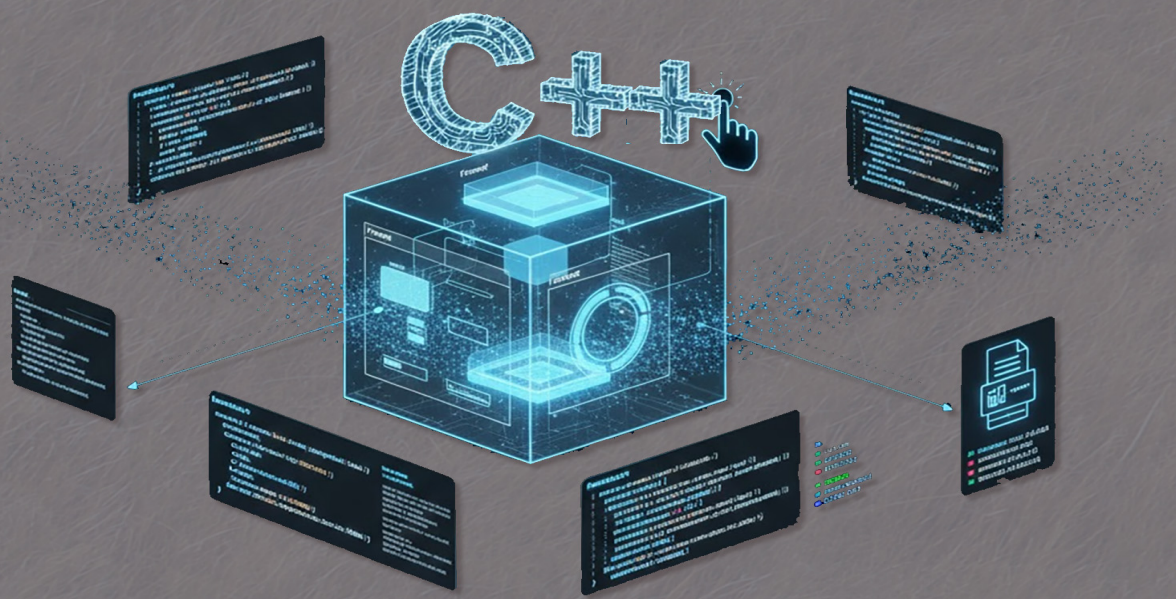
https://simplifycpp.org

Building a Simple Logger in Modern C++



Prepared by Ayman alheraki

Building a Simple Logger in Modern C++

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Author's Introduction

This booklet was written for C++ developers who want to move beyond the simplistic notion of "printing to the console" and begin designing **infrastructure-level components** with the same care, discipline, and rigor applied to core system architecture.
Logging is often perceived as a secondary concern. In many projects, it starts as a few `std::cout` statements, scattered across the codebase, added in moments of urgency to inspect values or trace execution paths. At small scale, this approach appears harmless. At scale, it becomes one of the most damaging architectural liabilities a system can carry.
Logging is deceptively simple. Almost every program writes text somewhere. Yet in professional, long-lived systems, logging rapidly evolves into one of the most critical subsystems. It directly affects:

- Debugging efficiency and incident response
- Runtime performance and latency characteristics
- Observability and operational insight
- Post-mortem diagnostics and failure analysis
- Overall system stability and predictability

A poorly designed logger can introduce subtle data races, hidden performance bottlenecks, excessive I/O contention, and even deadlocks in the most critical execution paths. Conversely,

a well-designed logging system can serve as a powerful lens through which complex systems become understandable, diagnosable, and maintainable over time.

Many existing logging tutorials focus almost exclusively on *usage*. They demonstrate how to call a logging macro, how to configure a library, or how to produce colorful output. While such material is useful, it often bypasses the deeper and more important question: *How should a logging system be designed in the first place?*

This work deliberately shifts the focus from consumption to construction. It emphasizes **engineering discipline** over convenience:

- Clear ownership and well-defined lifetimes
- Minimal and predictable runtime overhead
- Explicit separation of concerns
- Extensibility without fragility
- Correctness under concurrency

The logger developed in this booklet is intentionally simple, but not simplistic. Each design decision is motivated by real-world constraints encountered in production C++ systems: multi-threaded execution, error paths, performance-sensitive code, and the need for testability.

The goal is not to compete with established logging libraries, nor to reinvent them feature-for-feature. Instead, the objective is to **understand the principles** that govern their internal structure. By building a logger from first principles using only modern C++ and the standard library, the reader gains insight into why mature logging frameworks look the way they do, and how to evaluate them critically.

Ultimately, this booklet treats logging not as a convenience feature, but as a foundational architectural component. One that reflects the maturity of both the system and the engineer who designed it.

`Ayman Alheraki`

Preface

Modern C++ gives engineers an extraordinary degree of control. It allows precise management of memory, deterministic object lifetimes, explicit ownership models, fine-grained concurrency, and performance characteristics that can be tuned down to the instruction level. Few languages offer this level of power.

That power, however, is inseparable from responsibility.

In C++, design mistakes rarely remain local. A small architectural flaw can silently propagate across an entire codebase, surfacing months or years later as performance regressions, data races, undefined behavior, or systems that become too fragile to evolve safely. Infrastructure components are especially sensitive to such mistakes, and logging is among the most exposed of them all.

A logger sits at the intersection of multiple, often competing concerns:

- **I/O** — interacting with files, consoles, or external systems
- **Formatting** — transforming structured information into human-readable or machine-parsable text
- **Threading** — operating safely in multi-threaded environments without introducing contention or races
- **Error handling** — functioning correctly when the system is already in a failure state

- **Build configuration** — adapting behavior across debug, release, and production builds

Because of this central position, a logger is often invoked from the most delicate parts of a system: error paths, exception handlers, recovery logic, and diagnostic code that runs when invariants have already been violated. In such contexts, any weakness in the logging system is amplified.
A poorly designed logger can:

- Introduce subtle and difficult-to-reproduce data races
- Destroy performance by blocking critical execution paths
- Mask or suppress failures instead of revealing them
- Complicate testing by coupling unrelated components

These failures are rarely obvious at first. They tend to emerge under load, in production, and at the worst possible time.
This booklet approaches logging as an engineering problem rather than a utility problem. It builds a logger incrementally, starting from first principles and progressively introducing structure, constraints, and discipline. Every step is motivated by a concrete design concern rather than by convenience or fashion.
Only the C++ standard library is used. This constraint is intentional. It ensures that every mechanism involved is transparent, inspectable, and portable, and that the reader gains a deep understanding of the tools Modern C++ already provides.
Throughout the booklet, modern C++ idioms are respected and reinforced: RAII for resource management, strong typing for correctness, explicit ownership for clarity, and minimal abstractions for performance. The result is not merely a working logger, but a clear illustration of how infrastructure components should be approached in professional Modern C++ systems.

Chapter 1

Why Logging Matters

1.1 Logging as Infrastructure

Logging is not a feature. It is infrastructure.

This distinction is critical. Features are optional; they can be enabled, disabled, replaced, or even removed with limited impact. Infrastructure, on the other hand, forms the backbone of a system. When infrastructure is weak, everything built on top of it becomes fragile.

A system without reliable logging is blind. When failures occur, developers have no historical context, no execution trace, and no evidence of what actually happened. Debugging becomes guesswork.

A system with excessive or poorly controlled logging is slow. Performance degrades due to unnecessary I/O, excessive formatting, and contention between threads. In extreme cases, logging itself becomes the dominant workload.

Professional logging exists to strike a careful balance.

1.1.1 Signal vs Noise

Not all information is equally valuable. A professional logging system emphasizes *signal* while minimizing *noise*.
Consider the following naïve approach:

```
void process_request(int id) {
    std::cout << "Entering process_request\n";
    std::cout << "Processing id: " << id << "\n";
    std::cout << "Finished process_request\n";
}
```

At first glance, this appears harmless. In reality, it produces unstructured, repetitive output that becomes meaningless when scaled to thousands of requests per second.
A structured and intentional approach focuses on meaning:

```
logger.log(LogLevel::Info,
           "Processing request",
           {{"request_id", id}});
```

The second example communicates intent, not implementation detail. It logs what matters, not every execution step.

