
215     for (int i = 0; i < 5; ++i) {
216         if (stoken.stop_requested()) {
217             std::cout << "  [REST] Request handling interrupted\n";
218             return;
219         }
220
221         std::this_thread::sleep_for(100ms);
222     }
223
224     ++request_count_;
225     std::cout << "  [REST] Request completed (total: " << request_count_
226         << ")\n";
227
228     // Simulate occasional error in REST service
229     if (request.find("CAUSE_ERROR") != std::string::npos) {
230         throw std::runtime_error("REST service error (non-critical)");
231     }
232
233     void submit_request(const std::string& request) {
234         std::lock_guard<std::mutex> lock(mutex_);
235         request_queue_.push(request);
236         cv_.notify_one();
237     }
238
239     void stop() {
240         std::cout << "  [Main] Requesting REST service stop...\n";
241         thread_.request_stop();
242         // jthread automatically joins
243     }
244
245     bool is_running() const {
246         return thread_.joinable() && !thread_.get_stop_token().stop_requested
247             ();
248     }
249
250     void demonstrate() {
251         std::cout << "\n" << std::string(70, '=') << "\n";

```



```
301         // Interruptible sleep
302         auto wake_time = std::chrono::steady_clock::now() + 1s;
303         while (std::chrono::steady_clock::now() < wake_time) {
304             if (stoken.stop_requested()) {
305                 std::cout << "  [Monitor] Stop requested during sleep\n";
306                 return;
307             }
308             std::this_thread::sleep_for(100ms);
309         }
310     }
311
312     std::cout << "  [Monitor] Service stopped\n";
313 }
314
315 void stop() {
316     thread_.request_stop();
317 }
318
319 int get_health_checks() const {
320     return health_checks_.load();
321 }
322 };
323
324 void demonstrate() {
325     std::cout << "\n" << std::string(70, '=') << "\n";
326     std::cout << "==== SECTION 3: Monitoring Service with stop_token ===\n";
327     std::cout << std::string(70, '=') << "\n\n";
328
329     MonitorService monitor;
330     monitor.start();
331
332     std::cout << "Monitoring running...\n";
333     std::this_thread::sleep_for(2500ms);
334
335     std::cout << "\nStopping monitor service...\n";
336     monitor.stop();
337
338     std::cout << "\n Total health checks: " << monitor.get_health_checks() <<
339     "\n";
340 }
341
342 } // namespace monitor_service_example
343
344 /**
345 // SECTION 4: Thread Pool with stop_token
346 /**
347 =====
348 namespace thread_pool_example {
```

```
349
350 class ThreadPool {
351 private:
352     std::vector<std::jthread> workers_;
353     std::queue<std::function<void()>> tasks_;
354     std::mutex mutex_;
355     std::condition_variable_any cv_;
356
357 public:
358     explicit ThreadPool(size_t num_threads) {
359         for (size_t i = 0; i < num_threads; ++i) {
360             workers_.emplace_back([this, i](std::stop_token stoken) {
361                 std::cout << "  [Worker " << i << "] Started\n";
362
363                 while (!stoken.stop_requested()) {
364                     std::function<void()> task;
365
366                     {
367                         std::unique_lock<std::mutex> lock(mutex_);
368
369                         // Wait for task or stop signal
370                         bool stop = cv_.wait(lock, stoken, [this] {
371                             return !tasks_.empty();
372                         });
373
374                         if (stop) {
375                             std::cout << "  [Worker " << i << "] Stop
376                             requested\n";
377                             break;
378                         }
379
380                         if (!tasks_.empty()) {
381                             task = std::move(tasks_.front());
382                             tasks_.pop();
383                         }
384
385                         if (task) {
386                             std::cout << "  [Worker " << i << "] Executing task\n"
387                             ;
388                             task();
389                         }
390
391                         std::cout << "  [Worker " << i << "] Stopped\n";
392                     });
393                 }
394             }
395
396             template<typename F>
397             void enqueue(F&& task) {
398                 {
399                     std::lock_guard<std::mutex> lock(mutex_);
400                     tasks_.emplace(std::forward<F>(task));
401                 }
402             }
403         }
404     }
405 }
```

```
401     }
402     cv_.notify_one();
403 }
404
405 void stop() {
406     std::cout << "[ThreadPool] Requesting stop for all workers...\n";
407     for (auto& worker : workers_) {
408         worker.request_stop();
409     }
410     cv_.notify_all();
411     // jthread automatically joins all threads
412 }
413
414 ~ThreadPool() {
415     stop();
416 }
417 };
418
419 void demonstrate() {
420     std::cout << "\n" << std::string(70, '=') << "\n";
421     std::cout << "==== SECTION 4: Thread Pool with stop_token ====\n";
422     std::cout << std::string(70, '=') << "\n\n";
423
424     ThreadPool pool(3);
425
426     std::cout << "Enqueueing tasks...\n";
427     for (int i = 1; i <= 5; ++i) {
428         pool.enqueue([i] {
429             std::cout << "    Task " << i << " processing...\n";
430             std::this_thread::sleep_for(200ms);
431             std::cout << "    Task " << i << " completed\n";
432         });
433     }
434
435     std::this_thread::sleep_for(1s);
436
437     std::cout << "\nStopping thread pool...\n";
438     pool.stop();
439
440     std::cout << "\n All workers stopped gracefully\n";
441 }
442
443 } // namespace thread_pool_example
444
445 // =====
446 // SECTION 5: stop_callback - Cleanup on Stop
447 // =====
448
449 namespace stop_callback_example {
450 }
```

```
451 class ServiceWithCleanup {
452 private:
453     std::jthread thread_;
454
455 public:
456     void start() {
457         thread_ = std::jthread([](std::stop_token stoken) {
458             std::cout << "[Service] Started\n";
459
460             // Register multiple callbacks
461             std::stop_callback callback1(stoken, [] {
462                 std::cout << "[Cleanup 1] Closing network connections...\n";
463             });
464
465             std::stop_callback callback2(stoken, [] {
466                 std::cout << "[Cleanup 2] Flushing buffers...\n";
467             });
468
469             std::stop_callback callback3(stoken, [] {
470                 std::cout << "[Cleanup 3] Saving state...\n";
471             });
472
473             // Simulate work
474             int work_count = 0;
475             while (!stoken.stop_requested()) {
476                 ++work_count;
477                 std::cout << "[Service] Working... (" << work_count << ")\n";
478                 ;
479                 std::this_thread::sleep_for(500ms);
480             }
481
482             std::cout << "[Service] Stopped after " << work_count << " iterations\n";
483             // Callbacks execute here (in reverse order of registration)
484         });
485
486     void stop() {
487         thread_.request_stop();
488     }
489 };
490
491 void demonstrate() {
492     std::cout << "\n" << std::string(70, '=') << "\n";
493     std::cout << "==== SECTION 5: stop_callback for Cleanup ===\n";
494     std::cout << std::string(70, '=') << "\n\n";
495
496     ServiceWithCleanup service;
497     service.start();
498
499     std::this_thread::sleep_for(1500ms);
500
501     std::cout << "\nRequesting stop...\n";
502     service.stop();
503 }
```

```
503     std::cout << "\n Cleanup callbacks executed automatically\n";
504 }
505
506 } // namespace stop_callback_example
507
508 //
509 // =====
510 // SECTION 6: Complete Microservices Example
511 //
512 // =====
513
514 namespace microservices_example {
515
516 // Core service (critical - no stop_token, uses exception for abort)
517 class DatabaseService {
518 private:
519     std::thread thread_; // Regular thread, not jthread
520     std::atomic<bool> running_{false};
521
522 public:
523     void start() {
524         running_ = true;
525         thread_ = std::thread([this] {
526             std::cout << " [Database] CORE service started (critical)\n";
527
528             try {
529                 while (running_) {
530                     std::cout << " [Database] Processing transactions...\n";
531                     std::this_thread::sleep_for(500ms);
532
533                     // Simulate critical error (would call abort())
534                     // For demo, we just log it
535                 }
536             }
537             catch (const std::exception& e) {
538                 std::cerr << " [Database] CRITICAL ERROR: " << e.what() << "\n";
539                 std::cerr << " [Database] Would call abort() in real system\n";
540             }
541
542             std::cout << " [Database] Core service stopped\n";
543         });
544     }
545
546     void stop() {
547         running_ = false;
548         if (thread_.joinable()) {
549             thread_.join();
550         }
551     }
552 }
```

```
551 };
```

```
552
553 // Non-core service (non-critical - uses stop_token)
554 class MetricsService {
555 private:
556     std::jthread thread_;
557     std::atomic<int> metrics_collected_{0};
558
559 public:
560     void start() {
561         thread_ = std::jthread([this](std::stop_token stoken) {
562             std::cout << "  [Metrics] NON-CORE service started (stop_token)\n";
563
564             std::stop_callback cleanup(stoken, [this] {
565                 std::cout << "  [Metrics] Collected " << metrics_collected_.load()
566                                     << " metrics before stop\n";
567             });
568
569             while (!stoken.stop_requested()) {
570                 ++metrics_collected_;
571                 std::cout << "  [Metrics] Collecting metric #"
572                         << metrics_collected_.load() << "\n";
573
574                 // Interruptible sleep
575                 auto wake_time = std::chrono::steady_clock::now() + 800ms;
576                 while (std::chrono::steady_clock::now() < wake_time) {
577                     if (stoken.stop_requested()) {
578                         return;
579                     }
580                     std::this_thread::sleep_for(100ms);
581                 }
582             }
583         });
584     }
585
586     void stop() {
587         std::cout << "  [Metrics] Requesting graceful stop...\n";
588         thread_.request_stop();
589     }
590 };
591
592 void demonstrate() {
593     std::cout << "\n" << std::string(70, '=') << "\n";
594     std::cout << "==== SECTION 6: Complete Microservices Example ===\n";
595     std::cout << std::string(70, '=') << "\n\n";
596
597     DatabaseService database;
598     MetricsService metrics;
599
600     std::cout << "Starting services...\n";
601     database.start();
602     metrics.start();
```

```
603     std::this_thread::sleep_for(2s);
605
606     std::cout << "\n";
607     std::cout << "  Stopping NON-CORE service\n";
608     std::cout << "  (Metrics - uses stop_token)\n";
609     std::cout << "\n\n";
610
611     metrics.stop();
612
613     std::cout << "\n Metrics service stopped gracefully\n";
614     std::cout << "  Database service (core) still running\n";
615
616     std::this_thread::sleep_for(1s);
617
618     std::cout << "\nStopping all services...\n";
619     database.stop();
620
621     std::cout << "\n All services stopped\n";
622 }
623
624 } // namespace microservices_example
625
626 // =====
627 // SECTION 7: Summary and Best Practices
628 // =====
629
630 void show_summary() {
631     std::cout << "\n" << std::string(70, '=') << "\n";
632     std::cout << "==== Summary: stop_token for Non-Core Threads ===\n";
633     std::cout << std::string(70, '=') << "\n\n";
634
635     std::cout << "                                     \n";
636     std::cout << "  THREAD TYPE           STOP MECHANISM\n";
637     std::cout << "                                     \n";
638     std::cout << "  Core (Critical)      abort() on exception\n";
639     std::cout << "                                     \n";
640     std::cout << "                                     std::thread + atomic<bool>\n";
641     std::cout << "                                     \n";
642     std::cout << "                                     Ensures entire process stops\n";
643     std::cout << "                                     \n";
644     std::cout << "  Non-Core               std::jthread + stop_token\n";
645     std::cout << "                                     \n";
646     std::cout << "  (REST, Monitor)       Graceful shutdown\n";
647     std::cout << "                                     \n";
648     std::cout << "                                     Core services continue running\n";
649     std::cout << "                                     \n\n";
```

```

646
647     std::cout << "WHY stop_token FOR NON-CORE THREADS:\n";
648     std::cout << "    Type-safe cancellation signal\n";
649     std::cout << "    Standard C++20 interface\n";
650     std::cout << "    Works with condition_variable_any for immediate
651         interruption\n";
652     std::cout << "    std::jthread automatically joins (RAII)\n";
653     std::cout << "    stop_callback for automatic cleanup\n";
654     std::cout << "    Cannot accidentally set wrong value (unlike bool)\n\n";
655
656     std::cout << "COMPARISON: atomic<bool> vs stop_token\n\n";
657
658     std::cout << "    atomic<bool> approach:\n";
659     std::cout << "        while (!stop_flag.load()) {\n";
660     std::cout << "            cv.wait_for(lock, 100ms, ...); // Needs timeout\n";
661     std::cout << "            // Check flag regularly\n";
662     std::cout << "        }\n";
663     std::cout << "        // Manual join() required\n\n";
664
665     std::cout << "    stop_token approach:\n";
666     std::cout << "        std::jthread thread([](std::stop_token stoken) {\n";
667     std::cout << "            while (!stoken.stop_requested()) {\n";
668     std::cout << "                cv.wait(lock, stoken, ...); // Immediate interrupt
669         !\n";
670     std::cout << "            }\n";
671     std::cout << "        }\n";
672     std::cout << "        thread.request_stop(); // Automatic join on destruction
673         \n\n";
674
675     std::cout << "BEST PRACTICES:\n";
676     std::cout << " 1. Use stop_token for all non-critical threads\n";
677     std::cout << " 2. Use std::jthread (not std::thread) for automatic
678         joining\n";
679     std::cout << " 3. Use condition_variable_any (not condition_variable)\n";
680     std::cout << " 4. Check stop_requested() in loops\n";
681     std::cout << " 5. Use stop_callback for cleanup logic\n";
682     std::cout << " 6. Core services: use abort() for critical failures\n";
683     std::cout << " 7. Non-core services: graceful shutdown with stop_token\n";
684
685
686 }
687
688 // =====
689 // MAIN FUNCTION
690 // =====

```

```
691
692 int main() {
693     std::cout << "\n";
694     std::cout << "                                         \n";
695     std::cout << "             std::stop_token for Non-Core Thread Cancellation\n";
696     std::cout << "                                         \n";
697     std::cout << "                                         C++20 Cooperative Cancellation\n";
698     std::cout << "                                         \n";
699
700     // Section 1: Old vs New
701     old_vs_new::demonstrate();
702
703     // Section 2: REST Service
704     rest_service_example::demonstrate();
705
706     // Section 3: Monitoring Service
707     monitor_service_example::demonstrate();
708
709     // Section 4: Thread Pool
710     thread_pool_example::demonstrate();
711
712     // Section 5: stop_callback
713     stop_callback_example::demonstrate();
714
715     // Section 6: Complete Example
716     microservices_example::demonstrate();
717
718     // Section 7: Summary
719     show_summary();
720
721     std::cout << "\n" << std::string(70, '=') << "\n";
722     std::cout << "All demonstrations completed!\n";
723     std::cout << std::string(70, '=') << "\n\n";
724
725     std::cout << "KEY TAKEAWAY:\n";
726     std::cout << "Use std::stop_token with std::jthread for graceful
727             cancellation\n";
728     std::cout << "of non-critical threads (REST, monitoring, metrics).\n";
729     std::cout << "Core threads should still use abort() for critical failures
730             .\n\n";
731
732     return 0;
733 }
```

## 68 Source Code: StructuredBindings.cpp

File: src/StructuredBindings.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <map>
5 #include <tuple>
6 #include <array>
7 #include <set>
8 #include <utility>
9 #include <algorithm>
10
11 // =====
12 // 1. BASIC STRUCTURED BINDINGS WITH PAIR
13 // =====
14 void example_basic_structured_bindings() {
15     std::cout << "\n==== 1. BASIC STRUCTURED BINDINGS WITH PAIR ===" << std::endl;
16
17     std::pair<int, std::string> person{25, "Alice"};
18
19     // Old way (C++11/14)
20     int age_old = person.first;
21     std::string name_old = person.second;
22     std::cout << "Old way: " << name_old << " is " << age_old << " years old"
23         << std::endl;
24
25     // New way with structured bindings (C++17)
26     auto [age, name] = person;
27     std::cout << "Structured binding: " << name << " is " << age << " years
28         old" << std::endl;
29 }
30
31 // =====
32 // 2. STRUCTURED BINDINGS WITH TUPLE
33 // =====
34 void example_tuple_bindings() {
35     std::cout << "\n==== 2. STRUCTURED BINDINGS WITH TUPLE ===" << std::endl;
36
37     std::tuple<int, double, std::string, bool> data{42, 3.14, "Hello", true};
38
39     // Structured binding with tuple
40     auto [num, pi, text, flag] = data;
41
42     std::cout << "Number: " << num << std::endl;
43     std::cout << "Pi: " << pi << std::endl;
44     std::cout << "Text: " << text << std::endl;
45     std::cout << "Flag: " << std::boolalpha << flag << std::endl;
46 }
```

```
47 // 3. STRUCTURED BINDINGS WITH ARRAYS
48 // =====
49 void example_array_bindings() {
50     std::cout << "\n==== 3. STRUCTURED BINDINGS WITH ARRAYS ===" << std::endl;
51
52     // C-array
53     int c_array[3] = {10, 20, 30};
54     auto [a, b, c] = c_array;
55     std::cout << "C-array elements: " << a << ", " << b << ", " << c << std::
56         endl;
57
58     // std::array
59     std::array<std::string, 3> std_array = {"Red", "Green", "Blue"};
60     auto [color1, color2, color3] = std_array;
61     std::cout << "Colors: " << color1 << ", " << color2 << ", " << color3 <<
62         std::endl;
63 }
64
65 // =====
66 // 4. STRUCTURED BINDINGS WITH STRUCTS
67 // =====
68 struct Point {
69     int x;
70     int y;
71     int z;
72 };
73
74 struct Employee {
75     std::string name;
76     int id;
77     double salary;
78 };
79
80 void example_struct_bindings() {
81     std::cout << "\n==== 4. STRUCTURED BINDINGS WITH STRUCTS ===" << std::endl;
82
83     Point p{10, 20, 30};
84     auto [x, y, z] = p;
85     std::cout << "Point coordinates: x=" << x << ", y=" << y << ", z=" << z <<
86         std::endl;
87
88     Employee emp{"John Doe", 12345, 75000.50};
89     auto [emp_name, emp_id, emp_salary] = emp;
90     std::cout << "Employee: " << emp_name << " (ID: " << emp_id
91         << ", Salary: $" << emp_salary << ")" << std::endl;
92 }
93
94 // =====
95 // 5. STRUCTURED BINDINGS WITH MAP ITERATION
96 // =====
97 void example_map_iteration() {
98     std::cout << "\n==== 5. STRUCTURED BINDINGS WITH MAP ITERATION ===" << std
99         ::endl;
100 }
```

```
97     std::map<std::string, int> age_map = {  
98         {"Alice", 30},  
99         {"Bob", 25},  
100        {"Charlie", 35}  
101    };  
102  
103    // Old way  
104    std::cout << "Old way:" << std::endl;  
105    for (const auto& pair : age_map) {  
106        std::cout << " " << pair.first << " is " << pair.second << " years  
107        old" << std::endl;  
108    }  
109  
110    // New way with structured bindings  
111    std::cout << "With structured bindings:" << std::endl;  
112    for (const auto& [name, age] : age_map) {  
113        std::cout << " " << name << " is " << age << " years old" << std::  
114        endl;  
115    }  
116  
117 // =====  
118 // 6. STRUCTURED BINDINGS WITH REFERENCES  
119 // =====  
120 void example_reference_bindings() {  
121     std::cout << "\n==== 6. STRUCTURED BINDINGS WITH REFERENCES ===" << std::  
122         endl;  
123  
124     std::pair<int, std::string> data{100, "Original"};  
125  
126     // Non-reference binding (copy)  
127     auto [val1, str1] = data;  
128     val1 = 200;  
129     str1 = "Modified Copy";  
130     std::cout << "Original after copy modification: "  
131         << data.first << ", " << data.second << std::endl;  
132  
133     // Reference binding  
134     auto& [val2, str2] = data;  
135     val2 = 300;  
136     str2 = "Modified Reference";  
137     std::cout << "Original after reference modification: "  
138         << data.first << ", " << data.second << std::endl;  
139  
140 // =====  
141 // 7. STRUCTURED BINDINGS WITH CONST  
142 // =====  
143 void example_const_bindings() {  
144     std::cout << "\n==== 7. STRUCTURED BINDINGS WITH CONST ===" << std::endl;  
145  
146     const std::pair<int, std::string> data{42, "Constant"};  
147  
148     // Const structured binding
```

```
148     const auto& [value, text] = data;
149     std::cout << "Const binding: " << value << ", " << text << std::endl;
150
151     // value = 100; // ERROR: cannot modify const
152 }
153
154 // =====
155 // 8. STRUCTURED BINDINGS WITH FUNCTION RETURNS
156 // =====
157 std::tuple<int, double, std::string> get_data() {
158     return {42, 3.14159, "Function Return"};
159 }
160
161 std::pair<bool, std::string> validate_input(int value) {
162     if (value > 0) {
163         return {true, "Valid"};
164     } else {
165         return {false, "Invalid: must be positive"};
166     }
167 }
168
169 void example_function_return_bindings() {
170     std::cout << "\n==== 8. STRUCTURED BINDINGS WITH FUNCTION RETURNS ===" <<
171         std::endl;
172
173     auto [num, pi, message] = get_data();
174     std::cout << "From function: " << num << ", " << pi << ", " << message <<
175         std::endl;
176
177     auto [is_valid, error_msg] = validate_input(10);
178     std::cout << "Validation: " << std::boolalpha << is_valid << " - " <<
179         error_msg << std::endl;
180 }
181
182 // =====
183 // 9. STRUCTURED BINDINGS WITH NESTED STRUCTURES
184 // =====
185 struct Address {
186     std::string street;
187     std::string city;
188     int zipcode;
189 };
190
191 struct Person {
192     std::string name;
193     int age;
194     Address address;
195 };
196
197 void example_nested_bindings() {
```

```
198     std::cout << "\n==== 9. STRUCTURED BINDINGS WITH NESTED STRUCTURES ===" <<
199         std::endl;
200
200     Person person{"John Smith", 30, {"123 Main St", "Springfield", 12345}};
201
202     auto [name, age, address] = person;
203     auto [street, city, zipcode] = address;
204
205     std::cout << "Person: " << name << ", Age: " << age << std::endl;
206     std::cout << "Address: " << street << ", " << city << " " << zipcode <<
207         std::endl;
208 }
209
210 // =====
211 // 10. STRUCTURED BINDINGS WITH STL ALGORITHMS
212 // =====
213 void example_algorithms_with_bindings() {
214     std::cout << "\n==== 10. STRUCTURED BINDINGS WITH STL ALGORITHMS ===" <<
215         std::endl;
216
217     std::vector<std::pair<std::string, int>> scores = {
218         {"Alice", 95},
219         {"Bob", 87},
220         {"Charlie", 92},
221         {"David", 88}
222     };
223
224     // Find student with highest score
225     auto max_it = std::max_element(scores.begin(), scores.end(),
226         [] (const auto& a, const auto& b) {
227             auto [name1, score1] = a;
228             auto [name2, score2] = b;
229             return score1 < score2;
230         });
231
232     if (max_it != scores.end()) {
233         auto [top_student, top_score] = *max_it;
234         std::cout << "Top student: " << top_student << " with score " <<
235             top_score << std::endl;
236     }
237
238     // Print all with structured bindings
239     std::cout << "All scores:" << std::endl;
240     std::for_each(scores.begin(), scores.end(),
241         [] (const auto& entry) {
242             auto [name, score] = entry;
243             std::cout << " " << name << ":" << score << std::endl;
244         });
245 }
246
247 // =====
248 // 11. STRUCTURED BINDINGS WITH INSERT OPERATIONS
249 // =====
250 void example_insert_bindings() {
```

```
248     std::cout << "\n==== 11. STRUCTURED BINDINGS WITH INSERT OPERATIONS ===" <<
249         std::endl;
250
250     std::map<int, std::string> id_map;
251
252     // Insert returns pair<iterator, bool>
253     auto [it1, inserted1] = id_map.insert({1, "First"});
254     std::cout << "Insert 1: " << (inserted1 ? "Success" : "Failed") << std::endl;
255
256     auto [it2, inserted2] = id_map.insert({1, "Duplicate"});
257     std::cout << "Insert duplicate 1: " << (inserted2 ? "Success" : "Failed")
258         << std::endl;
259
260     auto [it3, inserted3] = id_map.insert({2, "Second"});
261     std::cout << "Insert 2: " << (inserted3 ? "Success" : "Failed") << std::endl;
262
263     std::cout << "Map contents:" << std::endl;
264     for (const auto& [id, value] : id_map) {
265         std::cout << " " << id << ":" << value << std::endl;
266     }
267
268 // =====
269 // 12. STRUCTURED BINDINGS WITH MULTIPLE RETURNS
270 // =====
271 struct Stats {
272     double mean;
273     double median;
274     double stddev;
275 };
276
277 Stats calculate_stats(const std::vector<int>& data) {
278     double sum = 0;
279     for (int val : data) sum += val;
280     double mean = sum / data.size();
281
282     return {mean, static_cast<double>(data[data.size()/2]), 10.5}; // simplified
283 }
284
285 void example_multiple_return_values() {
286     std::cout << "\n==== 12. STRUCTURED BINDINGS WITH MULTIPLE RETURNS ===" <<
287         std::endl;
288
289     std::vector<int> dataset = {10, 20, 30, 40, 50};
290
291     auto [mean, median, stddev] = calculate_stats(dataset);
292
293     std::cout << "Statistics:" << std::endl;
294     std::cout << " Mean: " << mean << std::endl;
295     std::cout << " Median: " << median << std::endl;
296     std::cout << " Std Dev: " << stddev << std::endl;
```

```
296 }
297
298 // =====
299 // MAIN FUNCTION
300 // =====
301 int main() {
302     std::cout << "\n=====" << std::
303         endl;
304     std::cout << "      C++17 STRUCTURED BINDINGS EXAMPLES" << std::endl;
305     std::cout << "=====" << std::
306         endl;
307     example_basic_structured_bindings();
308     example_tuple_bindings();
309     example_array_bindings();
310     example_struct_bindings();
311     example_map_iteration();
312     example_reference_bindings();
313     example_const_bindings();
314     example_function_return_bindings();
315     example_nested_bindings();
316     example_algorithms_with_bindings();
317     example_insert_bindings();
318     example_multiple_return_values();
319
320     std::cout << "\n=====" << std::
321         endl;
322     std::cout << "      ALL EXAMPLES COMPLETED" << std::endl;
323     std::cout << "=====\n" << std::
324         endl;
325
326     return 0;
327 }
```

## 69 Source Code: SystemInteractionAndParsing.cpp

File: src/SystemInteractionAndParsing.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // SYSTEM INTERACTION, FILESYSTEM, STRING_VIEW, AND REGEX
3 // =====
4 // This example demonstrates modern C++ for system programming:
5 //
6 // TOPICS COVERED:
7 // 1. Executing external commands (popen, system)
8 // 2. std::filesystem for file operations (C++17)
9 // 3. std::string_view for efficient parsing (C++17)
10 // 4. std::regex for pattern matching (C++11)
11 // 5. Parsing Linux tool output (ps, df, lsof, etc.)
12 // 6. Running bash and Python scripts from C++
13 // 7. File streams (ifstream, ofstream, fstream)
14 // 8. String streams (ostringstream, istringstream)
15 // 9. Stream manipulators and formatting
16 // 10. Binary I/O and stream state handling
17 // 11. Error handling with std::error_code
18 // 12. Thread synchronization (mutex, lock_guard, unique_lock)
19 // 13. File locking for concurrent access
20 // 14. Best practices for system integration
21 // 15. Security considerations
22 // 16. Cross-platform considerations
23 // =====
24
25 #include <iostream>
26 #include <iomanip>
27 #include <string>
28 #include <string_view>
29 #include <vector>
30 #include <map>
31 #include <regex>
32 #include <filesystem>
33 #include <fstream>
34 #include <sstream>
35 #include <array>
36 #include <memory>
37 #include <algorithm>
38 #include <cstdio>
39 #include <cstdlib>
40 #include <stdexcept>
41 #include <chrono>
42 #include <ctime>
43 #include <mutex>
44 #include <shared_mutex>
45 #include <thread>
46 #include <condition_variable>
47 #include <atomic>
48
49 namespace fs = std::filesystem;
```

```
50 // =====
51 // SECTION 1: BASIC COMMAND EXECUTION
52 // =====
53
54
55 // Execute command and capture output
56 std::string execute_command(std::string_view command) {
57     std::array<char, 128> buffer;
58     std::string result;
59
60     // popen is safer than system() - doesn't invoke shell if not needed
61     std::unique_ptr<FILE, decltype(&pclose)> pipe(
62         popen(command.data(), "r"), pclose);
63
64     if (!pipe) {
65         throw std::runtime_error("popen() failed!");
66     }
67
68     while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
69         result += buffer.data();
70     }
71
72     return result;
73 }
74
75 // Execute command with return code
76 struct CommandResult {
77     std::string output;
78     int return_code;
79     bool success;
80 };
81
82 CommandResult execute_command_with_status(std::string_view command) {
83     CommandResult result;
84
85     std::array<char, 256> buffer;
86     std::string output;
87
88     FILE* pipe = popen(command.data(), "r");
89     if (!pipe) {
90         return {.output = "", .return_code = -1, .success = false};
91     }
92
93     while (fgets(buffer.data(), buffer.size(), pipe) != nullptr) {
94         output += buffer.data();
95     }
96
97     int status = pclose(pipe);
98     result.output = output;
99     result.return_code = WEXITSTATUS(status);
100    result.success = (result.return_code == 0);
101
102    return result;
103 }
```

```
104 void demonstrate_basic_execution() {
105     std::cout << "\n==== 1. BASIC COMMAND EXECUTION ===" << std::endl;
106
107     // Simple command
108     std::cout << "\n1.1 Current directory:" << std::endl;
109     std::string pwd = execute_command("pwd");
110     std::cout << "    " << pwd;
111
112     // Command with pipes
113     std::cout << "1.2 Count files in current directory:" << std::endl;
114     std::string count = execute_command("ls -1 | wc -l");
115     std::cout << "    Files: " << count;
116
117     // With status check
118     std::cout << "1.3 Execute with status check:" << std::endl;
119     auto result = execute_command_with_status("echo 'Hello from shell'");
120     std::cout << "    Output: " << result.output;
121     std::cout << "    Return code: " << result.return_code << std::endl;
122     std::cout << "    Success: " << std::boolalpha << result.success << std::endl;
123
124     std::cout << "\n KEY POINTS:" << std::endl;
125     std::cout << " • Use popen() to capture output" << std::endl;
126     std::cout << " • Check return codes for errors" << std::endl;
127     std::cout << " • pclose() is critical - use RAI" << std::endl;
128
129 }
130
131 // =====
132 // SECTION 2: std::string_view FOR EFFICIENT PARSING
133 // =====
134
135 // Parse process info using string_view (no copies!)
136 struct ProcessInfo {
137     std::string pid;
138     std::string user;
139     std::string cpu;
140     std::string mem;
141     std::string command;
142 };
143
144 std::vector<std::string_view> split_string_view(std::string_view str, char delimiter) {
145     std::vector<std::string_view> tokens;
146     size_t start = 0;
147
148     while (start < str.size()) {
149         // Skip leading delimiters
150         while (start < str.size() && str[start] == delimiter) {
151             ++start;
152         }
153
154         if (start >= str.size()) break;
155
156         size_t end = start;
157         while (end < str.size() && str[end] != delimiter) {
158             ++end;
159         }
160
161         tokens.push_back(str.substr(start, end - start));
162         start = end;
163     }
164
165     return tokens;
166 }
```

```
156     // Find end of token
157     size_t end = str.find(delimiter, start);
158     if (end == std::string_view::npos) {
159         end = str.size();
160     }
161
162     tokens.push_back(str.substr(start, end - start));
163     start = end + 1;
164 }
165
166 return tokens;
167 }
168
169 // Efficient parsing with string_view - no memory allocations for substrings!
170 std::vector<ProcessInfo> parse_ps_output(std::string_view output) {
171     std::vector<ProcessInfo> processes;
172
173     size_t line_start = 0;
174     size_t line_end = output.find('\n');
175
176     // Skip header line
177     if (line_end != std::string_view::npos) {
178         line_start = line_end + 1;
179     }
180
181     while (line_start < output.size()) {
182         line_end = output.find('\n', line_start);
183         if (line_end == std::string_view::npos) {
184             line_end = output.size();
185         }
186
187         std::string_view line = output.substr(line_start, line_end -
188             line_start);
188         if (line.empty()) {
189             line_start = line_end + 1;
190             continue;
191         }
192
193         auto tokens = split_string_view(line, ' ');
194         if (tokens.size() >= 5) {
195             ProcessInfo info;
196             info.pid = std::string(tokens[0]);
197             info.user = std::string(tokens[1]);
198             info.cpu = std::string(tokens[2]);
199             info.mem = std::string(tokens[3]);
200
201             // Command is rest of line - find position of 5th token in
202             // original output
202             size_t token4_pos = line.find(tokens[4]);
203             if (token4_pos != std::string_view::npos) {
204                 size_t cmd_offset = line_start + token4_pos;
205                 info.command = std::string(output.substr(cmd_offset, line_end
206                     - cmd_offset));
206             }
207 }
```

```
207         processes.push_back(info);
208     }
209
210     line_start = line_end + 1;
211 }
212
213
214     return processes;
215 }
216
217 void demonstrate_string_view_parsing() {
218     std::cout << "\n== 2. std::string_view PARSING ==" << std::endl;
219
220     // Get process list
221     std::string ps_output = execute_command("ps aux | head -20");
222
223     std::cout << "\n2.1 Parsing ps output with string_view (zero-copy):" <<
224         std::endl;
225     auto processes = parse_ps_output(ps_output);
226
227     std::cout << "    Found " << processes.size() << " processes" << std::endl;
228     std::cout << "\n    Top 5 by CPU:" << std::endl;
229
230     // Sort by CPU
231     std::sort(processes.begin(), processes.end(),
232               [] (const ProcessInfo& a, const ProcessInfo& b) {
233                   return std::stod(a.cpu) > std::stod(b.cpu);
234               });
235
236     for (size_t i = 0; i < std::min(size_t(5), processes.size()); i++) {
237         std::cout << "    " << processes[i].pid << " "
238             << processes[i].cpu << "% CPU "
239             << processes[i].command.substr(0, 50) << std::endl;
240     }
241
242     std::cout << "\n    string_view benefits:" << std::endl;
243     std::cout << "    •    No memory allocations during parsing" << std::endl;
244     std::cout << "    •    Fast substring operations" << std::endl;
245     std::cout << "    •    Perfect for tokenizing large outputs" << std::endl;
246 }
247
248 // =====
249 // SECTION 3: std::regex FOR PATTERN MATCHING
250 // =====
251
252 // Parse network interface info with regex
253 struct NetworkInterface {
254     std::string name;
255     std::string ip_address;
256     std::string netmask;
257     std::string status;
258 };
259 std::vector<NetworkInterface> parse_ifconfig_with_regex(std::string_view
```

```
260     output) {
261         std::vector<NetworkInterface> interfaces;
262
263         // Regex patterns
264         std::regex iface_pattern(R"((\w+):\s+flags=.+)");
265         std::regex inet_pattern(R"(inet\s+(\d+\.\d+\.\d+\.\d+)\s+netmask\s+(\S+))");
266         std::regex status_pattern(R"(status:\s+(\w+))");
267
268         std::string output_str(output);
269         std::istringstream iss(output_str);
270         std::string line;
271
272         NetworkInterface current_iface;
273         bool has_current = false;
274
275         while (std::getline(iss, line)) {
276             std::smatch match;
277
278             // Check for interface name
279             if (std::regex_search(line, match, iface_pattern)) {
280                 // Save previous interface
281                 if (has_current && !current_iface.ip_address.empty()) {
282                     interfaces.push_back(current_iface);
283                 }
284
285                 current_iface = NetworkInterface{};
286                 current_iface.name = match[1].str();
287                 has_current = true;
288             }
289             // Check for IP address
290             else if (has_current && std::regex_search(line, match, inet_pattern))
291             {
292                 current_iface.ip_address = match[1].str();
293                 current_iface.netmask = match[2].str();
294             }
295             // Check for status
296             else if (has_current && std::regex_search(line, match, status_pattern))
297             {
298                 current_iface.status = match[1].str();
299             }
300
301             // Don't forget last interface
302             if (has_current && !current_iface.ip_address.empty()) {
303                 interfaces.push_back(current_iface);
304             }
305
306             return interfaces;
307         }
308
309         // Parse log files with regex
310         struct LogEntry {
```

```
310     std::string timestamp;
311     std::string level;
312     std::string message;
313 };
314
315 std::vector<LogEntry> parse_log_with_regex(std::string_view log_content) {
316     std::vector<LogEntry> entries;
317
318     // Match: [2024-01-15 10:30:45] ERROR: Something went wrong
319     std::regex log_pattern(R"(\[\d{4}-\d{2}-\d{2}\s+\d{2}:\d{2}:\d{2})\]\s+(\w+):\s+(.+))");
320
321     std::string log_str(log_content);
322     std::istringstream iss(log_str);
323     std::string line;
324
325     while (std::getline(iss, line)) {
326         std::smatch match;
327         if (std::regex_search(line, match, log_pattern)) {
328             entries.push_back({
329                 .timestamp = match[1].str(),
330                 .level = match[2].str(),
331                 .message = match[3].str()
332             });
333         }
334     }
335
336     return entries;
337 }
338
339 void demonstrate_regex_parsing() {
340     std::cout << "\n==== 3. std::regex PATTERN MATCHING ===" << std::endl;
341
342     // Network interface parsing
343     std::cout << "\n3.1 Parsing network interfaces with regex:" << std::endl;
344     std::string ifconfig_output = execute_command("ifconfig 2>/dev/null || ip
345         addr 2>/dev/null || echo 'No network tools available'");
346
347     auto interfaces = parse_ifconfig_with_regex(ifconfig_output);
348     for (const auto& iface : interfaces) {
349         std::cout << "    Interface: " << iface.name << std::endl;
350         std::cout << "        IP: " << iface.ip_address << std::endl;
351         std::cout << "        Netmask: " << iface.netmask << std::endl;
352         if (!iface.status.empty()) {
353             std::cout << "        Status: " << iface.status << std::endl;
354         }
355     }
356
357     // Create sample log for parsing
358     std::cout << "\n3.2 Parsing log files with regex:" << std::endl;
359     std::string sample_log =
360         "[2024-01-15 10:30:45] INFO: Application started\n"
361         "[2024-01-15 10:30:46] DEBUG: Loading configuration\n"
362         "[2024-01-15 10:30:47] ERROR: Database connection failed\n"
```

