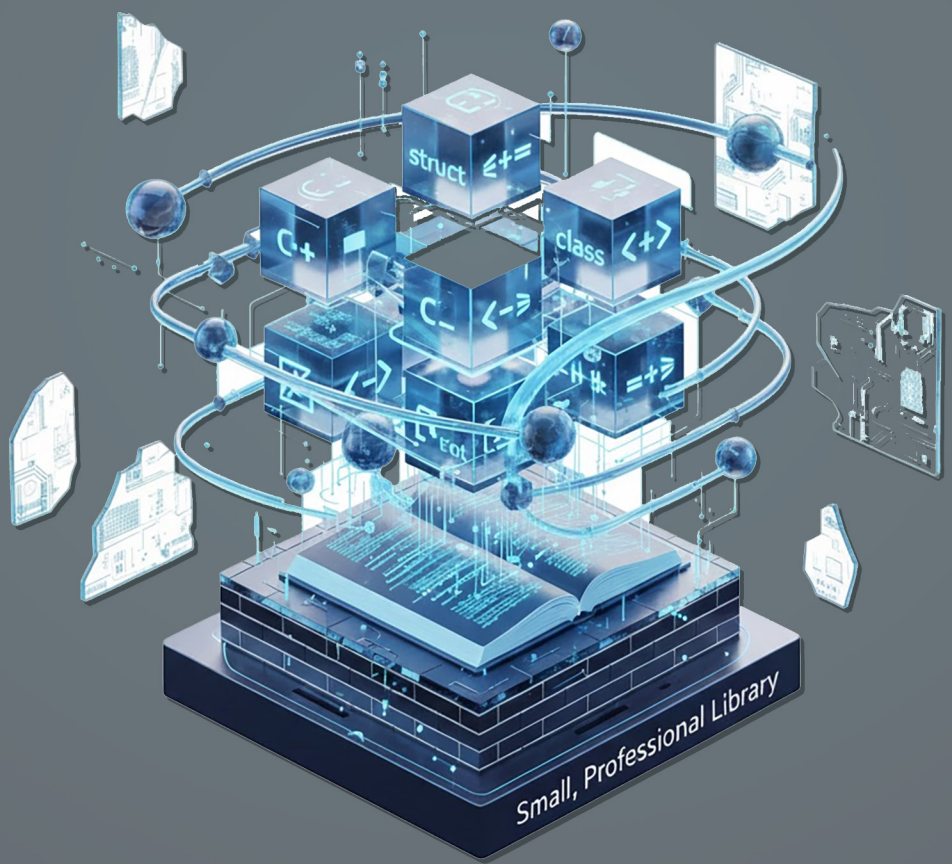
https://simplifycpp.org

API Design in Modern C++

How to Build a Small, Professional Library



Prepared By Ayman Alheraki

API Design in Modern C++

How to Build a Small, Professional Library

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Author’s Introduction

I have spent many years working with C++ in real production environments, where software longevity, correctness, and performance are not theoretical concerns, but daily engineering responsibilities.

Throughout this experience, one recurring problem has remained constant: poorly designed APIs cause more long-term damage than almost any other technical mistake. They increase maintenance cost, restrict evolution, introduce subtle bugs, and silently propagate design errors across entire systems.

This booklet was written to address that problem directly.

It is not a tutorial on C++ syntax, nor a catalog of language features. Instead, it focuses on how to think about API design in Modern C++, how to encode correctness into interfaces, and how to build small libraries that remain usable, stable, and trustworthy over time.

The principles presented here are drawn from real-world systems, modern C++ standards, and hard-earned engineering lessons.

Ayman Alheraki

Preface

Modern C++ provides one of the most expressive and powerful type systems available to systems programmers today. It enables precise control over resources, performance, and abstraction, often within the same codebase. However, power without discipline rarely produces clarity. More often, it produces complexity that is difficult to reason about, difficult to maintain, and costly to evolve.

A large number of C++ libraries fail not because their implementations are inefficient or incorrect, but because their interfaces are weak, ambiguous, or misleading. Ownership rules are unclear or undocumented. Error handling strategies are inconsistent or implicit. Invariants are assumed rather than enforced. As a result, misuse becomes easy, and correctness depends on discipline rather than design.