1.1.2 Cost vs Insight

Every log statement has a cost. That cost may include:

- String formatting
- Heap allocation
- Synchronization
- Disk or console I/O

Consider this common mistake:

```
logger.log(LogLevel::Debug,
           "Result = " + std::to_string(expensive_computation()));
```

Even if the debug level is disabled, the computation and string construction still occur. This wastes CPU cycles and memory bandwidth.
A disciplined design separates decision from execution:

```
if (logger.is_enabled(LogLevel::Debug)) {
    logger.log(LogLevel::Debug,
               std::format("Result = {}", expensive_computation()));
}
```

Professional logging ensures that insight is gained only when its cost is justified.

1.1.3 Simplicity vs Flexibility

A logger must be simple enough to use consistently, yet flexible enough to evolve with the system.
Hard-coded approaches often start like this:

```
std::ofstream log_file("app.log");

void log_error(const std::string& msg) {
    log_file << msg << std::endl;
}
```

This design quickly collapses when requirements change:

- Multiple output destinations
- Different log formats

- Runtime configuration
- Thread safety

Logging infrastructure must anticipate change without becoming over-engineered. This balance is a design problem, not a syntactic one.

1.2 Common Logging Failures

Despite its importance, logging is frequently implemented poorly. The following failures appear repeatedly in real-world C++ systems.

1.2.1 Global Mutable State

One of the most common mistakes is the use of global mutable loggers:

```
std::ofstream global_log("app.log");

void log(const std::string& msg) {
    global_log << msg << std::endl;
}
```

This design introduces multiple problems:

- Undefined initialization order across translation units
- No clear ownership or lifetime
- Data races in multi-threaded code

Once globals are involved, reasoning about correctness becomes difficult, especially during startup and shutdown phases.

1.2.2 Blocking I/O on Hot Paths

Logging is often added to performance-critical sections without regard to cost:

```
for (auto& item : items) {
    logger.log(LogLevel::Debug, "Processing item");
    process(item);
}
```

If logging performs blocking I/O, this loop becomes dramatically slower. In high-frequency code paths, even a single blocking write can destroy throughput.
Professional systems either:

- Avoid logging in hot paths
- Use conditional logging
- Defer I/O via buffering or asynchronous mechanisms

1.2.3 Mixing Formatting and Transport

Another frequent failure is coupling message formatting directly to the output mechanism:

```
void log_error(const std::string& msg) {
    std::cerr << "[ERROR] " << msg << std::endl;
}
```

This design makes it impossible to:

- Reuse formatting logic
- Change output destinations
- Test logging behavior in isolation

Formatting and transport are distinct responsibilities and must remain separate.

1.2.4 Hard-Coding Output Destinations

Hard-coded destinations lock the system into a single mode of operation:

```
logger.log(LogLevel::Info, "Server started");
std::cout << "Server started\n";
```

As systems grow, logging requirements evolve: logs may need to go to files, sockets, memory buffers, or external collectors. Hard-coded destinations resist change and force invasive refactoring.

1.2.5 Hidden Coupling and Untestability

Perhaps the most damaging consequence of poor logging design is loss of testability.
When logging cannot be observed or controlled in tests, it becomes invisible behavior.
Invisible behavior cannot be verified, and unverified behavior eventually fails.
A professional logging system is observable, controllable, and testable by design.

Understanding why logging matters is the foundation for every design decision that follows. The remaining chapters build on these principles, transforming logging from an afterthought into a deliberate and reliable infrastructure component.

Chapter 2

Design Goals

This chapter defines the architectural boundaries of the logger built in this booklet. Before writing a single line of code, it is essential to state explicitly what the system is intended to do and, just as importantly, what it is *not* intended to do.
Clear design goals act as a contract. They prevent accidental complexity, guide implementation decisions, and provide a reference point when trade-offs must be made.

2.1 Explicit Design Constraints

The logger developed in this booklet follows a set of deliberate and strict constraints. These constraints are not arbitrary; each one exists to address a specific class of problems commonly encountered in real-world C++ systems.

2.1.1 Header-Light by Design

Our logger will be header-light.
In large C++ codebases, heavy headers dramatically increase:

- Compile times
- Coupling between modules
- Risk of ODR violations

Logging headers are often included everywhere. If they are expensive to include, they silently tax the entire build.
A header-light design ensures that:

- Only essential declarations appear in headers
- Implementation details remain in source files
- Dependencies are minimized

```
// logger.hpp
#pragma once

enum class LogLevel;

class Logger {
public:
    virtual ~Logger() = default;
    virtual void log(LogLevel level, std::string_view message) = 0;
};
```

The interface is minimal. Formatting, threading, and I/O remain out of the header.

2.1.2 Avoiding Macros Where Possible

Macros bypass the C++ type system. They obscure control flow, hinder debugging, and complicate tooling.
Many logging systems rely heavily on macros:

```
#define LOG_INFO(msg) \
    if (current_level <= INFO) log_impl(msg)
```

Such macros:

- Hide function boundaries
- Complicate breakpoints
- Interact poorly with namespaces

This booklet favors functions and types:

```
logger.log(LogLevel::Info, "Server started");
```

Macros may still appear at the edges (for compile-time elimination), but they are not the foundation of the design.

2.1.3 Supporting Multiple Log Levels

Log levels are not cosmetic. They encode intent and severity.
Our design supports multiple levels as a first-class concept:

```
enum class LogLevel {
    Trace,
    Debug,
    Info,
    Warning,
    Error,
    Fatal
};
```

Each level exists to answer a different operational question:

- **Trace** — fine-grained execution flow
- **Debug** — diagnostic information
- **Info** — normal system behavior
- **Warning** — unexpected but recoverable states
- **Error** — failures requiring attention
- **Fatal** — unrecoverable conditions

The logger enforces these distinctions rather than treating levels as labels.

2.1.4 Thread Safety by Design

Modern C++ systems are multi-threaded by default. Logging must assume concurrent access. Thread safety is not an afterthought. It is a primary design constraint.

```
void log(LogLevel level, std::string_view msg) {
    std::lock_guard<std::mutex> lock(mutex_);
    sink_->write(level, msg);
}
```

The design guarantees correctness first. Performance optimizations are introduced only after correctness is established.

2.1.5 Correct Use of RAII

Resource management in C++ must be explicit and deterministic.
The logger uses RAII to manage:

- File handles

- Buffers
- Synchronization primitives

```
class FileSink {
    std::ofstream file_;
public:
    explicit FileSink(const std::string& path)
        : file_(path, std::ios::app) {}
};
```

RAII ensures:

- No resource leaks
- Exception safety
- Predictable shutdown behavior

2.2 Non-Goals

Equally important to design goals are explicit non-goals. They protect the project from uncontrolled scope expansion.

2.2.1 Not an Enterprise Logging Framework

This logger is not intended to replace established frameworks.
It does not aim to provide:

- Distributed tracing
- Remote aggregation
- Dynamic configuration systems

Those concerns belong to larger systems with different constraints.

2.2.2 No Platform-Specific APIs

The logger avoids platform-specific APIs.
It does not depend on:

- Windows Event Tracing
- POSIX syslog
- OS-specific file systems

This guarantees portability and clarity of implementation.

2.2.3 No External Dependencies

Only the C++ standard library is used.
This constraint ensures:

- Ease of integration
- Long-term maintainability
- Complete transparency of behavior

Every line of code can be read, understood, and reasoned about using standard C++ knowledge alone.

By clearly stating both goals and non-goals, this chapter establishes the design boundaries that guide every subsequent implementation decision. These constraints are not limitations; they are the foundation of a clean, disciplined, and professional logging system.

Chapter 3

Log Levels and Semantics

Log levels form the semantic backbone of any serious logging system. They are not cosmetic labels, nor are they mere verbosity switches. When designed correctly, log levels encode intent, severity, and operational meaning, and they define how information flows from code to operators, developers, and diagnostic tools.
A logging system without clear semantics degenerates into noise. A logging system with well-defined semantics becomes a powerful tool for understanding complex behavior.

3.1 Defining Log Levels

We begin with an explicit and strongly typed definition of log levels:

```
enum class LogLevel {
    Trace,
    Debug,
    Info,
    Warning,
    Error,
```

```
    Fatal
};
```

Using a scoped enumeration is a deliberate choice. It provides:

- Strong typing and namespace isolation
- No implicit conversion to integers
- Clear intent at call sites

Each level represents a distinct semantic category, not merely a numerical rank.