```
362     " [2024-01-15 10:30:48] WARN: Retrying connection\n"
363     " [2024-01-15 10:30:50] INFO: Connection established\n";
364
365     auto log_entries = parse_log_with_regex(sample_log);
366
367     std::cout << "    Found " << log_entries.size() << " log entries:" << std::
368     endl;
369     for (const auto& entry : log_entries) {
370         std::cout << "    [ " << entry.timestamp << " ] "
371             << entry.level << ": " << entry.message << std::endl;
372     }
373
374     // Filter errors
375     std::cout << "\n    Errors only:" << std::endl;
376     for (const auto& entry : log_entries) {
377         if (entry.level == "ERROR") {
378             std::cout << "        " << entry.message << std::endl;
379         }
380     }
381
382     std::cout << "\n    regex use cases:" << std::endl;
383     std::cout << "    • Structured text parsing (logs, config)" << std::endl;
384     std::cout << "    • Validation (emails, IPs, dates)" << std::endl;
385     std::cout << "    • Extraction from unstructured output" << std::endl;
386 }
387 // =====
388 // SECTION 4: std::filesystem OPERATIONS
389 // =====
390
391 void demonstrate_filesystem() {
392     std::cout << "\n==== 4. std::filesystem OPERATIONS ===" << std::endl;
393
394     std::error_code ec;
395
396     // Current path
397     std::cout << "\n4.1 Current path:" << std::endl;
398     fs::path current = fs::current_path(ec);
399     std::cout << "    " << current << std::endl;
400
401     // Create temporary test directory
402     fs::path test_dir = fs::temp_directory_path(ec) / "cpp_test_dir";
403     std::cout << "\n4.2 Creating test directory:" << std::endl;
404     std::cout << "    " << test_dir << std::endl;
405
406     if (fs::create_directories(test_dir, ec)) {
407         std::cout << "        Created" << std::endl;
408     } else {
409         std::cout << "        Already exists or error: " << ec.message() << std
410             ::endl;
411     }
412
413     // Create test files
414     std::cout << "\n4.3 Creating test files:" << std::endl;
```

```

414     for (int i = 0; i < 3; i++) {
415         fs::path file = test_dir / ("test_file_" + std::to_string(i) + ".txt")
416         ;
417         std::ofstream ofs(file);
418         ofs << "Test content " << i << "\n";
419         ofs << "Line 2 of file " << i << "\n";
420         std::cout << "    Created: " << file.filename() << std::endl;
421     }
422
423     // Iterate directory
424     std::cout << "\n4.4 Directory contents:" << std::endl;
425     for (const auto& entry : fs::directory_iterator(test_dir, ec)) {
426         std::cout << "    " << entry.path().filename()
427             << " (" << fs::file_size(entry, ec) << " bytes)" << std::
428             endl;
429     }
430
431     // File info
432     std::cout << "\n4.5 File properties:" << std::endl;
433     auto first_file = test_dir / "test_file_0.txt";
434     if (fs::exists(first_file, ec)) {
435         std::cout << "    Size: " << fs::file_size(first_file, ec) << " bytes"
436             << std::endl;
437         std::cout << "    Is regular file: " << fs::is_regular_file(first_file,
438             ec) << std::endl;
439         std::cout << "    Is directory: " << fs::is_directory(first_file, ec)
440             << std::endl;
441
442         auto ftime = fs::last_write_time(first_file, ec);
443         std::cout << "    Last modified: " << std::chrono::system_clock::
444             to_time_t(
445                 std::chrono::file_clock::to_sys(ftime)) << " (unix timestamp)" <<
446                 std::endl;
447     }
448
449     // Path operations
450     std::cout << "\n4.6 Path operations:" << std::endl;
451     fs::path example = "/home/user/documents/file.txt";
452     std::cout << "    Full path: " << example << std::endl;
453     std::cout << "    Filename: " << example.filename() << std::endl;
454     std::cout << "    Stem: " << example.stem() << std::endl;
455     std::cout << "    Extension: " << example.extension() << std::endl;
456     std::cout << "    Parent: " << example.parent_path() << std::endl;
457
458     // Cleanup
459     std::cout << "\n4.7 Cleanup:" << std::endl;
460     std::uintmax_t removed = fs::remove_all(test_dir, ec);
461     std::cout << "    Removed " << removed << " files/directories" << std::endl
462         ;
463
464     std::cout << "\n    filesystem features:" << std::endl;
465     std::cout << "    •    Cross-platform path handling" << std::endl;
466     std::cout << "    •    Iterator for directory traversal" << std::endl;
467     std::cout << "    •    Error codes for exception-free errors" << std::endl;

```

```
460 }
461
462 // =====
463 // SECTION 5: RUNNING BASH SCRIPTS
464 // =====
465
466 void create_test_bash_script(const fs::path& script_path) {
467     std::ofstream script(script_path);
468     script << "#!/bin/bash\n";
469     script << "# Test bash script called from C++\n\n";
470     script << "echo \"Script started with $# arguments\"\n";
471     script << "echo \"Arguments: $@\n\n";
472     script << "# Process arguments\n";
473     script << "for arg in \"$@\"; do\n";
474     script << "    echo \" - Processing: $arg\"\n";
475     script << "done\n\n";
476     script << "# Return some data\n";
477     script << "echo \"Hostname: $(hostname)\"\n";
478     script << "echo \"User: $(whoami)\"\n";
479     script << "echo \"Date: $(date +%Y-%m-%d)\"\n";
480     script << "\n";
481     script << "exit 0\n";
482
483     fs::permissions(script_path,
484                     fs::perms::owner_all | fs::perms::group_read | fs::perms::
485                     others_read,
486                     fs::perm_options::add);
487 }
488
489 void demonstrate_bash_scripts() {
490     std::cout << "\n== 5. RUNNING BASH SCRIPTS ==" << std::endl;
491
492     std::error_code ec;
493     fs::path temp_dir = fs::temp_directory_path(ec);
494     fs::path script_path = temp_dir / "test_script.sh";
495
496     std::cout << "\n5.1 Creating bash script:" << std::endl;
497     std::cout << "    " << script_path << std::endl;
498
499     create_test_bash_script(script_path);
500
501     // Execute script with arguments
502     std::cout << "\n5.2 Executing script with arguments:" << std::endl;
503     std::string command = "bash " + script_path.string() + " arg1 arg2 arg3";
504     std::string output = execute_command(command);
505     std::cout << output;
506
507     // Parse script output with string_view
508     std::cout << "\n5.3 Parsing script output:" << std::endl;
509     std::regex hostname_pattern(R"(Hostname:\s+(\S+))");
510     std::regex user_pattern(R"(User:\s+(\S+))");
511     std::regex date_pattern(R"(Date:\s+(\S+))");
512
513     std::smatch match;
```

```
513     std::string output_str(output);
514
515     if (std::regex_search(output_str, match, hostname_pattern)) {
516         std::cout << "    Hostname: " << match[1].str() << std::endl;
517     }
518     if (std::regex_search(output_str, match, user_pattern)) {
519         std::cout << "    User: " << match[1].str() << std::endl;
520     }
521     if (std::regex_search(output_str, match, date_pattern)) {
522         std::cout << "    Date: " << match[1].str() << std::endl;
523     }
524
525     // Cleanup
526     fs::remove(script_path, ec);
527
528     std::cout << "\n Script execution:" << std::endl;
529     std::cout << " • Use bash explicitly for portability" << std::endl;
530     std::cout << " • Pass arguments safely (avoid injection)" << std::endl;
531     std::cout << " • Parse structured output with regex" << std::endl;
532 }
533
534 // =====
535 // SECTION 6: RUNNING PYTHON SCRIPTS
536 // =====
537
538 void create_test_python_script(const fs::path& script_path) {
539     std::ofstream script(script_path);
540     script << "#!/usr/bin/env python3\n";
541     script << "import sys\n";
542     script << "import json\n";
543     script << "import platform\n\n";
544     script << "def main():\n";
545     script << "    # Get system info\n";
546     script << "    info = {\n";
547     script << "        'python_version': sys.version.split()[0],\n";
548     script << "        'platform': platform.system(),\n";
549     script << "        'architecture': platform.machine(),\n";
550     script << "        'arguments': sys.argv[1:]\n";
551     script << "    }\n";
552     script << "    \n";
553     script << "    # Output as JSON for easy parsing\n";
554     script << "    print(json.dumps(info, indent=2))\n\n";
555     script << "if __name__ == '__main__':\n";
556     script << "    main()\n";
557
558     fs::permissions(script_path,
559                     fs::perms::owner_all | fs::perms::group_read | fs::perms::
560                     others_read,
561                     fs::perm_options::add);
562 }
563
564 void demonstrate_python_scripts() {
565     std::cout << "\n==== 6. RUNNING PYTHON SCRIPTS ===" << std::endl;
566 }
```

```

566     std::error_code ec;
567     fs::path temp_dir = fs::temp_directory_path(ec);
568     fs::path script_path = temp_dir / "test_script.py";
569
570     std::cout << "\n6.1 Creating Python script:" << std::endl;
571     std::cout << "    " << script_path << std::endl;
572
573     create_test_python_script(script_path);
574
575     // Execute Python script
576     std::cout << "\n6.2 Executing Python script:" << std::endl;
577     std::string command = "python3 " + script_path.string() + " data1 data2";
578     auto result = execute_command_with_status(command);
579
580     if (result.success) {
581         std::cout << "    Script executed successfully" << std::endl;
582         std::cout << "\n6.3 JSON Output:\n" << result.output << std::endl;
583
584         // Parse JSON output with regex (simple approach)
585         std::cout << "6.4 Extracting data:" << std::endl;
586         std::regex version_pattern("\"python_version\"\\s*:\\s*\"([^\"]+)\"");
587         std::regex platform_pattern("\"platform\"\\s*:\\s*\"([^\"]+)\"");
588         std::regex arch_pattern("\"architecture\"\\s*:\\s*\"([^\"]+)\"");
589
590         std::smatch match;
591         if (std::regex_search(result.output, match, version_pattern)) {
592             std::cout << "    Python version: " << match[1].str() << std::endl;
593         }
594         if (std::regex_search(result.output, match, platform_pattern)) {
595             std::cout << "    Platform: " << match[1].str() << std::endl;
596         }
597         if (std::regex_search(result.output, match, arch_pattern)) {
598             std::cout << "    Architecture: " << match[1].str() << std::endl;
599         }
600     } else {
601         std::cout << "    Script failed with code: " << result.return_code <<
602             std::endl;
603     }
604
605     // Cleanup
606     fs::remove(script_path, ec);
607
608     std::cout << "\n Python integration:" << std::endl;
609     std::cout << " • Use JSON for structured data exchange" << std::endl;
610     std::cout << " • Check return codes" << std::endl;
611     std::cout << " • Python for quick prototyping/data processing" << std::endl;
612
613 // =====
614 // SECTION 7: PARSING LINUX TOOL OUTPUT
615 // =====
616
617 struct DiskInfo {

```

```
618     std::string filesystem;
619     std::string size;
620     std::string used;
621     std::string available;
622     std::string use_percent;
623     std::string mounted_on;
624 };
625
626 std::vector<DiskInfo> parse_df_output(std::string_view output) {
627     std::vector<DiskInfo> disks;
628
629     size_t line_start = 0;
630     size_t line_end = output.find('\n');
631
632     // Skip header
633     if (line_end != std::string_view::npos) {
634         line_start = line_end + 1;
635     }
636
637     while (line_start < output.size()) {
638         line_end = output.find('\n', line_start);
639         if (line_end == std::string_view::npos) {
640             line_end = output.size();
641         }
642
643         std::string_view line = output.substr(line_start, line_end -
644                                         line_start);
644         if (line.empty()) {
645             line_start = line_end + 1;
646             continue;
647         }
648
649         auto tokens = split_string_view(line, ' ');
650         if (tokens.size() >= 6) {
651             DiskInfo info;
652             info.filesystem = std::string(tokens[0]);
653             info.size = std::string(tokens[1]);
654             info.used = std::string(tokens[2]);
655             info.available = std::string(tokens[3]);
656             info.use_percent = std::string(tokens[4]);
657             info.mounted_on = std::string(tokens[5]);
658             disks.push_back(info);
659         }
660
661         line_start = line_end + 1;
662     }
663
664     return disks;
665 }
666
667 void demonstrate_linux_tools() {
668     std::cout << "\n==== 7. PARSING LINUX TOOL OUTPUT ===" << std::endl;
669
670     // df command
```

```

671     std::cout << "\n7.1 Disk usage (df -h):" << std::endl;
672     std::string df_output = execute_command("df -h 2>/dev/null | head -10");
673     auto disks = parse_df_output(df_output);
674
675     for (const auto& disk : disks) {
676         if (disk.filesystem.find("/dev/") == 0) { // Real devices only
677             std::cout << "    " << disk.mounted_on
678             << ":" << disk.used << "/" << disk.size
679             << " (" << disk.use_percent << ")" << std::endl;
680         }
681     }
682
683     // Memory info
684     std::cout << "\n7.2 Memory usage:" << std::endl;
685     std::string mem_output = execute_command("free -h | grep Mem");
686     std::regex mem_pattern(R"(Mem:\s+(\S+)\s+(\S+)\s+(\S+))");
687     std::smatch match;
688     if (std::regex_search(mem_output, match, mem_pattern)) {
689         std::cout << "    Total: " << match[1].str() << std::endl;
690         std::cout << "    Used: " << match[2].str() << std::endl;
691         std::cout << "    Free: " << match[3].str() << std::endl;
692     }
693
694     // CPU info
695     std::cout << "\n7.3 CPU information:" << std::endl;
696     std::string cpu_output = execute_command("lscpu 2>/dev/null | grep -E '"
697                                         "Model name|CPU\\(s\\\\)|Thread'");
698
699     std::istringstream iss(cpu_output);
700     std::string line;
701     while (std::getline(iss, line)) {
702         if (!line.empty()) {
703             std::cout << "    " << line << std::endl;
704         }
705     }
706
707     std::cout << "\n Linux tools:" << std::endl;
708     std::cout << " • Rich ecosystem of system tools" << std::endl;
709     std::cout << " • Standardized output formats" << std::endl;
710     std::cout << " • Combine with grep/awk for filtering" << std::endl;
711 }
712 // =====
713 // SECTION 8: FILE STREAMS - ifstream, ofstream, fstream
714 // =====
715
716 void demonstrate_file_streams() {
717     std::cout << "\n== 8. FILE STREAMS (ifstream, ofstream, fstream) ==" <<
718         std::endl;
719
720     std::error_code ec;
721     fs::path temp_dir = fs::temp_directory_path(ec);
722     fs::path test_file = temp_dir / "stream_test.txt";

```

```
723 // 8.1 Writing with ofstream
724 std::cout << "\n8.1 Writing with ofstream:" << std::endl;
725 {
726     std::ofstream ofs(test_file);
727     if (!ofs) {
728         std::cout << "      Failed to open file for writing" << std::endl;
729         return;
730     }
731
732     ofs << "Line 1: Hello, World!\n";
733     ofs << "Line 2: Modern C++ Streams\n";
734     ofs << "Line 3: Numbers: " << 42 << ", " << 3.14159 << "\n";
735     ofs << "Line 4: Boolean: " << std::boolalpha << true << "\n";
736
737     std::cout << "      Wrote 4 lines to " << test_file.filename() << std::endl;
738 } // File automatically closed when ofs goes out of scope
739
740 // 8.2 Reading with ifstream
741 std::cout << "\n8.2 Reading with ifstream (line by line):" << std::endl;
742 {
743     std::ifstream ifs(test_file);
744     if (!ifs) {
745         std::cout << "      Failed to open file for reading" << std::endl;
746         return;
747     }
748
749     std::string line;
750     int line_num = 1;
751     while (std::getline(ifs, line)) {
752         std::cout << "      [" << line_num++ << "] " << line << std::endl;
753     }
754 }
755
756 // 8.3 Reading entire file
757 std::cout << "\n8.3 Reading entire file at once:" << std::endl;
758 {
759     std::ifstream ifs(test_file);
760     std::string content((std::istreambuf_iterator<char>(ifs)),
761                         std::istreambuf_iterator<char>());
762     std::cout << "      File size: " << content.size() << " bytes" << std::endl;
763     std::cout << "      First 50 chars: " << content.substr(0, 50) << "..." << std::endl;
764 }
765
766 // 8.4 Appending to file
767 std::cout << "\n8.4 Appending with std::ios::app:" << std::endl;
768 {
769     std::ofstream ofs(test_file, std::ios::app); // Append mode
770     ofs << "Line 5: Appended line\n";
771     std::cout << "      Appended one line" << std::endl;
772 }
773
```

```
774 // 8.5 Random access with fstream
775 std::cout << "\n8.5 Random access with fstream:" << std::endl;
776 {
777     std::fstream fs(test_file, std::ios::in | std::ios::out);
778
779     // Seek to position
780     fs.seekg(0, std::ios::end);
781     std::streampos file_size = fs.tellg();
782     std::cout << "    File size: " << file_size << " bytes" << std::endl;
783
784     // Go back to start
785     fs.seekg(0, std::ios::beg);
786
787     // Read first line
788     std::string first_line;
789     std::getline(fs, first_line);
790     std::cout << "    First line: " << first_line << std::endl;
791
792     // Current position
793     std::cout << "    Current position: " << fs.tellg() << std::endl;
794 }
795
796 // 8.6 Stream state checking
797 std::cout << "\n8.6 Stream state checking:" << std::endl;
798 {
799     std::ifstream ifs(test_file);
800
801     std::cout << "    Initial state:" << std::endl;
802     std::cout << "        good(): " << ifs.good() << std::endl;
803     std::cout << "        eof(): " << ifs.eof() << std::endl;
804     std::cout << "        fail(): " << ifs.fail() << std::endl;
805     std::cout << "        bad(): " << ifs.bad() << std::endl;
806
807     // Read until EOF
808     std::string line;
809     while (std::getline(ifs, line)) {
810         // Reading...
811     }
812
813     std::cout << "    After reading to EOF:" << std::endl;
814     std::cout << "        eof(): " << ifs.eof() << std::endl;
815     std::cout << "        good(): " << ifs.good() << std::endl;
816
817     // Clear error state
818     ifs.clear();
819     std::cout << "    After clear():" << std::endl;
820     std::cout << "        good(): " << ifs.good() << std::endl;
821 }
822
823 // Cleanup
824 fs::remove(test_file, ec);
825
826 std::cout << "\n File stream tips:" << std::endl;
827 std::cout << " • RAI - files auto-close when stream destroyed" << std::endl;
```

```
        endl;
828     std::cout << " • Check state with good(), fail(), bad(), eof()" << std::
829     endl;
830     std::cout << " • Use seekg()/tellg() for reading position" << std::endl;
831     std::cout << " • Use seekp()/tellp() for writing position" << std::endl;
832 }
833 // =====
834 // SECTION 9: STRING STREAMS - ostringstream, istringstream
835 // =====
836
837 struct LogRecord {
838     std::string timestamp;
839     std::string level;
840     int thread_id;
841     std::string message;
842 };
843
844 void demonstrate_string_streams() {
845     std::cout << "\n== 9. STRING STREAMS (ostringstream, istringstream) =="
846     << std::endl;
847
848     // 9.1 ostringstream - building strings
849     std::cout << "\n9.1 Building strings with ostringstream:" << std::endl;
850     {
851         std::ostringstream oss;
852         oss << "System report:\n";
853         oss << " CPU cores: " << 8 << "\n";
854         oss << " Memory: " << 16.5 << " GB\n";
855         oss << " Load: " << std::fixed << std::setprecision(2) << 0.6789 << "
856             \n";
857
858         std::string report = oss.str();
859         std::cout << report;
860     }
861
862     // 9.2 istringstream - parsing strings
863     std::cout << "\n9.2 Parsing with istringstream:" << std::endl;
864     {
865         std::string data = "2024-01-15 ERROR 12345 Database connection failed"
866         ;
867         std::istringstream iss(data);
868
869         LogRecord record;
870         iss >> record.timestamp >> record.level >> record.thread_id;
871         std::getline(iss, record.message); // Rest is message
872
873         // Trim leading space from message
874         if (!record.message.empty() && record.message[0] == ' ')
875             record.message = record.message.substr(1);
876
877         std::cout << " Parsed log record:" << std::endl;
878         std::cout << " Timestamp: " << record.timestamp << std::endl;
879     }
```

```
877     std::cout << "      Level: " << record.level << std::endl;
878     std::cout << "      Thread: " << record.thread_id << std::endl;
879     std::cout << "      Message: " << record.message << std::endl;
880 }
881
882 // 9.3 Parsing CSV
883 std::cout << "\n9.3 Parsing CSV with istringstream:" << std::endl;
884 {
885     std::string csv = "John,Doe,30,Engineer\nJane,Smith,28,Designer\nBob,
886     Johnson,35,Manager";
887     std::istringstream iss(csv);
888     std::string line;
889
890     while (std::getline(iss, line)) {
891         std::istringstream line_stream(line);
892         std::string first, last, age, role;
893
894         std::getline(line_stream, first, ',');
895         std::getline(line_stream, last, ',');
896         std::getline(line_stream, age, ',');
897         std::getline(line_stream, role, ',');
898
899         std::cout << "    " << first << " " << last
900             << " (" << age << ") - " << role << std::endl;
901     }
902 }
903
904 // 9.4 Number conversion with stringstream
905 std::cout << "\n9.4 Type conversions:" << std::endl;
906 {
907     // String to number
908     std::string num_str = "42.5";
909     std::istringstream iss(num_str);
910     double value;
911     iss >> value;
912     std::cout << "    String \"<< num_str << "\" -> double: " << value <<
913         std::endl;
914
915     // Number to string (alternative to std::to_string with formatting)
916     std::ostringstream oss;
917     oss << std::fixed << std::setprecision(3) << 3.14159265;
918     std::cout << "    double 3.14159265 -> string: \"<< oss.str() << \"\""
919         << std::endl;
920
921     // Hex conversion
922     oss.str(""); // Clear
923     oss.clear(); // Clear state
924     oss << "0x" << std::hex << 255;
925     std::cout << "    int 255 -> hex string: \"<< oss.str() << \"\" <<
926         std::endl;
927 }
928
929 // 9.5 Building complex formatted strings
930 std::cout << "\n9.5 Complex string formatting:" << std::endl;
```

```
927 {
928     std::ostringstream oss;
929
930     auto now = std::chrono::system_clock::now();
931     auto now_time_t = std::chrono::system_clock::to_time_t(now);
932
933     oss << "
934             SYSTEM STATUS REPORT
935             "
936     oss << " Time: " << std::put_time(std::localtime(&now_time_t), "%Y-%m-%d %H:%M:%S") << "
937             "
938     oss << " Status: " << std::left << std::setw(22) << "Running" << "
939             "
940     oss << " Uptime: " << std::right << std::setw(10) << "99.9%" << "
941             "
942     oss << "
943             "
944     std::cout << "\n String stream benefits:" << std::endl;
945     std::cout << " • Type-safe string building" << std::endl;
946     std::cout << " • Parsing with >> operator" << std::endl;
947     std::cout << " • Full stream manipulator support" << std::endl;
948     std::cout << " • Reusable: clear() + str(\"\\\") to reset" << std::endl;
949 }
950
951 // =====
952 // SECTION 10: STREAM MANIPULATORS AND FORMATTING
953 // =====
954
955 void demonstrate_stream_manipulators() {
956     std::cout << "\n== 10. STREAM MANIPULATORS ==" << std::endl;
957
958     // 10.1 Numeric formatting
959     std::cout << "\n10.1 Numeric formatting:" << std::endl;
960     {
961         double pi = 3.14159265358979;
962         int num = 255;
963
964         std::cout << " Default: " << pi << std::endl;
965         std::cout << " Fixed (6 decimals): " << std::fixed << pi << std::endl;
966         std::cout << " Scientific: " << std::scientific << pi << std::endl;
967         std::cout << std::defaultfloat; // Reset to default
968
969         std::cout << " Precision 3: " << std::setprecision(3) << pi << std::endl;
970         std::cout << " Precision 10: " << std::setprecision(10) << pi << std::endl;
971
972         std::cout << " Decimal: " << std::dec << num << std::endl;
973         std::cout << " Hex: 0x" << std::hex << num << std::endl;
974         std::cout << " Octal: 0" << std::oct << num << std::endl;
```

```
975     std::cout << std::dec; // Reset to decimal
976 }
977
978 // 10.2 Width and alignment
979 std::cout << "\n10.2 Width and alignment:" << std::endl;
980 {
981     std::cout << "    |" << std::setw(15) << "Right" << "|" << std::endl;
982     std::cout << "    |" << std::left << std::setw(15) << "Left" << "|" <<
983         std::endl;
984     std::cout << "    |" << std::internal << std::setw(15) << -123 << "|" <<
985         std::endl;
986     std::cout << std::right; // Reset
987 }
988
989 // 10.3 Fill character
990 std::cout << "\n10.3 Fill character:" << std::endl;
991 {
992     std::cout << "    " << std::setfill('*') << std::setw(20) << "Title" <<
993         std::setfill(' ') << std::endl;
994     std::cout << "    " << std::setfill('0') << std::setw(8) << 42 << std::
995         setfill(' ') << std::endl;
996     std::cout << "    " << std::setfill('-') << std::setw(30) << "" << std::
997         setfill(' ') << std::endl;
998 }
999
1000 // 10.4 Boolean formatting
1001 std::cout << "\n10.4 Boolean formatting:" << std::endl;
1002 {
1003     bool flag = true;
1004     std::cout << "    Default: " << flag << std::endl;
1005     std::cout << "    Boolalpha: " << std::boolalpha << flag << std::endl;
1006     std::cout << std::noboolalpha; // Reset
1007 }
1008
1009 // 10.5 Sign formatting
1010 std::cout << "\n10.5 Sign formatting:" << std::endl;
1011 {
1012     int pos = 42, neg = -42;
1013     std::cout << "    Default: " << pos << ", " << neg << std::endl;
1014     std::cout << "    Showpos: " << std::showpos << pos << ", " << neg <<
1015         std::endl;
1016     std::cout << std::noshowpos; // Reset
1017 }
1018
1019 // 10.6 Table formatting example
1020 std::cout << "\n10.6 Table formatting:" << std::endl;
1021 {
1022     struct Product {
1023         std::string name;
1024         double price;
1025         int quantity;
1026     };
1027
1028     std::vector<Product> products = {
```

```
1023     {"Widget", 29.99, 15},
1024     {"Gadget", 149.50, 8},
1025     {"Doohickey", 9.95, 42}
1026 };
1027
1028     std::cout << "           " << std::endl;
1029     std::cout << "           " << std::left << std::setw(15) << "Product"
1030           << "           " << std::right << std::setw(8) << "Price"
1031           << "           " << std::setw(8) << "Qty" << "           " << std::endl;
1032     std::cout << "           " << std::endl;
1033
1034     for (const auto& p : products) {
1035         std::cout << "           " << std::left << std::setw(15) << p.name
1036           << "           $" << std::right << std::setw(7) << std::fixed
1037           << std::setprecision(2) << p.price
1038           << "           " << std::setw(8) << p.quantity << "           " << std::endl;
1039     }
1040
1041     std::cout << "           " << std::endl;
1042 }
1043
1044 std::cout << "\n Manipulator categories:" << std::endl;
1045 std::cout << " •  Numeric: fixed, scientific, precision, hex, oct, dec"
1046   << std::endl;
1047 std::cout << " •  Alignment: setw, left, right, internal, setfill" << std::endl;
1048 std::cout << " •  Boolean: boolalpha, noboolalpha" << std::endl;
1049 std::cout << " •  Sign: showpos, noshowpos, showbase" << std::endl;
1050 }
1051 // =====
1052 // SECTION 11: BINARY I/O
1053 // =====
1054
1055 struct BinaryRecord {
1056     int id;
1057     double value;
1058     char name[32];
1059 };
1060
1061 void demonstrate_binary_io() {
1062     std::cout << "\n== 11. BINARY I/O ==" << std::endl;
1063
1064     std::error_code ec;
1065     fs::path temp_dir = fs::temp_directory_path(ec);
1066     fs::path binary_file = temp_dir / "binary_data.bin";
1067
1068     // 11.1 Writing binary data
1069     std::cout << "\n11.1 Writing binary data:" << std::endl;
1070     {
1071         std::ofstream ofs(binary_file, std::ios::binary);
1072
1073         BinaryRecord records[] = {
```

```
1074     {1, 3.14159, "Record One"},  
1075     {2, 2.71828, "Record Two"},  
1076     {3, 1.41421, "Record Three"}  
1077 };  
1078  
1079     for (const auto& record : records) {  
1080         ofs.write(reinterpret_cast<const char*>(&record), sizeof(  
1081             BinaryRecord));  
1082     }  
1083  
1084     std::cout << "      Wrote " << sizeof(records) / sizeof(BinaryRecord)  
1085         << " binary records (" << sizeof(records) << " bytes)" <<  
1086             std::endl;  
1087 }  
1088  
1089 // 11.2 Reading binary data  
1090 std::cout << "\n11.2 Reading binary data:" << std::endl;  
1091 {  
1092     std::ifstream ifs(binary_file, std::ios::binary);  
1093  
1094     // Get file size  
1095     ifs.seekg(0, std::ios::end);  
1096     std::streamsize size = ifs.tellg();  
1097     ifs.seekg(0, std::ios::beg);  
1098  
1099     std::cout << "      File size: " << size << " bytes" << std::endl;  
1100     std::cout << "      Records: " << size / sizeof(BinaryRecord) << std:::  
1101         endl;  
1102  
1103     BinaryRecord record;  
1104     int count = 0;  
1105     while (ifs.read(reinterpret_cast<char*>(&record), sizeof(BinaryRecord))  
1106         ) {  
1107         std::cout << "      Record " << ++count << ": "  
1108             << "id=" << record.id << ", "  
1109             << "value=" << record.value << ", "  
1110             << "name=" << record.name << "\n" << std::endl;  
1111     }  
1112 }  
1113  
1114 // 11.3 Writing POD types  
1115 std::cout << "\n11.3 Writing POD (Plain Old Data) types:" << std::endl;  
1116 {  
1117     fs::path pod_file = temp_dir / "pod_data.bin";  
1118  
1119     std::ofstream ofs(pod_file, std::ios::binary);  
1120  
1121     int integer = 42;  
1122     double floating = 3.14159;  
1123     char character = 'X';  
1124  
1125     ofs.write(reinterpret_cast<const char*>(&integer), sizeof(integer));  
1126     ofs.write(reinterpret_cast<const char*>(&floating), sizeof(floating));  
1127     ofs.write(reinterpret_cast<const char*>(&character), sizeof(character))
```

```
    );  
1124  
1125     ofs.close();  
1126  
1127     // Read back  
1128     std::ifstream ifs(pod_file, std::ios::binary);  
1129  
1130     int read_int;  
1131     double read_double;  
1132     char read_char;  
1133  
1134     ifs.read(reinterpret_cast<char*>(&read_int), sizeof(read_int));  
1135     ifs.read(reinterpret_cast<char*>(&read_double), sizeof(read_double));  
1136     ifs.read(reinterpret_cast<char*>(&read_char), sizeof(read_char));  
1137  
1138     std::cout << "    Read back: " << read_int << ", "  
1139             << read_double << ", " << read_char << std::endl;  
1140  
1141     fs::remove(pod_file, ec);  
1142 }  
1143  
1144 // Cleanup  
1145 fs::remove(binary_file, ec);  
1146  
1147 std::cout << "\n Binary I/O notes:" << std::endl;  
1148 std::cout << " • Use std::ios::binary flag" << std::endl;  
1149 std::cout << " • Only for POD types (no pointers, virtual functions)" <<  
1150             std::endl;  
1151 std::cout << " • Platform-dependent (endianness, padding)" << std::endl;  
1152 std::cout << " • Use serialization libraries for portability" << std::  
1153             endl;  
1154 }  
1155 // =====  
1156 // SECTION 12: CUSTOM STREAM OPERATORS  
1157 // =====  
1158  
1159 struct Point {  
1160     double x, y, z;  
1161 };  
1162  
1163 // Custom output operator  
1164 std::ostream& operator<<(std::ostream& os, const Point& p) {  
1165     os << "Point(" << p.x << ", " << p.y << ", " << p.z << ")";  
1166     return os;  
1167 }  
1168  
1169 // Custom input operator  
1170 std::istream& operator>>(std::istream& is, Point& p) {  
1171     char paren, comma1, comma2;  
1172     std::string word;  
1173  
1174     // Expected format: Point(1.0, 2.0, 3.0)  
1175     is >> word >> paren >> p.x >> comma1 >> p.y >> comma2 >> p.z >> paren;
```

```
1175     return is;
1176 }
1178
1179 void demonstrate_custom_operators() {
1180     std::cout << "\n==== 12. CUSTOM STREAM OPERATORS ===" << std::endl;
1181
1182     // 12.1 Custom output
1183     std::cout << "\n12.1 Custom output operator:" << std::endl;
1184     {
1185         Point p1{1.5, 2.7, 3.9};
1186         std::cout << "    " << p1 << std::endl;
1187
1188         // Works with string streams too
1189         std::ostringstream oss;
1190         oss << "Point data: " << p1;
1191         std::cout << "    Stringified: " << oss.str() << std::endl;
1192     }
1193
1194     // 12.2 Custom input
1195     std::cout << "\n12.2 Custom input operator:" << std::endl;
1196     {
1197         std::string input = "Point(4.5, 5.6, 6.7)";
1198         std::istringstream iss(input);
1199
1200         Point p2;
1201         iss >> p2;
1202
1203         std::cout << "    Parsed from string: " << p2 << std::endl;
1204     }
1205
1206     // 12.3 Chaining
1207     std::cout << "\n12.3 Operator chaining:" << std::endl;
1208     {
1209         Point p1{1.0, 2.0, 3.0};
1210         Point p2{4.0, 5.0, 6.0};
1211
1212         std::cout << "    Multiple points: " << p1 << " and " << p2 << std::endl;
1213     }
1214
1215     std::cout << "\n Custom operator benefits:" << std::endl;
1216     std::cout << " • Natural syntax: cout << myObject" << std::endl;
1217     std::cout << " • Works with all streams (file, string, cout)" << std::endl;
1218     std::cout << " • Chainable for multiple operations" << std::endl;
1219     std::cout << " • Consistent with standard library" << std::endl;
1220 }
1221
1222 // =====
1223 // SECTION 13: MUTEXES AND LOCK_GUARD
1224 // =====
1225
1226 // Shared resource with thread-safe access
```

```
1227 class SharedCounter {
1228 private:
1229     int counter = 0;
1230     mutable std::mutex mtx; // mutable allows locking in const methods
1231
1232 public:
1233     void increment() {
1234         std::lock_guard<std::mutex> lock(mtx); // RAII lock
1235         ++counter;
1236     } // Automatic unlock when lock goes out of scope
1237
1238     void add(int value) {
1239         std::lock_guard<std::mutex> lock(mtx);
1240         counter += value;
1241     }
1242
1243     int get() const {
1244         std::lock_guard<std::mutex> lock(mtx);
1245         return counter;
1246     }
1247 };
1248
1249 // File logger with mutex protection
1250 class ThreadSafeLogger {
1251 private:
1252     std::ofstream log_file;
1253     std::mutex file_mutex;
1254
1255 public:
1256     ThreadSafeLogger(const std::string& filename) : log_file(filename, std::ios::app) {}
1257
1258     void log(const std::string& message) {
1259         std::lock_guard<std::mutex> lock(file_mutex);
1260
1261         auto now = std::chrono::system_clock::now();
1262         auto now_time_t = std::chrono::system_clock::to_time_t(now);
1263
1264         log_file << "[" << std::put_time(std::localtime(&now_time_t), "%H:%M:%S")
1265             << "] " << message << std::endl;
1266     }
1267
1268     ~ThreadSafeLogger() {
1269         if (log_file.is_open()) {
1270             log_file.close();
1271         }
1272     }
1273 };
1274
1275 void demonstrate_mutex_and_lock_guard() {
1276     std::cout << "\n==== 13. MUTEXES AND LOCK_GUARD ===" << std::endl;
1277
1278     // 13.1 Basic lock_guard usage
```

```
1279     std::cout << "\n13.1 std::lock_guard for automatic RAII locking:" << std::endl;
1280
1281     {
1282         SharedCounter counter;
1283         std::vector<std::thread> threads;
1284
1285         // Launch 10 threads that increment counter
1286         for (int i = 0; i < 10; ++i) {
1287             threads.emplace_back([&counter]() {
1288                 for (int j = 0; j < 100; ++j) {
1289                     counter.increment();
1290                 }
1291             });
1292         }
1293
1294         // Wait for all threads
1295         for (auto& t : threads) {
1296             t.join();
1297         }
1298
1299         std::cout << "    Final counter value: " << counter.get() << std::endl;
1300         std::cout << "    Expected: 1000" << std::endl;
1301         std::cout << "    Thread-safe with lock_guard!" << std::endl;
1302     }
1303
1304     // 13.2 Thread-safe file logging
1305     std::cout << "\n13.2 Thread-safe file logging:" << std::endl;
1306     {
1307         std::error_code ec;
1308         fs::path temp_dir = fs::temp_directory_path(ec);
1309         fs::path log_file = temp_dir / "threadsafe_log.txt";
1310
1311         ThreadSafeLogger logger(log_file.string());
1312         std::vector<std::thread> threads;
1313
1314         // Multiple threads writing to log
1315         for (int i = 0; i < 5; ++i) {
1316             threads.emplace_back([&logger, i]() {
1317                 for (int j = 0; j < 3; ++j) {
1318                     std::ostringstream oss;
1319                     oss << "Thread " << i << " message " << j;
1320                     logger.log(oss.str());
1321                     std::this_thread::sleep_for(std::chrono::milliseconds(10));
1322                 }
1323             });
1324
1325         for (auto& t : threads) {
1326             t.join();
1327         }
1328
1329         std::cout << "    15 log entries written safely" << std::endl;
1330         std::cout << "    Log file: " << log_file.filename() << std::endl;
```

```
1331
1332     // Read and display log
1333     std::ifstream ifs(log_file);
1334     std::string line;
1335     int count = 0;
1336     std::cout << "    First 5 entries:" << std::endl;
1337     while (std::getline(ifs, line) && count++ < 5) {
1338         std::cout << "    " << line << std::endl;
1339     }
1340
1341     fs::remove(log_file, ec);
1342 }
1343
1344 std::cout << "\n lock_guard benefits:" << std::endl;
1345 std::cout << " • RAI - automatic unlock on scope exit" << std::endl;
1346 std::cout << " • Exception-safe" << std::endl;
1347 std::cout << " • Simple and efficient" << std::endl;
1348 std::cout << " • Cannot forget to unlock" << std::endl;
1349 }
1350
1351 // =====
1352 // SECTION 14: UNIQUE_LOCK AND ADVANCED LOCKING
1353 // =====
1354
1355 class BankAccount {
1356 private:
1357     double balance = 0.0;
1358     mutable std::mutex mtx;
1359     std::condition_variable cv;
1360
1361 public:
1362     BankAccount(double initial) : balance(initial) {}
1363
1364     bool withdraw(double amount) {
1365         std::unique_lock<std::mutex> lock(mtx);
1366
1367         if (balance >= amount) {
1368             balance -= amount;
1369             return true;
1370         }
1371         return false;
1372     }
1373
1374     void deposit(double amount) {
1375         std::unique_lock<std::mutex> lock(mtx);
1376         balance += amount;
1377         cv.notify_all(); // Wake up waiting threads
1378     }
1379
1380     // Wait until sufficient funds available
1381     bool wait_and_withdraw(double amount, std::chrono::milliseconds timeout) {
1382         std::unique_lock<std::mutex> lock(mtx);
1383
1384         // Wait with timeout
```

