https://simplifycpp.org

Best Practices in Modern C++

The Golden Principles

Prepared By Ayman Alheraki

Best Practices in Modern C++

The Golden Principles

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Contents **2**

Author's Introduction **4**

1 Core Language Usage **7**

1.1 Rule 1 — Prefer Value Semantics by Default 7

1.2 Rule 2 — Use `auto` to Express Intent, Not Laziness 10

2 Object Lifetime and RAII **13**

2.1 Rule 3 — Resource Ownership Must Be Explicit 13

2.2 Rule 4 — Destructors Must Never Throw 17

3 Memory and Ownership **21**

3.1 Rule 5 — Never Use Raw `new` and `delete` in Application Code 21

3.2 Rule 6 — Use `shared_ptr` Only for Shared Ownership 24

4 Move Semantics and Value Categories **28**

4.1 Rule 7 — Always Implement the Rule of Five if You Manage Resources 28

5 Error Handling and Contracts **35**

5.1 Rule 8 — Exceptions Represent Failure to Fulfill a Contract 35

6 Concurrency and Atomics **42**

6.1 Rule 9 — Data Race Freedom Is a Design Property 42

7 Performance Engineering **48**

7.1 Rule 10 — Measure Before Optimizing 48

8 API and Library Design **56**

8.1 Rule 11 — APIs Must Encode Ownership in the Type System 56

9 Build Systems and Toolchains **63**

9.1 Rule 12 — Treat Warnings as Errors 63

References **75**

Author's Introduction

This booklet was written with a single, uncompromising goal: to define what it truly means to write **professional Modern C++** in real production environments.
This is not a language tutorial. It is not a collection of syntax explanations. It is not a reference manual.
It is a **distillation of engineering discipline**.
Over the past decades, C++ has evolved from a powerful low-level language into a highly expressive systems engineering platform. Yet most real-world failures in C++ software do not originate from missing language features — they originate from:

- Broken ownership models
- Undefined behavior
- Incorrect lifetime management
- Unsafe concurrency
- Unmeasured performance assumptions
- Poor API contracts

Modern C++ gives engineers the tools to eliminate these classes of failure entirely. What it does *not* give is forgiveness for ignoring them.
This booklet exists to formalize a set of **engineering survival laws** that separate:

- Systems that scale from
- Systems that eventually collapse

Every rule presented here is:

- Derived from real production failures
- Verified across large-scale systems
- Applicable across platforms and domains
- Independent of frameworks and trends

These rules apply equally to:

- Backend infrastructure
- Game engines
- Embedded systems
- Financial platforms
- Real-time control software
- Compilers and runtimes

The target reader of this booklet is not a beginner. It is the engineer who already writes C++, and now seeks:

- Predictable correctness
- Measurable performance
- Deterministic behavior

- Maintainable architectures
- APIs that resist misuse

Every page is written from the perspective that:

> *Modern C++ is not about what the language allows — it is about what professional engineering demands.*

If this booklet changes how you design ownership, structure concurrency, measure performance, or expose APIs — then it has fulfilled its purpose.
The responsibility for correctness in C++ has always belonged to the engineer. This booklet exists to clarify what that responsibility truly entails.

`Ayman Alheraki`

Chapter 1

Core Language Usage

1.1 Rule 1 — Prefer Value Semantics by Default

Principle: Value semantics mean that objects behave like values: they are copied, moved, and destroyed in a deterministic and predictable manner. This model provides the strongest guarantees for correctness, composability, testability, and optimization. Modern C++ is fundamentally designed around value semantics as the primary abstraction model.
When you pass objects by value or by const reference, you express:

- Clear ownership boundaries
- Deterministic lifetime
- Strong aliasing guarantees
- Maximum freedom for compiler optimization

Pointer-based APIs, by contrast, introduce:

- Implicit nullability

- Unclear ownership
- Aliasing ambiguity
- Hidden lifetime dependencies

Bad Example (Pointer Semantics):

```
void process(std::vector<int>* v);
```

This signature encodes multiple ambiguities:

- Is `v` allowed to be `nullptr`?
- Does the function own the vector?
- Is the function allowed to reassign the pointer?
- Who controls the lifetime?

Correct Modern C++ (Value Semantics):

```
void process(const std::vector<int>& v);
```

This version expresses:

- Non-null guarantee
- No ownership transfer
- Read-only access
- Stable lifetime requirement

Advanced Variant (Move-Enabled API):

```
void process(std::vector<int> v); // allows move optimization
```

This enables:

- Zero-copy transfer from temporaries
- Strong ownership transfer semantics
- Full move optimization under NRVO

Impact on Performance and Safety:

- Eliminates null-pointer dereferencing
- Improves alias analysis
- Enables better vectorization
- Removes pointer-based cache unpredictability
- Enhances inlining and escape analysis

Common Mistakes in Real Projects:

- Passing objects by raw pointer to avoid “copying”
- Using pointers to express optional values instead of `std::optional`
- Designing APIs that return raw owning pointers
- Treating value passing as inherently expensive

Engineering Rule of Thumb: *If your API accepts a pointer, you must justify why a reference or a value is not correct.*

1.2 Rule 2 — Use `auto` to Express Intent, Not Laziness

Principle: `auto` is a **type deduction tool**, not a typing shortcut. Its purpose is to:

- Preserve correctness under refactoring
- Express conceptual rather than concrete types
- Avoid iterator and lambda verbosity
- Prevent implicit narrowing and signedness bugs

Used correctly, `auto` strengthens code. Used lazily, it erodes type clarity.