3.1.1 Semantic Meaning of Each Level

Trace The `Trace` level represents the finest level of detail. It is intended for:

- Entry and exit points of functions
- Fine-grained execution flow
- Diagnostic experiments during development

```
logger.log(LogLevel::Trace, "Entering parse_expression()");
```

Trace logging is typically disabled in production environments due to its high volume and cost.

Debug The `Debug` level is used for diagnostic information that helps developers understand internal state.

```
logger.log(LogLevel::Debug,
           std::format("Cache size = {}", cache.size()));
```

Debug logs are valuable during development and troubleshooting but should not be required for normal operation.

Info The `Info` level describes normal, expected system behavior.

```
logger.log(LogLevel::Info, "Server started successfully");
```

Info logs answer the question: *What is the system doing under normal conditions?*

Warning The `Warning` level indicates unexpected situations that do not immediately break correctness but may require attention.

```
logger.log(LogLevel::Warning,
           "Configuration file missing optional field");
```

Warnings signal risk without declaring failure.

Error The `Error` level represents failures that affect correctness or functionality.

```
logger.log(LogLevel::Error,
           "Failed to open database connection");
```

Errors typically correspond to recoverable failures or failed operations.

Fatal The `Fatal` level represents unrecoverable conditions.

```
logger.log(LogLevel::Fatal,
           "Out of memory: shutting down");
```

Fatal logs often precede program termination and must be handled with extreme care.

3.1.2 Levels as Filtering Contracts

Each log level is a contract between the code and the logging system.
When a developer writes:

```
logger.log(LogLevel::Debug, "Computed hash value");
```

they are declaring:

- The information is optional
- It is intended for diagnostics
- It may be safely omitted in production

This contract enables filtering without ambiguity.

3.2 Level Filtering

Filtering determines whether a log statement produces observable output. This decision must satisfy strict engineering constraints.
Filtering must be:

- **Cheap** — minimal runtime overhead
- **Predictable** — consistent behavior across builds
- **Compile-time friendly** — allowing elimination when possible

3.2.1 Runtime Filtering

The simplest approach uses a runtime threshold:

```
class Logger {
public:
    explicit Logger(LogLevel level) : level_(level) {}

    bool is_enabled(LogLevel level) const {
        return level >= level_;
    }

private:
    LogLevel level_;
};
```

Usage becomes explicit and readable:

```
if (logger.is_enabled(LogLevel::Debug)) {
    logger.log(LogLevel::Debug,
               std::format("Value = {}", value));
}
```

This avoids unnecessary formatting and computation when a level is disabled.

3.2.2 Cost of Ignoring Filtering

Failing to filter correctly leads to subtle performance bugs:

```
logger.log(LogLevel::Debug,
           std::format("Expensive result = {}",
                       compute_expensive_result()));
```

Even if debug logging is disabled, the computation still occurs. This violates the cost vs insight balance discussed earlier.

3.2.3 Compile-Time Friendly Filtering

In performance-critical systems, some levels may be compiled out entirely.

```
constexpr bool enable_trace = false;

if constexpr (enable_trace) {
    logger.log(LogLevel::Trace, "Tracing enabled");
}
```

This approach allows the compiler to remove entire branches, ensuring zero runtime cost.

3.2.4 Predictability Across Builds

Filtering behavior must be predictable. A log statement should not change meaning between debug and release builds unless explicitly intended.
Implicit build-dependent behavior leads to confusion and inconsistent diagnostics.

3.2.5 Filtering as a Design Boundary

Filtering is not an implementation detail. It is a design boundary between:

- What the code expresses
- What the system observes

By treating log levels as semantic contracts and filtering as a first-class design concern, the logging system remains efficient, expressive, and trustworthy.

With clear semantics and disciplined filtering in place, the logger gains structure. The following chapters build upon this foundation, introducing formatting, sinks, and concurrency while preserving these guarantees.

Chapter 4

Core Logger Interface

At the heart of any logging system lies its public interface. This interface defines how the rest of the system *interacts* with logging, and therefore it must be designed with extreme care.
A poorly designed interface spreads complexity outward. A well-designed interface contains complexity inward.
This chapter focuses on defining a minimal, stable, and expressive core logger interface that can survive long-term evolution without forcing widespread refactoring.

4.1 Minimal Interface

We begin with the smallest interface that still captures the essence of logging:

```
class Logger {
public:
    virtual ~Logger() = default;
    virtual void log(LogLevel level,
                     const std::string& message) = 0;
};
```

At first glance, this interface may appear almost trivial. In reality, it encodes several important architectural decisions.

4.1.1 Why Minimalism Matters

Logging is ubiquitous. A logger is referenced from:

- Business logic
- Infrastructure code
- Error handling paths
- Diagnostic utilities

If the interface is heavy, unstable, or overly expressive, it becomes a liability. Every additional method:

- Increases coupling
- Reduces flexibility
- Raises the cost of change

By restricting the interface to a single responsibility—*recording a log entry*—we preserve long-term maintainability.

4.1.2 Virtual Destructor and Polymorphic Use

The presence of a virtual destructor is not optional.

```
virtual ~Logger() = default;
```

This guarantees correct destruction through base-class pointers:

```
std::unique_ptr<Logger> logger =
    std::make_unique<FileLogger>("app.log");
```

Without a virtual destructor, this pattern would result in undefined behavior.

4.1.3 Why a Virtual Interface?

The use of a virtual function establishes a clear abstraction boundary.

```
virtual void log(LogLevel level,
                 const std::string& message) = 0;
```

This allows:

- Multiple implementations (file, console, memory)
- Runtime substitution
- Dependency injection

The logger becomes a service rather than a concrete mechanism.

4.1.4 Separation of Policy and Mechanism

This interface deliberately separates:

- **Policy** — what should be logged and at which level
- **Mechanism** — how and where the log is written

Consider the calling code:

```
logger->log(LogLevel::Warning,
            "Connection retry limit exceeded");
```

The caller expresses *intent*. It does not care:

- Whether the log goes to a file or console
- Whether it is buffered or synchronous
- Whether it is formatted as text or JSON

Those concerns belong to the implementation.

4.1.5 Separating Formatting from Transport

Notice that the interface accepts a fully formed string.

```
virtual void log(LogLevel level,
                 const std::string& message) = 0;
```

This is intentional.
Formatting is a policy decision. Transport is a mechanism.
By keeping formatting outside the logger interface:

- The logger remains simple
- Formatting strategies can evolve independently
- Testing becomes easier

A caller may choose to format inline:

```
logger->log(LogLevel::Info,
            std::format("User {} logged in", user_id));
```

Or delegate formatting to a higher-level component. The logger does not impose a strategy.

4.1.6 Why Not Templates?

One might consider a templated interface:

```
template <typename... Args>
void log(LogLevel level, Args&&... args);
```

While attractive, such designs:

- Increase compile times
- Inflate headers
- Lock formatting into the logger

This booklet deliberately avoids that path to preserve clarity and separation of concerns.

4.1.7 Ownership and Lifetime Clarity

The interface does not manage ownership of the logger itself. That responsibility belongs to the surrounding architecture.
Typical usage patterns include:

```
void run(Logger& logger);
```

or

```
class Service {
    Logger& logger_;
};
```

By avoiding global access and static singletons, ownership remains explicit and testable.

4.1.8 Testability as a First-Class Concern

Because the interface is minimal, it is trivial to mock or replace:

```
class MemoryLogger : public Logger {
public:
    std::vector<std::string> entries;

    void log(LogLevel,
             const std::string& message) override {
        entries.push_back(message);
    }
};
```

This enables precise, deterministic tests without touching files or consoles.

4.1.9 Stability Over Time

A minimal interface is more likely to remain stable.
Once widely adopted, interfaces are extremely expensive to change. By keeping the surface area small, we increase the probability that this interface can survive years of evolution without modification.

This core interface is intentionally modest. It establishes a clean boundary between the rest of the system and the logging infrastructure. All subsequent features—formatting, filtering, threading, buffering, and sinks—will be layered on top of this foundation without violating its simplicity or clarity.

Chapter 5

Formatting Strategy

Formatting is one of the most underestimated aspects of logging design. It is often treated as a cosmetic concern, yet in reality it has deep implications for performance, correctness, maintainability, and long-term operational value.
This chapter explains why formatting must be treated as a deliberate design decision and demonstrates how Modern C++ provides powerful tools to do it correctly.