```
1385     if (!cv.wait_for(lock, timeout, [this, amount]() {
1386         return balance >= amount;
1387     })) {
1388         return false; // Timeout
1389     }
1390
1391     balance -= amount;
1392     return true;
1393 }
1394
1395 double get_balance() const {
1396     std::unique_lock<std::mutex> lock(mtx);
1397     return balance;
1398 }
1399 };
1400
1401 // Reader-writer lock example
1402 class SharedData {
1403 private:
1404     std::map<std::string, int> data;
1405     mutable std::shared_mutex rw_mutex;
1406
1407 public:
1408     // Multiple readers can access simultaneously
1409     int read(const std::string& key) const {
1410         std::shared_lock<std::shared_mutex> lock(rw_mutex);
1411         auto it = data.find(key);
1412         return (it != data.end()) ? it->second : 0;
1413     }
1414
1415     // Only one writer at a time
1416     void write(const std::string& key, int value) {
1417         std::unique_lock<std::shared_mutex> lock(rw_mutex);
1418         data[key] = value;
1419     }
1420
1421     size_t size() const {
1422         std::shared_lock<std::shared_mutex> lock(rw_mutex);
1423         return data.size();
1424     }
1425 };
1426
1427 void demonstrate_unique_lock() {
1428     std::cout << "\n==== 14. UNIQUE_LOCK AND ADVANCED LOCKING ===" << std::endl
1429     ;
1430
1431     // 14.1 unique_lock - more flexible than lock_guard
1432     std::cout << "\n14.1 std::unique_lock features:" << std::endl;
1433     {
1434         std::mutex mtx;
1435
1436         std::cout << " • Can be locked/unlocked multiple times:" << std::endl;
1437     }
1438 }
```

```
1437     std::unique_lock<std::mutex> lock(mtx);
1438     std::cout << "      Locked" << std::endl;
1439
1440     lock.unlock(); // Explicit unlock
1441     std::cout << "      Unlocked (doing non-critical work)" << std::endl;
1442     lock.lock(); // Relock
1443     std::cout << "      Locked again" << std::endl;
1444 }
1445
1446     std::cout << " *  Deferred locking:" << std::endl;
1447 {
1448     std::unique_lock<std::mutex> lock(mtx, std::defer_lock); // Don't
1449     lock.lock(); // Lock when needed
1450     std::cout << "      Created unlocked" << std::endl;
1451
1452     std::cout << "      Now locked" << std::endl;
1453 }
1454
1455 }
1456
1457 // 14.2 Condition variable with unique_lock
1458 std::cout << "\n14.2 Condition variable (wait with timeout):" << std::endl;
1459 ;
1460 {
1461     BankAccount account(100.0);
1462
1463     std::cout << "      Initial balance: $" << account.get_balance() << std::endl;
1464
1465     // Try to withdraw more than available
1466     std::thread withdrawal_thread([&account]() {
1467         std::cout << "      Attempting to withdraw $150 (waiting up to 500ms)"
1468         ...
1469         << std::endl;
1470         bool success = account.wait_and_withdraw(150.0, std::chrono::milliseconds(500));
1471
1472         if (success) {
1473             std::cout << "      Withdrawal successful" << std::endl;
1474         } else {
1475             std::cout << "      Timeout - insufficient funds" << std::endl;
1476             ;
1477         }
1478     });
1479
1480     std::this_thread::sleep_for(std::chrono::milliseconds(200));
1481
1482     // Deposit money to unblock
1483     std::cout << "      Depositing $75..." << std::endl;
1484     account.deposit(75.0);
1485
1486     withdrawal_thread.join();
1487     std::cout << "      Final balance: $" << account.get_balance() << std::endl;
```

```
        endl;
1484 }
1485
1486 // 14.3 Shared mutex for reader-writer locks
1487 std::cout << "\n14.3 std::shared_mutex (reader-writer lock):" << std::endl
1488 ;
1489 {
1490     SharedData shared;
1491     std::atomic<int> read_count{0};
1492     std::atomic<int> write_count{0};
1493
1494     std::vector<std::thread> threads;
1495
1496     // 8 readers
1497     for (int i = 0; i < 8; ++i) {
1498         threads.emplace_back([&shared, &read_count, i]() {
1499             for (int j = 0; j < 10; ++j) {
1500                 shared.read("key" + std::to_string(i % 3));
1501                 read_count++;
1502                 std::this_thread::sleep_for(std::chrono::microseconds(100));
1503             }
1504         });
1505     }
1506
1507     // 2 writers
1508     for (int i = 0; i < 2; ++i) {
1509         threads.emplace_back([&shared, &write_count, i]() {
1510             for (int j = 0; j < 5; ++j) {
1511                 shared.write("key" + std::to_string(j), i * 100 + j);
1512                 write_count++;
1513                 std::this_thread::sleep_for(std::chrono::milliseconds(1));
1514             }
1515         });
1516     }
1517
1518     for (auto& t : threads) {
1519         t.join();
1520     }
1521
1522     std::cout << "    Completed " << read_count << " reads" << std::endl;
1523     std::cout << "    Completed " << write_count << " writes" << std::endl;
1524     std::cout << "    Final data size: " << shared.size() << std::endl;
1525     std::cout << "    Multiple readers can run concurrently!" << std::endl;
1526 }
1527
1528 std::cout << "\n unique_lock vs lock_guard:" << std::endl;
1529 std::cout << "    lock_guard: Simple, RAII, no unlock() method" << std::endl;
1530 std::cout << "    unique_lock: Flexible, can unlock/relock, works with
1531         condition_variable" << std::endl;
1532 std::cout << "\n shared_mutex:" << std::endl;
1533 std::cout << "    shared_lock: Multiple readers simultaneously" << std::endl;
```

```
        endl;
1532     std::cout << " • unique_lock: Exclusive writer access" << std::endl;
1533     std::cout << " • Perfect for read-heavy workloads" << std::endl;
1534 }
1535
1536 // =====
1537 // SECTION 15: FILE LOCKING (OS-LEVEL)
1538 // =====
1539
1540 #ifdef __linux__
1541 #include <sys/file.h> // For flock
1542 #include <fcntl.h>
1543 #include <unistd.h>
1544
1545 class FileLock {
1546 private:
1547     int fd = -1;
1548     std::string filename;
1549
1550 public:
1551     FileLock(const std::string& file) : filename(file) {
1552         fd = open(filename.c_str(), O_RDWR | O_CREAT, 0666);
1553         if (fd == -1) {
1554             throw std::runtime_error("Failed to open file for locking");
1555         }
1556     }
1557
1558     bool try_lock() {
1559         return flock(fd, LOCK_EX | LOCK_NB) == 0; // Non-blocking exclusive
1560         lock
1561     }
1562
1563     void lock() {
1564         if (flock(fd, LOCK_EX) != 0) { // Blocking exclusive lock
1565             throw std::runtime_error("Failed to acquire lock");
1566         }
1567     }
1568
1569     void unlock() {
1570         flock(fd, LOCK_UN);
1571     }
1572
1573     ~FileLock() {
1574         if (fd != -1) {
1575             unlock();
1576             close(fd);
1577         }
1578     };
1579 #endif
1580
1581 void demonstrate_file_locking() {
1582     std::cout << "\n==== 15. FILE LOCKING (OS-LEVEL) ===" << std::endl;
1583 }
```

```
1584 #ifdef __linux__
1585     std::cout << "\n15.1 Advisory file locking with flock():" << std::endl;
1586 {
1587     std::error_code ec;
1588     fs::path temp_dir = fs::temp_directory_path(ec);
1589     fs::path lock_file = temp_dir / "test.lock";
1590
1591     try {
1592         FileLock lock1(lock_file.string());
1593
1594         std::cout << "    Thread 1: Acquiring lock..." << std::endl;
1595         lock1.lock();
1596         std::cout << "    Thread 1: Lock acquired" << std::endl;
1597
1598         // Simulate work
1599         std::this_thread::sleep_for(std::chrono::milliseconds(100));
1600
1601         // Try to acquire same lock from another thread
1602         std::thread other_thread([&lock_file]() {
1603             try {
1604                 FileLock lock2(lock_file.string());
1605                 std::cout << "    Thread 2: Trying non-blocking lock..." <<
1606                         std::endl;
1607
1608                 if (lock2.try_lock()) {
1609                     std::cout << "    Thread 2: Lock acquired" << std::endl;
1610                     lock2.unlock();
1611                 } else {
1612                     std::cout << "    Thread 2: Lock held by another
1613                         process" << std::endl;
1614                 }
1615             } catch (const std::exception& e) {
1616                 std::cout << "    Thread 2: Error - " << e.what() << std::endl;
1617             }
1618         });
1619
1620         other_thread.join();
1621
1622         std::cout << "    Thread 1: Releasing lock..." << std::endl;
1623         lock1.unlock();
1624
1625     } catch (const std::exception& e) {
1626         std::cout << "    Error: " << e.what() << std::endl;
1627     }
1628
1629     fs::remove(lock_file, ec);
1630 }
1631
1632 std::cout << "\n15.2 Preventing concurrent file access:" << std::endl;
1633 {
1634     std::error_code ec;
1635     fs::path temp_dir = fs::temp_directory_path(ec);
```

```
1634     fs::path shared_file = temp_dir / "shared_data.txt";
1635     fs::path lock_file_path = temp_dir / "shared_data.txt.lock";
1636
1637     // Writer with lock
1638     auto writer = [&shared_file, &lock_file_path](int id) {
1639         try {
1640             FileLock lock(lock_file_path.string());
1641             lock.lock();
1642
1643             std::cout << "    Writer " << id << ": Writing..." << std::endl
1644             ;
1645             std::ofstream ofs(shared_file, std::ios::app);
1646             ofs << "Line from writer " << id << "\n";
1647             ofs.close();
1648
1649             std::this_thread::sleep_for(std::chrono::milliseconds(50));
1650             lock.unlock();
1651             std::cout << "    Writer " << id << ": Done" << std::endl;
1652         } catch (const std::exception& e) {
1653             std::cout << "    Writer " << id << ": Error - " << e.what() <<
1654             std::endl;
1655         }
1656     };
1657
1658     std::thread w1(writer, 1);
1659     std::thread w2(writer, 2);
1660     std::thread w3(writer, 3);
1661
1662     w1.join();
1663     w2.join();
1664     w3.join();
1665
1666     // Read result
1667     std::ifstream ifs(shared_file);
1668     std::string line;
1669     int line_count = 0;
1670     while (std::getline(ifs, line)) {
1671         line_count++;
1672     }
1673
1674     std::cout << "    All writers completed safely" << std::endl;
1675     std::cout << "    Total lines written: " << line_count << std::endl;
1676
1677     fs::remove(shared_file, ec);
1678     fs::remove(lock_file_path, ec);
1679 }
1680
1681 std::cout << "\n File locking types:" << std::endl;
1682 std::cout << " • Advisory locks: Processes must cooperate (flock)" <<
1683     std::endl;
1684 std::cout << " • Mandatory locks: Enforced by kernel (rare)" << std::
1685     endl;
1686 std::cout << " • LOCK_EX: Exclusive lock (writer)" << std::endl;
1687 std::cout << " • LOCK_SH: Shared lock (reader)" << std::endl;
```

```
1684     std::cout << " •  LOCK_NB: Non-blocking flag" << std::endl;
1685
1686 #else
1687     std::cout << "\n          File locking examples require Linux (flock)" << std
1688         ::endl;
1689     std::cout << "      Windows equivalent: LockFileEx()" << std::endl;
1690     std::cout << "      Cross-platform: Use Boost.Interprocess" << std::endl;
1691 #endif
1692 }
1693 // =====
1694 // SECTION 16: RACE CONDITIONS - PROBLEM AND SOLUTION
1695 // =====
1696
1697 // Simulated device hotplug manager - WITHOUT proper synchronization (BROKEN!)
1698 class BrokenDeviceManager {
1699 private:
1700     std::map<std::string, std::string> devices; // device_id -> status
1701     int device_count = 0;
1702
1703 public:
1704     // Called by multiple threads - NO MUTEX!
1705     void add_device(const std::string& device_id) {
1706         // RACE CONDITION: Multiple threads can read/write simultaneously
1707         if (devices.find(device_id) == devices.end()) {
1708             std::this_thread::sleep_for(std::chrono::microseconds(100)); // 
1709             // Simulate processing
1710             devices[device_id] = "active";
1711             device_count++;
1712         }
1713     }
1714
1715     void remove_device(const std::string& device_id) {
1716         // RACE CONDITION: Map modification without synchronization
1717         if (devices.find(device_id) != devices.end()) {
1718             std::this_thread::sleep_for(std::chrono::microseconds(100));
1719             devices.erase(device_id);
1720             device_count--;
1721         }
1722     }
1723
1724     void process_script_output(const std::string& script_output) {
1725         // RACE CONDITION: Reading and updating shared state
1726         std::istringstream iss(script_output);
1727         std::string line;
1728         while (std::getline(iss, line)) {
1729             if (line.find("ADD:") == 0) {
1730                 std::string device = line.substr(4);
1731                 devices[device] = "pending";
1732             }
1733         }
1734     }
1735
1736     int get_count() const {
```

```
1736     return device_count; // RACE CONDITION: Reading without lock
1737 }
1738
1739 std::map<std::string, std::string> get_devices() const {
1740     return devices; // RACE CONDITION: Copying map without lock
1741 }
1742 };
1743
1744 // Fixed version - WITH proper synchronization
1745 class SafeDeviceManager {
1746 private:
1747     std::map<std::string, std::string> devices;
1748     int device_count = 0;
1749     mutable std::mutex mtx; // Protects all shared state
1750     std::ofstream log_file;
1751
1752 public:
1753     SafeDeviceManager() : log_file("/tmp/device_manager.log", std::ios::app)
1754     {}
1755
1756     void add_device(const std::string& device_id) {
1757         std::lock_guard<std::mutex> lock(mtx); // FIXED: Automatic locking
1758
1759         if (devices.find(device_id) == devices.end()) {
1760             std::this_thread::sleep_for(std::chrono::microseconds(100));
1761             devices[device_id] = "active";
1762             device_count++;
1763
1764             if (log_file.is_open()) {
1765                 log_file << "[ADD] " << device_id << std::endl;
1766             }
1767         }
1768
1769     void remove_device(const std::string& device_id) {
1770         std::lock_guard<std::mutex> lock(mtx); // FIXED: Protected
1771
1772         if (devices.find(device_id) != devices.end()) {
1773             std::this_thread::sleep_for(std::chrono::microseconds(100));
1774             devices.erase(device_id);
1775             device_count--;
1776
1777             if (log_file.is_open()) {
1778                 log_file << "[REMOVE] " << device_id << std::endl;
1779             }
1780         }
1781     }
1782
1783     void process_script_output(const std::string& script_output) {
1784         std::lock_guard<std::mutex> lock(mtx); // FIXED: Protected
1785
1786         std::istringstream iss(script_output);
1787         std::string line;
1788         while (std::getline(iss, line)) {
```

```
1789         if (line.find("ADD:") == 0) {
1790             std::string device = line.substr(4);
1791             devices[device] = "pending";
1792         }
1793     }
1794 }
1795
1796 int get_count() const {
1797     std::lock_guard<std::mutex> lock(mtx); // FIXED: Protected read
1798     return device_count;
1799 }
1800
1801 std::map<std::string, std::string> get_devices() const {
1802     std::lock_guard<std::mutex> lock(mtx); // FIXED: Protected copy
1803     return devices;
1804 }
1805
1806 ~SafeDeviceManager() {
1807     if (log_file.is_open()) {
1808         log_file.close();
1809     }
1810 }
1811 };
1812
1813 // Simulate hotplug events by calling external scripts
1814 void simulate_hotplug_script(int event_id) {
1815     // Create a temporary script that simulates device detection
1816     std::error_code ec;
1817     fs::path temp_dir = fs::temp_directory_path(ec);
1818     fs::path script_path = temp_dir / ("hotplug_" + std::to_string(event_id) +
1819         ".sh");
1820
1821     std::ofstream script(script_path);
1822     script << "#!/bin/bash\n";
1823     script << "echo 'ADD:device_" << event_id << "'\n";
1824     script << "sleep 0.01\n";
1825     script << "echo 'STATUS:online'\n";
1826     script.close();
1827
1828     fs::permissions(script_path, fs::perms::owner_all, fs::perm_options::add,
1829                     ec);
1830 }
1831
1832 void demonstrate_race_condition() {
1833     std::cout << "\n==== 16. RACE CONDITIONS - PROBLEM AND SOLUTION ===" << std::endl;
1834
1835     // 16.1 Demonstrate the BROKEN version
1836     std::cout << "\n16.1 BROKEN: Race condition without mutex:" << std::endl;
1837
1838     {
1839         BrokenDeviceManager broken_mngr;
1840         std::vector<std::thread> threads;
```

```
1839     std::cout << "    Launching 20 threads to add devices..." << std::endl;
1840
1841     // Multiple threads adding same devices
1842     for (int i = 0; i < 20; ++i) {
1843         threads.emplace_back([&broken_mgr, i]() {
1844             for (int j = 0; j < 10; ++j) {
1845                 std::string device_id = "dev_" + std::to_string(j);
1846                 broken_mgr.add_device(device_id);
1847             }
1848         });
1849     }
1850
1851     for (auto& t : threads) {
1852         t.join();
1853     }
1854
1855     int final_count = broken_mgr.get_count();
1856     auto devices = broken_mgr.get_devices();
1857
1858     std::cout << "    Expected: 10 unique devices" << std::endl;
1859     std::cout << "    Device count variable: " << final_count << std::endl;
1860     std::cout << "    Actual devices in map: " << devices.size() << std::endl;
1861
1862     if (final_count != 10 || devices.size() != 10) {
1863         std::cout << "        RACE CONDITION DETECTED!" << std::endl;
1864         std::cout << "        Count mismatch due to concurrent access" <<
1865             std::endl;
1866     } else {
1867         std::cout << "        (Race condition may not always show - timing
1868             dependent)" << std::endl;
1869     }
1870
1871 // 16.2 Demonstrate the FIXED version
1872 std::cout << "\n16.2  FIXED: With mutex protection:" << std::endl;
1873 {
1874     SafeDeviceManager safe_mgr;
1875     std::vector<std::thread> threads;
1876
1877     std::cout << "    Launching 20 threads to add devices..." << std::endl;
1878
1879     for (int i = 0; i < 20; ++i) {
1880         threads.emplace_back([&safe_mgr, i]() {
1881             for (int j = 0; j < 10; ++j) {
1882                 std::string device_id = "dev_" + std::to_string(j);
1883                 safe_mgr.add_device(device_id);
1884             }
1885         });
1886
1887     for (auto& t : threads) {
1888         t.join();
1889     }
```

```
1890
1891     int final_count = safe_mgr.get_count();
1892     auto devices = safe_mgr.get_devices();
1893
1894     std::cout << "    Expected: 10 unique devices" << std::endl;
1895     std::cout << "    Device count: " << final_count << std::endl;
1896     std::cout << "    Devices in map: " << devices.size() << std::endl;
1897
1898     if (final_count == 10 && devices.size() == 10) {
1899         std::cout << "    CORRECT: Mutex prevented race condition!" <<
1900             std::endl;
1901     }
1902
1903 // 16.3 Real-world scenario: File updates from multiple sources
1904 std::cout << "\n16.3 Race condition with file I/O and scripts:" << std::
1905     endl;
1906 {
1907     std::error_code ec;
1908     fs::path temp_dir = fs::temp_directory_path(ec);
1909     fs::path shared_file = temp_dir / "device_registry.txt";
1910
1911     // BROKEN: Multiple threads writing to file without coordination
1912     std::cout << "    Without file locking:" << std::endl;
1913     {
1914         // Clear file
1915         std::ofstream(shared_file).close();
1916
1917         std::vector<std::thread> writers;
1918
1919         for (int i = 0; i < 5; ++i) {
1920             writers.emplace_back([&shared_file, i]() {
1921                 for (int j = 0; j < 3; ++j) {
1922                     // RACE CONDITION: Multiple threads writing
1923                     // simultaneously
1924                     std::ofstream ofs(shared_file, std::ios::app);
1925                     ofs << "Thread_" << i << "_entry_" << j << "\n";
1926                     ofs.close();
1927                     std::this_thread::sleep_for(std::chrono::microseconds
1928                         (100));
1929                 }
1930             });
1931         }
1932
1933         for (auto& t : writers) {
1934             t.join();
1935         }
1936
1937         // Count lines
1938         std::ifstream ifs(shared_file);
1939         int line_count = 0;
1940         std::string line;
1941         while (std::getline(ifs, line)) {
1942             if (!line.empty()) line_count++;
1943         }
1944     }
1945 }
```

```
1940     }
1941
1942     std::cout << "      Expected 15 lines, got: " << line_count << std
1943         ::endl;
1944     if (line_count != 15) {
1945         std::cout << "      Some writes may have been lost!" << std
1946             ::endl;
1947     }
1948
1949 // FIXED: With mutex protection
1950 std::cout << "      With mutex protection:" << std::endl;
1951 {
1952     // Clear file
1953     std::ofstream(shared_file).close();
1954
1955     std::mutex file_mtx;
1956     std::vector<std::thread> writers;
1957
1958     for (int i = 0; i < 5; ++i) {
1959         writers.emplace_back([&shared_file, &file_mtx, i]() {
1960             for (int j = 0; j < 3; ++j) {
1961                 std::lock_guard<std::mutex> lock(file_mtx); // FIXED!
1962                 std::ofstream ofs(shared_file, std::ios::app);
1963                 ofs << "Thread_" << i << "_entry_" << j << "\n";
1964                 ofs.close();
1965                 std::this_thread::sleep_for(std::chrono::microseconds
1966                     (100));
1967             }
1968         });
1969     }
1970
1971     for (auto& t : writers) {
1972         t.join();
1973     }
1974
1975     // Count lines
1976     std::ifstream ifs(shared_file);
1977     int line_count = 0;
1978     std::string line;
1979     while (std::getline(ifs, line)) {
1980         if (!line.empty()) line_count++;
1981     }
1982
1983     std::cout << "      Expected 15 lines, got: " << line_count << std
1984         ::endl;
1985     if (line_count == 15) {
1986         std::cout << "      All writes preserved!" << std::endl;
1987     }
1988
1989     fs::remove(shared_file, ec);
1990 }
```

```
1990 // 16.4 Simulating hotplug events with script execution
1991 std::cout << "\n16.4 Hotplug simulation (scripts + threads):" << std::endl
1992 ;
1993 {
1994     SafeDeviceManager mgr;
1995     std::vector<std::thread> hotplug_threads;
1996
1997     std::cout << "    Simulating 3 concurrent hotplug events..." << std::endl;
1998
1999     for (int i = 0; i < 3; ++i) {
2000         hotplug_threads.emplace_back([&mgr, i](){
2001             // Simulate calling udev script or hotplug handler
2002             std::string script_output = "ADD:usb_device_" + std::to_string(i) + "\n";
2003             script_output += "ADD:usb_port_" + std::to_string(i) + "\n";
2004
2005             mgr.process_script_output(script_output);
2006
2007             // Simulate device becoming active
2008             std::this_thread::sleep_for(std::chrono::milliseconds(10));
2009             mgr.add_device("usb_device_" + std::to_string(i));
2010             mgr.add_device("usb_port_" + std::to_string(i));
2011         });
2012     }
2013
2014     for (auto& t : hotplug_threads) {
2015         t.join();
2016     }
2017
2018     auto devices = mgr.get_devices();
2019     std::cout << "    Total devices registered: " << devices.size() << std::endl;
2020
2021     std::cout << "    Devices:" << std::endl;
2022     for (const auto& [id, status] : devices) {
2023         std::cout << "        " << id << " -> " << status << std::endl;
2024     }
2025     std::cout << "    All hotplug events handled safely" << std::endl;
2026 }
2027
2028 std::cout << "\n Common race condition scenarios:" << std::endl;
2029 std::cout << "    1. Multiple threads modifying shared container" << std::endl;
2030 std::cout << "    2. Check-then-act pattern without atomicity" << std::endl;
2031 std::cout << "    3. Reading and writing without synchronization" << std::endl;
2032 std::cout << "    4. File I/O from multiple threads/processes" << std::endl;
2033 std::cout << "    5. Scripts updating shared resources" << std::endl;
2034
2035 std::cout << "\n Solutions:" << std::endl;
2036 std::cout << "    • Use std::mutex with lock_guard/unique_lock" << std::endl;
```

```

2035     std::cout << " • Use std::atomic for simple counters" << std::endl;
2036     std::cout << " • Use file locks for inter-process sync" << std::endl;
2037     std::cout << " • Protect ALL access to shared data" << std::endl;
2038     std::cout << " • Keep critical sections small" << std::endl;
2039 }
2040
2041 // =====
2042 // SECTION 17: SECURITY CONSIDERATIONS
2043 // =====
2044
2045 void explain_security_considerations() {
2046     std::cout << "\n" << std::string(70, '=') << std::endl;
2047     std::cout << "SECURITY CONSIDERATIONS:\n";
2048     std::cout << std::string(70, '=') << std::endl;
2049
2050     std::cout << "\n CRITICAL SECURITY ISSUES:" << std::endl;
2051
2052     std::cout << "\n1. COMMAND INJECTION:" << std::endl;
2053     std::cout << "    BAD: system(\"ls \" + user_input);" << std::endl;
2054     std::cout << "    GOOD: Validate and sanitize input first" << std::endl;
2055     std::cout << "    GOOD: Use execve() family for direct execution" << std
2056         ::endl;
2057
2058     std::cout << "\n2. SHELL METACHARACTERS:" << std::endl;
2059     std::cout << "    Dangerous: ; | & $ ` \\ \" ' < > ( ) { } [ ] ! #"
2060         << std
2061         ::endl;
2062     std::cout << "    User input: \"file.txt; rm -rf /\""
2063     std::cout << "    Result: DISASTER!" << std::endl;
2064
2065     std::cout << "\n3. PATH TRAVERSAL:" << std::endl;
2066     std::cout << "    BAD: open(\"/data/\" + user_input)" << std::endl;
2067     std::cout << "    User: \"../../etc/passwd\""
2068     std::cout << "    GOOD: Validate paths, use fs::canonical()" << std::endl
2069         ;
2070
2071     std::cout << "\n4. ENVIRONMENT VARIABLES:" << std::endl;
2072     std::cout << "    $PATH, $LD_PRELOAD can be hijacked" << std::endl;
2073     std::cout << "    Use absolute paths: /usr/bin/python3"
2074     std::cout << "    Don't trust $PATH from external sources" << std::endl;
2075
2076     std::cout << "\n BEST PRACTICES:" << std::endl;
2077     std::cout << "\n1. INPUT VALIDATION:" << std::endl;
2078     std::cout << "    Whitelist allowed characters" << std::endl;
2079     std::cout << "    Use regex to validate format" << std::endl;
2080     std::cout << "    Reject anything suspicious" << std::endl;
2081
2082     std::cout << "\n2. AVOID system():" << std::endl;
2083     std::cout << "    Use popen() for output capture"
2084     std::cout << "    Use fork() + execve() for full control"
2085     std::cout << "    Never pass user input directly to shell"
2086
2087
2088     std::cout << "\n3. LEAST PRIVILEGE:" << std::endl;
2089     std::cout << "    Run with minimum necessary permissions"
2090     std::cout << "    Drop privileges after initialization"

```

```
2086     std::cout << " • Use separate user accounts" << std::endl;
2087
2088     std::cout << "\n4. ERROR HANDLING:" << std::endl;
2089     std::cout << " • Check all return codes" << std::endl;
2090     std::cout << " • Don't expose system details in errors" << std::endl;
2091     std::cout << " • Log security events" << std::endl;
2092 }
2093
2094 // =====
2095 // SECTION 18: CROSS-PLATFORM CONSIDERATIONS
2096 // =====
2097
2098 void explain_cross_platform() {
2099     std::cout << "\n" << std::string(70, '=') << std::endl;
2100     std::cout << "CROSS-PLATFORM CONSIDERATIONS:\n";
2101     std::cout << std::string(70, '=') << std::endl;
2102
2103     std::cout << "\n LINUX vs   WINDOWS:" << std::endl;
2104
2105     std::cout << "\n1. COMMAND EXECUTION:" << std::endl;
2106     std::cout << "   Linux:  popen(\"/bin/ls\", \"r\")" << std::endl;
2107     std::cout << "   Windows: _popen(\"dir\", \"r\")" << std::endl;
2108     std::cout << "   Windows: CreateProcess() for full control" << std::endl;
2109
2110     std::cout << "\n2. PATH SEPARATORS:" << std::endl;
2111     std::cout << "   Linux:  /home/user/file.txt" << std::endl;
2112     std::cout << "   Windows: C:\\\\Users\\\\user\\\\file.txt" << std::endl;
2113     std::cout << "   Use fs::path - handles both!" << std::endl;
2114
2115     std::cout << "\n3. LINE ENDINGS:" << std::endl;
2116     std::cout << "   Linux:  \\n (LF)" << std::endl;
2117     std::cout << "   Windows: \\r\\n (CRLF)" << std::endl;
2118     std::cout << "   Open files in text mode for conversion" << std::endl;
2119
2120     std::cout << "\n4. SYSTEM TOOLS:" << std::endl;
2121     std::cout << "   Linux:  ps, df, grep, awk, sed" << std::endl;
2122     std::cout << "   Windows: tasklist, wmic, findstr, PowerShell" << std::endl;
2123     std::cout << "   Check platform and use appropriate tools" << std::endl;
2124
2125     std::cout << "\n PORTABLE CODE:" << std::endl;
2126     std::cout << "#ifdef _WIN32" << std::endl;
2127     std::cout << "   std::string cmd = \"dir\";" << std::endl;
2128     std::cout << "#else" << std::endl;
2129     std::cout << "   std::string cmd = \"ls\";" << std::endl;
2130     std::cout << "#endif" << std::endl;
2131
2132     std::cout << "\nOr use std::filesystem for platform independence!" << std::endl;
2133 }
2134
2135 // =====
2136 // SECTION 19: BEST PRACTICES SUMMARY
2137 // =====
```

```
2138
2139 void explain_best_practices() {
2140     std::cout << "\n" << std::string(70, '=') << std::endl;
2141     std::cout << "BEST PRACTICES SUMMARY:\n";
2142     std::cout << std::string(70, '=') << std::endl;
2143
2144     std::cout << "\n DO:" << std::endl;
2145     std::cout << "\n1. Use std::filesystem for file operations" << std::endl;
2146     std::cout << " • Cross-platform" << std::endl;
2147     std::cout << " • Exception-safe" << std::endl;
2148     std::cout << " • Modern and clean API" << std::endl;
2149
2150     std::cout << "\n2. Use std::string_view for parsing" << std::endl;
2151     std::cout << " • Zero-copy substring operations" << std::endl;
2152     std::cout << " • Perfect for tokenizing large outputs" << std::endl;
2153     std::cout << " • No memory allocations" << std::endl;
2154
2155     std::cout << "\n3. Use std::regex for pattern matching" << std::endl;
2156     std::cout << " • Structured text parsing" << std::endl;
2157     std::cout << " • Validation" << std::endl;
2158     std::cout << " • Extraction from unstructured data" << std::endl;
2159
2160     std::cout << "\n4. Check return codes and use error_code" << std::endl;
2161     std::cout << " • filesystem operations can fail" << std::endl;
2162     std::cout << " • Commands can fail" << std::endl;
2163     std::cout << " • Use std::error_code for exception-free errors" << std::endl;
2164
2165     std::cout << "\n5. Use RAII for resource management" << std::endl;
2166     std::cout << " • std::unique_ptr with custom deleter for FILE*" << std::endl;
2167     std::cout << " • Automatic cleanup on exception" << std::endl;
2168
2169     std::cout << "\n DON'T:" << std::endl;
2170     std::cout << "\n1. Don't use system() with user input" << std::endl;
2171     std::cout << " • Command injection vulnerability" << std::endl;
2172     std::cout << " • Use execve() family instead" << std::endl;
2173
2174     std::cout << "\n2. Don't ignore errors" << std::endl;
2175     std::cout << " • Check pclose() return value" << std::endl;
2176     std::cout << " • Check filesystem operation errors" << std::endl;
2177
2178     std::cout << "\n3. Don't parse with manual string manipulation" << std::endl;
2179     std::cout << " • Use string_view for efficiency" << std::endl;
2180     std::cout << " • Use regex for complex patterns" << std::endl;
2181
2182     std::cout << "\n4. Don't assume POSIX everywhere" << std::endl;
2183     std::cout << " • Windows is different" << std::endl;
2184     std::cout << " • Use std::filesystem for portability" << std::endl;
2185
2186     std::cout << "\n GOLDEN RULES:" << std::endl;
2187     std::cout << " 1. Validate all external input" << std::endl;
2188     std::cout << " 2. Use modern C++ facilities (filesystem, string_view)"
```

```
        << std::endl;
2189     std::cout << "    3. Check errors always" << std::endl;
2190     std::cout << "    4. Think security first!" << std::endl;
2191 }
2192
2193 // =====
2194 // MAIN FUNCTION
2195 // =====
2196
2197 int main() {
2198     std::cout << "\n";
2199     std::cout << "
2200         \n";
2201     std::cout << "
2202         \n";
2203     std::cout << "
2204         \n";
2205     try {
2206         demonstrate_basic_execution();
2207         demonstrate_string_view_parsing();
2208         demonstrate_regex_parsing();
2209         demonstrate_filesystem();
2210         demonstrate_bash_scripts();
2211         demonstrate_python_scripts();
2212         demonstrate_linux_tools();
2213         demonstrate_file_streams();
2214         demonstrate_string_streams();
2215         demonstrate_stream_manipulators();
2216         demonstrate_binary_io();
2217         demonstrate_custom_operators();
2218         demonstrate_mutex_and_lock_guard();
2219         demonstrate_unique_lock();
2220         demonstrate_file_locking();
2221         demonstrate_race_condition();
2222         explain_security_considerations();
2223         explain_cross_platform();
2224         explain_best_practices();
2225
2226         std::cout << "\n" << std::string(70, '=') << std::endl;
2227         std::cout << "SUMMARY:\n";
2228         std::cout << std::string(70, '=') << std::endl;
2229
2230         std::cout << "\n KEY FEATURES DEMONSTRATED:" << std::endl;
2231         std::cout << "\n1. COMMAND EXECUTION:" << std::endl;
2232         std::cout << "    • popen() for output capture" << std::endl;
2233         std::cout << "    • Return code checking" << std::endl;
2234         std::cout << "    • RAII for resource management" << std::endl;
2235
2236         std::cout << "\n2. std::string_view (C++17):" << std::endl;
2237         std::cout << "    • Zero-copy parsing" << std::endl;
2238         std::cout << "    • Efficient tokenization" << std::endl;
```

```
2239     std::cout << " • Perfect for large outputs" << std::endl;
2240
2241     std::cout << "\n3. std::regex (C++11):" << std::endl;
2242     std::cout << " • Pattern matching" << std::endl;
2243     std::cout << " • Log parsing" << std::endl;
2244     std::cout << " • Network output parsing" << std::endl;
2245
2246     std::cout << "\n4. std::filesystem (C++17):" << std::endl;
2247     std::cout << " • Cross-platform paths" << std::endl;
2248     std::cout << " • Directory iteration" << std::endl;
2249     std::cout << " • Error code handling" << std::endl;
2250
2251     std::cout << "\n5. STREAM I/O:" << std::endl;
2252     std::cout << " • File streams (ifstream, ofstream, fstream)" << std
2253         ::endl;
2254     std::cout << " • String streams (ostringstream, istringstream)" <<
2255         std::endl;
2256     std::cout << " • Stream manipulators (setw, setprecision, etc.)" <<
2257         std::endl;
2258     std::cout << " • Binary I/O for POD types" << std::endl;
2259     std::cout << " • Custom stream operators" << std::endl;
2260
2261     std::cout << "\n6. SCRIPT INTEGRATION:" << std::endl;
2262     std::cout << " • Bash scripts with arguments" << std::endl;
2263     std::cout << " • Python scripts with JSON output" << std::endl;
2264     std::cout << " • Structured data exchange" << std::endl;
2265
2266     std::cout << "\n7. THREAD SYNCHRONIZATION:" << std::endl;
2267     std::cout << " • std::mutex and std::lock_guard (RAII)" << std::endl
2268         ;
2269     std::cout << " • std::unique_lock (flexible locking)" << std::endl;
2270     std::cout << " • std::shared_mutex (reader-writer locks)" << std::
2271         endl;
2272     std::cout << " • std::condition_variable (wait/notify)" << std::endl
2273         ;
2274     std::cout << " • OS-level file locking (flock)" << std::endl;
2275     std::cout << " • Race condition examples and fixes" << std::endl;
2276
2277     std::cout << "\n SECURITY REMINDERS:" << std::endl;
2278     std::cout << " • Never pass unsanitized user input to shell" << std
2279         ::endl;
2280     std::cout << " • Validate all paths and arguments" << std::endl;
2281     std::cout << " • Use absolute paths for commands" << std::endl;
2282     std::cout << " • Check return codes always" << std::endl;
2283
2284     std::cout << "\n Modern C++: Safe, efficient system programming!\n"
2285         << std::endl;
2286
2287 } catch (const std::exception& e) {
2288     std::cerr << "\n Error: " << e.what() << std::endl;
2289     return 1;
2290 }
2291
2292 return 0;
```

2285 }

## 70 Source Code: TemplatizedCameraInterface.cpp

File: src/TemplatedCameraInterface.cpp

Repository: [View on GitHub](#)

```
1 #include <iostream>
2 #include <array>
3 #include <vector>
4 #include <memory>
5 #include <cstdint>
6 #include <type_traits>
7 #include <algorithm>
8 #include <cmath>
9 #include <chrono>
10
11 // =====
12 // TEMPLATED CAMERA INTERFACE FOR MULTIPLE PIXEL TYPES
13 // =====
14 // Demonstrates C++ templates for camera interfacing where pixel data
15 // can be various types: uint8_t, uint16_t, float, double
16 // =====
17
18 // =====
19 // 1. BASIC IMAGE CLASS (TEMPLATED BY PIXEL TYPE)
20 // =====
21
22 template<typename PixelType>
23 class Image {
24 private:
25     size_t width;
26     size_t height;
27     std::vector<PixelType> pixels;
28
29 public:
30     Image(size_t w, size_t h)
31         : width(w), height(h), pixels(w * h) {}
32
33     Image(size_t w, size_t h, PixelType initial_value)
34         : width(w), height(h), pixels(w * h, initial_value) {}
35
36     // Accessors
37     size_t get_width() const { return width; }
38     size_t get_height() const { return height; }
39     size_t get_size() const { return pixels.size(); }
40
41     // Pixel access
42     PixelType& at(size_t x, size_t y) {
43         return pixels[y * width + x];
44     }
45
46     const PixelType& at(size_t x, size_t y) const {
47         return pixels[y * width + x];
48     }
49
```

```
50 // Raw data access (for camera hardware interface)
51 PixelType* data() { return pixels.data(); }
52 const PixelType* data() const { return pixels.data(); }
53
54 // Memory size in bytes
55 size_t memory_bytes() const {
56     return pixels.size() * sizeof(PixelType);
57 }
58
59 // Fill with value
60 void fill(PixelType value) {
61     std::fill(pixels.begin(), pixels.end(), value);
62 }
63 };
64
65 // =====
66 // 2. CAMERA INTERFACE (TEMPLATED)
67 // =====
68
69 template<typename PixelType>
70 class Camera {
71 private:
72     size_t width;
73     size_t height;
74     std::string camera_name;
75
76 public:
77     Camera(const std::string& name, size_t w, size_t h)
78         : camera_name(name), width(w), height(h) {}
79
80     virtual ~Camera() = default;
81
82     // Pure virtual: capture image
83     virtual Image<PixelType> capture() = 0;
84
85     // Configuration
86     size_t get_width() const { return width; }
87     size_t get_height() const { return height; }
88     const std::string& get_name() const { return camera_name; }
89
90     // Get pixel type information
91     static constexpr size_t bits_per_pixel() {
92         return sizeof(PixelType) * 8;
93     }
94
95     static constexpr bool is_floating_point() {
96         return std::is_floating_point_v<PixelType>;
97     }
98
99     static constexpr bool is_integer() {
100         return std::is_integral_v<PixelType>;
101     }
102 };
103 }
```