Bad Example (Type-Erasing Laziness):

```
auto x = getComplexTemplateType();
```

Problems:

- The semantic meaning of `x` is invisible
- The return category (value, reference, proxy) is unclear
- Refactoring hazards increase
- Debugging becomes opaque

Correct Modern C++ (Intent-Expressive Use):

```
auto count = container.size();
```

This expresses:

- Logical meaning: “count”
- Type follows container contract

- No narrowing conversion
- Immunity to future container type changes

Advanced Correct Usage Patterns:

Iterator Deduction:

```
for (auto it = data.begin(); it != data.end(); ++it)
{
    process(*it);
}
```

Lambda Return Deduction:

```
auto transform = [](int x) { return x * x; };
```

Reference Preservation:

```
auto& ref = container[0];
```

Impact on Performance and Safety:

- Prevents accidental copies
- Preserves reference qualifiers
- Eliminates signed/unsigned mismatch bugs
- Avoids template instantiation coupling
- Improves refactor stability

Common Mistakes in Real Projects:

- Using `auto` for public API types
- Writing `auto x = 0;` and expecting type safety

- Losing reference qualifiers by omitting `&`
- Masking expensive proxy return types
- Using `auto` to avoid understanding the type

Engineering Rule of Thumb: *Use `auto` when the type is obvious from context, not when it is important to understand explicitly.*

Chapter 2

Object Lifetime and RAII

2.1 Rule 3 — Resource Ownership Must Be Explicit

Principle: Every resource must have exactly one clear, well-defined owner. Ownership determines:

- Who is responsible for releasing the resource
- When the resource is released
- Whether the resource can be transferred, shared, or borrowed

In Modern C++, **ownership is a semantic property that must be encoded in the type system**. Any ownership that exists only in comments or developer assumptions is a latent defect.
Resources include:

- Heap memory
- File descriptors

- Sockets
- Mutexes and locks
- Memory-mapped regions
- Kernel and OS handles

Bad Example (Implicit and Unsafe Ownership):

```
FILE* f = fopen("data.txt", "r");
```

This code introduces multiple hazards:

- Ownership is implicit and undocumented
- No exception safety
- Cleanup responsibility is manual
- Early returns can easily cause leaks
- Double-close is possible

Correct Modern C++ (Explicit RAII Ownership):

```
std::unique_ptr<FILE, decltype(&fclose)>
f(fopen("data.txt", "r"), &fclose);
```

This version guarantees:

- Single ownership
- Automatic cleanup on all exit paths
- Full exception safety

- No double-close
- Deterministic destruction

Preferred Modern Wrapper (Recommended in Production):

```
class File
{
public:
    explicit File(const char* path, const char* mode)
        : handle_(fopen(path, mode))
    {
        if (!handle_) throw std::runtime_error("File open failed");
    }

    ~File() noexcept
    {
        if (handle_) fclose(handle_);
    }

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

    File(File&& other) noexcept : handle_(other.handle_)
    {
        other.handle_ = nullptr;
    }

    File& operator=(File&& other) noexcept
    {
        if (this != &other)
        {
            if (handle_) fclose(handle_);
            handle_ = other.handle_;
```

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

    FILE* get() const noexcept { return handle_; }

private:
    FILE* handle_;
};
```

This wrapper provides:

- Explicit ownership model
- Move-only semantics
- Clear lifetime boundaries
- Strong exception safety
- Safe transfer of ownership

Impact on Performance and Safety:

- Eliminates resource leaks entirely
- Guarantees deterministic release timing
- Prevents double-free and use-after-free
- Improves reasoning about lifetime
- Enables aggressive compiler optimization

Common Mistakes in Real Projects:

- Using raw pointers for owning resources
- Mixing manual cleanup with RAII
- Returning raw owning pointers from functions
- Sharing ownership implicitly across modules
- Forgetting move semantics for RAII wrappers

Engineering Rule of Thumb: *If a resource is not owned by a type with a destructor, it is not owned safely.*

2.2 Rule 4 — Destructors Must Never Throw

Principle: Destructors participate in stack unwinding. If a destructor throws while another exception is already active, the program **must terminate**. This is mandated by the C++ language to preserve correctness.
Therefore:

- Destructors must be implicitly or explicitly `noexcept`
- All failure-prone operations must be isolated
- Error reporting must not rely on throwing from destructors

Bad Example (Catastrophic Termination):

```
~Logger()
{
    throw std::runtime_error("fail");
}
```

If this destructor executes during stack unwinding:

- `std::terminate()` is called immediately
- Program state becomes unrecoverable
- Resources leak catastrophically
- Logs are lost

Correct Modern C++ (Noexcept Destruction):

```
~Logger() noexcept
{
    try
    {
        flush();
    }
    catch (...)
    {
        // swallow all exceptions
    }
}
```

This guarantees:

- No termination during unwinding
- Exception safety at program shutdown
- Stable failure containment

Production-Grade Pattern (Two-Phase Cleanup):

```
class Logger
{
public:
    void close()
```

```
    {
        flush(); // may throw
        closed_ = true;
    }

    ~Logger() noexcept
    {
        if (!closed_)
        {
            try { flush(); } catch (...) {}
        }
    }

private:
    bool closed_ = false;
};
```

This model allows:

- Proper error reporting in `close()`
- Guaranteed safety in the destructor
- Explicit shutdown control

Impact on Performance and Safety:

- Prevents undefined termination paths
- Guarantees noexcept destruction for standard containers
- Enables stack unwinding optimizations
- Improves exception propagation reliability

- Prevents cascading failure at shutdown

Common Mistakes in Real Projects:

- Performing I/O in destructors without guards
- Throwing from cleanup code
- Relying on destructors for error reporting
- Forgetting that STL containers assume noexcept destructors
- Using exceptions inside global object destructors

Engineering Rule of Thumb: *If your destructor can fail, your design is already broken.*

Chapter 3

Memory and Ownership

3.1 Rule 5 — Never Use Raw `new` and `delete` in Application Code

Principle: Dynamic memory allocation must **always** be bound to automatic lifetime management through RAII. Any use of raw `new` and `delete` in application-level C++ code represents a:

- Loss of ownership clarity
- Breakdown of exception safety
- Violation of deterministic lifetime guarantees
- Direct reintroduction of C-style memory hazards

Raw allocation is only justifiable in:

- Custom allocators

- Low-level memory pools
- Embedded runtimes
- Kernel-level and freestanding environments

Bad Example (Manual Heap Ownership):