5.1 Why Formatting Matters

Every log entry is ultimately consumed by a human, a tool, or both. Formatting determines whether that consumption is efficient or painful.
Formatting directly affects three critical dimensions of a logging system.

5.1.1 Performance

Formatting is not free.
Even before a log message reaches its destination, formatting may incur:

- String allocations
- Temporary objects
- Locale handling
- Conversion overhead

Consider the following example:

```
logger.log(LogLevel::Debug,
           "Value = " + std::to_string(value));
```

This code performs dynamic allocation and string concatenation regardless of whether the log entry is ultimately emitted. In hot paths, this cost accumulates quickly.
Professional formatting strategies aim to:

- Avoid unnecessary work
- Defer formatting until needed
- Keep formatting logic explicit and visible

5.1.2 Readability

Logs are read under pressure.
They are examined:

- During incidents
- While debugging failures
- In post-mortem analysis

Poor formatting obscures meaning:

```
ERR42 usr17 fail op9
```

Clear formatting communicates intent:

```
[ERROR] user_id=17 operation=checkout failed
```

Readable logs reduce cognitive load and shorten recovery time.

5.1.3 Parsability

Logs are no longer consumed only by humans.
They are frequently:

- Parsed by scripts
- Indexed by log collectors
- Analyzed by monitoring systems

Unstructured formatting makes automated processing brittle. Consistent structure enables tooling.
This booklet focuses on text-based formatting, but the principles apply equally to structured formats such as JSON or key–value logs.

5.2 Formatting as a Separate Concern

A fundamental design rule is that formatting must be separated from transport.
Formatting answers the question: *How should information be represented?*
Transport answers the question: *Where should that representation go?*
Conflating these responsibilities leads to rigid and untestable designs.

```
// Poor design: formatting and transport mixed
void log_error(const std::string& msg) {
    std::cerr << "[ERROR] " << msg << std::endl;
}
```

This approach makes it impossible to:

- Reuse formatting logic
- Redirect output
- Test formatting independently

A professional logger keeps formatting outside the transport layer.

5.3 Using `std::format`

Modern C++ provides `std::format`, a type-safe, efficient, and expressive formatting facility introduced in C++20.

5.3.1 Basic Formatting Example

```
#include <format>

std::string format_message(LogLevel level,
                           std::string_view message) {
    return std::format("[{}] {}",
                       static_cast<int>(level),
                       message);
}
```

This function:

- Produces a complete, formatted message
- Has no side effects
- Is easy to test in isolation

The logger itself does not know how the message was formatted. It only receives a string.

5.3.2 Improving Semantic Clarity

Using raw integers for log levels is rarely ideal. We can improve clarity by mapping levels to names.

```
constexpr std::string_view to_string(LogLevel level) {
    switch (level) {
        case LogLevel::Trace:   return "TRACE";
        case LogLevel::Debug:   return "DEBUG";
        case LogLevel::Info:    return "INFO";
        case LogLevel::Warning: return "WARN";
        case LogLevel::Error:   return "ERROR";
        case LogLevel::Fatal:   return "FATAL";
    }
    return "UNKNOWN";
}
```

Now formatting becomes more expressive:

```
std::string format_message(LogLevel level,
                           std::string_view message) {
    return std::format("[{}] {}",
                       to_string(level),
                       message);
}
```

This improves both readability and downstream parsing.

5.3.3 Including Contextual Information

Formatting is also the natural place to add contextual metadata:

```
std::string format_message(LogLevel level,
                           std::string_view message,
                           std::string_view component) {
    return std::format("[{}][{}] {}",
                       to_string(level),
                       component,
                       message);
}
```

The logger remains unchanged. Only the formatting strategy evolves.

5.3.4 Formatting and Performance Discipline

Formatting should be performed only when required.

```
if (logger.is_enabled(LogLevel::Debug)) {
    logger.log(LogLevel::Debug,
               format_message(LogLevel::Debug,
                              compute_message(),
                              "cache"));
}
```

This ensures that expensive formatting does not occur when the message will be discarded.

5.3.5 Why `std::format` Over Alternatives

Compared to older approaches:

- `std::format` is type-safe

- It avoids printf-style mismatches
- It integrates naturally with C++ strings

Unlike streaming operators, it:

- Produces a single allocation-friendly result
- Avoids chained temporary objects
- Encourages explicit formatting boundaries

5.3.6 Testability of Formatting

Because formatting is a pure function, it is trivial to test:

```
auto msg = format_message(LogLevel::Info, "Started");
assert(msg == "[INFO] Started");
```

This level of testability is impossible when formatting is embedded inside I/O operations.

By treating formatting as a first-class design concern and keeping it outside the transport layer, we preserve clarity, performance, and flexibility. The next chapters build upon this separation, introducing concrete logging sinks and concurrency while maintaining these guarantees.

Chapter 6

File-Based Logger

Writing log entries to a file is one of the most common and enduring logging strategies. Files provide persistence, post-mortem analysis, and compatibility with existing tooling. However, file-based logging also introduces responsibility: ownership of resources, failure handling, buffering behavior, and interaction with the operating system.

This chapter demonstrates how to build a file-based logger that is correct, predictable, and aligned with Modern C++ principles.

6.1 RAII and File Ownership

At the core of a correct file-based logger lies one fundamental rule: **the file is a resource, and resources must be owned explicitly**.

In Modern C++, this responsibility is fulfilled through RAII (Resource Acquisition Is Initialization).

6.1.1 A Minimal File Logger

We begin with a simple implementation:

```
#include <fstream>

class FileLogger : public Logger {
    std::ofstream out;

public:
    explicit FileLogger(const std::string& path)
        : out(path, std::ios::app) {}

    void log(LogLevel level, const std::string& msg) override {
        out << msg << '\n';
    }
};
```

Despite its simplicity, this implementation already encodes several important design decisions.

6.1.2 Deterministic Resource Ownership

The `std::ofstream` object owns the file descriptor. Its lifetime is tied directly to the lifetime of the `FileLogger` instance.

```
{
    FileLogger logger("app.log");
    logger.log(LogLevel::Info, "Application started");
} // file is closed here
```

When the logger object goes out of scope:

- The file stream is flushed

- The file descriptor is closed
- All resources are released deterministically

There is no need for an explicit `close()` call. This is not convenience—it is correctness.

6.1.3 Exception Safety by Construction

RAII provides strong exception safety guarantees.
If an exception is thrown anywhere after the file has been opened, the destructor of `std::ofstream` will still execute.

```
void run(Logger& logger) {
    logger.log(LogLevel::Info, "Starting work");
    throw std::runtime_error("Unexpected failure");
}
```

Even in this scenario:

- The file remains in a valid state
- Resources are not leaked
- Log integrity is preserved up to the failure point

Manual resource management rarely provides this level of reliability.

6.1.4 Append Mode and Log Continuity

The file is opened using `std::ios::app`:

```
out(path, std::ios::app)
```

This ensures:

- Existing logs are preserved
- New entries are appended
- Multiple runs contribute to a continuous log history

Choosing the correct file mode is a policy decision. For some systems, truncation may be acceptable. For most production systems, continuity is essential.

6.1.5 Why the Logger Does Not Own the Path

Notice that the logger stores only the stream, not the file path.
This avoids:

- Redundant state
- Configuration leakage
- Path-dependent behavior

Once the stream is constructed, the logger's responsibility is to write, not to manage filesystem semantics.

6.1.6 Handling Write Failures

File I/O can fail. Disks fill up, permissions change, and filesystems become unavailable.
A minimal implementation does not ignore this reality.

```
void log(LogLevel, const std::string& msg) override {
    if (!out) {
        return; // or handle error
    }
    out << msg << '\n';
}
```

More advanced designs may:

- Track error states
- Disable logging after failure
- Escalate failures to monitoring systems

The key point is that file failure handling must be explicit, not assumed.

6.1.7 Flushing and Performance Trade-offs

By default, `std::ofstream` buffers output. This is desirable for performance.
Calling `std::endl` would force a flush:

```
out << msg << std::endl; // forces flush
```

This is intentionally avoided.
A professional logger:

- Uses `\n` for normal logging
- Flushes explicitly only when necessary
- Accepts that logs may lag slightly behind execution

Reliability does not require flushing on every write.