```
104 // =====
105 // 3. SIMULATED CAMERA IMPLEMENTATIONS
106 // =====
107
108 // 8-bit grayscale camera (most common)
109 class Camera8bit : public Camera<uint8_t> {
110 public:
111     Camera8bit(const std::string& name, size_t w, size_t h)
112         : Camera<uint8_t>(name, w, h) {}
113
114     Image<uint8_t> capture() override {
115         Image<uint8_t> img(get_width(), get_height());
116
117         // Simulate captured gradient pattern
118         for (size_t y = 0; y < get_height(); ++y) {
119             for (size_t x = 0; x < get_width(); ++x) {
120                 uint8_t value = static_cast<uint8_t>(
121                     (x * 255.0 / get_width()) * 0.5 +
122                     (y * 255.0 / get_height()) * 0.5
123                 );
124                 img.at(x, y) = value;
125             }
126         }
127         return img;
128     }
129 };
130
131 // 16-bit camera (scientific/medical imaging)
132 class Camera16bit : public Camera<uint16_t> {
133 public:
134     Camera16bit(const std::string& name, size_t w, size_t h)
135         : Camera<uint16_t>(name, w, h) {}
136
137     Image<uint16_t> capture() override {
138         Image<uint16_t> img(get_width(), get_height());
139
140         // Simulate high dynamic range data
141         for (size_t y = 0; y < get_height(); ++y) {
142             for (size_t x = 0; x < get_width(); ++x) {
143                 uint16_t value = static_cast<uint16_t>(
144                     (x * 65535.0 / get_width()) * 0.3 +
145                     (y * 65535.0 / get_height()) * 0.7
146                 );
147                 img.at(x, y) = value;
148             }
149         }
150         return img;
151     }
152 };
153
154 // Float camera (normalized values 0.0-1.0)
155 class CameraFloat : public Camera<float> {
156 public:
157     CameraFloat(const std::string& name, size_t w, size_t h)
```

```
158     : Camera<float>(name, w, h) {}
159
160     Image<float> capture() override {
161         Image<float> img(get_width(), get_height());
162
163         // Simulate normalized data with some pattern
164         for (size_t y = 0; y < get_height(); ++y) {
165             for (size_t x = 0; x < get_width(); ++x) {
166                 float value =
167                     0.5f + 0.5f * std::sin(x * 0.1f) * std::cos(y * 0.1f);
168                 img.at(x, y) = value;
169             }
170         }
171         return img;
172     }
173 };
174
175 // Double precision camera (research/astronomy)
176 class CameraDouble : public Camera<double> {
177 public:
178     CameraDouble(const std::string& name, size_t w, size_t h)
179         : Camera<double>(name, w, h) {}
180
181     Image<double> capture() override {
182         Image<double> img(get_width(), get_height());
183
184         // Simulate high-precision data
185         for (size_t y = 0; y < get_height(); ++y) {
186             for (size_t x = 0; x < get_width(); ++x) {
187                 double value =
188                     std::sin(x * 0.05) * std::cos(y * 0.05) +
189                     std::exp(-((x - get_width()/2.0) * (x - get_width()/2.0) +
190                               (y - get_height()/2.0) * (y - get_height()/2.0)
191                               ) / 1000.0);
192                 img.at(x, y) = value;
193             }
194         }
195         return img;
196     }
197
198 // =====
199 // 4. IMAGE PROCESSING ALGORITHMS (TEMPLATED)
200 // =====
201
202 template<typename PixelType>
203 class ImageProcessor {
204 public:
205     // Calculate average pixel value
206     static double calculate_mean(const Image<PixelType>& img) {
207         double sum = 0.0;
208         const PixelType* data = img.data();
209         size_t size = img.get_size();
210     }
```

```
211     for (size_t i = 0; i < size; ++i) {
212         sum += static_cast<double>(data[i]);
213     }
214
215     return sum / size;
216 }
217
218 // Find min and max pixel values
219 static std::pair<PixelType, PixelType> find_min_max(const Image<PixelType>
220 &img) {
221     const PixelType* data = img.data();
222     size_t size = img.get_size();
223
224     PixelType min_val = data[0];
225     PixelType max_val = data[0];
226
227     for (size_t i = 1; i < size; ++i) {
228         if (data[i] < min_val) min_val = data[i];
229         if (data[i] > max_val) max_val = data[i];
230     }
231
232     return {min_val, max_val};
233 }
234
235 // Scale pixel values
236 static Image<PixelType> scale(const Image<PixelType>& img, double factor)
237 {
238     Image<PixelType> result(img.get_width(), img.get_height());
239
240     for (size_t y = 0; y < img.get_height(); ++y) {
241         for (size_t x = 0; x < img.get_width(); ++x) {
242             double scaled = static_cast<double>(img.at(x, y)) * factor;
243             result.at(x, y) = static_cast<PixelType>(scaled);
244         }
245     }
246
247     return result;
248 }
249
250 // Threshold operation
251 static Image<PixelType> threshold(const Image<PixelType>& img, PixelType
252 threshold_value) {
253     Image<PixelType> result(img.get_width(), img.get_height());
254
255     for (size_t y = 0; y < img.get_height(); ++y) {
256         for (size_t x = 0; x < img.get_width(); ++x) {
257             result.at(x, y) = (img.at(x, y) >= threshold_value) ?
258                 threshold_value : PixelType(0);
259         }
260     }
261
262     return result;
263 }
264};
```

```
262 // =====
263 // 5. TYPE CONVERSION BETWEEN PIXEL TYPES
264 // =====
265
266 template<typename DestType, typename SrcType>
267 Image<DestType> convert_image(const Image<SrcType>& src) {
268     Image<DestType> dest(src.get_width(), src.get_height());
269
270     // Find source range for proper scaling
271     auto [min_val, max_val] = ImageProcessor<SrcType>::find_min_max(src);
272     double src_range = static_cast<double>(max_val) - static_cast<double>(min_val);
273
274     // Determine destination range
275     double dest_min, dest_max;
276     if constexpr (std::is_floating_point_v<DestType>) {
277         dest_min = 0.0;
278         dest_max = 1.0;
279     } else if constexpr (std::is_same_v<DestType, uint8_t>) {
280         dest_min = 0.0;
281         dest_max = 255.0;
282     } else if constexpr (std::is_same_v<DestType, uint16_t>) {
283         dest_min = 0.0;
284         dest_max = 65535.0;
285     } else {
286         dest_min = 0.0;
287         dest_max = 1.0;
288     }
289
290     dest_range = dest_max - dest_min;
291
292     // Convert with proper scaling
293     for (size_t y = 0; y < src.get_height(); ++y) {
294         for (size_t x = 0; x < src.get_width(); ++x) {
295             double normalized = (static_cast<double>(src.at(x, y)) -
296                 static_cast<double>(min_val)) / src_range;
297             double scaled = normalized * dest_range + dest_min;
298             dest.at(x, y) = static_cast<DestType>(scaled);
299         }
300     }
301
302     return dest;
303 }
304
305 // =====
306 // 6. GENERIC CAMERA HANDLER (WORKS WITH ANY PIXEL TYPE)
307 // =====
308
309 template<typename PixelType>
310 class CameraHandler {
311     private:
312         std::unique_ptr<Camera<PixelType>> camera;
```

```
314 | public:
315 |     CameraHandler(std::unique_ptr<Camera<PixelType>> cam)
316 |         : camera(std::move(cam)) {}
317 |
318 |     void display_camera_info() {
319 |         std::cout << "\n  Camera: " << camera->get_name() << std::endl;
320 |         std::cout << "  Resolution: " << camera->get_width() << "x" << camera
321 |             ->get_height() << std::endl;
322 |         std::cout << "  Pixel type: " << typeid(PixelType).name() << std::endl
323 |             ;
324 |         std::cout << "  Bits per pixel: " << Camera<PixelType>::bits_per_pixel
325 |             () << std::endl;
326 |         std::cout << "  Floating point: " << (Camera<PixelType>::
327 |             is_floating_point() ? "Yes" : "No") << std::endl;
328 |
329 |         // Calculate memory per frame
330 |         size_t bytes = camera->get_width() * camera->get_height() * sizeof(
331 |             PixelType);
332 |         std::cout << "  Memory per frame: " << bytes << " bytes";
333 |         if (bytes >= 1024 * 1024) {
334 |             std::cout << "(" << bytes / (1024.0 * 1024.0) << " MB)";
335 |         } else if (bytes >= 1024) {
336 |             std::cout << "(" << bytes / 1024.0 << " KB)";
337 |         }
338 |         std::cout << std::endl;
339 |
340 |     void capture_and_process() {
341 |         std::cout << "\n  Capturing image..." << std::endl;
342 |         auto start = std::chrono::high_resolution_clock::now();
343 |
344 |         Image<PixelType> img = camera->capture();
345 |
346 |         auto end = std::chrono::high_resolution_clock::now();
347 |         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(
348 |             end - start);
349 |         std::cout << "  Capture time: " << duration.count() << " us" << std::
350 |             endl;
351 |
352 |         // Process image
353 |         std::cout << "\n  Processing image..." << std::endl;
354 |         double mean = ImageProcessor<PixelType>::calculate_mean(img);
355 |         auto [min_val, max_val] = ImageProcessor<PixelType>::find_min_max(img)
356 |             ;
357 |
358 |         std::cout << "  Mean value: " << mean << std::endl;
359 |         std::cout << "  Min value: " << static_cast<double>(min_val) << std::
360 |             endl;
361 |         std::cout << "  Max value: " << static_cast<double>(max_val) << std::
362 |             endl;
363 |
364 |         // Sample some pixel values
365 |         std::cout << "\n  Sample pixels:" << std::endl;
366 |         std::cout << "    Top-left (0,0): " << static_cast<double>(img.at(0,
```

```
358     0)) << std::endl;
359     std::cout << "    Center (" << img.get_width()/2 << "," << img.
360     get_height()/2 << "): "
361         << static_cast<double>(img.at(img.get_width()/2, img.
362             get_height()/2)) << std::endl;
363     std::cout << "    Bottom-right (" << img.get_width()-1 << "," << img.
364             get_height()-1 << "): "
365         << static_cast<double>(img.at(img.get_width()-1, img.
366             get_height()-1)) << std::endl;
367     }
368 };
369
370 // =====
371 // 7. DEMONSTRATION FUNCTIONS
372 // =====
373
374 void demonstrate_8bit_camera() {
375     std::cout << "\n== 1. 8-BIT GRayscale Camera (uint8_t) ==" << std::endl;
376     std::cout << "Common in: Webcams, industrial cameras, surveillance" << std
377         ::endl;
378
379     CameraHandler<uint8_t> handler(
380         std::make_unique<Camera8bit>("Webcam HD", 640, 480)
381     );
382
383     handler.display_camera_info();
384     handler.capture_and_process();
385 }
386
387 void demonstrate_16bit_camera() {
388     std::cout << "\n== 2. 16-Bit Camera (uint16_t) ==" << std::endl;
389     std::cout << "Common in: Scientific imaging, medical X-ray, astronomy" <<
390         std::endl;
391
392     CameraHandler<uint16_t> handler(
393         std::make_unique<Camera16bit>("Scientific CCD", 1024, 1024)
394     );
395
396     handler.display_camera_info();
397     handler.capture_and_process();
398 }
399
400 void demonstrate_float_camera() {
401     std::cout << "\n== 3. FLOAT Camera (32-bit float) ==" << std::endl;
402     std::cout << "Common in: HDR imaging, computational photography, depth
403         sensors" << std::endl;
404
405     CameraHandler<float> handler(
406         std::make_unique<CameraFloat>("ToF Depth Camera", 320, 240)
407     );
408
409     handler.display_camera_info();
410     handler.capture_and_process();
411 }
```

```
404
405 void demonstrate_double_camera() {
406     std::cout << "\n==== 4. DOUBLE PRECISION CAMERA (64-bit double) ===" << std
407         ::endl;
408     std::cout << "Common in: Research, astronomical imaging, spectroscopy" <<
409         std::endl;
410
411     CameraHandler<double> handler(
412         std::make_unique<CameraDouble>("Telescope CCD", 512, 512)
413     );
414
415     handler.display_camera_info();
416     handler.capture_and_process();
417 }
418
419
420 void demonstrate_type_conversion() {
421     std::cout << "\n==== 5. IMAGE TYPE CONVERSION ===" << std::endl;
422
423     // Capture 8-bit image
424     Camera8bit cam8("Source Camera", 320, 240);
425     Image<uint8_t> img8 = cam8.capture();
426
427     std::cout << "\nOriginal 8-bit image:" << std::endl;
428     std::cout << "  Size: " << img8.memory_bytes() << " bytes" << std::endl;
429     auto [min8, max8] = ImageProcessor<uint8_t>::find_min_max(img8);
430     std::cout << "  Range: " << (int)min8 << " to " << (int)max8 << std::endl;
431
432     // Convert to 16-bit
433     std::cout << "\nConverting to 16-bit..." << std::endl;
434     Image<uint16_t> img16 = convert_image<uint16_t>(img8);
435     std::cout << "  Size: " << img16.memory_bytes() << " bytes" << std::endl;
436     auto [min16, max16] = ImageProcessor<uint16_t>::find_min_max(img16);
437     std::cout << "  Range: " << min16 << " to " << max16 << std::endl;
438
439     // Convert to float
440     std::cout << "\nConverting to float..." << std::endl;
441     Image<float> imgf = convert_image<float>(img8);
442     std::cout << "  Size: " << imgf.memory_bytes() << " bytes" << std::endl;
443     auto [minf, maxf] = ImageProcessor<float>::find_min_max(imgf);
444     std::cout << "  Range: " << minf << " to " << maxf << std::endl;
445
446     // Convert to double
447     std::cout << "\nConverting to double..." << std::endl;
448     Image<double> imgd = convert_image<double>(img8);
449     std::cout << "  Size: " << imgd.memory_bytes() << " bytes" << std::endl;
450     auto [mind, maxd] = ImageProcessor<double>::find_min_max(imgd);
451     std::cout << "  Range: " << mind << " to " << maxd << std::endl;
452 }
453
454 void demonstrate_processing_algorithms() {
455     std::cout << "\n==== 6. IMAGE PROCESSING ALGORITHMS ===" << std::endl;
456
457     Camera8bit cam("Processing Camera", 100, 100);
458     Image<uint8_t> img = cam.capture();
```

```
456
457     std::cout << "\nOriginal image statistics:" << std::endl;
458     double mean = ImageProcessor<uint8_t>::calculate_mean(img);
459     auto [min_val, max_val] = ImageProcessor<uint8_t>::find_min_max(img);
460     std::cout << "  Mean: " << mean << std::endl;
461     std::cout << "  Min: " << (int)min_val << ", Max: " << (int)max_val << std
462         ::endl;
463
464     // Scale by 2x
465     std::cout << "\nScaling by 2x:" << std::endl;
466     Image<uint8_t> scaled = ImageProcessor<uint8_t>::scale(img, 2.0);
467     double mean_scaled = ImageProcessor<uint8_t>::calculate_mean(scaled);
468     std::cout << "  Mean after scaling: " << mean_scaled << std::endl;
469
470     // Threshold
471     std::cout << "\nThresholding at 128:" << std::endl;
472     Image<uint8_t> thresholded = ImageProcessor<uint8_t>::threshold(img, 128);
473     size_t above_threshold = 0;
474     for (size_t i = 0; i < thresholded.get_size(); ++i) {
475         if (thresholded.data()[i] > 0) ++above_threshold;
476     }
477     std::cout << "  Pixels above threshold: " << above_threshold
478         << " (" << (100.0 * above_threshold / thresholded.get_size()) <<
479             "%)" << std::endl;
480 }
481
482 // =====
483 // MAIN FUNCTION
484 // =====
485
486 int main() {
487     std::cout << "\n
488         =====
489         =====" <<
490         std::endl;
491     std::cout << "  TEMPLATED CAMERA INTERFACE FOR MULTIPLE PIXEL TYPES" <<
492         std::endl;
493     std::cout << "
494         =====
495         =====" <<
496         std::endl;
497     demonstrate_8bit_camera();
498     demonstrate_16bit_camera();
499     demonstrate_float_camera();
500     demonstrate_double_camera();
501     demonstrate_type_conversion();
502     demonstrate_processing_algorithms();
503
504     std::cout << "\n
505         =====
506         =====" <<
507         std::endl;
508     std::cout << "  TEMPLATE BENEFITS FOR CAMERA INTERFACING" << std::endl;
509     std::cout << "
510         =====
511         =====" <<
512         std::endl;
```

```
499
500     std::cout << "\n ADVANTAGES OF TEMPLATES:" << std::endl;
501     std::cout << "\n1. TYPE SAFETY" << std::endl;
502     std::cout << " • Compile-time type checking" << std::endl;
503     std::cout << " • No runtime casting needed" << std::endl;
504     std::cout << " • Prevents mixing incompatible pixel types" << std::endl;
505
506     std::cout << "\n2. ZERO OVERHEAD" << std::endl;
507     std::cout << " • No virtual function calls" << std::endl;
508     std::cout << " • All types resolved at compile time" << std::endl;
509     std::cout << " • Fully optimizable by compiler" << std::endl;
510
511     std::cout << "\n3. CODE REUSABILITY" << std::endl;
512     std::cout << " • Write processing algorithms once" << std::endl;
513     std::cout << " • Works with uint8_t, uint16_t, float, double" << std::endl;
514     std::cout << " • Easy to add new pixel types" << std::endl;
515
516     std::cout << "\n4. COMPILE-TIME INFORMATION" << std::endl;
517     std::cout << " • sizeof(PixelType) known at compile time" << std::endl;
518     std::cout << " • Type traits (is_floating_point, etc.)" << std::endl;
519     std::cout << " • if constexpr for type-specific code" << std::endl;
520
521     std::cout << "\n MEMORY COMPARISON (640x480 image):" << std::endl;
522     size_t pixels = 640 * 480;
523     std::cout << "     uint8_t: " << pixels * sizeof(uint8_t) / 1024 << " KB" <<
524         std::endl;
525     std::cout << "     uint16_t: " << pixels * sizeof(uint16_t) / 1024 << " KB" <<
526         std::endl;
527     std::cout << "     float: " << pixels * sizeof(float) / 1024 << " KB" <<
528         std::endl;
529     std::cout << "     double: " << pixels * sizeof(double) / 1024 << " KB" <<
530         std::endl;
531
532     std::cout << "\n REAL-WORLD APPLICATIONS:" << std::endl;
533     std::cout << "\n     uint8_t (8-bit):" << std::endl;
534     std::cout << " • Consumer webcams" << std::endl;
535     std::cout << " • Surveillance cameras" << std::endl;
536     std::cout << " • Industrial inspection" << std::endl;
537     std::cout << " • Video streaming" << std::endl;
538
539     std::cout << "\n     uint16_t (16-bit):" << std::endl;
540     std::cout << " • Medical X-ray imaging" << std::endl;
541     std::cout << " • Scientific CCD cameras" << std::endl;
542     std::cout << " • Astronomy (star imaging)" << std::endl;
543     std::cout << " • High dynamic range photography" << std::endl;
544
545     std::cout << "\n     float (32-bit):" << std::endl;
546     std::cout << " • Time-of-Flight (ToF) depth sensors" << std::endl;
547     std::cout << " • HDR image processing" << std::endl;
548     std::cout << " • Computational photography" << std::endl;
549     std::cout << " • Normalized image data" << std::endl;
550
551     std::cout << "\n     double (64-bit):" << std::endl;
```

```
548     std::cout << " • Radio telescope data" << std::endl;
549     std::cout << " • Spectroscopy" << std::endl;
550     std::cout << " • High-precision research" << std::endl;
551     std::cout << " • Scientific analysis" << std::endl;
552
553     std::cout << "\n TEMPLATE TECHNIQUES DEMONSTRATED:" << std::endl;
554     std::cout << " • Class templates: Image<T>, Camera<T>" << std::endl;
555     std::cout << " • Function templates: convert_image<DestType, SrcType>" <<
556         std::endl;
556     std::cout << " • Type traits: std::is_floating_point_v<T>" << std::endl;
557     std::cout << " • if constexpr: Compile-time conditional code" << std::
558         endl;
558     std::cout << " • Template specialization: Type-specific behavior" << std
559         ::endl;
559     std::cout << " • constexpr functions: Compile-time computation" << std::
560         endl;
560
561     std::cout << "\n"
562         ======\n" <<
563         std::endl;
564
564 }
```

## 71 Source Code: TemplatizedCameraModules.cpp

File: src/TemplatedCameraModules.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // C++20 MODULES DEMONSTRATION: IMPORT AND USE CAMERA MODULE
3 // =====
4 // This file demonstrates:
5 // 1. Importing C++20 modules with "import"
6 // 2. Using exported templates, classes, and concepts
7 // 3. C++20 features: concepts, requires clauses, [[nodiscard]]
8 // =====
9
10 #include <iostream>
11 #include <iomanip>
12 #include <chrono>
13
14 // C++20 MODULE IMPORT
15 import camera;
16
17 // =====
18 // DEMONSTRATION FUNCTIONS
19 // =====
20
21 void demonstrate_concepts() {
22     std::cout << "\n==== C++20 CONCEPTS DEMONSTRATION ===" << std::endl;
23
24     std::cout << "\nPixelType concept checks:" << std::endl;
25     std::cout << "    uint8_t is PixelType: " << PixelType<uint8_t> << std::endl
26         ;
27     std::cout << "    uint16_t is PixelType: " << PixelType<uint16_t> << std::endl
28         ;
29     std::cout << "    float is PixelType: " << PixelType<float> << std::endl;
30     std::cout << "    double is PixelType: " << PixelType<double> << std::endl;
31     std::cout << "    int is PixelType: " << PixelType<int> << " (not in allowed
32         list)" << std::endl;
33
34     std::cout << "\nIntegerPixel concept checks:" << std::endl;
35     std::cout << "    uint8_t is IntegerPixel: " << IntegerPixel<uint8_t> << std::endl
36         ;
37     std::cout << "    float is IntegerPixel: " << IntegerPixel<float> << std::endl
38         ;
39
40     std::cout << "\nFloatingPixel concept checks:" << std::endl;
41     std::cout << "    float is FloatingPixel: " << FloatingPixel<float> << std::endl
42         ;
43     std::cout << "    uint8_t is FloatingPixel: " << FloatingPixel<uint8_t> <<
44         std::endl;
45 }
46
47 template<PixelType T>
48 void display_camera_info(const Camera<T>& camera) {
49     std::cout << "\n    Camera: " << camera.get_name() << std::endl;
```

```
43     std::cout << " Resolution: " << camera.get_width() << "x" << camera.
44         get_height() << std::endl;
45     std::cout << " Bits per pixel: " << Camera<T>::bits_per_pixel() << std::
46         endl;
47     std::cout << " Floating point: " << (Camera<T>::is_floating_point() ? "
48         Yes" : "No") << std::endl;
49
50     size_t bytes = camera.get_width() * camera.get_height() * sizeof(T);
51     std::cout << " Memory per frame: " << bytes << " bytes";
52     if (bytes >= 1024 * 1024) {
53         std::cout << "(" << std::fixed << std::setprecision(2)
54             << bytes / (1024.0 * 1024.0) << " MB)";
55     } else if (bytes >= 1024) {
56         std::cout << "(" << std::fixed << std::setprecision(2)
57             << bytes / 1024.0 << " KB)";
58     }
59     std::cout << std::endl;
60 }
61
62 template<PixelType T>
63 void demonstrate_camera_capture(CameraHandler<T>& handler) {
64     const Camera<T>* cam = handler.get_camera();
65
66     std::cout << "\n  Capturing image..." << std::endl;
67     auto start = std::chrono::high_resolution_clock::now();
68
69     Image<T> img = handler.capture();
70
71     auto end = std::chrono::high_resolution_clock::now();
72     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end
73         - start);
74     std::cout << " Capture time: " << duration.count() << " µs" << std::endl;
75
76     std::cout << "\n  Processing image..." << std::endl;
77     double mean = ImageProcessor<T>::calculate_mean(img);
78     auto [min_val, max_val] = ImageProcessor<T>::find_min_max(img);
79
80     std::cout << "  Mean value: " << mean << std::endl;
81     std::cout << "  Min value: " << static_cast<double>(min_val) << std::endl;
82     std::cout << "  Max value: " << static_cast<double>(max_val) << std::endl;
83
84     std::cout << "\n  Sample pixels:" << std::endl;
85     std::cout << "    Top-left (0,0): " << static_cast<double>(img.at(0, 0))
86         << std::endl;
87     std::cout << "    Center (" << img.get_width()/2 << "," << img.get_height()
88         ()/2 << "): "
89         << static_cast<double>(img.at(img.get_width()/2, img.get_height()
90             ()/2)) << std::endl;
91 }
92
93 void demonstrate_8bit_camera() {
94     std::cout << "\n==== 1. 8-BIT CAMERA (uint8_t) ===" << std::endl;
95
96     CameraHandler<uint8_t> handler(
```

```
90     std::make_unique<Camera8bit>("Webcam HD", 640, 480)
91 );
92
93     display_camera_info(*handler.get_camera());
94     demonstrate_camera_capture(handler);
95 }
96
97 void demonstrate_16bit_camera() {
98     std::cout << "\n== 2. 16-BIT CAMERA (uint16_t) ==" << std::endl;
99
100    CameraHandler<uint16_t> handler(
101        std::make_unique<Camera16bit>("Scientific CCD", 1024, 1024)
102    );
103
104    display_camera_info(*handler.get_camera());
105    demonstrate_camera_capture(handler);
106 }
107
108 void demonstrate_float_camera() {
109     std::cout << "\n== 3. FLOAT CAMERA (32-bit) ==" << std::endl;
110
111    CameraHandler<float> handler(
112        std::make_unique<CameraFloat>("ToF Depth Camera", 320, 240)
113    );
114
115    display_camera_info(*handler.get_camera());
116    demonstrate_camera_capture(handler);
117 }
118
119 void demonstrate_double_camera() {
120     std::cout << "\n== 4. DOUBLE CAMERA (64-bit) ==" << std::endl;
121
122    CameraHandler<double> handler(
123        std::make_unique<CameraDouble>("Telescope CCD", 512, 512)
124    );
125
126    display_camera_info(*handler.get_camera());
127    demonstrate_camera_capture(handler);
128 }
129
130 void demonstrate_type_conversion() {
131     std::cout << "\n== 5. IMAGE TYPE CONVERSION ==" << std::endl;
132
133     Camera8bit cam8("Source Camera", 320, 240);
134     Image<uint8_t> img8 = cam8.capture();
135
136     std::cout << "\nOriginal 8-bit image:" << std::endl;
137     std::cout << "  Size: " << img8.memory_bytes() << " bytes" << std::endl;
138     auto [min8, max8] = ImageProcessor<uint8_t>::find_min_max(img8);
139     std::cout << "  Range: " << (int)min8 << " to " << (int)max8 << std::endl;
140
141     std::cout << "\nConverting to 16-bit..." << std::endl;
142     Image<uint16_t> img16 = convert_image<uint16_t>(img8);
143     std::cout << "  Size: " << img16.memory_bytes() << " bytes" << std::endl;
```

```
144     auto [min16, max16] = ImageProcessor<uint16_t>::find_min_max(img16);
145     std::cout << "  Range: " << min16 << " to " << max16 << std::endl;
146
147     std::cout << "\nConverting to float..." << std::endl;
148     Image<float> imgf = convert_image<float>(img8);
149     std::cout << "  Size: " << imgf.memory_bytes() << " bytes" << std::endl;
150     auto [minf, maxf] = ImageProcessor<float>::find_min_max(imgf);
151     std::cout << "  Range: " << minf << " to " << maxf << std::endl;
152
153     std::cout << "\nConverting to double..." << std::endl;
154     Image<double> imgd = convert_image<double>(img8);
155     std::cout << "  Size: " << imgd.memory_bytes() << " bytes" << std::endl;
156     auto [mind, maxd] = ImageProcessor<double>::find_min_max(imgd);
157     std::cout << "  Range: " << mind << " to " << maxd << std::endl;
158 }
159
160 void demonstrate_concept_constrained_operations() {
161     std::cout << "\n==== 6. CONCEPT-CONSTRAINED OPERATIONS ===" << std::endl;
162
163     // Integer pixel operations
164     std::cout << "\nInteger pixel operations (uint8_t):" << std::endl;
165     Camera8bit cam8("Integer Camera", 100, 100);
166     Image<uint8_t> img8 = cam8.capture();
167
168     std::cout << "  Original mean: " << ImageProcessor<uint8_t>::
169     calculate_mean(img8) << std::endl;
170
171     // Threshold only works with IntegerPixel concept
172     Image<uint8_t> thresholded = ImageProcessor<uint8_t>::threshold(img8, 128)
173     ;
174     size_t above = 0;
175     for (size_t i = 0; i < thresholded.get_size(); ++i) {
176         if (thresholded.data()[i] > 0) ++above;
177     }
178     std::cout << "  Pixels above threshold: " << above
179     << " (" << (100.0 * above / thresholded.get_size()) << "%)" <<
180     std::endl;
181
182     // Floating point operations
183     std::cout << "\nFloating pixel operations (float):" << std::endl;
184     CameraFloat camf("Float Camera", 100, 100);
185     Image<float> imgf = camf.capture();
186
187     std::cout << "  Original mean: " << ImageProcessor<float>::calculate_mean(
188     imgf) << std::endl;
189
190     // Normalize only works with FloatingPixel concept
191     Image<float> normalized = ImageProcessor<float>::normalize(imgf);
192     std::cout << "  Normalized mean: " << ImageProcessor<float>::
193     calculate_mean(normalized) << std::endl;
194     auto [minf, maxf] = ImageProcessor<float>::find_min_max(normalized);
195     std::cout << "  Normalized range: " << minf << " to " << maxf << std::endl;
196     ;
```

```
192 // Note: The following would NOT compile (concept constraint):  
193 // Image<float> bad = ImageProcessor<float>::threshold(imgf, 0.5f); //  
194 //   ERROR: requires IntegerPixel  
195 // Image<uint8_t> bad2 = ImageProcessor<uint8_t>::normalize(img8); //  
196 //   ERROR: requires FloatingPixel  
197  
198 // =====  
199 // MAIN FUNCTION  
200 // =====  
201  
202 int main() {  
203     std::cout << "\n  
204         =====" <<  
205         std::endl;  
206     std::cout << "  C++20 MODULES: TEMPLATED CAMERA INTERFACE" << std::endl;  
207     std::cout << "\n  
208         =====" <<  
209         std::endl;  
210     demonstrate_concepts();  
211     demonstrate_8bit_camera();  
212     demonstrate_16bit_camera();  
213     demonstrate_float_camera();  
214     demonstrate_double_camera();  
215     demonstrate_type_conversion();  
216     demonstrate_concept_constrained_operations();  
217  
218     std::cout << "\n  
219         =====" <<  
220         std::endl;  
221     std::cout << "  C++20 FEATURES DEMONSTRATED" << std::endl;  
222     std::cout << "\n  
223         =====" <<  
224         std::endl;  
225     std::cout << "\n MODULES (import/export):" << std::endl;  
226     std::cout << " •  export module camera - Module interface declaration" <<  
227         std::endl;  
228     std::cout << " •  import camera - Import module in main program" << std::endl;  
229     std::cout << " •  Exported classes: Image<T>, Camera<T>, ImageProcessor<T  
230         >" << std::endl;  
231     std::cout << " •  Exported functions: convert_image<T1, T2>()" << std::endl;  
232     std::cout << " •  Exported concepts: PixelType, IntegerPixel,  
233         FloatingPixel" << std::endl;  
234  
235     std::cout << "\n CONCEPTS:" << std::endl;  
236     std::cout << " •  concept PixelType - Constrain allowed pixel types" <<  
237         std::endl;
```

```
229     std::cout << " • concept IntegerPixel - Only integer pixel types" << std::endl;
230     std::cout << " • concept FloatingPixel - Only floating-point pixel types" << std::endl;
231     std::cout << " • requires clauses on functions (threshold, normalize)" << std::endl;
232     std::cout << " • Compile-time type checking prevents invalid operations" << std::endl;
233
234     std::cout << "\n OTHER C++20 FEATURES:" << std::endl;
235     std::cout << " • [[nodiscard]] - Warn if return value ignored" << std::endl;
236     std::cout << " • Three-way comparison (operator<=) - Auto-generated comparisons" << std::endl;
237     std::cout << " • noexcept specifications - Exception safety guarantees" << std::endl;
238
239     std::cout << "\n MODULE BENEFITS:" << std::endl;
240     std::cout << " • Faster compilation - No header parsing overhead" << std::endl;
241     std::cout << " • Better encapsulation - Only exported entities visible" << std::endl;
242     std::cout << " • No include guards needed - Modules included once automatically" << std::endl;
243     std::cout << " • Order independent - No macro contamination" << std::endl;
244     std::cout << " • Cleaner code - Explicit export declarations" << std::endl;
245
246     std::cout << "\n COMPILATION NOTES:" << std::endl;
247     std::cout << "     GCC 11+:" << std::endl;
248     std::cout << "         g++ -std=c++20 -fmodules-ts -xc++-system-header iostream" << std::endl;
249     std::cout << "         g++ -std=c++20 -fmodules-ts -c CameraModule.cppm" << std::endl;
250     std::cout << "         g++ -std=c++20 -fmodules-ts TemplatedCameraModules.cpp CameraModule.o" << std::endl;
251
252     std::cout << "\n     MSVC 2022+:" << std::endl;
253     std::cout << "         cl /std:c++20 /experimental:module /c CameraModule.cppm" << std::endl;
254     std::cout << "         cl /std:c++20 /experimental:module TemplatedCameraModules.cpp CameraModule.obj" << std::endl;
255
256     std::cout << "\n     Clang 16+:" << std::endl;
257     std::cout << "         clang++ -std=c++20 -fmodules -c CameraModule.cppm" << std::endl;
258     std::cout << "         clang++ -std=c++20 -fmodules TemplatedCameraModules.cpp CameraModule.o" << std::endl;
259
260     std::cout << "\n     NOTE: C++20 modules support varies by compiler!" << std::endl;
261     std::cout << " • GCC: Experimental support with -fmodules-ts" << std::endl;
```

```
262     std::cout << " • MSVC: Best support with /std:c++20 and /experimental:  
263         module" << std::endl;  
264     std::cout << " • Clang: Good support in recent versions (16+)" << std::  
265         endl;  
266  
267     std::cout << "\n  
268         =====\n" << std::endl;  
269  
270     return 0;  
271 }
```