```
int* p = new int(42);
delete p;
```

This code introduces:

- No exception safety between allocation and delete
- Undefined behavior if delete is missed
- No ownership semantics in the type system
- Double-delete vulnerability
- Memory leak risks under early returns

Catastrophic Variant (Exception Leak):

```
int* p = new int(42);
throw std::runtime_error("fail");
delete p; // never executed
```

Correct Modern C++ (Exclusive Ownership via RAII):

```
auto p = std::make_unique<int>(42);
```

This guarantees:

- Single ownership enforced by type system

- Automatic cleanup on every control path
- Full exception safety
- Zero double-free possibility
- Move-only semantics for safe transfer

Correct Modern C++ (Container Ownership):

```
std::vector<int> v = {42};
```

This is superior to heap allocation because:

- Stack-allocated control object
- Contiguous memory
- Cache-friendly layout
- Automatic destruction

Correct Modern C++ (Polymorphic Ownership):

```
std::unique_ptr<Base> obj = std::make_unique<Derived>();
```

Impact on Performance and Safety:

- Eliminates memory leaks
- Prevents double-delete and use-after-free
- Enables move-based optimizations
- Improves cache behavior
- Enables whole-program lifetime analysis

Common Mistakes in Real Projects:

- Calling `delete` on memory owned by `unique_ptr`
- Mixing raw pointers with smart pointer ownership
- Returning raw pointers from factory functions
- Using `new` for STL containers
- Allocating for performance-critical paths without profiling

Engineering Rule of Thumb: *If `delete` appears in your application code, your ownership model is broken.*

3.2 Rule 6 — Use `shared_ptr` Only for Shared Ownership

Principle: `std::shared_ptr` represents **distributed ownership**. This is fundamentally different from:

- Exclusive ownership (`unique_ptr`)
- Borrowed access (raw pointer or reference)

Shared ownership introduces:

- Atomic reference counting
- Non-deterministic destruction timing
- Higher memory overhead
- Higher cache pressure
- Potential reference cycles

It must therefore be used **sparingly and intentionally**.

Bad Example (Control Block Duplication):

```
std::shared_ptr<int> p(new int(5));
```

This is flawed because:

- Two allocations occur (object + control block)
- Memory fragmentation increases
- Cache locality degrades
- More TLB pressure is introduced

Correct Modern C++ (Single Allocation):

```
auto p = std::make_shared<int>(5);
```

This guarantees:

- Single contiguous allocation
- Fewer cache misses
- Reduced memory overhead
- Faster reference count updates

Correct Usage (True Shared Ownership):

```
struct Node
{
    std::shared_ptr<Node> next;
};

auto a = std::make_shared<Node>();
auto b = std::make_shared<Node>();
a->next = b;
```

Dangerous Pattern (Reference Cycle):

```
struct A { std::shared_ptr<B> b; };
struct B { std::shared_ptr<A> a; };
```

This leaks permanently because:

- Reference counts never reach zero
- Destructors are never called
- Memory becomes unreachable but not freed

Correct Cycle-Breaking Strategy:

```
struct A { std::shared_ptr<B> b; };
struct B { std::weak_ptr<A> a; };
```

Impact on Performance and Safety:

- Atomic reference counting is costly
- Prevents deterministic destruction
- Inhibits compiler alias analysis
- Introduces hidden synchronization
- Can cause unbounded memory growth

Common Mistakes in Real Projects:

- Using `shared_ptr` as default pointer type
- Using it to avoid thinking about ownership
- Creating accidental reference cycles

- Passing `shared_ptr` by value excessively
- Using `shared_ptr` in performance-critical hot paths

Engineering Rule of Thumb: *If ownership is not truly shared by design,* `shared_ptr` *is the wrong tool.*

Chapter 4

Move Semantics and Value Categories

4.1 Rule 7 — Always Implement the Rule of Five if You Manage Resources

Principle: Any type that **directly manages a resource** (heap memory, file handles, sockets, mutexes, GPU buffers, memory-mapped files, OS descriptors) must explicitly control:

- Destruction
- Copy construction
- Copy assignment
- Move construction
- Move assignment

This is known as the **Rule of Five**. Violating it results in:

- Double-free

- Use-after-free
- Resource leaks
- Invalid moved-from states
- Silent data corruption

Bad Example (Implicitly Broken Ownership):

```
struct Buffer {
    int* p;
};
```

This type is **catastrophically unsafe** because:

- Compiler generates a shallow copy constructor
- Compiler generates a shallow copy assignment
- Two objects now believe they own the same memory
- Destruction triggers double-free

Fatal Usage Scenario:

```
Buffer a;
a.p = new int[1024];

Buffer b = a;      // shallow copy
delete[] a.p;      // first free
delete[] b.p;      // second free -> undefined behavior
```

Correct Modern C++ (Explicit Move-Only Ownership):

```
struct Buffer {
    int* p = nullptr;

    Buffer() = default;

    explicit Buffer(std::size_t n)
        : p(new int[n]) {}

    ~Buffer() {
        delete[] p;
    }

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

    Buffer(Buffer&& other) noexcept
        : p(other.p)
    {
        other.p = nullptr;
    }

    Buffer& operator=(Buffer&& other) noexcept
    {
        if (this != &other) {
            delete[] p;
            p = other.p;
            other.p = nullptr;
        }
        return *this;
    }
};
```

This implementation guarantees:

- Single, unambiguous ownership
- Safe transfer of resources via moves
- No accidental duplication
- Correct destruction semantics
- Exception safety

Key Engineering Properties:

- Copy is forbidden
- Move is cheap (pointer swap)
- Destruction is deterministic
- Ownership transfer is explicit

Correct Usage (Move into Containers):