6.1.8 Single Responsibility Preserved

The file-based logger:

- Does not format messages
- Does not filter by level

- Does not manage threading

It writes strings to a file. Nothing more.
This restraint is deliberate. Each additional responsibility increases complexity and coupling.

6.1.9 RAII Guarantees Recap

By relying on RAII, the file-based logger guarantees:

- Deterministic resource release
- Exception-safe cleanup
- Clear ownership semantics
- Predictable shutdown behavior

These guarantees are not optional in infrastructure code. They are foundational.

The file-based logger demonstrates how Modern C++ makes correct resource management the default rather than the exception. In the next chapter, we extend this design to address concurrency, where the same discipline must be applied under far more demanding conditions.

Chapter 7

Thread Safety

Modern software systems are inherently concurrent. Even applications that appear simple on the surface often rely on multiple threads internally, whether for I/O, background tasks, event loops, or parallel computation.
Logging infrastructure must assume concurrency by default. If it does not, it will eventually fail in subtle and dangerous ways.
This chapter examines the concurrency challenges inherent in logging and introduces a correct, disciplined approach to thread safety using Modern C++.

7.1 The Concurrency Problem

Logging is frequently invoked from the most sensitive execution paths in a system. These include:

- **Multiple threads** executing concurrently
- **Error paths** where invariants may already be broken
- **Shutdown sequences** where object lifetimes are collapsing

In such contexts, incorrect logging behavior can:

- Corrupt log output
- Introduce data races
- Cause deadlocks
- Mask or exacerbate failures

Consider a naïve file-based logger used concurrently:

```
FileLogger logger("app.log");

void worker(int id) {
    logger.log(LogLevel::Info,
               std::format("Worker {} started", id));
}
```

If multiple threads call `log()` simultaneously, the behavior is undefined. Output may interleave, streams may corrupt internal state, and failures may be silent.

7.1.1 Why Logging Is Especially Vulnerable

Logging is often added after the fact. It appears in:

- Debugging sessions
- Emergency patches
- Failure recovery code

As a result, it frequently ends up in code that was not designed with logging in mind. Assuming single-threaded access is a common and costly mistake.

7.1.2 Thread Safety Is Not Optional

In infrastructure code, thread safety is not a feature toggle. It is a correctness requirement. A logger that is not thread-safe is only conditionally correct. Such conditional correctness is unacceptable in production systems.

7.2 Mutex-Based Protection

The simplest and most reliable way to ensure thread safety is mutual exclusion.

7.2.1 A Thread-Safe Logger

```
#include <mutex>
#include <fstream>

class ThreadSafeLogger : public Logger {
    std::mutex mtx;
    std::ofstream out;

public:
    explicit ThreadSafeLogger(const std::string& path)
        : out(path, std::ios::app) {}

    void log(LogLevel, const std::string& msg) override {
        std::lock_guard<std::mutex> lock(mtx);
        out << msg << '\n';
    }
};
```

This implementation establishes a clear critical section: only one thread may write to the file at a time.

7.2.2 Correctness First

The use of `std::lock_guard` ensures:

- Mutual exclusion
- Exception safety
- Deterministic lock release

Even if an exception were thrown during logging, the mutex would be released correctly. Correctness always precedes optimization.

7.2.3 Why This Works

The critical section protects:

- The internal state of `std::ofstream`
- The integrity of log output
- The ordering of log entries

By serializing access, the logger guarantees that each log entry appears as a coherent unit.

7.2.4 The Cost of Mutual Exclusion

While correct, mutex-based logging has a cost.

- Threads may block waiting for the lock
- Contention increases with logging frequency
- Latency spikes may occur under load

In high-throughput systems, this cost can become significant.

7.2.5 Hot Paths and Logging Discipline

Logging should rarely appear in hot paths. When it does, the cost of synchronization becomes visible.

```
for (;;) {
    logger.log(LogLevel::Debug, "Polling...");
}
```

This code will scale poorly regardless of the logger's implementation.
Professional systems combine:

- Conditional logging
- Appropriate log levels
- Awareness of execution context

7.2.6 Why This Is Still the Right Starting Point

Despite its cost, mutex-based protection is the correct baseline.
It is:

- Easy to reason about
- Easy to verify
- Hard to misuse

More advanced approaches—such as lock-free queues or asynchronous logging—build upon this foundation. They do not replace the need to understand it.

7.2.7 Thread Safety and RAII

The mutex itself is a resource. By embedding it as a member, its lifetime is tied to the logger.

```
std::mutex mtx;
```

This avoids:

- Dangling references
- Global synchronization primitives
- Hidden dependencies

Once again, RAII ensures correctness by construction.

7.2.8 Interaction with Shutdown

Thread-safe logging must also consider shutdown.
If threads outlive the logger, logging becomes undefined. Ownership and lifetime must be clear:

```
void run(Logger& logger);
```

The surrounding architecture must guarantee that logging stops before the logger is destroyed.

7.2.9 Not Always Optimal

The mutex-based approach is correct, but not always optimal.
High-performance systems may require:

- Asynchronous logging
- Per-thread buffers

- Lock-free data structures

These techniques reduce contention but increase complexity.
This booklet intentionally begins with the simplest correct solution. Optimization is introduced only after correctness and clarity are firmly established.

Thread safety is non-negotiable in logging infrastructure. By starting with a mutex-protected design, we ensure correctness under concurrency and lay a solid foundation for more advanced techniques in later chapters.

Chapter 8

Performance Considerations

Performance is not an optional concern in logging infrastructure. Because logging often appears in widely executed code paths, even small inefficiencies can scale into significant performance degradation.
This chapter examines the true costs associated with logging and demonstrates disciplined techniques to control them without sacrificing diagnostic value.

8.1 Cost of I/O

Among all operations involved in logging, disk I/O is by far the most expensive.
Compared to CPU instructions and memory access, disk writes are orders of magnitude slower. Even with buffering, the cost of interacting with the operating system dominates the execution time of a log call.

8.1.1 Understanding the Cost Hierarchy

A simplified hierarchy of costs looks like this:

- CPU arithmetic and branching — extremely cheap
- Memory access — cheap but not free
- Heap allocation — moderately expensive
- System calls — expensive
- Disk I/O — very expensive

Logging typically involves several of these steps.

8.1.2 The Danger of Logging in Tight Loops

Consider the following example:

```
for (int i = 0; i < 1'000'000; ++i) {
    logger.log(LogLevel::Debug, "Processing item");
}
```

Even if the message is small, this loop can:

- Saturate I/O buffers
- Introduce massive contention
- Render the application unusable

Logging inside tight loops is almost always a design mistake.

8.1.3 Hidden I/O Costs

The cost of I/O is often underestimated because it is hidden behind library abstractions.

```
out << msg << '\n';
```

This single line may involve:

- Buffer management
- Locale handling
- Synchronization
- Kernel interaction

The abstraction is convenient, but the cost is real.

8.1.4 I/O and Latency Sensitivity

In latency-sensitive systems, blocking I/O can be catastrophic. A single slow write can delay:

- Request processing
- Real-time responses
- Critical background tasks

Professional systems treat logging as a background concern whenever possible.

8.2 Lazy Logging

Lazy logging is the practice of deferring work until it is known to be necessary.
The core idea is simple: *never perform expensive work for a log message that will not be emitted*.

8.2.1 The Naïve Approach

A common mistake looks like this:

```
logger.log(LogLevel::Debug,
           std::format("Result = {}", expensive_computation()));
```

Even if debug logging is disabled, the expensive computation and formatting still occur.
This violates the principle of cost vs insight.

8.2.2 Explicit Level Checks

The simplest form of lazy logging uses an explicit level check:

```
if (level >= current_level) {
    logger.log(level, expensive_format());
}
```

This ensures that:

- Formatting is performed only when needed
- Expensive computations are avoided
- Logging overhead is minimized

8.2.3 Encapsulating the Check

To avoid repetitive code, the check can be encapsulated:

```
if (logger.is_enabled(LogLevel::Debug)) {
    logger.log(LogLevel::Debug,
               std::format("Value = {}", compute()));
}
```

This pattern makes intent explicit and readable.

8.2.4 Using Lambdas for Deferred Execution

A more advanced lazy logging technique uses deferred execution:

```
logger.log_if(LogLevel::Debug, [&] {
    return std::format("Value = {}", compute());
});
```

In this design:

- The lambda is invoked only if the level is enabled
- No unnecessary computation occurs
- Call sites remain concise

Such techniques are common in high-performance logging frameworks.