## 72 Source Code: ThreadPoolExamples.cpp

File: src/ThreadPoolExamples.cpp

Repository: [View on GitHub](#)

```
1 // ThreadPoolExamples.cpp
2 // Comprehensive educational examples of thread pool implementations in Modern
3 // C++
4 // From basic to advanced patterns including work stealing, priority queues,
5 // and more
6
7 #include <iostream>
8 #include <thread>
9 #include <vector>
10 #include <queue>
11 #include <functional>
12 #include <mutex>
13 #include <condition_variable>
14 #include <future>
15 #include <atomic>
16 #include <chrono>
17 #include <memory>
18 #include <type_traits>
19 #include <deque>
20 #include <optional>
21 #include <iomanip>
22
23 // =====
24 // SECTION 1: Basic Thread Pool (Simplest Implementation)
25 // =====
26
27 class BasicThreadPool {
28 private:
29     std::vector<std::thread> workers_;
30     std::queue<std::function<void()>> tasks_;
31     std::mutex queue_mutex_;
32     std::condition_variable condition_;
33     bool stop_;
34
35 public:
36     explicit BasicThreadPool(size_t num_threads) : stop_(false) {
37         std::cout << "  Creating thread pool with " << num_threads << "
38             workers\n";
39
40         for (size_t i = 0; i < num_threads; ++i) {
41             workers_.emplace_back([this, i]() {
42                 std::cout << "    Worker " << i << " started (thread "
43                     << std::this_thread::get_id() << ")\n";
44             });
45         }
46     }
47
48     void add_task(std::function<void()> task) {
49         std::lock_guard<std::mutex> lock(queue_mutex_);
50         tasks_.push(task);
51         condition_.notify_one();
52     }
53
54     void stop() {
55         stop_ = true;
56         condition_.notify_all();
57     }
58
59     void join() {
60         for (auto& worker : workers_) {
61             if (worker.joinable())
62                 worker.join();
63         }
64     }
65 }
```

```
43     while (true) {
44         std::function<void()> task;
45
46         {
47             std::unique_lock<std::mutex> lock(queue_mutex_);
48
49             // Wait until there's work or we're stopping
50             condition_.wait(lock, [this]() {
51                 return stop_ || !tasks_.empty();
52             });
53
54             if (stop_ && tasks_.empty()) {
55                 return;
56             }
57
58             task = std::move(tasks_.front());
59             tasks_.pop();
60         }
61
62         task(); // Execute the task
63     }
64 }
65 }
66 }
67 }
68
69 // Submit a task (no return value)
70 void submit(std::function<void()> task) {
71 {
72     std::lock_guard<std::mutex> lock(queue_mutex_);
73
74     if (stop_) {
75         throw std::runtime_error("Cannot submit task to stopped thread
76             pool");
77     }
78
79     tasks_.push(std::move(task));
80     condition_.notify_one();
81 }
82
83 ~BasicThreadPool() {
84 {
85     std::lock_guard<std::mutex> lock(queue_mutex_);
86     stop_ = true;
87 }
88
89     condition_.notify_all();
90
91     for (std::thread& worker : workers_) {
92         if (worker.joinable()) {
93             worker.join();
94         }
95     }
96 }
```

```
96         std::cout << "  Thread pool destroyed\n";
97     }
98 };
99
100
101 void demonstrate_basic_thread_pool() {
102     std::cout << "\n" << std::string(70, '=') << "\n";
103     std::cout << "==== 1. Basic Thread Pool ===\n";
104     std::cout << std::string(70, '=') << "\n\n";
105
106     std::cout << "Concept: Fixed number of worker threads processing tasks
107         from a queue\n\n";
108
109     BasicThreadPool pool(3);
110
111     std::cout << "\nSubmitting 6 tasks...\n\n";
112
113     for (int i = 1; i <= 6; ++i) {
114         pool.submit([i]() {
115             std::cout << "      Task " << i << " executing on thread "
116                         << std::this_thread::get_id() << "\n";
117             std::this_thread::sleep_for(200ms);
118             std::cout << "      Task " << i << " completed\n";
119         });
120
121     std::this_thread::sleep_for(2s); // Wait for tasks to complete
122
123     std::cout << "\n All tasks processed by 3 workers\n";
124     std::cout << " Tasks automatically distributed across available threads\n
125         ";
126 }
127
128 // SECTION 2: Thread Pool with Futures (Return Values)
129 // =====
130
131 class ThreadPoolWithFutures {
132 private:
133     std::vector<std::thread> workers_;
134     std::queue<std::function<void()>> tasks_;
135     std::mutex queue_mutex_;
136     std::condition_variable condition_;
137     bool stop_;
138
139 public:
140     explicit ThreadPoolWithFutures(size_t num_threads) : stop_(false) {
141         for (size_t i = 0; i < num_threads; ++i) {
142             workers_.emplace_back([this]() {
143                 while (true) {
```

```
144         std::function<void()> task;
145
146         {
147             std::unique_lock<std::mutex> lock(queue_mutex_);
148             condition_.wait(lock, [this]() {
149                 return stop_ || !tasks_.empty();
150             });
151
152             if (stop_ && tasks_.empty()) {
153                 return;
154             }
155
156             task = std::move(tasks_.front());
157             tasks_.pop();
158         }
159
160         task();
161     }
162 }
163
164 }
165
166 // Submit task and get future for result
167 template<typename F, typename... Args>
168 auto submit(F&& f, Args&&... args)
169     -> std::future<typename std::invoke_result_t<F, Args...>> {
170
171     using return_type = typename std::invoke_result_t<F, Args...>;
172
173     // Create a packaged_task
174     auto task = std::make_shared<std::packaged_task<return_type()>>(
175         std::bind(std::forward<F>(f), std::forward<Args>(args)...));
176
177
178     std::future<return_type> result = task->get_future();
179
180     {
181         std::lock_guard<std::mutex> lock(queue_mutex_);
182
183         if (stop_) {
184             throw std::runtime_error("Cannot submit to stopped pool");
185         }
186
187         tasks_.emplace([task]() { (*task)(); });
188     }
189
190     condition_.notify_one();
191     return result;
192 }
193
194 ~ThreadPoolWithFutures() {
195     std::lock_guard<std::mutex> lock(queue_mutex_);
196     stop_ = true;
```

```
198     }
199
200     condition_.notify_all();
201
202     for (std::thread& worker : workers_) {
203         if (worker.joinable()) {
204             worker.join();
205         }
206     }
207 }
208 };
209
210 void demonstrate_thread_pool_with_futures() {
211     std::cout << "\n" << std::string(70, '=') << "\n";
212     std::cout << "==== 2. Thread Pool with Futures (Return Values) ====\n";
213     std::cout << std::string(70, '=') << "\n\n";
214
215     std::cout << "Benefit: Can return results from tasks using std::future\n\n";
216
217     ThreadPoolWithFutures pool(4);
218
219     std::vector<std::future<int>> results;
220
221     std::cout << "Submitting computation tasks...\n";
222
223     for (int i = 1; i <= 8; ++i) {
224         results.push_back(pool.submit([i]{
225             std::cout << "  Task " << i << ": Computing " << i << "\n";
226             std::this_thread::sleep_for(100ms);
227             return i * i;
228         }));
229     }
230
231     std::cout << "\nCollecting results...\n";
232
233     for (size_t i = 0; i < results.size(); ++i) {
234         int result = results[i].get(); // Blocks until result is ready
235         std::cout << "  Result " << (i + 1) << ": " << result << "\n";
236     }
237
238     std::cout << "\n All tasks returned results via std::future\n";
239 }
240
241 // =====
242 // SECTION 3: Thread Pool with Priority Queue
243 // =====
244
245 enum class TaskPriority {
246     LOW = 0,
```

```
247     NORMAL = 1,
248     HIGH = 2,
249     CRITICAL = 3
250 };
251
252 struct PrioritizedTask {
253     std::function<void()> func;
254     TaskPriority priority;
255     int sequence; // For FIFO within same priority
256
257     bool operator<(const PrioritizedTask& other) const {
258         if (priority != other.priority) {
259             return priority < other.priority; // Higher priority first
260         }
261         return sequence > other.sequence; // FIFO for same priority
262     }
263 };
264
265 class PriorityThreadPool {
266 private:
267     std::vector<std::thread> workers_;
268     std::priority_queue<PrioritizedTask> tasks_;
269     std::mutex queue_mutex_;
270     std::condition_variable condition_;
271     bool stop_;
272     std::atomic<int> sequence_counter_;
273
274 public:
275     explicit PriorityThreadPool(size_t num_threads)
276         : stop_(false), sequence_counter_(0) {
277
278         for (size_t i = 0; i < num_threads; ++i) {
279             workers_.emplace_back([this]() {
280                 while (true) {
281                     PrioritizedTask task{}, TaskPriority::NORMAL, 0);
282
283                     {
284                         std::unique_lock<std::mutex> lock(queue_mutex_);
285                         condition_.wait(lock, [this]() {
286                             return stop_ || !tasks_.empty();
287                         });
288
289                         if (stop_ && tasks_.empty()) {
290                             return;
291                         }
292
293                         task = tasks_.top();
294                         tasks_.pop();
295                     }
296
297                     task.func();
298                 }
299             });
300         }
301     }
302 }
```

```
301     }
302
303     void submit(std::function<void()> task, TaskPriority priority =
304         TaskPriority::NORMAL) {
305     {
306         std::lock_guard<std::mutex> lock(queue_mutex_);
307
308         if (stop_) {
309             throw std::runtime_error("Cannot submit to stopped pool");
310         }
311
312         tasks_.push({std::move(task), priority, sequence_counter_++});
313     }
314
315     condition_.notify_one();
316 }
317
318 ~PriorityThreadPool() {
319 {
320     std::lock_guard<std::mutex> lock(queue_mutex_);
321     stop_ = true;
322 }
323
324     condition_.notify_all();
325
326     for (std::thread& worker : workers_) {
327         if (worker.joinable()) {
328             worker.join();
329         }
330     }
331 }
332
333 void demonstrate_priority_thread_pool() {
334     std::cout << "\n" << std::string(70, '=') << "\n";
335     std::cout << "==== 3. Priority Thread Pool ===\n";
336     std::cout << std::string(70, '=') << "\n\n";
337
338     std::cout << "Concept: Higher priority tasks execute before lower priority
339     ones\n\n";
340
341     PriorityThreadPool pool(2);
342
343     std::cout << "Submitting tasks with different priorities...\n\n";
344
345     // Submit in mixed order
346     pool.submit([]() {
347         std::cout << "  [NORMAL] Task 1\n";
348         std::this_thread::sleep_for(100ms);
349     }, TaskPriority::NORMAL);
350
351     pool.submit([]() {
352         std::cout << "  [CRITICAL] Urgent task!\n";
353         std::this_thread::sleep_for(100ms);
354     }, TaskPriority::CRITICAL);
355 }
```

```
353 }, TaskPriority::CRITICAL);  
354  
355 pool.submit([]() {  
356     std::cout << " [LOW] Background task\n";  
357     std::this_thread::sleep_for(100ms);  
358 }, TaskPriority::LOW);  
359  
360 pool.submit([]() {  
361     std::cout << " [HIGH] Important task\n";  
362     std::this_thread::sleep_for(100ms);  
363 }, TaskPriority::HIGH);  
364  
365 pool.submit([]() {  
366     std::cout << " [NORMAL] Task 2\n";  
367     std::this_thread::sleep_for(100ms);  
368 }, TaskPriority::NORMAL);  
369  
370 std::this_thread::sleep_for(1s);  
371  
372 std::cout << "\n Tasks executed by priority: CRITICAL → HIGH → NORMAL →  
373     LOW\n";  
374 }  
375 //  
376 ======  
377 // SECTION 4: Work-Stealing Thread Pool (Advanced)  
378 //  
379 ======  
380  
381 class WorkStealingThreadPool {  
382 private:  
383     struct WorkerThread {  
384         std::deque<std::function<void()>> local_queue;  
385         std::mutex queue_mutex;  
386         std::thread thread;  
387     };  
388  
389     std::vector<std::unique_ptr<WorkerThread>> workers_;  
390     std::atomic<bool> stop_;  
391     std::atomic<size_t> next_worker_; // Round-robin submission  
392  
393 public:  
394     explicit WorkStealingThreadPool(size_t num_threads)  
395         : stop_(false), next_worker_(0) {  
396         std::cout << " Creating work-stealing pool with " << num_threads << "  
397             workers\n";  
398         for (size_t i = 0; i < num_threads; ++i) {  
399             auto worker = std::make_unique<WorkerThread>();  
400             worker->thread = std::thread([this, i, &w = *worker]() {
```

```

401         std::cout << "      Worker " << i << " started\n";
402
403     while (!stop_) {
404         std::function<void()> task;
405
406         // Try to get task from own queue first
407         {
408             std::lock_guard<std::mutex> lock(w.queue_mutex);
409             if (!w.local_queue.empty()) {
410                 task = std::move(w.local_queue.front());
411                 w.local_queue.pop_front();
412             }
413         }
414
415         // If no local task, try to steal from other workers
416         if (!task) {
417             task = try_steal_work(i);
418         }
419
420         if (task) {
421             task();
422         } else {
423             std::this_thread::sleep_for(10ms); // Prevent busy-
424             waiting
425         }
426     });
427
428     workers_.push_back(std::move(worker));
429 }
430
431 void submit(std::function<void()> task) {
432     // Round-robin distribution
433     size_t worker_idx = next_worker_++ % workers_.size();
434
435     std::lock_guard<std::mutex> lock(workers_[worker_idx]->queue_mutex);
436     workers_[worker_idx]->local_queue.push_back(std::move(task));
437 }
438
439 ~WorkStealingThreadPool() {
440     stop_ = true;
441
442     for (auto& worker : workers_) {
443         if (worker->thread.joinable()) {
444             worker->thread.join();
445         }
446     }
447
448     std::cout << "  Work-stealing pool destroyed\n";
449 }
450
451
452 private:
453     std::function<void()> try_steal_work(size_t my_index) {

```

```

454     // Try to steal from other workers
455     for (size_t i = 0; i < workers_.size(); ++i) {
456         if (i == my_index) continue;
457
458         std::lock_guard<std::mutex> lock(workers_[i]->queue_mutex);
459         if (!workers_[i]->local_queue.empty()) {
460             // Steal from the back (different from local FIFO)
461             auto task = std::move(workers_[i]->local_queue.back());
462             workers_[i]->local_queue.pop_back();
463             return task;
464         }
465     }
466
467     return nullptr;
468 }
469 };
470
471 void demonstrate_work_stealing_pool() {
472     std::cout << "\n" << std::string(70, '=') << "\n";
473     std::cout << "==== 4. Work-Stealing Thread Pool ===\n";
474     std::cout << std::string(70, '=') << "\n\n";
475
476     std::cout << "Concept: Idle workers steal tasks from busy workers' queues\n";
477     std::cout << "Benefit: Better load balancing for uneven task durations\n\n";
478
479     WorkStealingThreadPool pool(4);
480
481     std::cout << "Submitting 12 tasks with varying durations...\n\n";
482
483     for (int i = 1; i <= 12; ++i) {
484         pool.submit([i]() {
485             int duration = (i % 3 + 1) * 100; // 100ms, 200ms, or 300ms
486             std::cout << " Task " << i << " (duration: " << duration
487                 << "ms) on thread " << std::this_thread::get_id() << "\n";
488             std::this_thread::sleep_for(std::chrono::milliseconds(duration));
489         });
490     }
491
492     std::this_thread::sleep_for(2s);
493
494     std::cout << "\n Work stealing enabled better load distribution\n";
495     std::cout << " Idle workers helped busy workers finish faster\n";
496 }
497
498 // =====
499 // SECTION 5: Dynamic Thread Pool (Auto-scaling)
500 // =====

```

```
501
502 class DynamicThreadPool {
503 private:
504     std::vector<std::thread> workers_;
505     std::queue<std::function<void()>> tasks_;
506     std::mutex queue_mutex_;
507     std::condition_variable condition_;
508     bool stop_;
509
510     size_t min_threads_;
511     size_t max_threads_;
512     std::atomic<size_t> active_threads_;
513     std::atomic<size_t> idle_threads_;
514
515 public:
516     DynamicThreadPool(size_t min_threads, size_t max_threads)
517         : stop_(false), min_threads_(min_threads), max_threads_(max_threads),
518           active_threads_(0), idle_threads_(0) {
519
520         std::cout << "  Creating dynamic pool (min: " << min_threads
521             << ", max: " << max_threads << ")\n";
522
523         // Start with minimum threads
524         for (size_t i = 0; i < min_threads_; ++i) {
525             add_worker();
526         }
527     }
528
529     void submit(std::function<void()> task) {
530     {
531         std::lock_guard<std::mutex> lock(queue_mutex_);
532
533         if (stop_) {
534             throw std::runtime_error("Cannot submit to stopped pool");
535         }
536
537         tasks_.push(std::move(task));
538
539         // Auto-scale: add worker if all busy and below max
540         if (idle_threads_ == 0 && workers_.size() < max_threads_) {
541             std::cout << "  Scaling up: adding worker (total: "
542                 << (workers_.size() + 1) << ")\n";
543             add_worker();
544         }
545     }
546
547     condition_.notify_one();
548 }
549
550 ~DynamicThreadPool() {
551 {
552     std::lock_guard<std::mutex> lock(queue_mutex_);
553     stop_ = true;
554 }
```

```
555     condition_.notify_all();
556
557     for (std::thread& worker : workers_) {
558         if (worker.joinable()) {
559             worker.join();
560         }
561     }
562
563     std::cout << "  Dynamic pool destroyed\n";
564 }
565
566
567 private:
568     void add_worker() {
569         workers_.emplace_back([this]() {
570             while (true) {
571                 std::function<void()> task;
572
573                 {
574                     std::unique_lock<std::mutex> lock(queue_mutex_);
575
576                     ++idle_threads_;
577
578                     condition_.wait(lock, [this]() {
579                         return stop_ || !tasks_.empty();
580                     });
581
582                     --idle_threads_;
583
584                     if (stop_ && tasks_.empty()) {
585                         return;
586                     }
587
588                     if (!tasks_.empty()) {
589                         task = std::move(tasks_.front());
590                         tasks_.pop();
591                     }
592                 }
593
594                 if (task) {
595                     ++active_threads_;
596                     task();
597                     --active_threads_;
598                 }
599             }
600         });
601     }
602
603
604     void demonstrate_dynamic_thread_pool() {
605         std::cout << "\n" << std::string(70, '=') << "\n";
606         std::cout << "==== 5. Dynamic Thread Pool (Auto-scaling) ===\n";
607         std::cout << std::string(70, '=') << "\n\n";
608     }
```

```
609     std::cout << "Concept: Pool grows/shrinks based on workload\n\n";
610
611     DynamicThreadPool pool(2, 6);
612
613     std::cout << "Phase 1: Light load (2 tasks)\n\n";
614
615     for (int i = 1; i <= 2; ++i) {
616         pool.submit([i]() {
617             std::cout << "  Light task " << i << "\n";
618             std::this_thread::sleep_for(200ms);
619         });
620     }
621
622     std::this_thread::sleep_for(500ms);
623
624     std::cout << "\nPhase 2: Heavy load (10 tasks)\n\n";
625
626     for (int i = 1; i <= 10; ++i) {
627         pool.submit([i]() {
628             std::cout << "  Heavy task " << i << "\n";
629             std::this_thread::sleep_for(300ms);
630         });
631         std::this_thread::sleep_for(50ms); // Gradual submission
632     }
633
634     std::this_thread::sleep_for(2s);
635
636     std::cout << "\n Pool automatically scaled up during heavy load\n";
637 }
638
639 // =====
640 // SECTION 6: Thread Pool Best Practices and Anti-Patterns
641 // =====
642
643 void demonstrate_best_practices() {
644     std::cout << "\n" << std::string(70, '=') << "\n";
645     std::cout << "==== 6. Thread Pool Best Practices ===\n";
646     std::cout << std::string(70, '=') << "\n\n";
647
648     std::cout << "  BEST PRACTICES:\n";
649     std::cout << "      \n\n";
650
651     std::cout << "1. Pool Size Selection:\n";
652     std::cout << "    CPU-bound: num_threads    std::thread::
653         hardware_concurrency()\n";
654     std::cout << "    I/O-bound: num_threads > CPU cores (2x-4x)\n";
655     std::cout << "    Mixed:      Profile and tune based on workload\n\n";
656
657     std::cout << "2. Task Granularity:\n";
658     std::cout << "    Tasks should run for at least 1ms (avoid overhead)\n";
```

```

658     std::cout << "      Don't submit trivial tasks (e.g., single addition)\n";
659     std::cout << "      Batch small operations into larger tasks\n\n";
660
661     std::cout << "3. Exception Handling:\n";
662     std::cout << "      Tasks should catch their own exceptions\n";
663     std::cout << "      Use std::future to propagate exceptions\n";
664     std::cout << "      Uncaught exceptions terminate worker threads\n\n";
665
666     std::cout << "4. Shutdown:\n";
667     std::cout << "      Signal stop before destroying pool\n";
668     std::cout << "      Join all worker threads in destructor\n";
669     std::cout << "      Consider draining pending tasks gracefully\n\n";
670
671     std::cout << "5. Task Dependencies:\n";
672     std::cout << "      DEADLOCK RISK: Task waiting for another task in same
673         pool\n";
674     std::cout << "      Use separate pools for dependent tasks\n";
675     std::cout << "      Or ensure pool size > max dependency depth\n\n";
676
677     std::cout << "  ANTI-PATTERNS:\n";
678     std::cout << "      \n\n";
679
680     std::cout << "1. Too many pools: Creates thread explosion\n";
681     std::cout << "      One pool per operation type (100+ pools)\n";
682     std::cout << "      One global pool or few specialized pools\n\n";
683
684     std::cout << "2. Blocking in tasks:\n";
685     std::cout << "      Task calls blocking I/O (starves other tasks)\n";
686     std::cout << "      Use async I/O or separate I/O thread pool\n\n";
687
688     std::cout << "3. Long-running tasks:\n";
689     std::cout << "      Task runs for minutes/hours (blocks worker)\n";
690     std::cout << "      Break into smaller chunks or use dedicated thread\n\n";
691
692     std::cout << "4. Unbounded queue growth:\n";
693     std::cout << "      Submit faster than processing (memory exhaustion)\n";
694     std::cout << "      Use bounded queue with backpressure\n\n";
695 }
696
697 // =====
698 // SECTION 7: Real-World Example - Image Processing Pipeline
699 // =====
700
701 struct Image {
702     int id;
703     std::string name;
704     int processing_stage; // 0=load, 1=filter, 2=resize, 3=save
705 };
706
707 void demonstrate_real_world_example() {

```

```
707 std::cout << "\n" << std::string(70, '=') << "\n";
708 std::cout << "==== 7. Real-World Example: Image Processing Pipeline ===\n";
709 std::cout << std::string(70, '=') << "\n\n";
710
711 std::cout << "Pipeline: Load → Filter → Resize → Save\n";
712 std::cout << "Strategy: Use thread pool to parallelize each stage\n\n";
713
714 ThreadPoolWithFutures pool(std::thread::hardware_concurrency());
715
716 std::vector<Image> images = {
717     {1, "photo1.jpg", 0},
718     {2, "photo2.jpg", 0},
719     {3, "photo3.jpg", 0},
720     {4, "photo4.jpg", 0}
721 };
722
723 std::cout << "Processing " << images.size() << " images...\n\n";
724
725 // Stage 1: Load images
726 std::vector<std::future<Image>> loaded;
727 for (const auto& img : images) {
728     loaded.push_back(pool.submit([img]() {
729         std::cout << " [LOAD] " << img.name << "\n";
730         std::this_thread::sleep_for(100ms);
731         Image result = img;
732         result.processing_stage = 1;
733         return result;
734     }));
735 }
736
737 // Stage 2: Apply filters
738 std::vector<std::future<Image>> filtered;
739 for (auto& future : loaded) {
740     Image img = future.get();
741     filtered.push_back(pool.submit([img]() {
742         std::cout << " [FILTER] " << img.name << "\n";
743         std::this_thread::sleep_for(150ms);
744         Image result = img;
745         result.processing_stage = 2;
746         return result;
747     }));
748 }
749
750 // Stage 3: Resize
751 std::vector<std::future<Image>> resized;
752 for (auto& future : filtered) {
753     Image img = future.get();
754     resized.push_back(pool.submit([img]() {
755         std::cout << " [RESIZE] " << img.name << "\n";
756         std::this_thread::sleep_for(100ms);
757         Image result = img;
758         result.processing_stage = 3;
759         return result;
760     }));
761 }
```

```

761 }
762
763 // Stage 4: Save
764 std::vector<std::future<void>> saved;
765 for (auto& future : resized) {
766     Image img = future.get();
767     saved.push_back(pool.submit([img]() {
768         std::cout << "  [SAVE] " << img.name << " -> output/" << img.name
769         << "\n";
770         std::this_thread::sleep_for(80ms);
771     }));
772 }
773
774 // Wait for all to complete
775 for (auto& future : saved) {
776     future.get();
777 }
778
779 std::cout << "\n Pipeline completed for all images\n";
780 std::cout << "  Each stage parallelized using thread pool\n";
781 }
782 // -----
783 // SECTION 8: Comparison - When to Use What
784 // -----
785
786 void demonstrate_comparison() {
787     std::cout << "\n" << std::string(70, '=') << "\n";
788     std::cout << "==== 8. Thread Pool Comparison - When to Use What ===\n";
789     std::cout << std::string(70, '=') << "\n\n";
790
791     std::cout << "                                     \n";
792     std::cout << "  Pool Type           Use Case           Pros/Cons\n";
793     std::cout << "                                     \n";
794     std::cout << "  Basic               Simple tasks      + Easy to\n";
795     std::cout << "  implement            \n";
796     std::cout << "  values               Fire-and-forget   - No return\n";
797     std::cout << "                                     \n";
798     std::cout << "  With Futures         Computational work + Return values\n";
799     std::cout << "                                     \n";
800     std::cout << "                                     \n";
801     std::cout << "  Priority             Need results      + Type-safe\n";
802     std::cout << "  order                \n";
803     std::cout << "                                     \n";
804     std::cout << "                                     \n";
805     std::cout << "  Mixed importance     Real-time systems - More overhead\n";
806     std::cout << "                                     \n";
807 }

```

```

803     std::cout << " Work-Stealing           Uneven task sizes      + Better load
804         balance\n";
805     std::cout << "
806         \n";
807     std::cout << "
808         \n";
809
810     std::cout << " ALTERNATIVES TO THREAD POOLS:\n";
811     std::cout << "
812         \n\n";
813
814     std::cout << "1. std::async:\n";
815     std::cout << "   Use when: One-off async operations, automatic thread
816         management\n";
817     std::cout << "   Avoid when: High frequency task submission (creates many
818         threads)\n\n";
819
820
821     std::cout << "2. std::thread:\n";
822     std::cout << "   Use when: Long-running background tasks, dedicated worker
823         \n";
824     std::cout << "   Avoid when: Many short-lived tasks (thread creation
825         overhead)\n\n";
826
827
828     std::cout << "3. ASIO/Boost.Asio:\n";
829     std::cout << "   Use when: I/O-bound operations, async networking\n";
830     std::cout << "   Avoid when: CPU-bound computations\n\n";
831
832
833 }
834
835 // MAIN FUNCTION
836
837
838 int main() {
839     std::cout << "\n";
840     std::cout << "
841         \n";
842         Thread Pool Implementations - Educational
843         Examples      \n";

```

```
842     std::cout << "
843         ;
844         std::cout << "  From basic to advanced: Complete guide to C++ thread
845             pools      \n";
846         std::cout << "
847                         \n";
848
849         std::cout << "\nSystem Info: " << std::thread::hardware_concurrency()
850             << "  hardware threads available\n";
851
852         demonstrate_basic_thread_pool();
853         demonstrate_thread_pool_with_futures();
854         demonstrate_priority_thread_pool();
855         demonstrate_work_stealing_pool();
856         demonstrate_dynamic_thread_pool();
857         demonstrate_best_practices();
858         demonstrate_real_world_example();
859         demonstrate_comparison();
860
861         std::cout << "\n" << std::string(70, '=') << "\n";
862         std::cout << "All thread pool demonstrations completed!\n";
863         std::cout << "\nKEY TAKEAWAYS:\n";
864         std::cout << "  1. Thread pools reuse threads → avoid creation overhead\n"
865             ;
866         std::cout << "  2. Choose pool type based on workload characteristics\n";
867         std::cout << "  3. Use futures for tasks that return values\n";
868         std::cout << "  4. Profile before optimizing thread count\n";
869         std::cout << "  5. Watch for deadlocks with task dependencies\n";
870
871         std::cout << std::string(70, '=') << "\n\n";
872
873     return 0;
874 }
```

## 73 Source Code: TuplesAndStructuredBindings.cpp

File: src/TuplesAndStructuredBindings.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // COMPREHENSIVE TUPLES AND STRUCTURED BINDINGS
3 // =====
4 // This example demonstrates modern C++ tuple features and best practices.
5 //
6 // TOPICS COVERED:
7 // 1. Basic tuple operations (C++11)
8 // 2. std::tie and unpacking
9 // 3. Structured bindings (C++17)
10 // 4. std::apply (C++17)
11 // 5. Tuple comparison and algorithms
12 // 6. Tuple concatenation
13 // 7. std::tuple_cat
14 // 8. Real-world use cases
15 // 9. Performance considerations
16 // 10. When to use tuples vs structs
17 // =====
18
19 #include <iostream>
20 #include <string>
21 #include <tuple>
22 #include <vector>
23 #include <map>
24 #include <algorithm>
25 #include <numeric>
26
27 // =====
28 // SECTION 1: BASIC TUPLE OPERATIONS
29 // =====
30
31 void demonstrate_basic_tuples() {
32     std::cout << "\n== 1. BASIC TUPLE OPERATIONS ==" << std::endl;
33
34     // Creating tuples
35     std::tuple<int, std::string, double> person1{25, "Alice", 5.6};
36     auto person2 = std::make_tuple(30, "Bob", 6.0);
37
38     // Accessing elements with std::get
39     std::cout << "\nPerson 1: " << std::get<0>(person1) << " years, "
40             << std::get<1>(person1) << ", "
41             << std::get<2>(person1) << "ft" << std::endl;
42
43     // Accessing by type (C++14) - only if type is unique!
44     std::cout << "Name: " << std::get<std::string>(person1) << std::endl;
45
46     // Modifying elements
47     std::get<0>(person1) = 26;
48     std::cout << "After birthday: " << std::get<0>(person1) << " years" << std::endl;
```

```
49 // Tuple size
50 std::cout << "\nTuple size: " << std::tuple_size<decltype(person1)>::value
51     << std::endl;
52
53 std::cout << "\n KEY POINTS:" << std::endl;
54 std::cout << " • std::get<N>(tuple) - access by index (compile-time)" <<
55     std::endl;
56 std::cout << " • std::get<Type>(tuple) - access by type (if unique)" <<
57     std::endl;
58 std::cout << " • Zero runtime overhead vs struct" << std::endl;
59 }
60
61 // =====
62 // SECTION 2: std::tie AND UNPACKING
63 // =====
64
65 std::tuple<int, int, int> divide_full(int dividend, int divisor) {
66     return {dividend / divisor, dividend % divisor, dividend};
67 }
68
69 void demonstrate_tie() {
70     std::cout << "\n== 2. std::tie AND UNPACKING ==" << std::endl;
71
72     // Old way: extract tuple elements
73     auto result = divide_full(17, 5);
74     int quotient = std::get<0>(result);
75     int remainder = std::get<1>(result);
76     int original = std::get<2>(result);
77
78     std::cout << "\nOld way: 17 / 5 = " << quotient
79         << " remainder " << remainder << std::endl;
80
81     // Better: std::tie (C++11)
82     int q, r, o;
83     std::tie(q, r, o) = divide_full(17, 5);
84     std::cout << "With tie: 17 / 5 = " << q << " remainder " << r << std::endl
85         ;
86
87     // Ignore some values with std::ignore
88     std::tie(q, std::ignore, std::ignore) = divide_full(20, 3);
89     std::cout << "Quotient only: 20 / 3 = " << q << std::endl;
90
91     // Using tie for swap
92     int a = 10, b = 20;
93     std::cout << "\nBefore swap: a=" << a << ", b=" << b << std::endl;
94     std::tie(a, b) = std::tie(b, a);
95     std::cout << "After swap: a=" << a << ", b=" << b << std::endl;
96
97     std::cout << "\n std::tie benefits:" << std::endl;
98     std::cout << " • Unpack multiple return values" << std::endl;
99     std::cout << " • Use std::ignore for unwanted values" << std::endl;
100    std::cout << " • Create reference tuples" << std::endl;
101 }
```

```
99
100 // =====
101 // SECTION 3: STRUCTURED BINDINGS (C++17)
102 // =====
103
104 struct Sensor {
105     int id;
106     double temperature;
107     double humidity;
108     std::string location;
109 };
110
111 std::tuple<double, double, double> compute_statistics(const std::vector<double>& values) {
112     double sum = std::accumulate(values.begin(), values.end(), 0.0);
113     double mean = sum / values.size();
114     double min = *std::min_element(values.begin(), values.end());
115     double max = *std::max_element(values.begin(), values.end());
116     return {mean, min, max};
117 }
118
119 void demonstrate_structured_bindings() {
120     std::cout << "\n==== 3. STRUCTURED BINDINGS (C++17) ===" << std::endl;
121
122     // Structured bindings - most modern and readable!
123     auto [quotient, remainder, original] = divide_full(25, 7);
124     std::cout << "\n25 / 7 = " << quotient << " remainder " << remainder << std::endl;
125
126     // Works with pairs
127     std::map<std::string, int> scores{{"Alice", 95}, {"Bob", 87}};
128     for (const auto& [name, score] : scores) {
129         std::cout << name << ": " << score << " points" << std::endl;
130     }
131
132     // Works with structs
133     Sensor sensor{101, 22.5, 65.0, "Lab-A"};
134     auto [id, temp, humid, loc] = sensor;
135     std::cout << "\nSensor " << id << " at " << loc
136         << " : " << temp << "°C, " << humid << "% humidity" << std::endl;
137
138     // With functions
139     std::vector<double> temps{20.1, 22.5, 21.8, 23.0, 19.5};
140     auto [mean, min, max] = compute_statistics(temps);
141     std::cout << "\nTemperature stats: mean=" << mean
142         << ", min=" << min << ", max=" << max << std::endl;
143
144     std::cout << "\n BEST PRACTICE:" << std::endl;
145     std::cout << " • Use structured bindings (C++17) over std::tie" << std::endl;
146     std::cout << " • More readable, less verbose" << std::endl;
147     std::cout << " • Works with tuples, pairs, structs, arrays" << std::endl
148         ;
```

```
149
150 // =====
151 // SECTION 4: std::apply (C++17)
152 // =====
153
154 int add_three(int a, int b, int c) {
155     return a + b + c;
156 }
157
158 double multiply(double x, double y, double z) {
159     return x * y * z;
160 }
161
162 void demonstrate_apply() {
163     std::cout << "\n== 4. std::apply (C++17) ==" << std::endl;
164
165     // std::apply - call function with tuple as arguments
166     std::tuple<int, int, int> args{10, 20, 30};
167     int sum = std::apply(add_three, args);
168     std::cout << "\nsum of (10, 20, 30) = " << sum << std::endl;
169
170     std::tuple<double, double, double> factors{2.0, 3.0, 4.0};
171     double product = std::apply(multiply, factors);
172     std::cout << "product of (2.0, 3.0, 4.0) = " << product << std::endl;
173
174     // With lambdas
175     auto printer = [] (auto... args) {
176         std::cout << "Values: ";
177         ((std::cout << args << " "), ...);
178         std::cout << std::endl;
179     };
180
181     std::apply(printer, std::make_tuple(1, "hello", 3.14, 'X'));
182
183     std::cout << "\n std::apply use cases:" << std::endl;
184     std::cout << " • Call function with tuple arguments" << std::endl;
185     std::cout << " • Forward arguments stored in tuple" << std::endl;
186     std::cout << " • Unpack tuple into function call" << std::endl;
187 }
188
189 // =====
190 // SECTION 5: TUPLE COMPARISON AND ALGORITHMS
191 // =====
192
193 void demonstrate_tuple_comparison() {
194     std::cout << "\n== 5. TUPLE COMPARISON ==" << std::endl;
195
196     std::tuple<int, std::string> t1{1, "apple"};
197     std::tuple<int, std::string> t2{1, "banana"};
198     std::tuple<int, std::string> t3{2, "apple"};
199
200     // Lexicographic comparison (left to right)
201     std::cout << "\nComparisons:" << std::endl;
202     std::cout << " (1, 'apple') < (1, 'banana') : "
```

```

203         << std::boolalpha << (t1 < t2) << std::endl;
204     std::cout << "    (1, 'apple') < (2, 'apple'): " << (t1 < t3) << std::endl;
205     std::cout << "    (1, 'banana') < (2, 'apple'): " << (t2 < t3) << std::endl
206     ;
207
208     // Sorting with tuples
209     std::vector<std::tuple<int, std::string, double>> students{
210         {85, "Charlie", 3.2},
211         {92, "Alice", 3.8},
212         {85, "Bob", 3.5},
213         {92, "David", 3.6}
214     };
215
216     std::cout << "\nBefore sort:" << std::endl;
217     for (const auto& [score, name, gpa] : students) {
218         std::cout << "    " << name << ":" << score << ", GPA " << gpa << std
219             ::endl;
220     }
221
222     // Sort by tuple (score desc, then name asc)
223     std::sort(students.begin(), students.end(),
224               [] (const auto& a, const auto& b) {
225                   return std::tie(std::get<0>(b), std::get<1>(a)) <
226                           std::tie(std::get<0>(a), std::get<1>(b));
227               });
228
229     std::cout << "\nAfter sort (score desc, name asc):" << std::endl;
230     for (const auto& [score, name, gpa] : students) {
231         std::cout << "    " << name << ":" << score << ", GPA " << gpa << std
232             ::endl;
233     }
234
235     std::cout << "\n Tuple comparison:" << std::endl;
236     std::cout << " •  Lexicographic (element by element)" << std::endl;
237     std::cout << " •  Perfect for sorting by multiple keys" << std::endl;
238     std::cout << " •  Use std::tie for custom comparison" << std::endl;
239 }
240
241 // =====
242 // SECTION 6: std::tuple_cat AND CONCATENATION
243 // =====
244
245 void demonstrate_tuple_cat() {
246     std::cout << "\n== 6. std::tuple_cat ==" << std::endl;
247
248     std::tuple<int, std::string> t1{42, "answer"};
249     std::tuple<double, char> t2{3.14, 'X'};
250
251     // Concatenate tuples
252     auto combined = std::tuple_cat(t1, t2);
253     auto [num, text, pi, letter] = combined;
254
255     std::cout << "\nCombined tuple: " << num << ", " << text
256         << ", " << pi << ", " << letter << std::endl;

```

```
254
255 // Combine multiple tuples
256 std::tuple<int> id{100};
257 std::tuple<std::string, std::string> name{"John", "Doe"};
258 std::tuple<int> age{30};
259
260 auto person = std::tuple_cat(id, name, age);
261 std::cout << "\nPerson: ID=" << std::get<0>(person)
262     << ", Name=" << std::get<1>(person) << " " << std::get<2>(person)
263     )
264     << ", Age=" << std::get<3>(person) << std::endl;
265
266 std::cout << "\n std::tuple_cat:" << std::endl;
267 std::cout << " • Concatenate multiple tuples" << std::endl;
268 std::cout << " • All done at compile-time" << std::endl;
269 std::cout << " • Zero runtime overhead" << std::endl;
270 }
271
272 // =====
273 // SECTION 7: REAL-WORLD USE CASE - PARTICLE SYSTEM
274 // =====
275
276 using State = std::tuple<double, double, double, double, double>;
277 using ParticleWeight = double;
278 using Particle = std::map<ParticleWeight, State>;
279 using Particles = std::vector<Particle>;
280
281 double compute_total_weight(const Particles& particles) {
282     double total = 0.0;
283     for (const auto& particle : particles) {
284         for (const auto& [weight, state] : particle) {
285             total += weight;
286         }
287     }
288     return total;
289 }
290
291 void print_particle_state(const State& state) {
292     auto [x, y, z, vx, vy] = state;
293     std::cout << "    Position(" << x << "," << y << "," << z << ") "
294         << "Velocity(" << vx << "," << vy << ")" << std::endl;
295 }
296
297 void demonstrate_particle_system() {
298     std::cout << "\n==== 7. REAL-WORLD: PARTICLE SYSTEM ===" << std::endl;
299
300     Particles particles;
301
302     // Create particles with weighted states
303     for (int i = 0; i < 4; i++) {
304         State state = std::make_tuple(i*1.0, i*1.1, i*1.2, i*0.5, i*0.6);
305         Particle particle;
306         particle.emplace(5.0 + i, state);
307         particles.push_back(particle);
308     }
309 }
```

```

307 }
308
309 std::cout << "\nParticle states:" << std::endl;
310 for (size_t i = 0; i < particles.size(); i++) {
311     for (const auto& [weight, state] : particles[i]) {
312         std::cout << "Particle " << i << " (weight=" << weight << "):" <<
313             std::endl;
314         print_particle_state(state);
315     }
316 }
317
318 double total_weight = compute_total_weight(particles);
319 std::cout << "\nTotal weight: " << total_weight << std::endl;
320
321 std::cout << "\n Why use tuples here:" << std::endl;
322 std::cout << " • Simple data aggregation" << std::endl;
323 std::cout << " • No need for named struct" << std::endl;
324 std::cout << " • Easy to decompose with structured bindings" << std::endl;
325 }
326
327 // =====
328 // SECTION 8: TUPLE VS STRUCT - WHEN TO USE EACH
329 // =====
330 void explain_tuple_vs_struct() {
331     std::cout << "\n" << std::string(70, '=') << std::endl;
332     std::cout << "TUPLE VS STRUCT - WHEN TO USE EACH:\n";
333     std::cout << std::string(70, '=') << std::endl;
334
335     std::cout << "\n USE TUPLES WHEN:" << std::endl;
336     std::cout << "\n1. RETURNING MULTIPLE VALUES:" << std::endl;
337     std::cout << " std::tuple<int, int> divide(int a, int b) { ... }" <<
338         std::endl;
339     std::cout << " • Quick and easy multiple returns" << std::endl;
340     std::cout << " • Use structured bindings to unpack" << std::endl;
341
342     std::cout << "\n2. TEMPORARY DATA GROUPING:" << std::endl;
343     std::cout << " • Local scope only" << std::endl;
344     std::cout << " • No need for named type" << std::endl;
345     std::cout << " • One-off data combinations" << std::endl;
346
347     std::cout << "\n3. GENERIC PROGRAMMING:" << std::endl;
348     std::cout << " • Template metaprogramming" << std::endl;
349     std::cout << " • Variadic template arguments" << std::endl;
350     std::cout << " • Type computations" << std::endl;
351
352     std::cout << "\n4. QUICK PROTOTYPING:" << std::endl;
353     std::cout << " • Experimenting with data structures" << std::endl;
354     std::cout << " • Before defining proper types" << std::endl;
355
356     std::cout << "\n USE STRUCTS WHEN:" << std::endl;
357     std::cout << "\n1. NAMED SEMANTICS MATTER:" << std::endl;
358     std::cout << " struct Person { int age; string name; double height; }";

```