```
std::vector<Buffer> buffers;
buffers.emplace_back(1024); // move-constructed into container
```

Correct Usage (Return by Value with Move Elision):

```
Buffer create_buffer()
{
    Buffer tmp(2048);
    return tmp; // guaranteed NRVO or move
}
```

Why Copy Must Be Disabled:

- Copying raw resources is ambiguous

- Deep copy is expensive and often incorrect
- Shallow copy is almost always wrong
- Ownership cannot be duplicated safely

When Copy *Should* Exist: Only if you explicitly implement:

- Deep copy
- Independent resource duplication
- Separate allocation per instance

Example of Safe Deep Copy:

```
struct Image {
    std::size_t size;
    std::uint8_t* data;

    Image(const Image& other)
        : size(other.size),
          data(new std::uint8_t[other.size])
    {
        std::memcpy(data, other.data, size);
    }
};
```

Performance Implications:

- Move is O(1)
- Copy is O(n)
- Correct move enables:

  - Zero-copy returns
  - Fast container reallocation
  - Efficient thread handoff

- Incorrect ownership disables:
  - Vector growth
  - RVO/NRVO
  - Lock-free data transfer

Interaction with Value Categories:

- Lvalues invoke copy if enabled
- Rvalues invoke move if available
- Temporaries always prefer move
- const objects disable move

Subtle but Critical Rule:

A const object can never be moved from.

Broken Pattern (Move Disabled by const):

```
const Buffer buf = create_buffer(); // move forbidden
```

Common Mistakes in Real Projects:

- Forgetting to delete copy operations
- Implementing move constructor but forgetting move assignment

- Not nulling moved-from pointers
- Throwing from move operations
- Making move operations non-`noexcept` (breaking containers)
- Using raw owning pointers instead of RAII wrappers

Engineering Rule of Thumb:

> *If your type owns something, the compiler must never guess how it is copied or moved.*

Advanced Production Insight:

- Types without a `noexcept` move constructor:
  - Break `std::vector` reallocation optimization
  - Force expensive fallback copies
  - Cause performance cliffs in hot paths
- Well-implemented move semantics are a **hard requirement** for:
  - Lock-free queues
  - Message-passing systems
  - Actor models
  - Job systems
  - High-frequency trading engines

Chapter 5

Error Handling and Contracts

5.1 Rule 8 — Exceptions Represent Failure to Fulfill a Contract

Principle: Exceptions in Modern C++ exist to represent a **violation of a function's contract**, not as a mechanism for ordinary control flow, iteration, or state transitions. A function either:

- Successfully fulfills its documented contract, or
- Fails to do so and reports that failure via an exception

Using exceptions as an alternative control path corrupts:

- Program logic
- Performance predictability
- Compiler optimization assumptions
- Code readability and maintainability

Formal Contract Model:

- **Preconditions:** What must be true before the function executes
- **Postconditions:** What the function guarantees on success
- **Invariants:** What remains true throughout execution

An exception means:

> *The function was unable to satisfy its postconditions despite valid invocation.*

Bad Example (Exceptions as Control Flow):

```
for (...)
{
    if (done())
        throw EndOfLoop{};
}
```

This is incorrect because:

- The loop does not describe a failure
- Normal termination is incorrectly modeled as exceptional behavior
- Stack unwinding occurs repeatedly
- Performance becomes non-deterministic
- Debugging becomes misleading

Catastrophic Variant (Hidden Hot-Path Cost):

```
while (true)
{
    try {
        work();
    } catch (...) {
        continue;
    }
}
```

This code:

- Defeats branch prediction
- Forces exception table lookups
- Triggers stack unwinding machinery
- Breaks zero-cost exception assumptions

Correct Modern C++ (Contract Enforcement):

```
if (!valid())
    throw std::logic_error("Invalid object state");
```

Here:

- The exception communicates a violated invariant
- The failure is not a normal execution path
- Error semantics are preserved

Correct Usage for Runtime Failures:

```
if (!file.is_open())
    throw std::runtime_error("File open failed");
```

Correct Usage for Programmer Errors:

```
if (ptr == nullptr)
    throw std::logic_error("Null pointer dereference risk");
```

Exception Categories and Their Meaning:

- `std::logic_error`
  - Violated preconditions
  - Broken invariants
  - Programmer errors
- `std::runtime_error`
  - I/O failures
  - Resource exhaustion
  - External system failure

Engineering Rule:

> *If recovery is expected and routine, do not use exceptions.*

Correct Alternative for Expected Failure (Value-Based Error Handling):

```
std::optional<int> parse(std::string_view s);

auto value = parse(input);
if (!value)
{
    // expected failure path
}
```

Correct Alternative with Diagnostics:

```
std::expected<int, ParseError> parse(std::string_view s);
```

This separates:

- **Expected failure** → return value
- **Unexpected failure** → exception

Exception Safety Guarantees:

- **No-throw guarantee:** Operation never throws
- **Strong guarantee:** State is unchanged on failure
- **Basic guarantee:** Invariants preserved, state may change

Example (Strong Guarantee via Copy-and-Swap):

```
void set_value(Data d)
{
    Data temp = d;
    swap(data_, temp);
}
```

Performance Implications:

- Zero-cost only applies to the *non-throwing* path
- Throwing is extremely expensive
- Stack unwinding touches:
  - Destructors
  - Personality routines
  - Landing pads

- Exceptions defeat:
  - Inlining
  - Vectorization
  - Control-flow flattening

Critical Engineering Observation:

Exceptions are free only when you do not use them.

Interaction with RAII:

- RAII ensures:
  - No resource leaks under exception
  - Deterministic cleanup
  - Invariant preservation
- Without RAII, exceptions become memory corruption accelerators

Correct RAII + Exception Example:

```
std::lock_guard<std::mutex> lock(m);

if (!condition)
    throw std::runtime_error("Failure under protection");
```

Common Mistakes in Real Projects:

- Using exceptions to break loops
- Using exceptions for normal I/O conditions
- Catching by value instead of reference

- Catching `...` and ignoring the error
- Throwing from destructors
- Throwing non-exception types
- Using exceptions as error codes

Professional Engineering Notes:

- In low-latency systems, exceptions are typically disabled entirely
- In safety-critical systems, exceptions are often banned by policy
- In financial systems, exceptions must never occur in hot paths
- In libraries, exception behavior is part of the ABI contract

Final Engineering Law:

> *Exceptions are a correctness tool, not a flow-control tool.*

Chapter 6

Concurrency and Atomics

6.1 Rule 9 — Data Race Freedom Is a Design Property

Principle: Data race freedom is not something you "add" via locks or atomics after writing code. It is a **design property** that must be explicitly established before introducing any concurrency mechanism.
A concurrent system is correct when:

- No two threads access the same memory location simultaneously unless at least one access is protected
- The ordering of operations is consistent with the program's synchronization model
- Shared state is minimized, well-defined, and controlled

Concurrency bugs are not fixed with mutexes or atomics—they are prevented by design.
Bad Example (Unprotected Shared State):

```
int shared;
shared++;     // undefined behavior under concurrency
```

Issues:

- Read-modify-write is not atomic
- Two threads incrementing may interleave
- Lost updates occur (write tearing)
- Undefined behavior per the C++ memory model

Concrete Race Example:

```
Thread 1: shared = shared + 1;
Thread 2: shared = shared + 1;
```

Both threads may read the same old value, write back the same incremented value, and lose one update.

Correct Modern C++ (Atomic Operation):

```
std::atomic<int> shared{0};
shared.fetch_add(1, std::memory_order_relaxed);
```

This ensures:

- No undefined behavior
- Correct RMW (read-modify-write) semantics
- No lost updates
- Efficient lock-free execution

More Explicit Form:

```
shared.fetch_add(1, std::memory_order_acq_rel);
```

Deeper Understanding: What Is a Data Race? A data race occurs when:

1. Two or more threads access the same memory location
2. At least one access is a write
3. There is no synchronization ordering between them

If all three conditions hold → **undefined behavior**. This is one of the most catastrophic forms of UB because:

- Compilers optimize under the assumption that data races do not exist
- Out-of-thin-air values may appear
- Writes may be silently dropped
- Reads may observe impossible states

Common Incorrect "Fix": Adding a Mutex After the Fact

```
int shared;
std::mutex m;

void inc() {
    std::lock_guard<std::mutex> lock(m);
    shared++;
}
```

This is often mistaken as a complete fix. It is not, because:

- Other code may still touch `shared` without locking
- Lifetime and aliasing of `shared` is uncontrolled
- Readers may bypass the mutex (undefined behavior)
- Mutex use requires consistent ownership discipline

A lock is not protection unless **every single access** uses the same lock.

Correct Design: Make the Shared State Atomic From Day One

```
class Counter {
public:
    void increment() noexcept {
        value_.fetch_add(1, std::memory_order_relaxed);
    }

    int get() const noexcept {
        return value_.load(std::memory_order_relaxed);
    }

private:
    std::atomic<int> value_{0};
};
```

This design:

- Encapsulates the shared state
- Prevents accidental non-atomic access
- Defines a clear synchronization boundary
- Enforces thread safety through the type system

Atomic Operations and Memory Ordering:

C++ atomics guarantee:

- Atomicity (no tearing)
- Ordering constraints
- Lock-free or wait-free behavior when possible

Memory orderings include:

- `memory_order_relaxed` — no ordering constraints
- `memory_order_acquire` — no reads after may move before
- `memory_order_release` — no writes before may move after
- `memory_order_acq_rel` — combined acquire/release
- `memory_order_seq_cst` — global total ordering

Most high-performance counters use `relaxed` ordering.

Why Locks Do Not Fix Broken Designs

Locks only serialize access; they do not:

- Establish ownership discipline
- Prevent aliasing of shared resources
- Fix broken invariants
- Solve design-level race conditions
- Prevent misuse by other threads

A correct design minimizes shared mutable state. Locks then become a mechanism of refinement—not a patch.

Impact on Safety and Performance:

- Eliminates undefined behavior from data races
- Makes program behavior predictable

- Enables compiler optimizations
- Allows safe lock-free programming
- Reduces need for heavy synchronization

Performance Footnote: Data races generate UB, which allows compilers to:

- Reorder instructions arbitrarily
- Eliminate reads or writes entirely
- Introduce optimizations that break multithreading

Thus, correctness must be established before parallelism.

Common Mistakes in Real Projects:

- “Protecting” only writes and not reads
- Using atomics without understanding memory ordering
- Accessing shared variables both atomically and non-atomically
- Assuming mutexes automatically ensure correctness
- Creating implicit sharing via reference captures in lambdas
- Returning references to shared mutable state

Professional Engineering Law:

> *If a variable is accessed by multiple threads, its thread-safety must be encoded in its type—not in comments.*

Chapter 7

Performance Engineering

7.1 Rule 10 — Measure Before Optimizing

Principle: *Optimization without measurement is speculation.* Any performance decision not backed by empirical data is a guess that risks:

- Optimizing the wrong code path
- Regressing overall performance
- Increasing complexity without benefit
- Introducing subtle correctness bugs

Modern C++ performance engineering is a **data-driven discipline**. The only valid optimization workflow is:

1. Measure
2. Analyze

3. Identify the bottleneck
4. Optimize
5. Re-measure

Skipping any step invalidates the result.

Bad Example (Myth-Based Micro-Optimization):

```
register int x;
```

This is meaningless in Modern C++ because:

- `register` is deprecated and ignored by modern compilers
- Register allocation is performed by the optimizer
- The programmer has no visibility into actual register usage
- It provides zero measurable performance benefit

Another Bad Example (Blind Manual Inlining):