8.2.5 Compile-Time Elimination

Some log levels may be completely eliminated at compile time:

```
constexpr bool enable_trace = false;

if constexpr (enable_trace) {
    logger.log(LogLevel::Trace, "Tracing enabled");
}
```

This results in zero runtime overhead. The compiler removes the code entirely.

8.2.6 Performance vs Clarity

Performance optimizations must not obscure intent.
Overly clever logging macros and templates often:

- Obscure control flow
- Complicate debugging
- Increase compile times

This booklet favors explicit, readable patterns that balance performance with clarity.

8.2.7 When Performance Truly Matters

Not all systems require extreme optimization. However, logging is frequently present in:

- Infrastructure code
- Networking paths
- Error handling logic

In these areas, careless logging can dominate runtime behavior.

8.2.8 Key Principle

The central rule is simple and non-negotiable:

> Never format messages that will be discarded.

By respecting this principle, logging remains a diagnostic asset rather than a performance liability.

Understanding the true cost of logging and applying lazy techniques allows systems to remain observable without sacrificing performance. The next chapters build upon these principles to introduce extensibility and more advanced logging patterns while preserving efficiency.

Chapter 9

Extensibility

A logging system that cannot evolve will eventually be replaced. Extensibility is therefore not an optional feature, but a core design requirement.
This chapter explores how a clean, minimal logger interface enables powerful extension without sacrificing correctness, performance, or clarity. The emphasis is on architectural techniques that scale with system complexity while remaining easy to reason about.

9.1 Multiple Sinks

A *sink* is a destination for log messages. Different environments and use cases require different sinks, sometimes simultaneously.
A well-designed logger must not assume a single output destination.

9.1.1 Why Multiple Sinks Matter

Consider the following scenarios:

- During development, logs should appear on the console

- In production, logs must be written to files
- In tests, logs should be captured in memory

Hard-coding a single destination forces invasive refactoring whenever requirements change.

9.1.2 Sink as an Abstraction

Instead of embedding output logic directly into the logger, we introduce a sink abstraction:

```
class LogSink {
public:
    virtual ~LogSink() = default;
    virtual void write(const std::string& message) = 0;
};
```

The logger delegates output to a sink:

```
class Logger {
public:
    explicit Logger(std::unique_ptr<LogSink> sink)
        : sink_(std::move(sink)) {}

    void log(LogLevel, const std::string& msg) {
        sink_->write(msg);
    }

private:
    std::unique_ptr<LogSink> sink_;
};
```

This separation allows new sinks to be added without modifying existing code.

9.1.3 Console Sink

A simple console sink illustrates the idea:

```
class ConsoleSink : public LogSink {
public:
    void write(const std::string& message) override {
        std::cout << message << '\n';
    }
};
```

This sink is ideal for development and interactive debugging.

9.1.4 File Sink

The file-based sink encapsulates persistent storage:

```
class FileSink : public LogSink {
    std::ofstream out_;

public:
    explicit FileSink(const std::string& path)
        : out_(path, std::ios::app) {}

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

Notice that file ownership and buffering remain localized within the sink.

9.1.5 In-Memory Sink

For testing and diagnostics, an in-memory sink is invaluable:

```
class MemorySink : public LogSink {
public:
    std::vector<std::string> entries;

    void write(const std::string& message) override {
        entries.push_back(message);
    }
};
```

This sink allows precise assertions in unit tests without touching I/O.

9.1.6 Benefits of Multiple Sinks

This design enables:

- Environment-specific logging behavior
- Testability without mocks
- Clean separation of concerns

Most importantly, sinks can be added or replaced without modifying the logger's public interface.

9.2 Composition Over Inheritance

Extensibility does not require deep inheritance hierarchies. In fact, excessive inheritance often leads to rigid and fragile designs.
This booklet favors **composition over inheritance**.

9.2.1 The Limits of Inheritance

A naïve approach might attempt to model behavior using subclasses:

```
class FilteringFileLogger : public FileLogger { /* ... */ };
class AsyncFileLogger : public FileLogger { /* ... */ };
```

Such designs quickly explode combinatorially. Each new feature multiplies the number of required subclasses.
Inheritance encodes behavior statically. Composition allows behavior to be assembled dynamically.

9.2.2 Fan-Out Logging

Fan-out logging sends a single log entry to multiple sinks.

```
class FanOutSink : public LogSink {
    std::vector<std::unique_ptr<LogSink>> sinks_;

public:
    void add_sink(std::unique_ptr<LogSink> sink) {
        sinks_.push_back(std::move(sink));
    }

    void write(const std::string& message) override {
        for (auto& sink : sinks_) {
            sink->write(message);
        }
    }
};
```

This enables scenarios such as:

- Writing logs to both file and console

- Persisting logs while capturing them for tests
- Broadcasting logs to multiple subsystems

9.2.3 Filtering Decorators

Filtering behavior can also be implemented as a composable layer.

```
class FilteringSink : public LogSink {
    LogLevel min_level_;
    std::unique_ptr<LogSink> wrapped_;

public:
    FilteringSink(LogLevel level,
                  std::unique_ptr<LogSink> sink)
        : min_level_(level), wrapped_(std::move(sink)) {}

    void write(const std::string& message) override {
        wrapped_->write(message);
    }
};
```

In a complete design, the filtering sink would receive structured data (level, message) rather than a raw string. The important point is architectural: behavior is layered, not inherited.

9.2.4 Advantages of Composition

Composition provides:

- Flexible behavior assembly
- Fewer class explosions
- Clear, testable components

- Runtime configurability

Each component has a single responsibility. Complex behavior emerges from simple building blocks.

9.2.5 Extensibility Without Fragility

Because extensions are built by composing existing components:

- Existing code remains unchanged
- Interfaces remain stable
- The risk of regressions is reduced

This is the hallmark of a mature infrastructure design.

Extensibility is not achieved by predicting every future requirement. It is achieved by designing clear boundaries and assembling behavior from small, well-defined parts.
By supporting multiple sinks and favoring composition over inheritance, the logging system remains adaptable, testable, and robust as the system around it evolves.

Chapter 10

Testing the Logger

Testing is where design quality is revealed. Components that were designed with clear boundaries and explicit responsibilities are easy to test. Components that mix concerns, hide behavior, or depend on global state are not.

Logging is often excluded from testing under the false assumption that it is "just output". In reality, logging is observable behavior, and observable behavior must be testable.

This chapter explains why logging must be treated as a testable subsystem and demonstrates practical techniques for achieving deterministic, reliable tests in Modern C++.

10.1 Why Logging Must Be Testable

Logs are not an implementation detail. They are part of a system's externally observable behavior.

When a system fails, logs are often the *only* evidence of what occurred. If logging behavior cannot be verified, then an essential part of the system remains unvalidated.

10.1.1 Logging as Behavioral Output

Consider the following function:

```
bool authenticate(User& user, Logger& logger) {
    if (!user.is_valid()) {
        logger.log(LogLevel::Warning, "Invalid user credentials");
        return false;
    }
    return true;
}
```

The return value alone does not fully describe the behavior. The emitted log message is part of the contract:

- It communicates failure
- It provides diagnostic context
- It supports auditing and monitoring

If tests only check the return value and ignore logging, they verify only half of the behavior.

10.1.2 The Cost of Untestable Logging

When logging is untestable, several problems emerge:

- Log messages silently regress
- Important diagnostics disappear unnoticed
- Error paths lose visibility

These failures often surface only in production, when fixing them is most expensive.

10.1.3 Why File and Console Logs Are Poor Test Targets

Testing against files or consoles is brittle:

- Tests depend on the filesystem
- Output ordering may vary
- Parallel tests interfere with each other

Such tests are slow, flaky, and difficult to maintain.
Professional testing requires isolation and determinism.

10.2 In-Memory Logger

The simplest and most effective solution is an in-memory logger. Instead of writing to I/O devices, it records log entries in a container that can be inspected directly by tests.

10.2.1 A Minimal In-Memory Logger

```
class MemoryLogger : public Logger {
public:
    std::vector<std::string> entries;

    void log(LogLevel, const std::string& msg) override {
        entries.push_back(msg);
    }
};
```

This implementation is intentionally minimal. Yet it immediately unlocks powerful testing capabilities.

10.2.2 Deterministic Behavior

Because all log entries are stored in memory:

- No I/O occurs
- No timing dependencies exist
- Results are fully deterministic

This makes tests fast, repeatable, and reliable.

10.2.3 Testing Logging Side Effects

Using the in-memory logger, logging behavior becomes directly observable.