```
    " << std::endl;
358 std::cout << " • Clear what each field means" << std::endl;
359 std::cout << " • Self-documenting code" << std::endl;
360
361 std::cout << "\n2. MEMBER FUNCTIONS NEEDED:" << std::endl;
362 std::cout << " • Methods to manipulate data" << std::endl;
363 std::cout << " • Encapsulation" << std::endl;
364 std::cout << " • Invariants to maintain" << std::endl;
365
366 std::cout << "\n3. PUBLIC API:" << std::endl;
367 std::cout << " • Library interfaces" << std::endl;
368 std::cout << " • Long-term maintainability" << std::endl;
369 std::cout << " • Clear documentation" << std::endl;
370
371 std::cout << "\n4. COMPLEX TYPES:" << std::endl;
372 std::cout << " • Many fields (>5)" << std::endl;
373 std::cout << " • Constructors needed" << std::endl;
374 std::cout << " • Relationships with other types" << std::endl;
375
376 std::cout << "\n ANTI-PATTERNS:" << std::endl;
377 std::cout << "\n DON'T: tuple<int, int, int, int, int, int>" << std
      ::endl;
378 std::cout << "    Too many fields - use struct!" << std::endl;
379 std::cout << "\n DON'T: Use tuples in public API" << std::endl;
380 std::cout << "    Hard to understand: get<2>(result) means what?" << std::
      endl;
381 std::cout << "\n DON'T: Store tuples long-term" << std::endl;
382 std::cout << "    Use proper named types for persistence" << std::endl;
383
384 std::cout << "\n GOLDEN RULE:" << std::endl;
385 std::cout << "    Tuples for quick local returns," << std::endl;
386 std::cout << "    Structs for everything else!" << std::endl;
387 }
388
389 // =====
390 // SECTION 9: PERFORMANCE CONSIDERATIONS
391 // =====
392
393 struct Point3D {
394     double x, y, z;
395 };
396
397 void demonstrate_performance() {
398     std::cout << "\n" << std::string(70, '=') << std::endl;
399     std::cout << "PERFORMANCE CONSIDERATIONS:\n";
400     std::cout << std::string(70, '=') << std::endl;
401
402     using TuplePoint = std::tuple<double, double, double>;
403
404     std::cout << "\n MEMORY LAYOUT:" << std::endl;
405     std::cout << "    struct Point3D: " << sizeof(Point3D) << " bytes" << std
      ::endl;
406     std::cout << "    tuple<d,d,d>: " << sizeof(TuplePoint) << " bytes" <<
      std::endl;
407 }
```

```

407     std::cout << "      SAME SIZE - no overhead!" << std::endl;
408
409     std::cout << "\n RUNTIME PERFORMANCE:" << std::endl;
410     std::cout << "      Struct: point.x (direct access)" << std::endl;
411     std::cout << "      Tuple: std::get<0>(point) (compile-time index)" << std::endl;
412     std::cout << "      SAME SPEED - both inline to same code!" << std::endl;
413
414     std::cout << "\n COMPIILATION TIME:" << std::endl;
415     std::cout << "      Struct: Fast - simple type" << std::endl;
416     std::cout << "      Tuple: Slower - template instantiation" << std::endl;
417     std::cout << "      Tuples increase compile time slightly" << std::endl;
418
419     std::cout << "\n PERFORMANCE TIPS:" << std::endl;
420     std::cout << " • Tuples have ZERO runtime overhead" << std::endl;
421     std::cout << " • std::get<N> is compile-time constant" << std::endl;
422     std::cout << " • Use references to avoid copies" << std::endl;
423     std::cout << " • Move semantics work perfectly" << std::endl;
424 }
425
426 // =====
427 // SECTION 10: ADVANCED TECHNIQUES
428 // =====
429
430 // Tuple element type at index
431 template<size_t I, typename Tuple>
432 using tuple_element_t = typename std::tuple_element<I, Tuple>::type;
433
434 void demonstrate_advanced() {
435     std::cout << "\n==== 10. ADVANCED TECHNIQUES ===" << std::endl;
436
437     // Type introspection
438     using MyTuple = std::tuple<int, double, std::string>;
439
440     std::cout << "\nTuple type information:" << std::endl;
441     std::cout << "      Size: " << std::tuple_size_v<MyTuple> << std::endl;
442
443     // Get element types
444     using FirstType = tuple_element_t<0, MyTuple>;
445     using SecondType = tuple_element_t<1, MyTuple>;
446
447     std::cout << "      First element is int: "
448             << std::boolalpha << std::is_same_v<FirstType, int> << std::endl
449             ;
450     std::cout << "      Second element is double: "
451             << std::is_same_v<SecondType, double> << std::endl;
452
453     // Forward as tuple (perfect forwarding)
454     auto forward_example = [] (auto&&... args) {
455         auto tuple = std::forward_as_tuple(args...);
456         std::cout << "\n      Created reference tuple of size "
457             << std::tuple_size_v<decltype(tuple)> << std::endl;
458     };

```

```
459 int x = 10;
460 double y = 20.5;
461 forward_example(x, y, std::string("test"));

462
463 std::cout << "\n Advanced features:" << std::endl;
464 std::cout << " • std::tuple_element - get element type" << std::endl;
465 std::cout << " • std::tuple_size - get tuple size" << std::endl;
466 std::cout << " • std::forward_as_tuple - create reference tuple" << std
467     ::endl;
468 std::cout << " • Perfect for template metaprogramming" << std::endl;
469 }

470 // =====
471 // MAIN FUNCTION
472 // =====
473
474 int main() {
475     std::cout << "\n";
476     std::cout << "                                     \n";
477     std::cout << "             COMPREHENSIVE TUPLES AND STRUCTURED BINDINGS
478             \n";
479     std::cout << "             \n";
480     std::cout << "             \n";
481
482     demonstrate_basic_tuples();
483     demonstrate_tie();
484     demonstrate_structured_bindings();
485     demonstrate_apply();
486     demonstrate_tuple_comparison();
487     demonstrate_tuple_cat();
488     demonstrate_particle_system();
489     explain_tuple_vs_struct();
490     demonstrate_performance();
491     demonstrate_advanced();

492
493     std::cout << "\n" << std::string(70, '=') << std::endl;
494     std::cout << "SUMMARY:\n";
495     std::cout << std::string(70, '=') << std::endl;

496
497     std::cout << "\n KEY TAKEAWAYS:" << std::endl;
498     std::cout << "\n1. MODERN SYNTAX:" << std::endl;
499     std::cout << " • Prefer structured bindings (C++17) over std::tie" <<
500         std::endl;
501     std::cout << " • Use std::apply for tuple-to-function unpacking" << std
502         ::endl;
503     std::cout << " • auto [a, b, c] = func(); // Most readable!" << std::
504         endl;

505     std::cout << "\n2. WHEN TO USE:" << std::endl;
506     std::cout << "     Multiple return values" << std::endl;
507     std::cout << "     Temporary data grouping" << std::endl;
```

```
506     std::cout << "      Generic programming" << std::endl;
507     std::cout << "      Public APIs (use structs)" << std::endl;
508     std::cout << "      Many fields (>5) (use structs)" << std::endl;
509
510     std::cout << "\n3. PERFORMANCE:" << std::endl;
511     std::cout << " • Zero runtime overhead vs struct" << std::endl;
512     std::cout << " • std::get<N> is compile-time" << std::endl;
513     std::cout << " • Slightly slower compilation" << std::endl;
514
515     std::cout << "\n4. BEST PRACTICES:" << std::endl;
516     std::cout << " • Keep tuples small (<= 5 elements)" << std::endl;
517     std::cout << " • Use for local/temporary data" << std::endl;
518     std::cout << " • Switch to struct when semantics matter" << std::endl;
519     std::cout << " • Document what tuple fields mean!" << std::endl;
520
521     std::cout << "\n Tuples: Quick and powerful for the right use cases!\n"
522             << std::endl;
523
524     return 0;
525 }
```

## 74 Source Code: UniversalResourceManager.cpp

File: src/UniversalResourceManager.cpp

Repository: [View on GitHub](#)

```

1 // =====
2 // UNIVERSAL RESOURCE MANAGER - CUSTOM DELETORS IN C++
3 // =====
4 // Comprehensive guide to using custom deletors with smart pointers
5 // to manage ANY type of resource (not just heap memory).
6 //
7 // KEY INSIGHT: Custom deletors turn smart pointers from simple memory
8 // managers into universal RAII resource managers for:
9 // • File handles (FILE*, fds)
10 // • Network connections (sockets, SSL contexts)
11 // • Database connections
12 // • GPU resources (OpenGL, DirectX, Vulkan, CUDA)
13 // • OS handles (Windows HANDLE, POSIX fds)
14 // • Shared memory, memory-mapped files
15 // • Locks, mutexes, semaphores
16 // • Custom allocators and memory pools
17 //
18 // Build: g++ -std=c++20 -O2 -o UniversalResourceManager
19 //           UniversalResourceManager.cpp
20 // =====
21
22 #include <iostream>
23 #include <memory>
24 #include <cstdio>
25 #include <vector>
26 #include <string>
27 #include <cstring>
28 #include <cstdlib>
29
30 using namespace std;
31
32 // =====
33 // PART 1: WHY CUSTOM DELETORS?
34 // =====
35 void explain_the_problem() {
36     cout << "\n";
37     cout << "          WHY DO WE NEED CUSTOM DELETORS?\n";
38     cout << "          \n";
39
40     cout << "\n THE FUNDAMENTAL PROBLEM:\n";
41     cout << "      Default unique_ptr/shared_ptr call 'delete ptr' or 'delete[]\n";
42     cout << "          ptr'\n";
43     cout << "      But NOT ALL resources are heap-allocated C++ objects!\n";
44
45     cout << "\n RESOURCE TYPES THAT NEED CUSTOM CLEANUP:\n";
46     cout << "      1. C Library Resources:    FILE* → fclose()\n";
47     cout << "      2. OS Handles:           HANDLE → CloseHandle()\n";

```

```

47     cout << "    3. Network:           socket → close(), shutdown()\n";
48     cout << "    4. Database:          connection → disconnect(), rollback
49             ()\n";
50     cout << "    5. GPU:               texture → glDeleteTextures()\n";
51     cout << "    6. Shared Memory:     mapping → munmap(), shm_unlink()\n";
52             ;
53     cout << "    7. Custom Allocators: pool → return_to_pool()\n";
54     cout << "    8. Legacy C++ Code:   Cannot modify destructor\n";
55
56     cout << "\n SOLUTION: Custom Deleters\n";
57     cout << "    Transform smart pointers into universal resource managers\n";
58     cout << "    Apply RAII to ANY resource, not just memory\n";
59 }
60
61 // =====
62 // PART 2: REAL-WORLD EXAMPLES
63 // =====
64
65 // Example 1: FILE HANDLING
66 void file_example() {
67     cout << "\n== EXAMPLE 1: FILE HANDLING ==\n";
68
69     cout << " WITHOUT custom deleter (DANGEROUS):\n";
70     {
71         FILE* raw_file = fopen("test.txt", "w");
72         if (raw_file) {
73             fputs("Hello World\n", raw_file);
74             // What if exception occurs here? File won't be closed!
75             // Must manually remember to call fclose()
76             fclose(raw_file);
77         }
78     }
79
80     cout << "\n WITH custom deleter (SAFE):\n";
81     {
82         auto file_closer = [](FILE* f) {
83             if (f) {
84                 cout << "    Auto-closing file\n";
85                 fclose(f);
86             }
87         };
88
89         unique_ptr<FILE, decltype(file_closer)>
90             file(fopen("test.txt", "w"), file_closer);
91
92         if (file) {
93             fputs("Auto-closed!\n", file.get());
94             // Exception-safe! File always closed
95         }
96     }
97
98 // Example 2: DATABASE CONNECTIONS
99 void database_example() {

```

```
99  cout << "\n==== EXAMPLE 2: DATABASE CONNECTIONS ===\n";
100
101 struct DatabaseConnection {
102     string connection_id;
103     bool is_open;
104
105     DatabaseConnection(string id) : connection_id(id), is_open(true) {
106         cout << "    Connected to: " << connection_id << endl;
107     }
108
109     void execute_query(const string& query) {
110         if (is_open) {
111             cout << "    Executing: " << query << endl;
112         }
113     }
114 };
115
116 auto db_closer = [] (DatabaseConnection* db) {
117     if (db && db->is_open) {
118         cout << "    Closing connection: " << db->connection_id << endl;
119         db->is_open = false;
120         // Real code: db->rollback(), db->disconnect()
121     }
122     delete db;
123 };
124
125 {
126     unique_ptr<DatabaseConnection, decltype(db_closer)>
127         db(new DatabaseConnection("prod_db"), db_closer);
128
129     db->execute_query("SELECT * FROM users");
130 } // Auto-closed, even if exception!
131 }
132
133 // Example 3: NETWORK SOCKETS
134 void socket_example() {
135     cout << "\n==== EXAMPLE 3: NETWORK SOCKETS ===\n";
136
137     struct Socket {
138         int socket_fd;
139         bool is_connected;
140
141         Socket() : socket_fd(42), is_connected(true) {
142             cout << "    Socket created, FD: " << socket_fd << endl;
143         }
144
145         void send_data(const string& data) {
146             if (is_connected) {
147                 cout << "    Sending: " << data << endl;
148             }
149         }
150     };
151
152     auto socket_cleanup = [] (Socket* sock) {
```

```
153     if (sock) {
154         cout << "      Cleaning up socket:\n";
155         if (sock->is_connected) {
156             cout << " •      Flushing send buffer\n";
157             cout << " •      Shutting down connection\n";
158             cout << " •      Closing FD: " << sock->socket_fd << endl;
159             // Real: shutdown(fd, SHUT_RDWR); close(fd);
160         }
161         delete sock;
162     }
163 };
164 {
165     unique_ptr<Socket, decltype(socket_cleanup)>
166     socket(new Socket(), socket_cleanup);
167     socket->send_data("Hello server!");
168 }
169 }
170 }
171
172 // Example 4: GPU/OPENGL RESOURCES
173 void gpu_resource_example() {
174     cout << "\n==== EXAMPLE 4: GPU RESOURCES ===\n";
175
176     struct Texture {
177         unsigned int texture_id;
178         int width, height;
179
180         Texture(int w, int h) : texture_id(1001), width(w), height(h) {
181             cout << "      GPU Texture: ID=" << texture_id
182             << " (" << w << "x" << h << ")" << endl;
183             // Real: glGenTextures(1, &texture_id);
184         }
185
186         void bind() {
187             cout << "      Binding texture: " << texture_id << endl;
188         }
189     };
190
191     auto texture_deleter = [] (Texture* tex) {
192         if (tex) {
193             cout << "      Deleting GPU texture: " << tex->texture_id << endl;
194             cout << " •      Unbinding texture\n";
195             cout << " •      Freeing GPU memory\n";
196             // Real: glBindTexture(GL_TEXTURE_2D, 0);
197             //         glDeleteTextures(1, &tex->texture_id);
198             delete tex;
199         }
200     };
201
202     {
203         unique_ptr<Texture, decltype(texture_deleter)>
204         texture(new Texture(1024, 1024), texture_deleter);
205         texture->bind();
206     }
}
```

```
207 }
208
209 // Example 5: SHARED MEMORY
210 void shared_memory_example() {
211     cout << "\n== EXAMPLE 5: SHARED MEMORY ==\n";
212
213     struct SharedMemory {
214         void* memory;
215         size_t size;
216         string name;
217
218         SharedMemory(const string& n, size_t s)
219             : memory(malloc(s)), size(s), name(n) {
220                 cout << "      Shared memory: " << name << " (" << s << " bytes)"
221                     << endl;
222             }
223
224         void write(const string& data) {
225             if (memory && data.size() <= size) {
226                 memcpy(memory, data.data(), data.size());
227                 cout << "      Written: " << data << endl;
228             }
229         }
230     };
231
232     auto shared_mem_deleter = [](SharedMemory* sm) {
233         if (sm) {
234             cout << "      Cleaning up: " << sm->name << endl;
235             cout << " •      Syncing with other processes\n";
236             cout << " •      Unmapping memory\n";
237             cout << " •      Closing handle\n";
238             // Real: munmap(), shm_unlink() (POSIX)
239             //       UnmapViewOfFile(), CloseHandle() (Windows)
240             if (sm->memory) free(sm->memory);
241             delete sm;
242         }
243     };
244
245     {
246         unique_ptr<SharedMemory, decltype(shared_mem_deleter)>
247             shmem(new SharedMemory("app_data", 4096), shared_mem_deleter);
248         shmem->write("Shared data");
249     }
250
251 // Example 6: RESOURCE POOL (Return to pool instead of delete!)
252 void resource_pool_example() {
253     cout << "\n== EXAMPLE 6: RESOURCE POOL (Don't Delete, Reuse!) ==\n";
254
255     struct DatabaseConnection {
256         int connection_id;
257         bool in_use;
258
259         DatabaseConnection(int id) : connection_id(id), in_use(true) {
```

```
260         cout << "      Connection " << id << " created (expensive!)" <<
261             endl;
262     }
263
264     void query() {
265         if (in_use) {
266             cout << "      Query on connection " << connection_id << endl;
267         }
268     }
269
270     static vector<DatabaseConnection*> pool;
271
272     // DON'T delete! Return to pool for reuse
273     auto pool_deleter = [] (DatabaseConnection* conn) {
274         if (conn) {
275             cout << "      Returning conn " << conn->connection_id
276                 << " to pool (NOT deleting)" << endl;
277             conn->in_use = false;
278             pool.push_back(conn); // Reuse later!
279         }
280     };
281
282     auto get_connection = [&] (int id) {
283         if (!pool.empty()) {
284             auto conn = pool.back();
285             pool.pop_back();
286             conn->in_use = true;
287             cout << "      Reusing connection " << conn->connection_id << endl
288                 ;
289             return unique_ptr<DatabaseConnection, decltype(pool_deleter)>(conn,
290                 pool_deleter);
291         }
292         auto conn = new DatabaseConnection(id);
293         return unique_ptr<DatabaseConnection, decltype(pool_deleter)>(conn,
294             pool_deleter);
295     };
296
297     // Initialize pool
298     pool.push_back(new DatabaseConnection(1));
299     pool.push_back(new DatabaseConnection(2));
300
301     {
302         auto conn1 = get_connection(3); // Reuses from pool
303         auto conn2 = get_connection(4); // Reuses from pool
304         auto conn3 = get_connection(5); // Creates new
305
306         conn1->query();
307         conn2->query();
308         conn3->query();
309     } // Returned to pool
310
311     cout << "      Pool size: " << pool.size() << " connections\n";
```

```

310     // Cleanup at program end
311     for (auto conn : pool) delete conn;
312     pool.clear();
313 }
314
315 // Example 7: DEBUG ALLOCATOR with statistics
316 void allocator_with_stats() {
317     cout << "\n==== EXAMPLE 7: DEBUG ALLOCATOR (Tracking & Logging) ===\n";
318
319     struct AllocatedBlock {
320         void* memory;
321         size_t size;
322         string allocation_site;
323
324         AllocatedBlock(size_t s, const string& site)
325             : memory(malloc(s)), size(s), allocation_site(site) {
326                 cout << "      Allocated " << s << " bytes at: " << site << endl;
327             }
328     };
329
330     static int total_allocs = 0;
331     static size_t total_memory = 0;
332
333     auto debug_deleter = [](AllocatedBlock* block) {
334         if (block) {
335             cout << "      Freeing " << block->size << " bytes from: "
336                 << block->allocation_site << endl;
337             total_allocs--;
338             total_memory -= block->size;
339             cout << "      Stats: " << total_allocs << " blocks, "
340                 << total_memory << " bytes\n";
341             if (block->memory) free(block->memory);
342             delete block;
343         }
344     };
345
346     auto debug_allocate = [&](size_t size, const string& site) {
347         auto block = new AllocatedBlock(size, site);
348         total_allocs++;
349         total_memory += size;
350         return unique_ptr<AllocatedBlock, decltype(debug_deleter)>(block,
351             debug_deleter);
352     };
353
354     {
355         auto block1 = debug_allocate(100, "function_a()");
356         auto block2 = debug_allocate(200, "function_b()");
357     } // Auto-tracked cleanup
358
359     cout << "      Final: " << total_allocs << " blocks remaining\n";
360 }
361
362 // Example 8: COMPOSITE DELETER (Multiple sub-resources)
363 void composite_deleter_example() {

```

```

363     cout << "\n==== EXAMPLE 8: COMPOSITE DELETER (Multiple Resources) ===\n";
364
365     struct ComplexResource {
366         FILE* log_file;
367         void* shared_memory;
368         int network_socket;
369
370         ComplexResource()
371             : log_file(fopen("app.log", "w"))
372             , shared_memory(malloc(1024))
373             , network_socket(42) {
374                 cout << "      Complex resource created (3 sub-resources)" << endl
375                 ;
376             }
377     };
378
379     // Single deleter handles ALL sub-resources in correct order
380     auto composite_deleter = [](ComplexResource* res) {
381         if (res) {
382             cout << "      Cleanup sequence:\n";
383             if (res->log_file) {
384                 cout << "          1. Close log file\n";
385                 fclose(res->log_file);
386             }
387             if (res->shared_memory) {
388                 cout << "          2. Free shared memory\n";
389                 free(res->shared_memory);
390             }
391             if (res->network_socket > 0) {
392                 cout << "          3. Close socket: " << res->network_socket <<
393                     endl;
394             }
395             delete res;
396         }
397     };
398
399     {
400         unique_ptr<ComplexResource, decltype(composite_deleter)>
401             resource(new ComplexResource(), composite_deleter);
402     } // All 3 resources cleaned up!
403 }
404
405 // =====
406 // PART 3: CUSTOM DELETER PATTERNS
407 // =====
408
409 void deleter_patterns() {
410     cout << "\n";
411     cout << "      CUSTOM DELETER PATTERNS & BEST PRACTICES\n";
412     cout << "\n";
413
414     cout << "\n PATTERN 1: Simple Lambda (Most Common)\n";
415     cout << "      auto deleter = [](Resource* r) { cleanup(r); delete r; };\n";

```

```

414     cout << "    unique_ptr<Resource, decltype(deleter)> ptr(res, deleter);\n";
415
416     cout << "\n PATTERN 2: Stateful Lambda (Captures context)\n";
417     cout << "    auto logger_deleter = [&logger](Resource* r) {\n";
418     cout << "        logger.log(\"Cleanup\");\n";
419     cout << "        cleanup(r); delete r;\n";
420     cout << "    };\n";
421
422     cout << "\n PATTERN 3: Function Object (For shared_ptr)\n";
423     cout << "    struct Deleter {\n";
424     cout << "        void operator()(Resource* r) { cleanup(r); delete r; }\n";
425     cout << "    };\n";
426     cout << "    shared_ptr<Resource> ptr(res, Deleter{});\n";
427
428     cout << "\n PATTERN 4: Function Pointer (C-style)\n";
429     cout << "    void cleanup_func(Resource* r) { cleanup(r); delete r; }\n";
430     cout << "    unique_ptr<Resource, void(*)(Resource*)> ptr(res, cleanup_func
        );\n";
431
432     cout << "\n PATTERN 5: Templated Deleter Factory\n";
433     cout << "    template<typename Cleanup>\n";
434     cout << "    auto make_managed(T* ptr, Cleanup cleanup) {\n";
435     cout << "        return unique_ptr<T, Cleanup>(ptr, cleanup);\n";
436     cout << "    };\n";
437 }
438
439 // =====
440 // PART 4: DRAWBACKS & PITFALLS
441 // =====
442
443 void drawbacks_and_pitfalls() {
444     cout << "\n
445             DRAWBACKS, PITFALLS & COMMON MISTAKES
446             \n";
447
448     cout << "\n PITFALL 1: Type Bloat (unique_ptr)\n";
449     cout << "    Problem: Deleter type is part of unique_ptr type!\n";
450     cout << "    unique_ptr<int, DeleterA> != unique_ptr<int, DeleterB>\n";
451     cout << "    Impact: Cannot store in same container, harder to pass around\n
452             ";
453     cout << "    Solution: Use shared_ptr (type-erases deleter) or function
454             pointers\n";
455
456     cout << "\n PITFALL 2: Size Overhead (unique_ptr with stateful deleters)
457             ";
458     cout << "    sizeof(unique_ptr<int>) = 8 bytes\n";
459     cout << "    sizeof(unique_ptr<int, LambdaDeleter>) = 16+ bytes (deleter
460             stored)\n";
461     cout << "    Impact: Increases memory usage, cache misses\n";
462     cout << "    Solution: Prefer stateless deleters, or use shared_ptr\n";
463
464     cout << "\n PITFALL 3: Don't Forget nullptr Check!\n";
465     cout << "    BAD: auto del = [](T* p) { cleanup(p); delete p; }\n";

```

```

462     cout << "      GOOD: auto del = [](T* p) { if(p) { cleanup(p); delete p; } }\\n";
463     cout << "      Why: Smart pointer may be reset() or moved-from\\n";
464
465     cout << "\\n  PITFALL 4: Don't Capture by Reference (for shared_ptr)\\n";
466     cout << "      BAD: shared_ptr<T>(ptr, [&logger](T* p) { logger.log(); })\\n";
467     cout << "      GOOD: shared_ptr<T>(ptr, [logger=&logger](T* p) { logger->log(); })\\n";
468     cout << "      Why: shared_ptr may outlive captured references (dangling!)\\n";
469     ;
470
471     cout << "\\n  PITFALL 5: Exception Safety in Deleters\\n";
472     cout << "      Deleters should be noexcept (or handle exceptions internally)\\n";
473     cout << "      BAD: auto del = [](T* p) { may_throw(); delete p; };\\n";
474     cout << "      GOOD: auto del = [](T* p) noexcept { try_cleanup(p); delete p; };\\n";
475     cout << "      Why: Exception in destructor = std::terminate()\\n";
476
477     cout << "\\n  PITFALL 6: Don't Delete Twice!\\n";
478     cout << "      DANGEROUS:\\n";
479     cout << "      T* raw = new T();\\n";
480     cout << "      unique_ptr<T, Deleter> p1(raw, deleter);\\n";
481     cout << "      unique_ptr<T, Deleter> p2(raw, deleter); // Double delete !\\n";
482
483     cout << "\\n  PITFALL 7: Don't Use delete on Non-Heap!\\n";
484     cout << "      CRASH:\\n";
485     cout << "      int stack_var = 42;\\n";
486     cout << "      shared_ptr<int>(&stack_var, [](int* p) { delete p; }); // BOOM!\\n";
487     cout << "      CORRECT:\\n";
488     cout << "      shared_ptr<int>(&stack_var, [](int*) {}); // No-op deleter\\n";
489
490     cout << "\\n  PITFALL 8: Performance Cost\\n";
491     cout << "      Every smart pointer deallocation calls deleter (virtual call for shared_ptr)\\n";
492     cout << "      Impact: Slower than raw delete, especially in tight loops\\n";
493     cout << "      Mitigation: Batch allocations, use object pools\\n";
494
495     cout << "\\n  PITFALL 9: Deleter Must Match Allocation!\\n";
496     cout << "      new -> delete           \\n";
497     cout << "      new[] -> delete[]       \\n";
498     cout << "      malloc -> free           \\n";
499     cout << "      new -> free             UNDEFINED BEHAVIOR!\\n";
500     cout << "      new[] -> delete          MEMORY LEAK & CORRUPTION!\\n";
501
502     cout << "\\n  PITFALL 10: Thread Safety (shared_ptr only)\\n";
503     cout << "      shared_ptr ref counting is thread-safe\\n";
504     cout << "      BUT custom deleter execution is NOT synchronized\\n";
505     cout << "      If deleter modifies shared state, YOU must synchronize!\\n";
506 }
```

```

506
507 // =====
508 // PART 5: WHEN TO USE WHAT
509 // =====
510
511 void when_to_use_what() {
512     cout << "\n"                                n\n";
513     cout << "          DECISION TREE: WHEN TO USE CUSTOM DELETERS?\n";
514     cout << "                                \n";
515
516     cout << "\n USE unique_ptr with custom deleter when:\n";
517     cout << " • Single ownership\n";
518     cout << " • Resource is NOT heap memory (FILE*, socket, handle)\n";
519     cout << " • Need custom cleanup (fclose, close, CloseHandle)\n";
520     cout << " • Don't need to share ownership\n";
521     cout << " • Type can vary per deleter (okay with type bloat)\n";
522
523     cout << "\n USE shared_ptr with custom deleter when:\n";
524     cout << " • Shared ownership needed\n";
525     cout << " • Want type erasure (all shared_ptr<T> same type)\n";
526     cout << " • Need to store in containers with different deleters\n";
527     cout << " • Resource may be shared across threads\n";
528     cout << " • Don't mind small overhead (control block)\n";
529
530     cout << "\n DON'T USE custom deleters when:\n";
531     cout << " • Standard delete/delete[] works fine\n";
532     cout << " • Resource is simple heap memory\n";
533     cout << " • Performance is critical (use raw pointers + manual RAII)\n";
534     cout << " • Can use standard containers (vector, string, etc.)\n";
535
536     cout << "\n PREFER alternatives:\n";
537     cout << " • vector<T> instead of unique_ptr<T[]>\n";
538     cout << " • string instead of unique_ptr<char[]>\n";
539     cout << " • RAII wrapper class for complex resources\n";
540     cout << " • Standard library types (fstream, thread, mutex)\n";
541 }
542
543 // =====
544 // PART 6: SIMPLIFIED PATTERNS
545 // =====
546
547 void simplified_patterns() {
548     cout << "\n"                                n\n";
549     cout << "          SIMPLIFIED PATTERNS FOR COMMON CASES\n";
550     cout << "                                \n";
551
552     cout << "\n PATTERN: Generic RAII Wrapper (Simplest!)\n";
553     cout << "     template<typename T, auto Cleanup>\n";
554     cout << "     struct RAIIWrapper {\n";
555     cout << "         T resource;\n";
556     cout << "         RAIIWrapper(T r) : resource(r) {};\n";
557     cout << "         ~RAIIWrapper() { if(resource) Cleanup(resource); }\n";

```

```
558     cout << "         operator T() { return resource; }\n";
559     cout << "     };\n";
560     cout << "     \n";
561     cout << "     Usage: RAIWrapper<FILE*, fclose> file(fopen(\"f.txt\", \"r\"))\n";
562
563     cout << "\n PATTERN: Factory Function (Hide deleter type)\n";
564     cout << "     auto make_file(const char* name, const char* mode) {\n";
565     cout << "         return unique_ptr<FILE, decltype(&fclose)>(\n";
566     cout << "             fopen(name, mode), &fclose);\n";
567     cout << "     }\n";
568     cout << "     \n";
569     cout << "     Usage: auto file = make_file(\"data.txt\", \"r\");\n";
570
571     cout << "\n PATTERN: Type Alias (Reusable)\n";
572     cout << "     using FilePtr = unique_ptr<FILE, decltype(&fclose)>;\n";
573     cout << "     \n";
574     cout << "     FilePtr make_file(const char* name) {\n";
575     cout << "         return FilePtr(fopen(name, \"r\"), &fclose);\n";
576     cout << "     }\n";
577 }
578
579 // =====
580 // MAIN - COMPREHENSIVE DEMO
581 // =====
582
583 int main() {
584     cout << "                                \n";
585     cout << "     UNIVERSAL RESOURCE MANAGER - CUSTOM DELETORS IN C++\n";
586     cout << "                                \n";
587     cout << "     Transform Smart Pointers Into Universal RAIManagers\n";
588     cout << "                                \n";
589
590     // Part 1: Understanding the problem
591     explain_the_problem();
592
593     // Part 2: Real-world examples
594     cout << "\n\n";
595     cout << "                                \n";
596     cout << "     REAL-WORLD EXAMPLES & USE CASES\n";
597     cout << "                                \n";
598
599     file_example();
600     database_example();
601     socket_example();
602     gpu_resource_example();
603     shared_memory_example();
604     resource_pool_example();
605     allocator_with_stats();
606     composite_deleter_example();
607
608     // Part 3: Patterns
```

```
608     deleter_patterns();
609
610     // Part 4: Pitfalls
611     drawbacks_and_pitfalls();
612
613     // Part 5: Decision guide
614     when_to_use_what();
615
616     // Part 6: Simplified patterns
617     simplified_patterns();
618
619     // Summary
620     cout << "\n\n";
621     cout << "                                     \n";
622     cout << "                                     KEY TAKEAWAYS
623     cout << "                                     \n";
624     cout << "                                     \n";
625     cout << "\n BENEFITS:\n";
626     cout << "     1. RAII for ANY resource (not just memory)\n";
627     cout << "     2. Exception-safe cleanup\n";
628     cout << "     3. No manual cleanup code\n";
629     cout << "     4. Prevents resource leaks\n";
630     cout << "     5. Self-documenting (cleanup is obvious)\n";
631     cout << "     6. Composable (combine multiple resources)\n";
632
633     cout << "\n WATCH OUT FOR:\n";
634     cout << "     1. Type bloat with unique_ptr\n";
635     cout << "     2. Size overhead with stateful deleters\n";
636     cout << "     3. nullptr checks in deleters\n";
637     cout << "     4. Exception safety (use noexcept)\n";
638     cout << "     5. Matching allocation/deallocation\n";
639     cout << "     6. Thread safety for shared state\n";
640
641     cout << "\n BEST PRACTICES:\n";
642     cout << "     1. Prefer simple lambdas for deleters\n";
643     cout << "     2. Make deleters noexcept\n";
644     cout << "     3. Always check for nullptr\n";
645     cout << "     4. Use factory functions to hide deleter type\n";
646     cout << "     5. Consider RAII wrapper classes for complex cases\n";
647     cout << "     6. Document deleter behavior clearly\n";
648     cout << "     7. Test cleanup paths (use sanitizers!)\n";
649
650     cout << "\n REMEMBER:\n";
651     cout << "     Custom deleters turn smart pointers into UNIVERSAL resource\n";
652     cout << "     managers. Use them to apply RAII to ANY resource type!\n";
653
654     cout << "\n                                     \n\n";
655     cout << "                                     ALL EXAMPLES COMPLETED SUCCESSFULLY!
656     cout << "                                     \n\n";
657
658     return 0;
```

659 }

## 75 Source Code: VariadicTemplateRecursion.cpp

File: src/VariadicTemplateRecursion.cpp

Repository: [View on GitHub](#)

```
1 // =====
2 // VARIADIC TEMPLATES WITH COMPILE-TIME RECURSION
3 // =====
4 // This example demonstrates how variadic templates enable compile-time
5 // recursion and metaprogramming, eliminating runtime overhead.
6 //
7 // TOPICS COVERED:
8 // 1. Basic variadic template recursion
9 // 2. Compile-time computation (constexpr)
10 // 3. Type manipulation at compile-time
11 // 4. Fold expressions (C++17)
12 // 5. Practical use cases
13 // 6. Embedded systems applications
14 //
15 // KEY CONCEPT: Template recursion happens during compilation,
16 // resulting in ZERO runtime cost - perfect for embedded systems!
17 // =====
18
19 #include <iostream>
20 #include <string>
21 #include <type_traits>
22 #include <array>
23 #include <tuple>
24 #include <cstdint>
25 #include <limits>
26 #include <bitset>
27
28 // =====
29 // SECTION 1: BASIC VARIADIC RECURSION - COMPILE-TIME
30 // =====
31
32 // Base case: no arguments
33 void print_recursive() {
34     std::cout << std::endl;
35 }
36
37 // Recursive case: process first argument, recurse on rest
38 template<typename First, typename... Rest>
39 void print_recursive(First first, Rest... rest) {
40     std::cout << first << " ";
41     print_recursive(rest...); // Compile-time recursion
42 }
43
44 void demonstrate_basic_recursion() {
45     std::cout << "\n==== 1. BASIC VARIADIC RECURSION ===" << std::endl;
46
47     std::cout << "Printing: ";
48     print_recursive(1, "hello", 3.14, 'X', "world");
49 }
```

```
50     std::cout << "\n KEY POINT:" << std::endl;
51     std::cout << " • Compiler generates 5 functions at compile-time" << std
52         ::endl;
53     std::cout << " • Each function handles one argument type" << std::endl;
54     std::cout << " • Zero runtime overhead - all resolved at compile-time"
55         << std::endl;
56     std::cout << " • No loops, no vtables, no dynamic dispatch" << std::endl
57         ;
58 }
59
60 // =====
61 // SECTION 2: COMPILE-TIME COMPUTATION - SUM
62 // =====
63
64 // Base case
65 constexpr int sum() {
66     return 0;
67 }
68
69 // Recursive case
70 template<typename First, typename... Rest>
71 constexpr auto sum(First first, Rest... rest) {
72     return first + sum(rest...);
73 }
74
75 // Compile-time maximum
76 constexpr int max_value() {
77     return -2147483648; // INT_MIN
78 }
79
80 template<typename First, typename... Rest>
81 constexpr auto max_value(First first, Rest... rest) {
82     auto rest_max = max_value(rest...);
83     return (first > rest_max) ? first : rest_max;
84 }
85
86 void demonstrate_compile_time_computation() {
87     std::cout << "\n== 2. COMPILE-TIME COMPUTATION ==" << std::endl;
88
89     // These are computed at COMPILE TIME!
90     constexpr int total = sum(1, 2, 3, 4, 5);
91     constexpr int max_val = max_value(10, 25, 15, 30, 5);
92
93     std::cout << "sum(1,2,3,4,5) = " << total << " (computed at compile-time!)"
94         << std::endl;
95     std::cout << "max(10,25,15,30,5) = " << max_val << " (computed at compile-
96         time!)" << std::endl;
97
98     std::cout << "\n COST ANALYSIS:" << std::endl;
99     std::cout << " Runtime cost: ZERO" << std::endl;
100    std::cout << " Binary size: Just the constant values" << std::endl;
101    std::cout << " CPU cycles: ZERO (values are literals in binary)" << std
102        ::endl;
```

```
98 // Verify with static_assert (compile-time only!)
99 static_assert(sum(1, 2, 3, 4, 5) == 15, "Sum should be 15");
100 static_assert(max_value(10, 25, 15, 30, 5) == 30, "Max should be 30");
101 std::cout << "    static_assert passed - verified at compile-time!" <<
102     std::endl;
103 }
104 // =====
105 // SECTION 3: TYPE MANIPULATION - COMPILE-TIME
106 // =====
107
108 // Check if all types are integral
109 template<typename... Types>
110 struct all_integral;
111
112 // Base case: empty parameter pack
113 template<>
114 struct all_integral<> : std::true_type {};
115
116 // Recursive case
117 template<typename First, typename... Rest>
118 struct all_integral<First, Rest...> {
119     static constexpr bool value =
120         std::is_integral_v<First> && all_integral<Rest...>::value;
121 };
122
123 // Helper variable template (C++17)
124 template<typename... Types>
125 inline constexpr bool all_integral_v = all_integral<Types...>::value;
126
127 // Check if all types are the same
128 template<typename... Types>
129 struct all_same;
130
131 template<typename T>
132 struct all_same<T> : std::true_type {};
133
134 template<typename First, typename Second, typename... Rest>
135 struct all_same<First, Second, Rest...> {
136     static constexpr bool value =
137         std::is_same_v<First, Second> && all_same<Second, Rest...>::value;
138 };
139
140 template<typename... Types>
141 inline constexpr bool all_same_v = all_same<Types...>::value;
142
143 // Get size of largest type
144 template<typename... Types>
145 struct max_sizeof;
146
147 template<typename T>
148 struct max_sizeof<T> {
149     static constexpr size_t value = sizeof(T);
150 };
```