```
inline int add(int a, int b) { return a + b; }
```

Modern compilers:

- Decide inlining based on cost models
- Inline across translation units (LTO)
- Inline under profile-guided optimization (PGO)

Manual `inline` is rarely relevant.

Correct Modern C++ (Establishing a Measurement Baseline):

```
auto t1 = std::chrono::steady_clock::now();

// code under measurement

auto t2 = std::chrono::steady_clock::now();
auto dt = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1);
```

This provides:

- High-resolution timing
- Monotonic clock source
- Reproducible measurements
- Platform-independent profiling

Important Rule: *Wall-clock time alone is never sufficient for deep optimization.*
You must measure:

- Instruction count
- Cache misses
- Branch mispredictions
- Memory bandwidth
- Allocation frequency

Correct Measurement at Scale (Statistical Profiling):

```
for (std::size_t i = 0; i < 10'000'000; ++i)
    work();
```

Why repetition is required:

- Warm up CPU caches
- Train branch predictors
- Stabilize dynamic frequency scaling
- Average out measurement noise

Single-shot measurements are statistically meaningless.

False Optimization vs Real Optimization

False Optimizations Include:

- Manual loop unrolling without profiling
- Premature bit-level tricks
- Hand-written pointer arithmetic over STL
- Avoiding function calls without evidence
- Rewriting readable code for assumed speed

Real Optimizations Are Typically:

- Algorithmic (O(n) $\rightarrow$ O(log n))
- Data layout changes (AoS $\rightarrow$ SoA)
- Cache locality improvements
- Allocation elimination
- Branch elimination

Engineering Fact: *Algorithmic improvement dominates micro-optimization by orders of magnitude.*

Measurement Tools by Optimization Layer

High-Level:

- Wall-clock timing
- Throughput measurement
- Latency distribution

Mid-Level:

- CPU sampling profilers
- Call graph analysis
- Allocation tracking

Low-Level:

- Hardware performance counters
- Cache miss analysis
- TLB pressure
- Branch predictor efficiency

Correct Microbenchmark Structure

```
void benchmark()
{
    warm_up();
```

```
    auto t1 = std::chrono::steady_clock::now();

    for (int i = 0; i < 10'000'000; ++i)
        work();

    auto t2 = std::chrono::steady_clock::now();
    report(t2 - t1);
}
```

Rules:

- Never benchmark cold code
- Never benchmark a single execution
- Never benchmark in debug mode
- Never trust one run

Impact on Performance Engineering Discipline:

- Prevents placebo optimizations
- Keeps performance work targeted
- Reduces code churn
- Improves engineering confidence
- Enables predictable scalability

Performance Myths Destroyed by Measurement

- “Pointers are always faster than references”
- “Manual memory management is always faster”

- “Exceptions are always slow”
- “Virtual functions are too expensive”
- “STL is slow”

Every one of these statements is **architecture-, workload-, and compiler-dependent**.
Optimization Ordering Law (Engineering Hierarchy):

1. Algorithm selection
2. Data layout
3. Memory access patterns
4. Allocation strategy
5. Concurrency structure
6. Instruction-level tuning

Optimizing step 6 before step 1 is engineering malpractice.
Common Mistakes in Real Projects:

- Optimizing before measuring
- Measuring only in debug builds
- Ignoring cache effects
- Assuming CPU time equals performance
- Focusing on micro-optimizations
- Trusting intuition over profiler output

- Optimizing cold paths

Professional Engineering Rule:

You are not allowed to optimize what you cannot measure.

Final Technical Law:

The profiler is always right, even when it contradicts your intuition.

Chapter 8

API and Library Design

8.1 Rule 11 — APIs Must Encode Ownership in the Type System

Principle: If ownership is unclear, the API is broken. Ownership must never be documented only in comments or external guidelines. It must be:

- **Statically enforced by the type system**
- **Unambiguous at the call site**
- **Impossible to misuse accidentally**

An API that does not encode ownership invites:

- Memory leaks
- Double-free
- Use-after-free

- Lifetime mismatches across modules
- ABI-level resource corruption

Fundamental Ownership Categories in Modern C++ APIs:

- **Exclusive ownership** → `std::unique_ptr<T>`
- **Shared ownership** → `std::shared_ptr<T>`
- **Borrowed access** → `T&`, `const T&`, `T*`
- **Optional presence** → `std::optional<T>`

The return type and parameter type must precisely communicate which category applies.

Bad Example (Ambiguous Ownership):

```
Widget* create();
```

This API is broken because the caller cannot know:

- Who owns the returned pointer
- Whether it must be deleted
- With which allocator it must be deleted
- Whether the pointer remains valid after the next call

This ambiguity leads to:

- Memory leaks
- Double delete
- Heap corruption across module boundaries

- Undefined behavior under exception paths

Catastrophic Usage Pattern:

```
Widget* w = create();
use(w);
delete w;        // Possibly wrong
```

The caller is forced to guess.

Correct Modern C++ (Exclusive Ownership Encoded):

```
std::unique_ptr<Widget> create();
```

This API enforces:

- The caller **must** take ownership
- Exactly one owner exists
- Destruction is deterministic
- Ownership transfer is explicit via move

Correct Usage:

```
auto w = create();    // ownership clearly transferred
use(*w);
```

There is no delete, no ambiguity, no contract violation possible.

Borrowed Access Must Never Transfer Ownership

Correct Borrowing API:

```
void render(const Widget& w);
```

This guarantees:

- No ownership transfer

- No lifetime responsibility
- Safe, non-null usage

Broken Borrowing API:

```
void render(const Widget* w);
```

This introduces:

- Possible null pointer
- No lifetime guarantee
- Hidden ownership ambiguity

Shared Ownership Must Be Explicit and Rare
Correct Shared Ownership API:

```
std::shared_ptr<Widget> acquire();
```

This indicates:

- The caller participates in reference counting
- Lifetime is non-deterministic
- Destruction occurs at last release

Warning: Shared ownership is an architectural decision, not a convenience feature.
Returning References: Lifetime Must Be Stable
Correct:

```
const Widget& get() const;
```

Valid only when:

- The referenced object outlives the caller
- The reference is not stored long-term
- The object is not invalidated by later API calls

Broken API:

```
const Widget& create();
```

This creates:

- Immediate dangling references
- Undefined behavior on first use

Factory Function Ownership Matrix

Return Type	Ownership	Destruction
`T`	Value ownership	Automatic
`std::unique_ptr<T>`	Exclusive	Deterministic
`std::shared_ptr<T>`	Shared	Last-release
`T&`	Borrowed	External
`T*`	Unspecified	Undefined

Engineering Law: *If your API returns a raw pointer, your design is incomplete.*

Ownership in Function Parameters

Correct Patterns:

```
void take(std::unique_ptr<Widget> w);    // consumes ownership
void observe(const Widget& w);          // borrows
void share(std::shared_ptr<Widget> w); // shares
```

Each signature enforces:

- Transfer semantics
- Lifetime responsibility
- Thread-safety implications

Broken Pattern:

```
void take(Widget* w);
```

This encodes nothing and allows every possible misuse.

ABI and Module Boundary Implications

At shared-library boundaries:

- Raw pointers hide allocator compatibility
- Different CRTs may manage memory differently
- Deallocating across modules can corrupt heaps

Correct ABI-safe Pattern:

```
std::unique_ptr<Widget, Deleter> create();
```

The deleter ensures:

- Deallocation occurs in the correct module
- Allocator symmetry is preserved

Impact on Safety and Maintainability:

- Eliminates lifetime ambiguity
- Makes misuse impossible by construction
- Prevents cross-module heap corruption

- Improves static analysis accuracy
- Enables whole-program ownership reasoning

Common Mistakes in Real Projects:

- Returning raw pointers from factories
- Using raw pointers to mean “sometimes owning”
- Using `shared_ptr` to avoid design work
- Passing owning pointers through multiple layers
- Storing borrowed references past their valid lifetime

Professional Engineering Law:

If ownership is not visible in the type, it does not exist in the design.

Final API Design Rule:

APIs must make invalid lifetime usage unrepresentable.

Chapter 9

Build Systems and Toolchains

9.1 Rule 12 — Treat Warnings as Errors

Principle: Every compiler warning represents a **latent defect**, a **broken assumption**, or a **future bug waiting to surface**. A professional C++ codebase must enforce a strict policy:

> *No warning is acceptable in production-quality code.*

Warnings are not suggestions. They are:

- Early indicators of undefined behavior
- Signals of type system violations
- Alerts about dangerous implicit conversions
- Evidence of broken control flow or lifetime
- Symptoms of non-portable constructs

Allowing warnings into the build pipeline normalizes technical debt at the compiler level.

Bad Example (Warnings Enabled but Ignored):

```
-Wall
```

This configuration is fundamentally insufficient because:

- `-Wall` does *not* mean "all warnings"
- Many critical warnings are not included
- Warnings do not fail the build
- Broken code can be merged silently

This policy communicates:

> *"We acknowledge defects, but we tolerate them."*

Correct Practice (Professional Enforcement):

```
-Wall -Wextra -Werror
```

This enforces:

- Maximum diagnostic coverage
- Zero tolerance for suspicious constructs
- Immediate developer feedback
- Defect elimination at commit time

Extended Professional Sets (Compiler-Dependent):

For GCC / Clang:

```
-Wall -Wextra -Wpedantic -Werror
```

For MSVC:

```
/W4 /WX
```

These configurations:

- Enforce strict language conformance
- Reject non-standard extensions
- Expose narrowing conversions
- Detect shadowed variables

Why Warnings Must Be Fatal

Allowing warnings means:

- Implicit conversions silently truncate data
- Signed/unsigned mismatches corrupt logic
- Uninitialized reads invoke undefined behavior
- Missing virtual destructors cause memory leaks
- Incorrect overrides introduce slicing bugs

Engineering Reality:

> *Every critical production crash begins life as a compiler warning.*

Example: Narrowing Conversion Bug

Warning Ignored:

```
int size = buffer.size();
```

If `buffer.size()` is `std::size_t`, this may:

- Truncate large values
- Break 64-bit builds
- Corrupt memory bounds

Correct Code:

```
std::size_t size = buffer.size();
```

With `-Werror`, this mistake is caught immediately.

Example: Missing Virtual Destructor

Warning Ignored:

```
struct Base {
    virtual void f();
};
```

Deleting derived objects via base pointer:

- Leaks memory
- Skips destructor execution
- Breaks invariants

Correct Code:

```
struct Base {
    virtual ~Base() = default;
};
```

Warnings and Undefined Behavior

Many compiler warnings are not “style” warnings. They are direct UB detectors:

- Use of uninitialized variables
- Out-of-bounds array access
- Invalid pointer arithmetic
- Dangling reference escapes
- Non-returning control paths

Allowing any of the above to compile is a correctness failure, not a stylistic choice.

Interaction with Static Analysis

Treating warnings as errors integrates naturally with:

- Static analyzers
- Sanitizers
- Continuous integration pipelines

This enables:

- Zero-warning baselines
- Regression detection at commit time
- Automated gatekeeping of code quality

Warning Suppression Is a Surgical Operation

Correct Local Suppression:

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wsign-conversion"
// justified narrow conversion
#pragma GCC diagnostic pop
```

Rules for suppression:

- Only suppress locally
- Always document the reason
- Never suppress at project scope
- Never suppress in headers

Forbidden Pattern:

```
-Wno-everything
```

This disables the compiler as a diagnostic instrument entirely.