```
MemoryLogger logger;
User user{/* invalid state */};

bool result = authenticate(user, logger);

assert(result == false);
assert(logger.entries.size() == 1);
assert(logger.entries[0] == "Invalid user credentials");
```

The test verifies:

- Functional behavior (return value)
- Logging behavior (message content)
- Log emission count

This is full behavioral verification.

10.2.4 Testing Error Paths

Error paths are notoriously hard to test. Logging often provides the only insight into those paths.

```
void load_config(const std::string& path, Logger& logger) {
    if (!file_exists(path)) {
        logger.log(LogLevel::Error,
                   "Configuration file not found");
        return;
    }
}
```

Test coverage becomes straightforward:

```
MemoryLogger logger;
load_config("missing.cfg", logger);

assert(logger.entries.size() == 1);
assert(logger.entries[0] == "Configuration file not found");
```

No filesystem setup is required. The test remains isolated and clear.

10.2.5 Avoiding Mocking Frameworks

Because the logger interface is minimal, no mocking framework is needed.
The in-memory logger:

- Is simpler than a mock
- Behaves like a real logger
- Requires no external tooling

This aligns with the philosophy of using real objects where possible.

10.2.6 Testing Order and Frequency

Logs often encode sequencing information.

```
logger.log(LogLevel::Info, "Start");
logger.log(LogLevel::Info, "Processing");
logger.log(LogLevel::Info, "Done");
```

Tests can assert ordering explicitly:

```
assert(logger.entries[0] == "Start");
assert(logger.entries[2] == "Done");
```

This is impossible when logs are written directly to I/O.

10.2.7 Extending the In-Memory Logger

The in-memory logger can be extended to support richer assertions:

```
struct LogEntry {
    LogLevel level;
    std::string message;
};

class StructuredMemoryLogger : public Logger {
public:
    std::vector<LogEntry> entries;

    void log(LogLevel level,
             const std::string& msg) override {
        entries.push_back({level, msg});
    }
};
```

This allows tests to assert both content and severity.

10.2.8 Logging Tests as Design Feedback

If logging is difficult to test, it is a design smell.
Difficult-to-test logging often indicates:

- Hidden global state
- Excessive coupling
- Mixed responsibilities

Testing pressure exposes architectural weaknesses early.

10.2.9 Key Principle

The guiding principle is simple:

> If a log message matters, it must be testable.

By treating logging as observable behavior and providing in-memory implementations, we ensure that diagnostics remain reliable, intentional, and verifiable throughout the system's lifetime.

With testable logging in place, the system gains confidence. Behavior is no longer inferred from production failures but verified continuously during development. This is a hallmark of professional Modern C++ engineering.

Chapter 11

Common Design Mistakes

Even experienced engineers repeat the same logging mistakes across projects. These errors rarely appear catastrophic in isolation, but in aggregate they undermine correctness, performance, testability, and long-term maintainability.
This chapter catalogues the most common design mistakes encountered in real-world C++ logging systems and explains why they are dangerous. Each mistake is illustrated with concrete examples and corrective guidance.

11.1 Logging Inside Destructors

One of the most subtle and dangerous mistakes is logging from destructors.

11.1.1 The Temptation

At first glance, logging in a destructor seems reasonable:

```
class Resource {
public:
```

```
    ~Resource() {
        logger.log(LogLevel::Info, "Resource destroyed");
    }
};
```

The intention is usually diagnostic: tracking lifetimes or detecting leaks.

11.1.2 Why This Is Dangerous

Destructors execute under fragile conditions:

- During stack unwinding
- During program shutdown
- After partial object destruction

If the logger itself:

- Has already been destroyed
- Depends on other destroyed resources
- Throws or blocks

the behavior becomes undefined.

11.1.3 Destructor Logging During Exceptions

Consider stack unwinding:

```
void process() {
    Resource r;
    throw std::runtime_error("Failure");
}
```

The destructor runs while an exception is active. If logging throws or blocks, the program may terminate via `std::terminate()`.

11.1.4 Correct Approach

Destructors should be:

- `noexcept`
- Free of side effects
- Limited to resource release

If lifecycle tracing is required, use explicit lifecycle methods:

```
void shutdown(Logger& logger) {
    logger.log(LogLevel::Info, "Resource shutting down");
}
```

> Destructors are for cleanup, not diagnostics.

11.2 Global Static Initialization Order

Another common failure involves global loggers.

11.2.1 The Problematic Pattern

```
Logger global_logger;

void foo() {
    global_logger.log(LogLevel::Info, "Hello");
}
```

This looks harmless, but it hides a well-known C++ pitfall: *static initialization order fiasco*.

11.2.2 Why This Breaks

Across translation units, the order of global initialization is undefined.

```
// file_a.cpp
extern Logger global_logger;

// file_b.cpp
Logger global_logger;
```

If another global object logs during its constructor, the logger may not yet be initialized. During shutdown, the reverse happens: the logger may be destroyed before other globals that attempt to log.

11.2.3 Symptoms

These bugs manifest as:

- Missing log entries
- Crashes during startup or shutdown
- Intermittent failures

They are notoriously difficult to diagnose.

11.2.4 Correct Approach

Avoid global loggers. Prefer explicit ownership and dependency injection:

```
int main() {
    FileLogger logger("app.log");
    run(logger);
}
```

This makes initialization order explicit and verifiable.

11.3 Excessive Formatting

Formatting is often the hidden cost of logging.

11.3.1 The Anti-Pattern

```
logger.log(LogLevel::Debug,
           std::format("x={}, y={}, z={}", x, y, z));
```

When placed in frequently executed code, this becomes a performance trap.

11.3.2 Why This Hurts

Even if the log level is disabled:

- Formatting still occurs
- Temporary strings are allocated
- CPU time is wasted

The cost is paid regardless of output.

11.3.3 Amplification Through Scale

A single formatting call may seem trivial. Millions of them per second are not.
Logging overhead scales with execution frequency, not perceived importance.

11.3.4 Correct Approach

Always guard expensive formatting:

```
if (logger.is_enabled(LogLevel::Debug)) {
    logger.log(LogLevel::Debug,
               std::format("x={}, y={}, z={}", x, y, z));
}
```

Or use deferred formatting:

```
logger.log_if(LogLevel::Debug, [&] {
    return std::format("x={}, y={}, z={}", x, y, z);
});
```

> If a message might be discarded, its formatting must be discardable too.

11.4 Mixing Concerns

Perhaps the most pervasive mistake is mixing unrelated responsibilities inside the logging system.

11.4.1 A Typical Example

```
void log_error(const std::string& msg) {
    std::cerr << "[ERROR] "
              << current_time()
              << " | "
              << msg
              << std::endl;
}
```

This single function performs:

- Formatting

- Time retrieval
- Output
- Severity labeling

Such designs become rigid and untestable.

11.4.2 Consequences of Mixed Concerns

Mixing concerns leads to:

- Tight coupling
- Code duplication
- Inflexible output formats
- Difficult testing

Changing any one aspect forces changes everywhere.

11.4.3 Correct Separation

Each responsibility should be isolated:

- Formatting logic
- Severity semantics
- Transport mechanism
- Filtering policy