```
151
152 template<typename First, typename... Rest>
153 struct max_sizeof<First, Rest...> {
154     static constexpr size_t value =
155         (sizeof(First) > max_sizeof<Rest...>::value)
156         ? sizeof(First)
157         : max_sizeof<Rest...>::value;
158 };
159
160 template<typename... Types>
161 inline constexpr size_t max_sizeof_v = max_sizeof<Types...>::value;
162
163 void demonstrate_type_manipulation() {
164     std::cout << "\n==== 3. TYPE MANIPULATION AT COMPILE-TIME ===" << std::endl
165         ;
166
167     std::cout << "\nall_integral_v:" << std::endl;
168     std::cout << "    <int, long, short> = "
169         << std::boolalpha << all_integral_v<int, long, short> << std::endl;
170
171     std::cout << "    <int, double, char> = "
172         << all_integral_v<int, double, char> << std::endl;
173
174     std::cout << "\nall_same_v:" << std::endl;
175     std::cout << "    <int, int, int> = "
176         << all_same_v<int, int, int> << std::endl;
177     std::cout << "    <int, long, int> = "
178         << all_same_v<int, long, int> << std::endl;
179
180     std::cout << "\nmax_sizeof_v:" << std::endl;
181     std::cout << "    <char, int, long> = "
182         << max_sizeof_v<char, int, long> << " bytes" << std::endl;
183     std::cout << "    <double, int, float> = "
184         << max_sizeof_v<double, int, float> << " bytes" << std::endl;
185
186     std::cout << "\n ALL COMPUTED AT COMPILE-TIME!" << std::endl;
187     std::cout << "    No runtime type checking needed" << std::endl;
188     std::cout << "    Perfect for template constraints" << std::endl;
189 }
190
191 // =====
192 // SECTION 4: FOLD EXPRESSIONS (C++17) - SIMPLIFIED RECURSION
193 // =====
194
195 // Old way: explicit recursion
196 template<typename... Args>
197 constexpr auto sum_old_way(Args... args) {
198     return (args + ...); // Fold expression!
199 }
200
201 // Fold expressions for logical operations
202 template<typename... Args>
203 constexpr bool all_positive(Args... args) {
204     return ((args > 0) && ...); // Fold with &&
```

```

203 }
204
205 template<typename... Args>
206 constexpr bool any_negative(Args... args) {
207     return ((args < 0) || ...); // Fold with ||
208 }
209
210 // Fold for comma operator (call function for each)
211 template<typename... Args>
212 void print_all_fold(Args... args) {
213     ((std::cout << args << " "), ...);
214     std::cout << std::endl;
215 }
216
217 void demonstrate_fold_expressions() {
218     std::cout << "\n==== 4. FOLD EXPRESSIONS (C++17) ===" << std::endl;
219
220     std::cout << "\nArithmetic folds:" << std::endl;
221     std::cout << "    sum(1,2,3,4,5) = " << sum_old_way(1, 2, 3, 4, 5) << std::endl;
222
223     std::cout << "\nLogical folds:" << std::endl;
224     std::cout << "    all_positive(1,2,3) = "
225         << std::boolalpha << all_positive(1, 2, 3) << std::endl;
226     std::cout << "    all_positive(1,-2,3) = "
227         << all_positive(1, -2, 3) << std::endl;
228     std::cout << "    any_negative(1,2,3) = "
229         << any_negative(1, 2, 3) << std::endl;
230     std::cout << "    any_negative(1,-2,3) = "
231         << any_negative(1, -2, 3) << std::endl;
232
233     std::cout << "\nPrint with fold: ";
234     print_all_fold(1, "hello", 3.14, "world");
235
236     std::cout << "\n FOLD EXPRESSIONS:" << std::endl;
237     std::cout << " • Simpler than explicit recursion" << std::endl;
238     std::cout << " • Still compile-time only" << std::endl;
239     std::cout << " • More readable code" << std::endl;
240     std::cout << " • Zero runtime cost" << std::endl;
241 }
242
243 // =====
244 // SECTION 5: PRACTICAL USE CASE - TYPE-SAFE PRINTF
245 // =====
246
247 // Validate format string at compile-time
248 template<typename... Args>
249 void safe_printf(const char* format, Args... args) {
250     // Count format specifiers
251     int format_count = 0;
252     for (const char* p = format; *p; ++p) {
253         if (*p == '%' && *(p+1) != '%') {
254             ++format_count;
255         }

```

```
256     }
257
258     int arg_count = sizeof...(args);
259
260     if (format_count != arg_count) {
261         std::cout << "ERROR: Format string has " << format_count
262             << " specifiers but " << arg_count << " arguments!" << std:::
263             endl;
264         return;
265     }
266
267     printf(format, args...);
268 }
269
270 void demonstrate_safe_printf() {
271     std::cout << "\n==== 5. TYPE-SAFE PRINTF ===" << std::endl;
272
273     std::cout << "\n Correct usage:" << std::endl;
274     safe_printf("Integer: %d, String: %s, Float: %.2f\n", 42, "hello", 3.14);
275
276     std::cout << "\n Incorrect usage (caught at runtime):" << std::endl;
277     safe_printf("Two specifiers: %d %s\n", 42); // Missing argument!
278
279     std::cout << "\n BETTER APPROACH:" << std::endl;
280     std::cout << "    Use C++20 std::format for complete type safety!" << std:::
281             endl;
282     std::cout << "    Or compile-time format string validation" << std::endl;
283 }
284
285 // =====
286 // SECTION 6: COMPILE-TIME ARRAY INITIALIZATION
287 // =====
288
289 // Create array with compile-time computed values
290 template<typename... Args>
291 constexpr auto make_array(Args... args) {
292     return std::array<std::common_type_t<Args...>, sizeof...(Args)>{args...};
293 }
294
295 // Fibonacci at compile-time
296 constexpr int fib(int n) {
297     return (n <= 1) ? n : fib(n-1) + fib(n-2);
298 }
299
300 // Generate array of first N fibonacci numbers
301 template<size_t... Is>
302 constexpr auto fib_array_impl(std::index_sequence<Is...>) {
303     return std::array<int, sizeof...(Is)>{fib(Is)...};
304 }
305
306 template<size_t N>
307 constexpr auto fib_array() {
308     return fib_array_impl(std::make_index_sequence<N>{});
309 }
```

```
308
309 void demonstrate_compile_time_array() {
310     std::cout << "\n==== 6. COMPILE-TIME ARRAY INITIALIZATION ===" << std::endl
311     ;
312
313     // Array created at compile-time!
314     constexpr auto arr = make_array(10, 20, 30, 40, 50);
315     std::cout << "\nArray: ";
316     for (auto val : arr) {
317         std::cout << val << " ";
318     }
319     std::cout << std::endl;
320
321     // Fibonacci array computed at compile-time!
322     constexpr auto fibs = fib_array<10>();
323     std::cout << "\nFirst 10 Fibonacci numbers (compile-time): ";
324     for (auto val : fibs) {
325         std::cout << val << " ";
326     }
327     std::cout << std::endl;
328
329     std::cout << "\n PERFORMANCE:" << std::endl;
330     std::cout << " • Arrays are in .rodata section (read-only)" << std::endl
331     ;
332     std::cout << " • No initialization code at runtime" << std::endl;
333     std::cout << " • Values embedded in binary" << std::endl;
334 }
335
336 // =====
337 // SECTION 7: EMBEDDED SYSTEMS - REGISTER CONFIGURATION
338 // =====
339
340 // Compile-time register bit field computation
341 template<uint32_t... Bits>
342 struct RegisterBits {
343     static constexpr uint32_t value = ((1u << Bits) | ...);
344 };
345
346 // Compile-time register configuration
347 enum class GPIO_Pin : uint8_t {
348     Pin0 = 0, Pin1 = 1, Pin2 = 2, Pin3 = 3,
349     Pin4 = 4, Pin5 = 5, Pin6 = 6, Pin7 = 7
350 };
351
352 template<GPIO_Pin... Pins>
353 struct GPIO_Config {
354     static constexpr uint8_t mask = ((1u << static_cast<uint8_t>(Pins)) | ...
355     ;
356
357     static void set_output() {
358         // This would write to actual hardware register
359         std::cout << "    Setting pins 0x" << std::hex
360             << static_cast<int>(mask) << std::dec
361             << " as output" << std::endl;
362 }
```

```

359     }
360 };
361
362 // Compile-time validation of peripheral configuration
363 template<uint32_t ClockFreq, uint32_t DesiredBaud>
364 struct UART_Divider {
365     static constexpr uint32_t divider = ClockFreq / (16 * DesiredBaud);
366     static constexpr uint32_t actual_baud = ClockFreq / (16 * divider);
367     static constexpr uint32_t error_percent =
368         (actual_baud > DesiredBaud)
369             ? ((actual_baud - DesiredBaud) * 100 / DesiredBaud)
370             : ((DesiredBaud - actual_baud) * 100 / DesiredBaud);
371
372     static_assert(divider > 0 && divider < 65536,
373                     "UART divider out of range!");
374     static_assert(error_percent < 3,
375                     "Baud rate error exceeds 3%!");
376 };
377
378 void demonstrate_embedded_usage() {
379     std::cout << "\n==== 7. EMBEDDED SYSTEMS APPLICATIONS ===" << std::endl;
380
381     std::cout << "\n Register bit configuration (compile-time):" << std::endl
382         ;
383     constexpr uint32_t bits = RegisterBits<0, 2, 4, 7>::value;
384     std::cout << "    Bits 0,2,4,7 = 0x" << std::hex << bits << std::dec
385         << " (0b" << std::bitset<8>(bits) << ")" << std::endl;
386
387     std::cout << "\n GPIO configuration (compile-time):" << std::endl;
388     using MyGPIO = GPIO_Config<GPIO_Pin::Pin0, GPIO_Pin::Pin3, GPIO_Pin::Pin7
389         >;
390     std::cout << "    GPIO mask: 0x" << std::hex
391         << static_cast<int>(MyGPIO::mask) << std::dec << std::endl;
392     MyGPIO::set_output();
393
394     std::cout << "\n UART configuration (compile-time validated):" << std::endl;
395
396     using UART = UART_Divider<16000000, 9600>;
397     std::cout << "    Clock: 16MHz, Baud: 9600" << std::endl;
398     std::cout << "    Divider: " << UART::divider
399         << " (computed at compile-time)" << std::endl;
400
401     std::cout << "\n EMBEDDED BENEFITS:" << std::endl;
402     std::cout << "    Configuration errors caught at compile-time" << std::endl;
403     std::cout << "    Zero runtime overhead" << std::endl;
404     std::cout << "    No magic numbers - everything is named" << std::endl;
405     std::cout << "    Hardware constraints validated at compile-time" << std::endl;
406
407 }
408
409 // =====
410 // SECTION 8: TUPLE OPERATIONS - COMPILE-TIME RECURSION
411 // =====

```

```
408 // Print tuple recursively at compile-time
409 template<size_t Index = 0, typename... Types>
410 void print_tuple(const std::tuple<Types...>& t) {
411     if constexpr (Index < sizeof...(Types)) {
412         std::cout << std::get<Index>(t);
413         if constexpr (Index + 1 < sizeof...(Types)) {
414             std::cout << ", ";
415         }
416         print_tuple<Index + 1>(t);
417     }
418 }
419
420
421 // Sum all tuple elements (if numeric)
422 template<size_t Index = 0, typename... Types>
423 auto sum_tuple(const std::tuple<Types...>& t) {
424     if constexpr (Index < sizeof...(Types)) {
425         if constexpr (Index + 1 < sizeof...(Types)) {
426             return std::get<Index>(t) + sum_tuple<Index + 1>(t);
427         } else {
428             return std::get<Index>(t);
429         }
430     } else {
431         return 0;
432     }
433 }
434
435 void demonstrate_tuple_operations() {
436     std::cout << "\n== 8. TUPLE OPERATIONS WITH RECURSION ==" << std::endl;
437
438     auto mixed_tuple = std::make_tuple(1, "hello", 3.14, 'X');
439     std::cout << "\nTuple: ";
440     print_tuple(mixed_tuple);
441     std::cout << std::endl;
442
443     auto numeric_tuple = std::make_tuple(10, 20, 30, 40, 50);
444     std::cout << "\nNumeric tuple: ";
445     print_tuple(numeric_tuple);
446     std::cout << std::endl;
447
448     auto total = sum_tuple(numeric_tuple);
449     std::cout << "Sum: " << total << std::endl;
450
451     std::cout << "\n if constexpr:" << std::endl;
452     std::cout << " •  Compile-time conditional" << std::endl;
453     std::cout << " •  Dead branches eliminated by compiler" << std::endl;
454     std::cout << " •  Perfect for template recursion termination" << std::endl;
455 }
456
457 // =====
458 // SECTION 9: WHEN TO USE COMPILE-TIME RECURSION
459 // =====
```

```
461 void explain_when_to_use() {
462     std::cout << "\n" << std::string(70, '=') << std::endl;
463     std::cout << "WHEN TO USE COMPILE-TIME RECURSION:\n";
464     std::cout << std::string(70, '=') << std::endl;
465
466     std::cout << "\n USE FOR:" << std::endl;
467     std::cout << "\n1. TYPE COMPUTATIONS:" << std::endl;
468     std::cout << " • Finding largest type in parameter pack" << std::endl;
469     std::cout << " • Type trait combinations" << std::endl;
470     std::cout << " • Template metaprogramming" << std::endl;
471
472     std::cout << "\n2. COMPILE-TIME CONSTANTS:" << std::endl;
473     std::cout << " • Mathematical computations (Fibonacci, factorial)" <<
474         std::endl;
474     std::cout << " • Configuration values" << std::endl;
475     std::cout << " • Lookup tables" << std::endl;
476
477     std::cout << "\n3. HETEROGENEOUS COLLECTIONS:" << std::endl;
478     std::cout << " • Tuple operations" << std::endl;
479     std::cout << " • Variant handling" << std::endl;
480     std::cout << " • Type-safe function wrappers" << std::endl;
481
482     std::cout << "\n4. EMBEDDED SYSTEMS:" << std::endl;
483     std::cout << " • Hardware register configuration" << std::endl;
484     std::cout << " • Peripheral setup validation" << std::endl;
485     std::cout << " • Zero-overhead abstractions" << std::endl;
486     std::cout << " • Compile-time constraint checking" << std::endl;
487
488     std::cout << "\n5. CODE GENERATION:" << std::endl;
489     std::cout << " • Unrolling loops at compile-time" << std::endl;
490     std::cout << " • Generating specialized functions" << std::endl;
491     std::cout << " • Avoiding runtime dispatch" << std::endl;
492
493     std::cout << "\n DON'T USE FOR:" << std::endl;
494     std::cout << "\n1. RUNTIME DATA:" << std::endl;
495     std::cout << " • User input processing" << std::endl;
496     std::cout << " • Dynamic collections" << std::endl;
497     std::cout << " • Data-dependent logic" << std::endl;
498
499     std::cout << "\n2. EXCESSIVE RECURSION:" << std::endl;
500     std::cout << " • Deep recursion slows compilation" << std::endl;
501     std::cout << " • Can cause compiler errors (recursion limits)" << std::
502         endl;
502     std::cout << " • Use fold expressions when possible" << std::endl;
503
504     std::cout << "\n3. SIMPLE CASES:" << std::endl;
505     std::cout << " • Use standard library when available" << std::endl;
506     std::cout << " • Don't reinvent the wheel" << std::endl;
507
508     std::cout << "\n MODERN C++ ALTERNATIVES:" << std::endl;
509     std::cout << " • C++17: Fold expressions (replace simple recursion)" <<
510         std::endl;
510     std::cout << " • C++17: if constexpr (cleaner recursion)" << std::endl;
511     std::cout << " • C++20: Concepts (constrain templates)" << std::endl;
```

```

512     std::cout << " • C++20: consteval (force compile-time evaluation)" <<
513         std::endl;
514     }
515
516 // =====
517 // SECTION 10: PERFORMANCE COMPARISON
518 // =====
519
520 // Runtime version
521 int sum_runtime(int* values, size_t count) {
522     int total = 0;
523     for (size_t i = 0; i < count; ++i) {
524         total += values[i];
525     }
526     return total;
527 }
528
529 void demonstrate_performance_comparison() {
530     std::cout << "\n" << std::string(70, '=') << std::endl;
531     std::cout << "PERFORMANCE: COMPILE-TIME VS RUNTIME:\n";
532     std::cout << std::string(70, '=') << std::endl;
533
534     std::cout << "\n COMPILE-TIME (variadic template):" << std::endl;
535     constexpr int compile_time_sum = sum(1, 2, 3, 4, 5);
536     std::cout << "    Result: " << compile_time_sum << std::endl;
537     std::cout << "    Binary: MOV EAX, 15 (single instruction!)" << std::endl;
538     std::cout << "    Cost: 0 CPU cycles at runtime" << std::endl;
539     std::cout << "    Size: ~4 bytes (just the constant)" << std::endl;
540
541     std::cout << "\n RUNTIME (loop):" << std::endl;
542     int values[] = {1, 2, 3, 4, 5};
543     int runtime_sum = sum_runtime(values, 5);
544     std::cout << "    Result: " << runtime_sum << std::endl;
545     std::cout << "    Binary: ~20+ instructions (loop, increment, add)" << std::endl;
546     std::cout << "    Cost: ~10+ CPU cycles" << std::endl;
547     std::cout << "    Size: ~40+ bytes (loop code)" << std::endl;
548
549     std::cout << "\n SPEEDUP: Infinite (0 vs ~10 cycles)" << std::endl;
550     std::cout << "    SIZE: 10x smaller (4 vs 40+ bytes)" << std::endl;
551
552     std::cout << "\n WHEN COMPILE-TIME WINS:" << std::endl;
553     std::cout << " • Values known at compile-time" << std::endl;
554     std::cout << " • Constant configuration" << std::endl;
555     std::cout << " • Type computations" << std::endl;
556     std::cout << " • Safety-critical embedded systems" << std::endl;
557 }
558
559 // =====
560 // SECTION 11: DRAWBACKS AND PITFALLS
561 // =====
562
563 // Pitfall 1: Deep recursion causes slow compilation
template<int N>

```

```
564 struct Fibonacci {
565     static constexpr int value = Fibonacci<N-1>::value + Fibonacci<N-2>::value
566     ;
567 };
568 template<> struct Fibonacci<0> { static constexpr int value = 0; };
569 template<> struct Fibonacci<1> { static constexpr int value = 1; };
570
571 // Pitfall 2: Exponential template instantiations
572 template<typename... Args>
573 struct InstantiationExplosion {
574     // Each call creates multiple instantiations
575     using type = std::tuple<Args..., Args...>; // Doubles the types!
576 };
577
578 // Pitfall 3: Cryptic error messages
579 template<typename... Args>
580 constexpr auto bad_function(Args... args) {
581     // This will produce a HORRIBLE error message if types don't support
582     // operator+
583     return (args + ...);
584 }
585
586 // Pitfall 4: Code bloat - each instantiation generates code
587 template<typename T>
588 T generic_sort(T value) {
589     // Even though this does nothing different, compiler generates
590     // separate code for EACH type instantiation
591     std::cout << "Sorting single value: " << value << std::endl;
592     return value;
593 }
594
595 // Pitfall 5: Hidden recursion limits
596 template<int N>
597 struct DeepRecursion {
598     static constexpr int value = DeepRecursion<N-1>::value + 1;
599 };
600 template<> struct DeepRecursion<0> { static constexpr int value = 0; };
601
602 void demonstrate_drawbacks_and_pitfalls() {
603     std::cout << "\n" << std::string(70, '=') << std::endl;
604     std::cout << "DRAWBACKS AND POTENTIAL PITFALLS:\n";
605     std::cout << std::string(70, '=') << std::endl;
606
607     std::cout << "\n PITFALL 1: SLOW COMPILATION" << std::endl;
608     std::cout << "    Problem: Deep template recursion is SLOW to compile" <<
609     std::endl;
610     std::cout << "    Example: Fibonacci<40>::value takes FOREVER to compile!" <<
611     std::endl;
612
613     // Safe to compute small values
614     constexpr int fib10 = Fibonacci<10>::value;
615     std::cout << "    Fibonacci<10> = " << fib10 << " (tolerable)" << std::endl;
616 }
```

```
613 // Fibonacci<40> would take minutes to compile!
614 // constexpr int fib40 = Fibonacci<40>::value; // DON'T DO THIS!
615
616
617 std::cout << "\n      Solution:" << std::endl;
618 std::cout << " •      Use constexpr functions instead of template
619   recursion" << std::endl;
620 std::cout << " •      constexpr int fib(int n) { return (n<=1) ? n : fib(n
621   -1)+fib(n-2); }" << std::endl;
622 std::cout << " •      Much faster compilation, same runtime performance"
623   << std::endl;
624
625 std::cout << "\n PITFALL 2: CRYPTIC ERROR MESSAGES" << std::endl;
626 std::cout << "      Problem: Template errors are EXTREMELY hard to read" <<
627   std::endl;
628 std::cout << "      Example error (trying to add strings with +):" << std::
629   endl;
630 std::cout << "      error: invalid operands to binary expression" << std::
631   endl;
632 std::cout << "      ('std::string' and 'std::string')" << std::endl;
633 std::cout << "      in instantiation of function template specialization"
634   << std::endl;
635 std::cout << "      'bad_function<std::string, std::string>' requested
636   here" << std::endl;
637 std::cout << "      ... 50 more lines of template instantiation backtrace
638   ..." << std::endl;
639
640 std::cout << "\n      Solution:" << std::endl;
641 std::cout << " •      Use C++20 concepts to constrain templates EARLY" <<
642   std::endl;
643 std::cout << " •      Add static_assert with clear messages" << std::endl;
644 std::cout << " •      Use std::enable_if with meaningful names" << std::
645   endl;
646
647 std::cout << "\n PITFALL 3: CODE BLOAT" << std::endl;
648 std::cout << "      Problem: Each template instantiation generates SEPARATE
649   code" << std::endl;
650
651 // Each call generates separate function in binary
652 generic_sort(10);
653 generic_sort(20.5);
654 generic_sort(std::string("hello"));
655
656 std::cout << "\n      Impact on binary size:" << std::endl;
657 std::cout << " •      generic_sort<int>      : ~50 bytes" << std::endl;
658 std::cout << " •      generic_sort<double>   : ~50 bytes" << std::endl;
659 std::cout << " •      generic_sort<string>  : ~80 bytes" << std::endl;
660 std::cout << " •      Total                  : ~180 bytes for same logic!"
661   << std::endl;
662
663 std::cout << "\n      Solution:" << std::endl;
664 std::cout << " •      Use type erasure for runtime polymorphism" << std::
665   endl;
666 std::cout << " •      Extract common code into non-template functions" <<
```

```
        std::endl;
653  std::cout << " •      Use extern template to prevent duplicate
654    instantiations" << std::endl;
655  std::cout << "\n PITFALL 4: COMPILER RECURSION LIMITS" << std::endl;
656  std::cout << " •      Problem: Compilers have hard limits on template depth" <<
657    std::endl;
658  std::cout << " •      Typical limit: 256-1024 instantiation depth" << std::endl
659    ;
660
661 // This works (small depth)
662 constexpr int depth_100 = DeepRecursion<100>::value;
663 std::cout << " •      DeepRecursion<100> = " << depth_100 << " (works)" << std
664    ::endl;
665
666 // DeepRecursion<2000> would hit the limit!
667 // constexpr int depth_2000 = DeepRecursion<2000>::value; // ERROR!
668
669 std::cout << "\n •      Error you'd see:" << std::endl;
670 std::cout << " •      fatal error: recursive template instantiation exceeded
671    " << std::endl;
672 std::cout << " •      maximum depth of 1024" << std::endl;
673
674 std::cout << "\n •      Solution:" << std::endl;
675 std::cout << " •      Use constexpr functions instead" << std::endl;
676 std::cout << " •      Use fold expressions (no recursion)" << std::endl;
677 std::cout << " •      Increase limit: -ftemplate-depth=2000 (not
678    recommended!)" << std::endl;
679
680 std::cout << "\n PITFALL 5: DEBUGGING NIGHTMARE" << std::endl;
681 std::cout << " •      Problem: Can't step through template code" << std::endl;
682 std::cout << " •      Debugger shows instantiated code, not template" << std
683    ::endl;
684 std::cout << " •      Type names are mangled and unreadable" << std::endl;
685 std::cout << " •      Call stack filled with template instantiation frames"
686    << std::endl;
687
688 std::cout << "\n •      Solution:" << std::endl;
689 std::cout << " •      Use std::cout for compile-time debugging" << std::
690    endl;
691 std::cout << " •      Use static_assert to validate assumptions" << std::
692    endl;
693 std::cout << " •      Test with simple types first" << std::endl;
694 std::cout << " •      Use __PRETTY_FUNCTION__ to see instantiated types"
695    << std::endl;
696
697 std::cout << "\n PITFALL 6: EXCESSIVE MEMORY DURING COMPILATION" << std::
698    endl;
699 std::cout << " •      Problem: Complex templates use LOTS of compiler memory"
700    << std::endl;
701 std::cout << " •      Impact:" << std::endl;
702 std::cout << " •      Simple project: ~500MB compiler memory" << std::endl
703    ;
704 std::cout << " •      Heavy template use: 4GB+ compiler memory" << std::endl
```

```
        endl;
692     std::cout << " •      Can cause out-of-memory errors in CI/CD" << std::
693         endl;
694
694     std::cout << "\n      Solution:" << std::endl;
695     std::cout << " •      Split template-heavy code across files" << std::endl
696         ;
696     std::cout << " •      Use forward declarations" << std::endl;
697     std::cout << " •      Use precompiled headers for common instantiations"
698         << std::endl;
698     std::cout << " •      Consider compilation time in design decisions" <<
699         std::endl;
699 }
700
701 // =====
702 // SECTION 12: REAL-WORLD EXAMPLES OF THINGS GOING WRONG
703 // =====
704
705 // Example 1: Accidental O(2^N) compilation complexity
706 template<int N, typename... Args>
707 struct BadMetaFunction {
708     // Each level DOUBLES the work - exponential explosion!
709     using result = typename BadMetaFunction<N-1, Args..., Args...>::result;
710 };
711
712 template<typename... Args>
713 struct BadMetaFunction<0, Args...> {
714     using result = std::tuple<Args...>;
715 };
716
717 // Example 2: Forgetting base case - infinite recursion
718 // template<typename T, typename... Rest>
719 // struct ForgetBaseCase {
720 //     static constexpr int value = 1 + ForgetBaseCase<Rest...>::value;
721 //     // OOPS! No base case for empty pack!
722 // };
723
724 // Example 3: Ambiguous overload resolution
725 template<typename T>
726 void overload_problem(T value) {
727     std::cout << "Generic version" << std::endl;
728 }
729
730 template<typename T, typename... Args>
731 void overload_problem(T first, Args... rest) {
732     std::cout << "Variadic version" << std::endl;
733 }
734
735 // overload_problem(42); // Ambiguous! Which one to call?
736
737 // Example 4: Type deduction failure
738 template<typename... Args>
739 auto broken_deduction(Args... args) {
740     // This fails if args is empty!
```

```
741 // auto first = args...; // ERROR: can't expand to nothing
742 return 0;
743 }
744
745 void demonstrate_real_world_failures() {
746     std::cout << "\n" << std::string(70, '=') << std::endl;
747     std::cout << "REAL-WORLD EXAMPLES OF FAILURES:\n";
748     std::cout << std::string(70, '=') << std::endl;
749
750     std::cout << "\n FAILURE 1: EXPONENTIAL COMPILE TIME" << std::endl;
751     std::cout << "    Code: BadMetaFunction<N, Types...>" << std::endl;
752     std::cout << "    Problem: Each recursion DOUBLES the type list" << std::endl;
753     std::cout << "    Complexity: O(2^N) template instantiations!" << std::endl;
754     std::cout << "    Result:" << std::endl;
755     std::cout << "    •      N=5: 32 instantiations (0.1s)" << std::endl;
756     std::cout << "    •      N=10: 1024 instantiations (5s)" << std::endl;
757     std::cout << "    •      N=15: 32768 instantiations (MINUTES!)" << std::endl;
758     std::cout << "    •      N=20: Out of memory / compiler crash" << std::endl;
759
760     std::cout << "\n FAILURE 2: MISSING BASE CASE" << std::endl;
761     std::cout << "    Error message:" << std::endl;
762     std::cout << "    fatal error: template instantiation depth exceeds
763     maximum" << std::endl;
764     std::cout << "    note: use -ftemplate-depth= to increase the maximum"
765     << std::endl;
766     std::cout << "    Root cause: Forgot to specialize for empty parameter pack
767     " << std::endl;
768     std::cout << "    Fix: Always provide template<> specialization" << std::endl;
769
770     std::cout << "\n FAILURE 3: AMBIGUOUS OVERLOADS" << std::endl;
771     std::cout << "    Problem: overload_problem(42)" << std::endl;
772     std::cout << "    Both match:" << std::endl;
773     std::cout << "    •      void overload_problem(T)           // T=int" << std::endl;
774     std::cout << "    •      void overload_problem(T, Args...) // T=int, Args=<>
775     " << std::endl;
776     std::cout << "    Solution: Use SFINAE or concepts to disambiguate" << std::endl;
777
778     std::cout << "\n FAILURE 4: EMPTY PARAMETER PACK" << std::endl;
779     std::cout << "    Problem: broken_deduction() called with no arguments" <<
780     std::endl;
781     std::cout << "    Can't expand empty pack in certain contexts" << std::endl;
782     std::cout << "    Solution: Always check sizeof...(Args) or provide base
783     case" << std::endl;
784
785     std::cout << "\n FAILURE 5: TYPE DEDUCTION CONFUSION" << std::endl;
786     std::cout << "    Problem: auto deduction with parameter packs" << std::endl;
787     std::cout << "    Example:" << std::endl;
```

```

782     std::cout << "      template<typename... Args>" << std::endl;
783     std::cout << "      auto func(Args... args) {" << std::endl;
784     std::cout << "          return args; // Which one? ERROR!" << std::endl;
785     std::cout << "      }" << std::endl;
786     std::cout << "      Solution: Be explicit about which argument to return" <<
787         std::endl;
788 }
789
790 // =====
791 // SECTION 13: BEST PRACTICES TO AVOID PITFALLS
792 // =====
793 void explain_best_practices() {
794     std::cout << "\n" << std::string(70, '=') << std::endl;
795     std::cout << "BEST PRACTICES TO AVOID PITFALLS:\n";
796     std::cout << std::string(70, '=') << std::endl;
797
798     std::cout << "\n DO:" << std::endl;
799
800     std::cout << "\n1. USE FOLD EXPRESSIONS (C++17+):" << std::endl;
801     std::cout << "      BAD: Explicit recursion for sum" << std::endl;
802     std::cout << "          template<typename T>" << std::endl;
803     std::cout << "          T sum(T t) { return t; }" << std::endl;
804     std::cout << "          template<typename T, typename... Args>" << std::endl;
805     std::cout << "          T sum(T t, Args... args) { return t + sum(args...); }"
806         << std::endl;
807     std::cout << "\n      GOOD: Fold expression" << std::endl;
808     std::cout << "          template<typename... Args>" << std::endl;
809     std::cout << "          auto sum(Args... args) { return (args + ...); }" <<
810         std::endl;
811
812     std::cout << "\n2. USE CONSTEXPR FUNCTIONS OVER TEMPLATE RECURSION:" <<
813         std::endl;
814     std::cout << "      BAD: template<int N> struct Fib { ... };" << std::endl;
815     std::cout << "          ;"
816     std::cout << "      GOOD: constexpr int fib(int n) { ... }" << std::endl;
817     std::cout << "      Benefits: Faster compilation, clearer code, easier to
818         debug" << std::endl;
819
820     std::cout << "\n3. USE CONCEPTS TO CONSTRAIN EARLY (C++20):" << std::endl;
821     std::cout << "      BAD: template<typename T> void func(T t) { t + t; }"
822         << std::endl;
823     std::cout << "      GOOD: template<std::integral T> void func(T t) { t + t;
824         }" << std::endl;
825     std::cout << "      Benefits: Clear error messages, compile-time validation"
826         << std::endl;
827
828     std::cout << "\n4. ALWAYS PROVIDE BASE CASE:" << std::endl;
829     std::cout << "      BAD: Only recursive case" << std::endl;
830     std::cout << "      GOOD: template<> struct Base { }; + recursive case"
831         << std::endl;
832     std::cout << "      Benefits: Avoid infinite recursion errors" << std::endl;
833
834     std::cout << "\n5. USE static_assert WITH CLEAR MESSAGES:" << std::endl;

```

```
826     std::cout << "      GOOD: static_assert(sizeof...(Args) > 0, \"Need at"
827     least one argument\");" << std::endl;
828     std::cout << "      Benefits: Catch errors early with helpful messages" <<
829     std::endl;
830
831     std::cout << "\n6. TEST WITH SIMPLE TYPES FIRST:" << std::endl;
832     std::cout << " • Start with int, double before complex types" << std::
833     endl;
834     std::cout << " • Verify logic works before adding complexity" << std::
835     endl;
836     std::cout << " • Use small parameter counts during development" << std::
837     endl;
838
839     std::cout << "\n7. MEASURE COMPILATION TIME:" << std::endl;
840     std::cout << " • Use -ftime-report (GCC/Clang)" << std::endl;
841     std::cout << " • Track build times in CI/CD" << std::endl;
842     std::cout << " • Set reasonable limits (e.g., <10s per file)" << std::
843     endl;
844
845     std::cout << "\n8. DOCUMENT CONSTRAINTS:" << std::endl;
846     std::cout << " • Specify required type traits" << std::endl;
847     std::cout << " • Document recursion depth limits" << std::endl;
848     std::cout << " • Provide usage examples" << std::endl;
849
850     std::cout << "\n DON'T:" << std::endl;
851     std::cout << "1. DON'T use template recursion for deep computations" <<
852     std::endl;
853     std::cout << "2. DON'T instantiate with many types unless necessary" <<
854     std::endl;
855     std::cout << "3. DON'T ignore compilation time during development" << std
856     ::endl;
857     std::cout << "4. DON'T use variadic templates for runtime data" << std::
858     endl;
859     std::cout << "5. DON'T nest template recursion deeply" << std::endl;
860
861     std::cout << "\n GOLDEN RULE:" << std::endl;
862     std::cout << " If it takes more than 1 second to compile," << std::endl;
863     std::cout << " you're probably doing it wrong!" << std::endl;
864 }
865
866 // =====
867 // MAIN FUNCTION
868 // =====
869
870 int main() {
871     std::cout << "\n";
872     std::cout << "                                     \n";
873     std::cout << "      VARIADIC TEMPLATES & COMPILE-TIME RECURSION
874                                     \n";
875     std::cout << "                                     \n";
876     std::cout << "      Zero runtime cost • Type safety • Perfect for embedded
877                                     \n";
878     std::cout << "                                     \n";
879
```

```
867 demonstrate_basic_recursion();
868 demonstrate_compile_time_computation();
869 demonstrate_type_manipulation();
870 demonstrate_fold_expressions();
871 demonstrate_safe_printf();
872 demonstrate_compile_time_array();
873 demonstrate_embedded_usage();
874 demonstrate_tuple_operations();
875 explain_when_to_use();
876 demonstrate_performance_comparison();
877 demonstrate_drawbacks_and_pitfalls();
878 demonstrate_real_world_failures();
879 explain_best_practices();

880
881 std::cout << "\n" << std::string(70, '=')
882 << std::endl;
883 std::cout << "SUMMARY:\n";
884 std::cout << std::string(70, '=')
885 << std::endl;

886 std::cout << "\n KEY TAKEAWAYS:" << std::endl;
887 std::cout << "\n1. COMPILE-TIME RECURSION:" << std::endl;
888 std::cout << " • All computation happens during compilation" << std::endl;
889 std::cout << " • Zero runtime cost" << std::endl;
890 std::cout << " • Results embedded in binary as constants" << std::endl;

891
892 std::cout << "\n2. VARIADIC TEMPLATES:" << std::endl;
893 std::cout << " • Handle any number of arguments" << std::endl;
894 std::cout << " • Type-safe heterogeneous processing" << std::endl;
895 std::cout << " • Perfect for metaprogramming" << std::endl;

896
897 std::cout << "\n3. EMBEDDED SYSTEMS:" << std::endl;
898 std::cout << " • Validate hardware configurations at compile-time" <<
899 std::endl;
900 std::cout << " • Zero-overhead abstractions" << std::endl;
901 std::cout << " • Catch errors before deployment" << std::endl;
902 std::cout << " • Minimal binary size impact" << std::endl;

903
904 std::cout << "\n4. MODERN FEATURES:" << std::endl;
905 std::cout << " • Use fold expressions (C++17) when possible" << std::endl;
906 std::cout << " • Use if constexpr for cleaner recursion" << std::endl;
907 std::cout << " • Use constexpr/constexpr for clarity" << std::endl;
908 std::cout << " • Combine with concepts for constraints" << std::endl;

909
910 std::cout << "\n CRITICAL PITFALLS TO AVOID:" << std::endl;
911 std::cout << "\n1. COMPILATION TIME:" << std::endl;
912 std::cout << " • Template recursion can be VERY slow" << std::endl;
913 std::cout << " • Prefer constexpr functions over template recursion" <<
914 std::endl;
915 std::cout << " • Use fold expressions to eliminate recursion" << std::endl;
```

```
916     std::cout << " • Each instantiation = separate code in binary" << std::endl;
917     std::cout << " • Can increase binary size significantly" << std::endl;
918     std::cout << " • Extract common code to non-template functions" << std::endl;
919
920     std::cout << "\n3. ERROR MESSAGES:" << std::endl;
921     std::cout << " • Template errors are cryptic and verbose" << std::endl;
922     std::cout << " • Use concepts (C++20) for better errors" << std::endl;
923     std::cout << " • Add static_assert with clear messages" << std::endl;
924
925     std::cout << "\n4. RECURSION LIMITS:" << std::endl;
926     std::cout << " • Compilers limit template depth (256-1024)" << std::endl;
927     std::cout << " • Don't increase limits - redesign instead!" << std::endl;
928     std::cout << " • Always provide proper base cases" << std::endl;
929
930     std::cout << "\n Compile-Time = Type Safety + Zero Cost!\n" << std::endl;
931     std::cout << " But use responsibly - compilation time matters!\n" << std::endl;
932
933     return 0;
934 }
```

## 76 Source Code: VirtualFunctionsInTemplates.cpp

File: src/VirtualFunctionsInTemplates.cpp

Repository: [View on GitHub](#)