This mini-booklet was written to address that failure mode directly.

> To demonstrate how Modern C++ can be used to design small, professional APIs that encode correctness into their interfaces, resist misuse by construction, and remain stable as systems grow and evolve.

The scope of this booklet is intentionally limited. It does not attempt to catalog language features, compare frameworks, or promote stylistic preferences. It avoids platform-specific dependencies and domain-specific assumptions. Instead, it focuses on fundamental design principles that apply to any serious C++ library, regardless of its purpose or deployment environment.

The emphasis throughout is on correctness, clarity, and longevity. Examples are chosen to illustrate design trade-offs rather than cleverness. Abstractions are evaluated not by how flexible they appear, but by how well they prevent errors and communicate intent.
The reader is expected to have prior experience with C++ and to be familiar with modern language features. This text is written for engineers who care about long-term code quality, maintainability, and correctness, and who view API design as an engineering responsibility rather than an afterthought.

Chapter 1

What a Professional C++ API Really Is

A professional C++ API is not defined by syntax, fashion, or popularity. It is not measured by the number of features it exposes, nor by how convenient it appears at first glance. Instead, a professional API is defined by the **guarantees it provides** and the **failures it actively prevents**.

In real-world systems, APIs outlive implementations. Functions are rewritten, algorithms are optimized, and internal data structures change, but the public interface often remains fixed for years or even decades. Every design decision made at the API level therefore carries long-term consequences.

An API is a promise. Once published and adopted, it becomes a dependency embedded in build systems, deployment pipelines, and organizational processes. Changing it is rarely free. Breaking it can be catastrophic.

For this reason, professional API design must be approached as a form of engineering discipline rather than a matter of taste.

1.1 APIs as Engineering Contracts

A professional C++ API is an engineering contract between the library author and its users. This contract defines not only what the API does, but also what it *guarantees*, what it *forbids*, and what it *assumes*.
At a minimum, a serious API must make the following aspects explicit:

- Ownership and lifetime rules
- Error handling strategy
- Valid and invalid states
- Performance expectations
- Thread-safety guarantees

These properties are not optional documentation details. They are fundamental aspects of correctness.

1.1.1 Ownership and Lifetime Rules

In C++, ownership is inseparable from correctness. If an API does not clearly define who owns an object, how long it lives, and when it may be destroyed, undefined behavior becomes inevitable.

1.1.1.1 Bad Example: Implicit Ownership

```
Widget* create_widget();
void use_widget(Widget* w);
```

This API raises immediate questions: Who deletes the widget? Is ownership transferred? Can the pointer be null? Nothing in the type system answers these questions.

1.1.1.2 Professional Example: Explicit Ownership

```
std::unique_ptr<Widget> create_widget();
void use_widget(const Widget& w);
```

Ownership is explicit. Lifetime is deterministic. Misuse is mechanically prevented.
If users must guess whether they are responsible for deleting an object, the API has already failed.

1.1.2 Error Handling Strategy

Error handling is not an implementation detail. It is a visible and structural part of the API contract.
A professional API defines:

- How errors are reported
- Which failures are recoverable
- Which failures are fatal
- Whether errors are exceptional or expected

1.1.2.1 Bad Example: Implicit Error Signaling

```
File open_file(const std::string& path);
// returns invalid File on failure
```

This design silently propagates errors and invites misuse.

1.1.2.2 Professional Example: Explicit Errors

```
std::expected<File, FileError> open_file(const std::string& path);
```

Failure is explicit and impossible to ignore accidentally. The caller must acknowledge error conditions.
Mixing multiple error-handling models within the same API introduces ambiguity and misuse. A consistent strategy is mandatory.

1.1.3 Valid and Invalid States

An API must clearly define which states are valid and which are not. Even more importantly, it should make invalid states *unrepresentable whenever possible*.

1.1.3.1 Bad Example: Partially Initialized State

```
class Connection {
public:
    void connect(const std::string& addr);
    void send(const Data& d);
};
```

Nothing prevents calling `send()` before `connect()`.

1.1.3.2 Professional Example: Invariant-Enforced Construction

```
class Connection {
public:
    explicit Connection(const Address& addr);
    void send(const Data& d);
};
```

Invalid states cannot exist. The invariant is enforced by construction.