Build System Enforcement

In professional build systems:

- Warning-as-error is enforced in:
  - CI pipelines
  - Release builds
  - Pull request validation
- No developer may bypass it without formal justification

CMake Example:

```
add_compile_options(-Wall -Wextra -Werror)
```

MSVC CMake Example:

```
add_compile_options(/W4 /WX)
```

Impact on Safety, Stability, and Maintainability:

- Prevents undefined behavior from entering the codebase
- Enforces strict type correctness
- Improves long-term maintainability
- Reduces debugging time dramatically
- Increases trust in refactoring

Common Mistakes in Real Projects:

- Suppressing warnings globally
- Ignoring warnings in third-party code
- Allowing “temporary” warnings to accumulate
- Downgrading warning levels for convenience
- Treating warnings as stylistic noise

Professional Engineering Law:

If your code builds with warnings, it is not finished.

Final Toolchain Rule:

A warning-free build is the minimum standard of professional C++ engineering.

Conclusion

Modern C++ is not about language features. It is about discipline, correctness, and engineering responsibility.
C++ today is a **fully mature systems engineering language**. It no longer tolerates:

- Accidental ownership
- Implicit lifetime assumptions
- Uncontrolled concurrency
- Undefined behavior as an engineering strategy
- Performance myths driven by intuition instead of measurement

Every rule in this booklet exists because **thousands of production systems failed without it**. These are not stylistic preferences. They are **hard-learned survival laws** of real-world software engineering.

Modern C++ Is a Contract-Based Engineering Discipline

Modern C++ enforces:

- **Explicit ownership** through the type system

- **Deterministic lifetime** through RAII
- **Defined error semantics** through contracts and exceptions
- **Predictable concurrency** through atomics and memory ordering
- **Measurable performance** through profiling, not guessing
- **Correct-by-construction APIs** that make misuse impossible
- **Toolchain-enforced correctness** through warnings-as-errors

C++ does not protect you by default. It gives you the tools to build **hard guarantees**—or to destroy yourself if you ignore them.

What Separates Professionals from Amateurs

The difference is not syntax knowledge. It is **behavior under complexity**:

- Professionals:
  - Design ownership before writing code
  - Encode invariants in types
  - Measure before optimizing
  - Make invalid states unrepresentable
  - Treat warnings as failures
  - Respect the memory model
  - Build APIs that resist misuse
- Amateurs:

  - Rely on intuition for performance
  - Use shared ownership by default
  - Patch race conditions with locks
  - Use exceptions for flow control
  - Suppress compiler diagnostics
  - Trust undefined behavior

C++ amplifies both excellence and negligence.

Modern C++ Is a Language of Accountability

In Modern C++:

- Every allocation is your responsibility
- Every thread interaction is your responsibility
- Every destructor is a promise
- Every API signature is a contract
- Every warning is a defect signal
- Every benchmark is a lie until it is repeated

There is no abstraction layer that absolves the engineer. C++ places the full cost of correctness directly on the author.
That is not a weakness. That is why C++ remains dominant in:

- Operating systems

- Game engines
- Financial trading platforms
- Embedded and real-time systems
- Compilers and runtimes
- High-performance scientific computing

The Real Meaning of "Best Practices"

Best practices are not trends. They are **converged knowledge**.
Each rule in this booklet represents:

- Years of production failures
- Postmortems on memory corruption
- Outages caused by race conditions
- Performance collapses caused by poor data layout
- Security vulnerabilities caused by lifetime misuse

Ignoring best practices does not make you a rebel. It makes you a future postmortem.

Final Engineering Law

> *C++ rewards engineers who think in lifetimes, ownership, memory, and concurrency—not in syntax and features.*

> *If you write Modern C++ as if it were a safer C, you will still get C's failures—just more subtly.*
>
> *The language does not guarantee correctness. You do.*

Closing Statement

This booklet does not teach you C++. It defines the **behavioral standard** required to use it professionally.
If you follow these rules consistently:

- Your systems will scale
- Your performance will be explainable
- Your failures will be diagnosable
- Your concurrency will be predictable
- Your APIs will resist misuse

And most importantly:

> **Your code will fail less in production.**

That is the only metric that ultimately matters.

References

- ISO/IEC 14882:2020 — *Programming Languages — C++*, International Organization for Standardization, 2020.
- ISO/IEC 14882:2023 — *Programming Languages — C++*, International Organization for Standardization, 2023.
- Bjarne Stroustrup, Herb Sutter, Andrei Alexandrescu et al., *C++ Core Guidelines*, ISO C++ Foundation, continuously updated (post-2020 editions).
- Herb Sutter, *The Free Lunch Is Over Revisited — Concurrency, Performance, and Modern C++*, CppCon, post-2020 editions.
- Andrei Alexandrescu, *Declarative Control Flow and Modern C++ Design*, CppCon, post-2020.
- Anthony Williams, *C++ Concurrency in Action*, 2nd Edition (Modern relevance through C++20/23), Manning Publications, actively used post-2020.
- Nicolai Josuttis, *C++ Move Semantics — The Complete Guide*, Updated Editions for C++20/23 Era.
- Nicolai Josuttis, *C++ Templates — The Complete Guide*, 2nd Edition (Industry standard post-2020 usage).

- Google Engineering Team, *Abseil C++ Tips of the Week*, Post-2020 Engineering Practices.
- LLVM Project, *LLVM Coding Standards and Modern C++ Practices*, Active post-2020 revisions.
- Microsoft C++ Team, *Modern C++ Best Practices for High-Performance Systems*, Visual C++ Blog, post-2020.
- Jason Turner, *C++ Weekly — Modern C++ Engineering Practices*, Post-2020 Content.
- Chandler Carruth, *Cache Locality, Data-Oriented Design, and Modern C++*, Post-2020 Professional Talks.
- ISO C++ Committee Papers (WG21), *C++20, C++23 Core Language & Library Evolution*, 2020–2024.