```
1  /*
2   * VIRTUAL FUNCTIONS IN TEMPLATES: THE CODE BLOAT PROBLEM
3   *
4   * This file demonstrates WHY you should AVOID virtual functions in class
5   * templates
6   * and shows the severe code bloat that results from naive templatization.
7   *
8   * KEY PRINCIPLE: Virtual functions in templates are instantiated EVERY TIME,
9   * even if never called, causing exponential code bloat.
10  *
11  * References:
12  * - C++ Core Guidelines T.80: Do not naively templatize a class hierarchy
13  * - C++ Core Guidelines T.5: Combine generic and OO techniques judiciously
14  * - C++ Core Guidelines T.84: Use a non-template core implementation
15  */
16
17 #include <iostream>
18 #include <vector>
19 #include <string>
20 #include <memory>
21 #include <algorithm>
22
23 // =====
24 // SECTION 1: THE PROBLEM - WHY VIRTUAL FUNCTIONS CAUSE BLOAT
25 // =====
26
27 namespace the_problem {
28
29 /*
30  * THE CORE ISSUE:
31  *
32  * In a class template:
33  * - NON-VIRTUAL functions: Only instantiated if explicitly called (lazy)
34  * - VIRTUAL functions: ALWAYS instantiated when class is instantiated (eager)
35  *
36  * WHY?
37  * Virtual functions MUST be in the vtable, and the vtable is constructed
38  * at compile-time during template instantiation. Therefore, ALL virtual
39  * functions must be compiled, even if your code never calls them.
40  *
41  * This is the root cause of code bloat in template hierarchies.
42  */
43
44 template<typename T>
45 class DemoInstantiation {
```

```
45 public:
46     // VIRTUAL: Compiled when class is instantiated
47     virtual void always_compiled() {
48         std::cout << "Virtual: always compiled for " << typeid(T).name() << "\n";
49     }
50
51     virtual void never_called_but_still_compiled() {
52         std::cout << "Virtual: wasting space for " << typeid(T).name() << "\n";
53     }
54
55     virtual ~DemoInstantiation() = default;
56
57     // NON-VIRTUAL: Only compiled if actually called
58     void only_if_used() {
59         std::cout << "Non-virtual: only compiled if called for " << typeid(T).name()
60             () << "\n";
61     }
62
63     void also_only_if_used() {
64         std::cout << "Non-virtual: lazy instantiation for " << typeid(T).name()
65             () << "\n";
66     }
67 };
68
69 void demonstrate() {
70     std::cout << "\n" << std::string(70, '=') << "\n";
71     std::cout << "THE PROBLEM: Virtual Functions Cause Template Bloat\n";
72     std::cout << std::string(70, '=') << "\n\n";
73
74     std::cout << "Creating DemoInstantiation<int>:\n\n";
75     DemoInstantiation<int> demo;
76
77     std::cout << "What gets compiled?\n";
78     std::cout << "    always_compiled() - YES (virtual, in vtable)\n";
79     std::cout << "    never_called_but_still_compiled() - YES (virtual, in
80         vtable)\n";
81     std::cout << "    only_if_used() - NO (not called yet)\n";
82     std::cout << "    also_only_if_used() - NO (not called yet)\n\n";
83
84     demo.always_compiled();
85
86     std::cout << "\n THE BLOAT:\n";
87     std::cout << " • never_called_but_still_compiled() is compiled\n";
88     std::cout << " • It's in the binary taking up space\n";
89     std::cout << " • But we never call it!\n";
90     std::cout << " • This is WASTED CODE\n\n";
91
92     std::cout << "Now imagine:\n";
93     std::cout << " • 10 virtual functions\n";
94     std::cout << " • 10 template instantiations\n";
95     std::cout << " • = 100 compiled functions\n";
96     std::cout << " • If you only use 2-3 per type = 70-80 wasted functions!\n";
```

```
        n";
94 }
95
96 } // namespace the_problem
97
98 // =====
99 // SECTION 2: THE BAD EXAMPLE - C++ Core Guidelines T.80
100 // =====
101
102 namespace bad_example_t80 {
103
104 /*
105 * BAD DESIGN: Template with virtual functions
106 *
107 * This is the EXACT example from C++ Core Guidelines T.80 showing
108 * what NOT to do when designing template hierarchies.
109 */
110
111 template<typename T>
112 struct Container {           // an interface
113     virtual T* get(int i) = 0;
114     virtual T* first() = 0;
115     virtual T* next() = 0;
116     virtual void sort() = 0;
117     virtual ~Container() = default;
118 };
119
120 template<typename T>
121 class Vector : public Container<T> {
122 private:
123     std::vector<T> data_;
124     typename std::vector<T>::iterator current_;
125
126 public:
127     Vector() : current_(data_.begin()) {}
128
129     T* get(int i) override {
130         return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
131             nullptr;
132     }
133
134     T* first() override {
135         current_ = data_.begin();
136         return current_ != data_.end() ? &(*current_) : nullptr;
137     }
138
139     T* next() override {
140         if (current_ != data_.end()) ++current_;
141         return current_ != data_.end() ? &(*current_) : nullptr;
142     }
143 }
```

```
142
143     void sort() override {
144         std::sort(data_.begin(), data_.end());
145     }
146
147     void add(const T& item) { data_.push_back(item); }
148 };
149
150 void demonstrate() {
151     std::cout << "\n" << std::string(70, '=') << "\n";
152     std::cout << "BAD EXAMPLE: C++ Core Guidelines T.80\n";
153     std::cout << std::string(70, '=') << "\n\n";
154
155     std::cout << "Code:\n";
156     std::cout << "    template<typename T>\n";
157     std::cout << "    struct Container {\n";
158     std::cout << "        virtual T* get(int i);\n";
159     std::cout << "        virtual T* first();\n";
160     std::cout << "        virtual T* next();\n";
161     std::cout << "        virtual void sort();\n";
162     std::cout << "    };\n\n";
163
164     std::cout << "Creating two instantiations:\n";
165     std::cout << "    Vector<int> vi;\n";
166     std::cout << "    Vector<string> vs;\n\n";
167
168     Vector<int> vi;
169     Vector<std::string> vs;
170
171     std::cout << "    WHAT GETS COMPILED:\n\n";
172
173     std::cout << "Vector<int>:\n";
174     std::cout << "    •    get(int) - compiled\n";
175     std::cout << "    •    first() - compiled\n";
176     std::cout << "    •    next() - compiled\n";
177     std::cout << "    •    sort() - compiled\n";
178     std::cout << "    Even if you only use get()!\n\n";
179
180     std::cout << "Vector<string>:\n";
181     std::cout << "    •    get(int) - compiled AGAIN\n";
182     std::cout << "    •    first() - compiled AGAIN\n";
183     std::cout << "    •    next() - compiled AGAIN\n";
184     std::cout << "    •    sort() - compiled AGAIN\n";
185     std::cout << "    Complete duplication!\n\n";
186
187     std::cout << "    CODE BLOAT CALCULATION:\n";
188     std::cout << "    •    2 types × 4 virtual functions = 8 functions\n";
189     std::cout << "    •    10 types × 10 virtual functions = 100 functions\n";
190     std::cout << "    •    Average function: 50-100 bytes\n";
191     std::cout << "    •    100 functions × 75 bytes = 7.5 KB of bloat\n";
192     std::cout << "    •    In large projects: MEGABYTES of wasted code\n\n";
193
194     std::cout << "    WHY THIS IS BAD:\n";
195     std::cout << "    1. Binary size increases unnecessarily\n";
```

```
196     std::cout << "    2. Longer compilation times\n";
197     std::cout << "    3. More pressure on instruction cache\n";
198     std::cout << "    4. Harder to maintain\n";
199     std::cout << "    5. Embedded systems: wastes precious flash memory\n";
200 }
201
202 } // namespace bad_example_t80
203
204 // =====
205 // SECTION 3: WHY IT HAPPENS - THE VTABLE REQUIREMENT
206 // =====
207
208 namespace why_it_happens {
209
210     void demonstrate() {
211         std::cout << "\n" << std::string(70, '=') << "\n";
212         std::cout << "WHY VIRTUAL FUNCTIONS MUST BE INSTANTIATED\n";
213         std::cout << std::string(70, '=') << "\n\n";
214
215         std::cout << "    THE MECHANISM:\n\n";
216
217         std::cout << "1. VTABLE CONSTRUCTION:\n";
218         std::cout << "    • Each class with virtual functions has a vtable\n";
219         std::cout << "    • vtable is built at COMPILE-TIME\n";
220         std::cout << "    • vtable contains pointers to ALL virtual functions\n";
221         std::cout << "    • Therefore, all virtual functions MUST exist\n\n";
222
223         std::cout << "2. TEMPLATE INSTANTIATION:\n";
224         std::cout << "    • Template<int> creates a new class type\n";
225         std::cout << "    • This class needs its own vtable\n";
226         std::cout << "    • Compiler must compile ALL virtual functions\n";
227         std::cout << "    • Even those never called\n\n";
228
229         std::cout << "3. COMPARISON:\n\n";
230
231         std::cout << "    Non-virtual function:\n";
232         std::cout << "        Compiler sees call site\n";
233         std::cout << "        \"Oh, this function is used, compile it\"\n";
234         std::cout << "        No call = no compilation (lazy)\n\n";
235
236         std::cout << "    Virtual function:\n";
237         std::cout << "        Compiler instantiates class\n";
238         std::cout << "        \"Must build vtable for this class\"\n";
239         std::cout << "        \"Need ALL virtual function addresses\"\n";
240         std::cout << "        ALL compiled regardless of use (eager)\n\n";
241
242         std::cout << "    STANDARD LIBRARY MISTAKE:\n";
243         std::cout << "    • std::locale facets (std::ctype<T>) made this mistake\n";
244         ;
245         std::cout << "    • ~15 virtual functions per facet\n";
```

```
245     std::cout << " • Most programs use 2-3 of them\n";
246     std::cout << " • But all 15 are instantiated\n";
247     std::cout << " • Acknowledged in C++ Core Guidelines as historical error
248     \n";
249 }
250 } // namespace why_it_happens
251 //
252 //=====
253 // SOLUTION 1: NON-TEMPLATE BASE CLASS (TYPE ERASURE)
254 //=====
255
256 namespace solution_type_erasure {
257
258 /*
259  * GOOD: Move virtual interface to non-template base
260  *
261  * Virtual functions compiled ONCE, not per template instantiation.
262  * This is the T.84 guideline: "Use a non-template core implementation"
263  */
264
265 // Non-template base - compiled ONCE
266 class ContainerBase {
267 public:
268     virtual ~ContainerBase() = default;
269     virtual void* get_impl(int i) = 0;
270     virtual void* first_impl() = 0;
271     virtual void* next_impl() = 0;
272     virtual void sort_impl() = 0;
273 };
274
275 // Template provides type safety
276 template<typename T>
277 class Vector : public ContainerBase {
278 private:
279     std::vector<T> data_;
280     typename std::vector<T>::iterator current_;
281
282 public:
283     Vector() : current_(data_.begin()) {}
284
285     void* get_impl(int i) override {
286         return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
287             nullptr;
288     }
289
290     void* first_impl() override {
291         current_ = data_.begin();
292         return current_ != data_.end() ? static_cast<void*>(&(*current_)) :
293             nullptr;
294 }
```

```

292     }
293
294     void* next_implementation() override {
295         if (current_ != data_.end()) ++current_;
296         return current_ != data_.end() ? static_cast<void*>(&(*current_)) :
297             nullptr;
298     }
299
300     void sort_implementation() override {
301         std::sort(data_.begin(), data_.end());
302     }
303
304     // Type-safe wrappers
305     T* get(int i) { return static_cast<T*>(get_implementation(i)); }
306     T* first() { return static_cast<T*>(first_implementation()); }
307     T* next() { return static_cast<T*>(next_implementation()); }
308     void sort() { sort_implementation(); }
309     void add(const T& item) { data_.push_back(item); }
310 };
311
312 void demonstrate() {
313     std::cout << "\n" << std::string(70, '=') << "\n";
314     std::cout << "SOLUTION 1: Non-Template Base (Type Erasure)\n";
315     std::cout << std::string(70, '=') << "\n\n";
316
317     Vector<int> vi;
318     Vector<std::string> vs;
319
320     std::cout << " FIXED DESIGN:\n\n";
321
322     std::cout << "ContainerBase (non-template):\n";
323     std::cout << " • Virtual functions compiled ONCE\n";
324     std::cout << " • get_implementation(), first_implementation(), next_implementation(), sort_implementation()\n";
325     std::cout << " • Shared across ALL instantiations\n\n";
326
327     std::cout << "Vector<T> (template):\n";
328     std::cout << " • Overrides virtual functions (minimal code)\n";
329     std::cout << " • Type-specific logic only\n";
330     std::cout << " • Type-safe wrapper functions\n\n";
331
332     std::cout << " BLOAT REDUCTION:\n";
333     std::cout << " • 10 types × 4 virtuals = 4 base + 10 overrides\n";
334     std::cout << " • vs Bad design: 40 separate functions\n";
335     std::cout << " • Reduction: 50-80% less code\n\n";
336
337     std::cout << " TRADE-OFFS:\n";
338     std::cout << " • Massive code size reduction\n";
339     std::cout << " • Runtime polymorphism preserved\n";
340     std::cout << " • Slight runtime overhead (void* casts)\n";
341     std::cout << " • Loss of type information in base\n";
342 }
343 } // namespace solution_type_erasure
344

```

```
345 // =====
346 // SOLUTION 2: CRTP - NO VIRTUAL FUNCTIONS
347 // =====
348
349 namespace solution_crtp {
350
351 /*
352 * BETTER: CRTP for compile-time polymorphism
353 *
354 * No vtable = No forced instantiation = No bloat
355 * Zero runtime overhead
356 */
357
358 template<typename Derived>
359 class ContainerCRTP {
360 public:
361     auto* get(int i) { return derived()->get_impl(i); }
362     auto* first() { return derived()->first_impl(); }
363     auto* next() { return derived()->next_impl(); }
364     void sort() { derived()->sort_impl(); }
365
366 private:
367     Derived* derived() { return static_cast<Derived*>(this); }
368 };
369
370 template<typename T>
371 class Vector : public ContainerCRTP<Vector<T>> {
372 private:
373     std::vector<T> data_;
374     typename std::vector<T>::iterator current_;
375
376 public:
377     Vector() : current_(data_.begin()) {}
378
379     T* get_impl(int i) {
380         return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
381             nullptr;
382     }
383
384     T* first_impl() {
385         current_ = data_.begin();
386         return current_ != data_.end() ? &(*current_) : nullptr;
387     }
388
389     T* next_impl() {
390         if (current_ != data_.end()) ++current_;
391         return current_ != data_.end() ? &(*current_) : nullptr;
392     }
393
394     void sort_impl() {
```

```
394     std::sort(data_.begin(), data_.end());
395 }
396
397 void add(const T& item) { data_.push_back(item); }
398 };
399
400 void demonstrate() {
401     std::cout << "\n" << std::string(70, '=') << "\n";
402     std::cout << "SOLUTION 2: CRTP (Static Polymorphism)\n";
403     std::cout << std::string(70, '=') << "\n\n";
404
405     Vector<int> vi;
406     vi.add(5); vi.add(3); vi.add(9);
407
408     std::cout << " NO VIRTUAL FUNCTIONS:\n\n";
409
410     std::cout << "Benefits:\n";
411     std::cout << " • No vtable\n";
412     std::cout << " • No vptr (8 bytes saved per object)\n";
413     std::cout << " • Functions ONLY compiled if called\n";
414     std::cout << " • Can be fully inlined\n";
415     std::cout << " • Zero runtime overhead\n\n";
416
417     vi.sort();
418
419     std::cout << "What got compiled:\n";
420     std::cout << " sort_impl<int> - used\n";
421     std::cout << " add - used\n";
422     std::cout << " get_impl - NOT compiled (not used)\n";
423     std::cout << " first_impl - NOT compiled (not used)\n";
424     std::cout << " next_impl - NOT compiled (not used)\n\n";
425
426     std::cout << " BLOAT ELIMINATION:\n";
427     std::cout << " • Only used functions compiled\n";
428     std::cout << " • Lazy instantiation = minimal code\n";
429     std::cout << " • Perfect for performance-critical code\n\n";
430
431     std::cout << " TRADE-OFFS:\n";
432     std::cout << " Zero code bloat\n";
433     std::cout << " Zero runtime overhead\n";
434     std::cout << " No runtime polymorphism\n";
435     std::cout << " Cannot use heterogeneous containers\n";
436 }
437
438 } // namespace solution_crtp
439
440 // =====
441 // SOLUTION 3: C++20 CONCEPTS
442 // =====
```

```
444 namespace solution_concepts {
445
446 #if __cplusplus >= 202002L
447
448 template<typename T>
449 concept Container = requires(T c, int i) {
450     { c.get(i) };
451     { c.first() };
452     { c.next() };
453     { c.sort() } -> std::same_as<void>;
454 };
455
456 template<typename T>
457 class Vector {
458 private:
459     std::vector<T> data_;
460     typename std::vector<T>::iterator current_;
461
462 public:
463     Vector() : current_(data_.begin()) {}
464
465     T* get(int i) {
466         return (i >= 0 && i < static_cast<int>(data_.size())) ? &data_[i] :
467             nullptr;
468     }
469
470     T* first() {
471         current_ = data_.begin();
472         return current_ != data_.end() ? &(*current_) : nullptr;
473     }
474
475     T* next() {
476         if (current_ != data_.end()) ++current_;
477         return current_ != data_.end() ? &(*current_) : nullptr;
478     }
479
480     void sort() { std::sort(data_.begin(), data_.end()); }
481     void add(const T& item) { data_.push_back(item); }
482 };
483
484 template<Container C>
485 void process(C& cont) {
486     cont.sort();
487 }
488
489 #endif
490
491 void demonstrate() {
492     std::cout << "\n" << std::string(70, '=') << "\n";
493     std::cout << "SOLUTION 3: C++20 Concepts\n";
494     std::cout << std::string(70, '=') << "\n\n";
495
496 #if __cplusplus >= 202002L
497     Vector<int> vi;
```

```
497     vi.add(5);
498     process(vi);
499
500     std::cout << "  NO INHERITANCE, NO VIRTUAL:\n\n";
501
502     std::cout << "Concepts provide:\n";
503     std::cout << " •  Compile-time interface checking\n";
504     std::cout << " •  Duck typing with type safety\n";
505     std::cout << " •  No base class needed\n";
506     std::cout << " •  Better error messages\n";
507     std::cout << " •  Zero runtime overhead\n\n";
508
509     std::cout << "  BLOAT ELIMINATION:\n";
510     std::cout << " •  No virtual functions at all\n";
511     std::cout << " •  Only used functions instantiated\n";
512     std::cout << " •  Cleanest solution\n";
513 #else
514     std::cout << "  C++20 concepts not available.\n";
515     std::cout << "  Compile with -std=c++20 to see this solution.\n";
516 #endif
517 }
518
519 } // namespace solution_concepts
520
521 // =====
522 // GUIDELINES AND RECOMMENDATIONS
523 // =====
524
525 namespace guidelines {
526
527 void demonstrate() {
528     std::cout << "\n" << std::string(70, '=') << "\n";
529     std::cout << "GUIDELINES: AVOID VIRTUAL FUNCTIONS IN TEMPLATES\n";
530     std::cout << std::string(70, '=') << "\n\n";
531
532     std::cout << "  AVOID THIS:\n\n";
533     std::cout << "    template<typename T>\n";
534     std::cout << "    class Container {\n";
535     std::cout << "        virtual void operation(); // BAD!\n";
536     std::cout << "    };\n\n";
537
538     std::cout << "  PROBLEMS:\n";
539     std::cout << "    1. CODE BLOAT: All virtuals instantiated every time\n";
540     std::cout << "    2. COMPILATION: Slower, more memory\n";
541     std::cout << "    3. BINARY SIZE: Can add megabytes in large projects\n";
542     std::cout << "    4. PERFORMANCE: More pressure on I-cache\n";
543     std::cout << "    5. MAINTENANCE: Harder to track what's compiled\n\n";
544
545     std::cout << "  WHEN IT'S A PROBLEM:\n";
546     std::cout << " •  Many virtual functions (5+)\n";
```



```
593     std::cout << "          WHY TO AVOID & BETTER ALTERNATIVES
594                           \n";
595     std::cout << "                           \n";
596     the_problem::demonstrate();
597     bad_example_t80::demonstrate();
598     why_it_happens::demonstrate();
599     solution_type_erasure::demonstrate();
600     solution_crtpp::demonstrate();
601     solution_concepts::demonstrate();
602     guidelines::demonstrate();
603
604     std::cout << "\n" << std::string(70, '=') << "\n";
605     std::cout << "KEY TAKEAWAYS\n";
606     std::cout << std::string(70, '=') << "\n\n";
607
608     std::cout << "  THE PROBLEM:\n";
609     std::cout << "  •  Virtual functions in templates = code bloat\n";
610     std::cout << "  •  ALL virtuals instantiated for EVERY type\n";
611     std::cout << "  •  Even if never called\n";
612     std::cout << "  •  Can add megabytes to binary size\n\n";
613
614     std::cout << "  WHY IT HAPPENS:\n";
615     std::cout << "  •  vtable must be built at compile-time\n";
616     std::cout << "  •  vtable needs ALL virtual function addresses\n";
617     std::cout << "  •  Compiler must compile them all\n";
618     std::cout << "  •  Non-virtual functions use lazy instantiation\n\n";
619
620     std::cout << "  SOLUTIONS:\n";
621     std::cout << "    1. C++20 Concepts - no inheritance needed\n";
622     std::cout << "    2. CRTP - compile-time polymorphism\n";
623     std::cout << "    3. Non-template base - runtime polymorphism\n\n";
624
625     std::cout << "  RECOMMENDATION:\n";
626     std::cout << "    AVOID virtual functions in class templates\n";
627     std::cout << "    unless you have a compelling reason and few
628     instantiations.\n\n";
629
630     std::cout << "                           \n";
631     std::cout << "           Follow C++ Core Guidelines T.80 and T.84
632                           \n";
633     std::cout << "                           \n\n";
634
635     return 0;
636 }
```

## 77 Appendix: Comprehensive Index

## 78 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.

---

### 78.1 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
- 

### 78.2 Interview Preparation

#### 78.2.1 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/constexpr
  - **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](#)

---

## 78.3 C++ Standards Features

### 78.3.1 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`

### 78.3.2 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

### 78.3.3 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

### 78.3.4 C++20

- **File:** [Cpp20Examples.cpp](#)

- **Topics:** Concepts, ranges, coroutines, modules, three-way comparison ( $\langle=\rangle$ ), 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

### 78.3.5 C++23

- **File:** [Cpp23Examples.cpp](#)
- **Topics:** `std::expected`, deducing `this`, `if consteval`, multidimensional subscript operator, `std::print`, `std::stacktrace`

[Back to Top](#)

---

## 78.4 Design Patterns & Idioms

### 78.4.1 CRTP (Curiously Recurring Template Pattern)

- **File:** [CRTPvsVirtualFunctions.cpp](#)
- **Topics:** CRTP vs virtual functions, static polymorphism, performance comparison, vtable overhead

### 78.4.2 Pimpl (Pointer to Implementation)

- **File:** [PimplIdiom.cpp](#)
- **Topics:** Pimpl idiom, ABI stability, compilation firewall, real-time concerns, cache locality, FastPimpl alternative

### 78.4.3 NVI (Non-Virtual Interface)

- **File:** [NVIIdiomTemplateMethod.cpp](#)
- **Topics:** Non-virtual interface idiom, template method pattern, Herb Sutter's virtuality guidelines

### 78.4.4 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

### 78.4.5 Dependency Injection

- **File:** [DependencyInjection.cpp](#)
- **Topics:** Constructor injection, setter injection, interface injection, composition over inheritance

#### 78.4.6 SOLID Principles

- **File:** [SOLIDPrinciples.cpp](#)
- **Topics:** Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion

#### 78.4.7 Factory Pattern

- **File:** [CreatingCApiFromCpp.cpp](#)
- **Topics:** Factory functions for C API

[Back to Top](#)

---

### 78.5 Object-Oriented Programming

#### 78.5.1 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

#### 78.5.2 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

#### 78.5.3 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](#)

---

### 78.6 Memory Management

#### 78.6.1 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

### 78.6.2 Universal Resource Management with Custom Deleters

- **File:** [UniversalResourceManager.cpp](#)
- **Topics:** Custom deleters for RAII, universal resource management beyond memory
- **Description:** Comprehensive guide to using smart pointers with custom deleters for managing non-memory resources (files, sockets, databases, GPU resources, shared memory, resource pools). Demonstrates 8 real-world examples, 10 common pitfalls with solutions, deleter patterns (lambdas, function objects, stateful deleters), decision tree for choosing `unique_ptr` vs `shared_ptr`, and simplified RAII patterns
- **Key Examples:**
  - `FILE*` management with `fclose()` deleter
  - Database connection lifecycle with RAII
  - Socket/network resource cleanup
  - GPU resource management (buffers, textures)
  - Shared memory mapping/unmapping
  - Resource pool integration with custom deleters
  - Debug allocators with statistics tracking
  - Composite deleters for multi-step cleanup
- **Pitfalls Covered:** Type bloat, size overhead, `nullptr` checks, exception safety, dangling references, double delete, stack vs heap, performance implications, allocation mismatch, thread safety
- **Design Patterns:** Lambdas, stateful lambdas, function objects, function pointers, templated factory functions, RAII wrapper templates

### 78.6.3 Move Semantics

- **File:** [MoveSemantics.cpp](#)
- **Topics:** Rvalue references, move constructors, move assignment, perfect forwarding
- **File:** [MoveSemanticsPerfectForwarding.cpp](#)
- **Topics:** Move semantics and perfect forwarding patterns
- **File:** [PerfectForwardingAndRequires.cpp](#)
- **Topics:** Perfect forwarding with `std::forward`, requires clauses

### 78.6.4 Memory Pools & Allocators

- **File:** [RealTimeProgramming.cpp](#)
- **Topics:** Memory pools, custom allocators, pre-allocation strategies
- **File:** [STLContainersNoHeap.cpp](#)
- **Topics:** Custom allocators, stack allocators, pool allocators

### 78.6.5 Resource Management

- **File:** [ResourceLeaks.cpp](#)
- **Topics:** Common resource leak patterns and prevention

[Back to Top](#)

---

## 78.7 Concurrency & Parallelism

### 78.7.1 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

### 78.7.2 Parallel Algorithms

- **File:** [Cpp17Concurrency.cpp](#)
- **Topics:** Parallel STL, execution policies (`std::execution::par`)

### 78.7.3 Future/Promise/Async

- **File:** [FuturePromiseAsync.cpp](#)
- **Topics:** `std::future`, `std::promise`, `std::async`, `std::packaged_task`, ASIO relationship

### 78.7.4 Stop Tokens

- **File:** [StopTokenExample.cpp](#)
- **Topics:** `std::stop_token`, `std::stop_source`, cooperative cancellation (C++20)

### 78.7.5 Microservices

- **File:** [MultiThreadedMicroservices.cpp](#)
- **Topics:** Multi-threaded microservice architecture

[Back to Top](#)

---

## 78.8 Real-Time & Embedded Systems

### 78.8.1 Real-Time Programming

- **File:** [RealTimeProgramming.cpp](#)
- **Topics:** Deterministic behavior, WCET, Big O notation, memory pre-allocation, `std::bitset`, `std::list::splice()`, real-time patterns

### 78.8.2 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`)

### 78.8.3 ARM & Architecture

- **File:** [ARMInstructionSets.cpp](#)
- **Topics:** ARM instruction sets, NEON, SVE, architecture-specific optimization

### 78.8.4 ROM Placement

- **File:** [ROMAbility.cpp](#)
- **Topics:** `constexpr`, `consteval`, `constinit`, ROM placement strategies

[Back to Top](#)

---

## 78.9 Safety-Critical & Standards

### 78.9.1 MISRA C++

- **File:** [MISRACppDemo.cpp](#)
- **Topics:** MISRA C++ rules, safety-critical coding guidelines

### 78.9.2 ISO 26262 (Automotive)

- **File:** [FunctionalSafetyISO26262.cpp](#)
- **Topics:** ISO 26262 functional safety, ASIL levels, automotive safety

### 78.9.3 STL for Safety-Critical

- **File:** [SafetyCriticalSTLContainers.cpp](#)
- **Topics:** Which STL containers to avoid/use in safety-critical systems, MISRA compliance, AUTOSAR, heap allocation issues

### 78.9.4 AUTOSAR

- **File:** [SafetyCriticalSTLContainers.cpp](#)
- **Topics:** AUTOSAR C++14 guidelines, container usage

[Back to Top](#)

---

## 78.10 STL Containers & Algorithms

### 78.10.1 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 $\_*$ )

### 78.10.2 `std::bitset`

- **File:** [RealTimeProgramming.cpp](#)
- **Topics:** `std::bitset` vs `std::vector<bool>`, real-time usage

### 78.10.3 `std::list::splice()`

- **File:** [RealTimeProgramming.cpp](#)
- **Topics:** O(1) element movement, splice operations, real-time patterns

### 78.10.4 Algorithms

- **File:** [BinarySearch.cpp](#)
- **Topics:** Binary search algorithms, STL algorithm usage

[Back to Top](#)

---

## 78.11 Template Metaprogramming

### 78.11.1 Templates & Concepts

- **File:** [ConceptsExamples.cpp](#)
- **Topics:** C++20 concepts, type constraints, `requires` clauses
- **File:** [PerfectForwardingAndRequires.cpp](#)
- **Topics:** Perfect forwarding with concepts

### 78.11.2 CRTP (Static Polymorphism)

- **File:** [CRTPvsVirtualFunctions.cpp](#)
- **Topics:** Curiously Recurring Template Pattern, compile-time polymorphism

### 78.11.3 Variadic Templates

- **File:** [VariadicTemplateRecursion.cpp](#)
- **Topics:** Variadic templates, template recursion, parameter packs

### 78.11.4 Template Interface

- **File:** [TemplatedCameraInterface.cpp](#)
- **Topics:** Generic templated interfaces

[Back to Top](#)

---

## 78.12 Error Handling

### 78.12.1 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

### 78.12.2 noexcept

- **File:** [NoexceptBestPractices.cpp](#)
- **Topics:** noexcept specifier, move operations, best practices

### 78.12.3 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](#)

---

## 78.13 Interoperability

### 78.13.1 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

### 78.13.2 Python Binding

- **File:** [Pybind11Example.cpp](#)
- **Topics:** pybind11, exposing C++ to Python, automatic binding generation

### 78.13.3 Protobuf

- **File:** [ProtobufExample.cpp](#)
- **Topics:** Protocol Buffers, serialization, cross-language data exchange

### 78.13.4 JSON

- **File:** [NlohmannJsonExample.cpp](#)
- **Topics:** nlohmann/json library, JSON parsing and serialization

### 78.13.5 REST API

- **File:** [RestApiExample.cpp](#)
- **Topics:** REST API client, HTTP requests, CURL integration

[Back to Top](#)

---

## 78.14 Performance & Optimization

### 78.14.1 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

### 78.14.2 Cache Locality

- **File:** [PimplIdiom.cpp](#)
- **Topics:** Cache-friendly design, avoiding pointer indirection
- **File:** [RealTimeProgramming.cpp](#)
- **Topics:** Memory layout for real-time systems

### 78.14.3 Compile-Time Computation

- **File:** [ROMability.cpp](#)
- **Topics:** `constexpr`, `consteval`, compile-time evaluation

[Back to Top](#)

---

## 78.15 Common Problems & Solutions

### 78.15.1 Fragile Base Class

- **File:** [FragileBaseClass.cpp](#)
- **Topics:** Fragile base class problem, ABI breaks, composition over inheritance

### 78.15.2 Diamond Problem

- **File:** [DiamondProblem.cpp](#)
- **Topics:** Multiple inheritance diamond problem, virtual inheritance solution

### 78.15.3 Object Slicing

- **File:** [ObjectSlicingCpp20.cpp](#)
- **Topics:** Object slicing, prevention strategies
- **File:** [ObjectSlicingSmartPtr.cpp](#)
- **Topics:** Smart pointers to prevent slicing

### 78.15.4 Resource Leaks

- **File:** [ResourceLeaks.cpp](#)
- **Topics:** Memory leaks, file handle leaks, RAII solutions

### 78.15.5 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](#)

---

## 78.16 Keywords & Language Features

### 78.16.1 Keywords

#### 78.16.1.1 `auto`

- [Cpp11Examples.cpp](#) - C++11 `auto` keyword
- [Cpp14Examples.cpp](#) - C++14 `auto` return type
- [GenericLambdas.cpp](#) - Generic lambdas with `auto`

#### 78.16.1.2 `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

#### 78.16.1.3 `virtual`

- [RuntimePolymorphism.cpp](#) - Virtual functions basics
- [CRTPvsVirtualFunctions.cpp](#) - Virtual vs CRTP
- [VirtualFunctionsInTemplates.cpp](#) - Virtual in templates problem
- [NVIIdiomTemplateMethod.cpp](#) - Virtual in NVI pattern

#### 78.16.1.4 `noexcept`

- [NoexceptBestPractices.cpp](#) - `noexcept` best practices
- [MoveSemantics.cpp](#) - `noexcept` move operations

#### 78.16.1.5 `requires` (Concepts)

- [ConceptsExamples.cpp](#) - Requires clauses
- [PerfectForwardingAndRequires.cpp](#) - Requires with forwarding

#### 78.16.1.6 `final`

- [FragileBaseClass.cpp](#) - Final keyword usage

#### 78.16.1.7 `override`

- [RuntimePolymorphism.cpp](#) - Override specifier

#### 78.16.1.8 `nullptr`

- [Cpp11Examples.cpp](#) - `nullptr` vs `NULL`

### 78.16.1.9 `decltype`

- [Cpp11Examples.cpp](#) - decltype usage
- [PerfectForwardingAndRequires.cpp](#) - decltype with forwarding

[Back to Top](#)

---

## 78.17 Event-Driven Programming

### 78.17.1 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](#)

---

## 78.18 Lambda Expressions

### 78.18.1 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](#)

---

## 78.19 Data Structures & Algorithms

### 78.19.1 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

### 78.19.2 Search & Sort

- **File:** [BinarySearch.cpp](#)
- **Topics:** Binary search implementation
- **File:** [SearchAnagramsDictionary.cpp](#)
- **Topics:** Anagram searching in dictionary

### 78.19.3 Array Algorithms

- **File:** [FindMaxNoOfConsecutiveOnesFromIntArray.cpp](#)
- **Topics:** Maximum consecutive elements

[Back to Top](#)

---

## 78.20 Tuples & Structured Bindings

### 78.20.1 Tuples

- **File:** [TuplesAndStructuredBindings.cpp](#)
- **Topics:** `std::tuple`, tuple operations, structured bindings

### 78.20.2 Structured Bindings

- **File:** [StructuredBindings.cpp](#)
- **Topics:** Structured bindings syntax and patterns

[Back to Top](#)

---

## 78.21 Configuration & Parsing

### 78.21.1 Configuration

- **File:** [ConfigLoaderAndChecker.cpp](#)
- **Topics:** Configuration file loading and validation

### 78.21.2 System Interaction

- **File:** [SystemInteractionAndParsing.cpp](#)
- **Topics:** System calls, command execution, parsing

[Back to Top](#)

---

## 78.22 Sensor Fusion & Scientific Computing

### 78.22.1 Eigen Library

- **File:** [EigenSensorFusion.cpp](#)
- **Topics:** Eigen library, Kalman filter, sensor fusion, matrix operations

[Back to Top](#)

---

## 78.23 Quick Reference Tables

### 78.23.1 By Language Version

C++11	C++14	C++17	C++20	C++23
<a href="#">Cpp11Examples.cpp</a>	<a href="#">Cpp14Examples.cpp</a>	<a href="#">Cpp17Examples.cpp</a>	<a href="#">Cpp20Examples.cpp</a>	<a href="#">Cpp23Examples.cpp</a>
<a href="#">MoveSemantics.cpp</a>	<a href="#">GenericLambdas.cpp</a>	<a href="#">StructuredBindings.cpp</a>	<a href="#">ConceptsExamples.cpp</a>	
<a href="#">LambdaCaptures.cpp</a>		<a href="#">OptionalExamples.cpp</a>	<a href="#">RangesExamples.cpp</a>	
-	-	<a href="#">Cpp17Concurrency.cpp</a>	<a href="#">StopTokenExample.cpp</a>	

### 78.23.2 By Use Case

Use Case	Recommended Files
<b>Learning Modern C++</b>	Start with <a href="#">Cpp11Examples.cpp</a> , then progress through C++14, 17, 20, 23 examples
<b>Real-Time Systems</b>	<a href="#">RealTimeProgramming.cpp</a> , <a href="#">SafetyCriticalSTLContainers.cpp</a> , <a href="#">PimplIdiom.cpp</a>
<b>Embedded Systems</b>	<a href="#">EmbeddedSystemsProgramming.cpp</a> , <a href="#">EmbeddedSystemsAvoid.cpp</a> , <a href="#">STLContainersNoHeap.cpp</a>
<b>Safety-Critical</b>	<a href="#">MISRACppDemo.cpp</a> , <a href="#">FunctionalSafetyISO26262.cpp</a> , <a href="#">SafetyCriticalSTLContainers.cpp</a>
<b>Performance</b>	<a href="#">CRTPvsVirtualFunctions.cpp</a> , <a href="#">PimplIdiom.cpp</a> , <a href="#">RealTimeProgramming.cpp</a>
<b>OOP Design</b>	<a href="#">InheritanceTypes.cpp</a> , <a href="#">DependencyInjection.cpp</a> , <a href="#">SOLIDPrinciples.cpp</a>
<b>Memory Management</b>	<a href="#">RuleOf3_5_0.cpp</a> , <a href="#">MoveSemantics.cpp</a> , <a href="#">ResourceLeaks.cpp</a>
<b>Concurrency</b>	<a href="#">AsioAndModernCppConcurrency.cpp</a> , <a href="#">FuturePromiseAsync.cpp</a> , <a href="#">ThreadPoolExamples.cpp</a>

### 78.23.3 By Problem You're Trying to Solve

Problem	Solution Files
“My code has memory leaks”	<a href="#">ResourceLeaks.cpp</a> , <a href="#">RuleOf3_5_0.cpp</a>
“Objects are getting sliced”	<a href="#">ObjectSlicingCpp20.cpp</a> , <a href="#">ObjectSlicingSmartPtr.cpp</a>
“Base class changes break derived classes”	<a href="#">FragileBaseClass.cpp</a>
“Multiple inheritance is causing problems”	<a href="#">DiamondProblem.cpp</a>
“Virtual functions are too slow”	<a href="#">CRTPvsVirtualFunctions.cpp</a> , <a href="#">VirtualFunctionsInTemplates.cpp</a>
“Pimpl hurts performance”	<a href="#">PimplIdiom.cpp</a>

---

Problem	Solution Files
“Need containers without heap”	<a href="#">STLContainersNoHeap.cpp</a> , <a href="#">SafetyCriticalSTLContainers.cpp</a>
“Timing is non-deterministic”	<a href="#">RealTimeProgramming.cpp</a>
“Need C interop”	<a href="#">CppWrappingCLibrary.cpp</a> , <a href="#">CreatingCApiFromCpp.cpp</a>

---

[Back to Top](#)

---

## 78.24 Additional Resources

### 78.24.1 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

### 78.24.2 Build Scripts

- [scripts/](#) - Build and execution scripts
- [build\\_modules.ps1](#) - PowerShell build script

[Back to Top](#)

---

## 78.25 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
- 

## 78.26 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:**

```
1 ModernCppExamples/
2   src/           # Source files (.cpp, .cppm)
3   MarkDownDocuments/ # Markdown documentation
4   scripts/        # Build scripts
5   proto/          # Protocol buffer definitions
6   CMakeLists.txt  # CMake build configuration
7   INDEX.md        # This file
```

[Back to Top](#)

---

*Last updated: January 3, 2026*