```
std::string msg = formatter.format(level, text);
sink.write(msg);
```

This separation allows each part to evolve independently.

11.4.4 Design Smell Indicator

If adding a new feature requires modifying many unrelated parts of the logging system, concerns are mixed.
Well-designed logging systems grow by composition, not modification.

11.5 Summary of Common Mistakes

The most frequent logging design mistakes can be summarized as follows:

- Logging from destructors
- Relying on global static loggers
- Performing unnecessary formatting
- Mixing formatting, policy, and transport

Each of these mistakes violates a core Modern C++ principle: *explicit ownership, clear boundaries, and predictable behavior*.

Avoiding these pitfalls does not require advanced techniques. It requires discipline, restraint, and respect for infrastructure design.
A logging system that avoids these mistakes becomes an asset rather than a liability—and that distinction defines professional software engineering.

Conclusion

A logger is not a peripheral utility. It is a foundational component of any serious software system. Its design choices echo throughout the entire codebase, influencing debuggability, performance, reliability, and long-term maintainability.
The quality of a logging system often reflects the maturity of the engineering culture behind it. A hastily assembled logger reveals short-term thinking. A carefully designed logger demonstrates architectural discipline.
Throughout this booklet, logging has been treated not as a convenience, but as infrastructure. This perspective changes how decisions are made:

- Simplicity is favored over cleverness
- Correctness precedes optimization
- Explicit design replaces hidden behavior

Modern C++ provides all the tools required to build such infrastructure without compromise.

RAII

RAII enables deterministic resource management. Files, buffers, mutexes, and other resources are acquired and released predictably. This is essential for logging, which often operates during failure and shutdown scenarios.

Strong Typing

Strong typing encodes intent directly into the interface. Log levels are not integers. Sinks are not interchangeable functions. Clear types prevent misuse and make invalid states harder to represent.

Concurrency Primitives

Modern C++ offers well-defined concurrency primitives. Mutexes, locks, and atomics allow thread-safe logging to be implemented correctly and portably. Concurrency is addressed explicitly rather than ignored or assumed away.

Zero-Cost Abstractions

Abstractions in C++ need not impose runtime penalties. When designed carefully, interfaces, composition, and layering preserve performance while improving clarity and extensibility.

Building a logger is not about producing text output. It is about:

- Defining clear boundaries
- Managing resources explicitly
- Balancing performance and observability
- Designing for evolution and testing

The logger constructed in this booklet is intentionally modest. It does not attempt to solve every problem or replicate enterprise-scale frameworks. Instead, it demonstrates how sound engineering principles translate into a reliable, extensible infrastructure component.
The same principles apply far beyond logging. They scale to configuration systems, metrics, tracing, and every other form of infrastructure code.

A well-designed logger is not merely useful. It is a signal that the system was built with care, foresight, and professional responsibility.

Appendices

The appendices serve as a practical reference section. They consolidate the architectural principles discussed throughout the booklet into actionable checklists, outline natural extension paths, and define terminology with precision.
These sections are intended to be revisited repeatedly during real-world design and implementation work.

Appendix A: Design Checklist

This checklist summarizes the essential questions that should be asked when designing, reviewing, or refactoring a logging system. Each item represents a core architectural invariant rather than a feature.

Separation of Responsibilities

- Is message formatting clearly separated from output transport?
- Can formatting be changed without modifying file or console logic?
- Can transport be changed without rewriting formatting code?

A failure in this category usually indicates tightly coupled code and poor extensibility.

Thread Safety

- Can multiple threads call the logger concurrently?
- Are shared resources properly synchronized?
- Is logging safe during error handling and shutdown paths?

If thread safety is unclear, the logger is not production-ready.

Performance Discipline

- Are log levels filtered before expensive work is performed?
- Is unnecessary formatting avoided when logs are disabled?
- Is blocking I/O avoided in hot paths?

Logging must never dominate runtime cost.

Testability

- Can logging behavior be observed in unit tests?
- Is there an in-memory or test sink available?
- Can log content, order, and severity be asserted deterministically?

If logging cannot be tested, it cannot be trusted.

Ownership and Lifetime

- Is logger ownership explicit and well-scoped?
- Are global or static loggers avoided?
- Is shutdown behavior predictable?

Explicit ownership prevents the most subtle and dangerous failures.

This checklist should be treated as a gate. If any item cannot be answered confidently, the design requires further work.

Appendix B: Possible Extensions

The logger built in this booklet is intentionally minimal. Its design, however, allows natural extension in several directions. This appendix outlines common enhancements and the design considerations they introduce.

Asynchronous Logging

Asynchronous logging decouples log production from log consumption.
Typical characteristics:

- Log calls enqueue messages
- A background thread performs I/O
- Reduced contention in hot paths

Design considerations include:

- Queue capacity and backpressure

- Shutdown synchronization
- Failure handling in background threads

Asynchronous logging improves performance but increases complexity.

Binary Log Formats

Instead of human-readable text, logs may be stored in binary form.
Benefits:

- Reduced storage size
- Faster serialization
- Precise data representation

Costs:

- Reduced immediate readability
- Need for dedicated tooling
- Compatibility concerns

Binary formats are most useful in high-throughput systems.

Structured Logging

Structured logging records logs as key–value pairs rather than free-form text.
Example conceptual structure:

- Timestamp
- Severity

- Component
- Message
- Context fields

Benefits include:

- Machine-friendly parsing
- Better integration with monitoring systems
- More precise querying

Structured logging is a natural evolution of the formatting strategies introduced earlier.

Log Rotation

Long-running systems must manage log growth.
Rotation strategies include:

- Size-based rotation
- Time-based rotation
- External rotation via system tools

Design concerns:

- Atomic file switching
- Data loss prevention
- Coordination with external processes

Log rotation should be explicit and predictable.

Each extension builds upon the same foundational principles: clear ownership, separation of concerns, and disciplined performance management.

Appendix C: Terminology

This glossary defines key terms as they are used throughout the booklet. The definitions are precise and consistent with Modern C++ design practice.

Logger A component responsible for receiving log entries and coordinating their processing according to defined policies.

Log Entry A single recorded event, consisting of a severity level and a message, and optionally additional contextual information.

Severity The semantic importance of a log entry, expressed as a log level such as `Info`, `Warning`, or `Error`.

Sink A destination for log entries, such as a file, console, or in-memory buffer.

Formatting The transformation of structured log data into a textual or binary representation.

Transport The mechanism by which formatted log entries are delivered to a sink.

Filtering The process of determining whether a log entry should be emitted based on severity or other criteria.

RAII Resource Acquisition Is Initialization. A C++ idiom that ties resource lifetime to object lifetime, ensuring deterministic acquisition and release.

Lazy Logging A technique in which formatting and computation are deferred until it is certain that a log entry will be emitted.

Hot Path A frequently executed code path where performance sensitivity is high.

These terms form a shared vocabulary. Using them consistently reduces ambiguity and improves communication between engineers working on logging and other infrastructure components.

References

This booklet is grounded in established standards, long-standing engineering practices, and real-world experience from large-scale C++ systems. The following references represent the conceptual and technical foundations upon which the material in this work is based.
They are not cited to encourage dependency on external frameworks, but to emphasize that the design principles presented here are aligned with the broader Modern C++ ecosystem and professional software engineering practice.

ISO C++ Standard

- **ISO/IEC 14882:2020 (C++20)** — Introduces `std::format`, enhanced concurrency primitives, and modern language features that enable safer and more expressive infrastructure design.

- **ISO/IEC 14882:2023 (C++23)** — Refines and extends the Modern C++ toolset, reinforcing zero-cost abstractions, constexpr capabilities, and library consistency relevant to performance-sensitive components such as logging.

The logger design in this booklet intentionally relies only on standard C++ facilities to ensure portability, longevity, and clarity.

C++ Core Guidelines

- The C++ Core Guidelines provide authoritative guidance on:
  - Resource management via RAII
  - Ownership and lifetime semantics
  - Concurrency safety and data race avoidance
  - Interface design and abstraction boundaries

Many of the architectural decisions discussed—such as avoiding global state, minimizing interfaces, and preferring composition—are direct applications of these guidelines.

Compiler Documentation

- **GCC** — Behavior of standard library implementations, optimization strategies, and code generation implications relevant to logging performance.
- **Clang/LLVM** — Diagnostics, tooling, and optimization models that influence how logging code is analyzed and optimized.
- **MSVC** — Platform-specific considerations, ABI stability, and standard library behavior on Windows systems.

Understanding compiler behavior is essential when evaluating the real cost of abstractions used in logging infrastructure.

Large-Scale C++ System Design Practices

- Experience from long-lived C++ systems highlights recurring themes:

  - Infrastructure code must favor correctness over cleverness
  - Observability must not compromise performance
  - Testability is a design requirement, not an afterthought

- Logging systems in production environments consistently reflect these constraints, regardless of domain.

The patterns and warnings presented throughout this booklet are drawn from these collective practices rather than from theoretical models alone.

Closing Note on References

This booklet intentionally avoids dependency on third-party logging libraries or proprietary tooling. The goal is not to promote a specific implementation, but to cultivate the ability to reason about logging systems critically and design them responsibly using Modern C++ alone. The references listed here serve as intellectual anchors, ensuring that the principles presented are aligned with the broader C++ community and with professional engineering standards that have proven their value over time.