1.1.4 Performance Expectations

Performance is part of the contract. Even when exact complexity guarantees are not specified, an API must make clear whether operations are:

- Constant time
- Linear
- Potentially expensive
- Allocation-heavy

1.1.4.1 Bad Example: Hidden Cost

```
std::string get_data();
```

This signature hides allocations and copies.

1.1.4.2 Professional Example: Cost Visibility

```
std::string get_data();                 // owning, allocation
std::string_view view_data() noexcept;  // non-owning, cheap
```

Users can reason about cost without inspecting the implementation.

1.1.5 Thread-Safety Guarantees

Thread safety must never be assumed.

1.1.5.1 Ambiguous Example

```
class Cache {
public:
    void insert(Key k, Value v);
    Value get(Key k) const;
};
```

Is this thread-safe? Thread-compatible? Not thread-safe?

1.1.5.2 Professional Documentation via Design

```
// Not thread-safe
class Cache {
public:
    void insert(Key k, Value v);
    Value get(Key k) const;
};
```

Or by construction:

```
class ThreadSafeCache {
public:
    void insert(Key k, Value v);
    Value get(Key k) const;
private:
    mutable std::mutex m_;
};
```

Ambiguity is eliminated.

1.1.6 Why Enforcement Matters

If these rules are expressed only in documentation, they will eventually be violated. Documentation can be ignored, misunderstood, or forgotten. The C++ type system, however, cannot be negotiated with.
Professional APIs rely on types, constructors, and language mechanisms to enforce contracts mechanically rather than socially.

1.2 Why Modern C++ Changes API Design

Modern C++ fundamentally changes how APIs can and should be designed. Earlier C++ standards forced developers to rely heavily on conventions, comments, and discipline. Modern C++ provides language-level tools that allow correctness to be encoded directly into interfaces.
Among the most important of these tools are:

- RAII as a universal ownership model
- Strong types instead of informal conventions
- Compile-time enforcement using concepts
- Deterministic and explicit resource management

These features are not syntactic sugar. They are correctness mechanisms.

1.2.1 RAII as a Universal Ownership Model

1.2.1.1 Legacy Pattern

```
FILE* f = fopen("data.txt", "r");
// error paths forget fclose
```

1.2.1.2 RAII-Based Design

```
std::ifstream file("data.txt");
// automatically closed
```

RAII eliminates entire classes of bugs:

- Memory leaks
- Double frees
- Resource starvation
- Error-path cleanup failures

1.2.2 Strong Types Instead of Conventions

1.2.2.1 Weak Convention

```
void connect(int port); // what is valid?
```

1.2.2.2 Strong Type

```
struct Port {
    explicit Port(uint16_t value);
};

void connect(Port port);
```

Semantic correctness is enforced at compile time.

1.2.3 Compile-Time Enforcement with Concepts

1.2.3.1 Unconstrained Template

```
template <typename T>
void save(const T& t) {
    t.serialize();
}
```

1.2.3.2 Concept-Constrained Contract

```
template <typename T>
concept Serializable = requires(T t) {
    t.serialize();
};

template <Serializable T>
void save(const T& t) {
    t.serialize();
}
```

Invalid types are rejected immediately with clear diagnostics.

1.2.4 Deterministic Resource Management

```
class Buffer {
public:
    Buffer(size_t size);
    ~Buffer(); // deterministic release
};
```

Users always know when resources are acquired and released. Hidden allocations and implicit global state are avoided.

1.2.5 Correctness as the Primary Goal

The defining shift in Modern C++ API design is philosophical. Correctness is no longer an aspiration; it is a design requirement.

Convenience is secondary. Flexibility is constrained. Misuse is treated as a design failure.

Modern C++ provides the tools necessary to design APIs that are robust, predictable, and difficult to abuse. Using these tools is not optional for professional-grade libraries.

Chapter 2

Scoping a Small Library Correctly

A small library is not a weak library. It is a focused library with a deliberately narrow responsibility and a clearly defined problem domain.
In professional software systems, size is not a measure of value. Stability, clarity, and predictability are. Many of the most successful C++ libraries in long-lived systems are small, not because they lack ambition, but because they actively resist uncontrolled growth.
A well-scoped library provides fewer features, but stronger guarantees. It does not attempt to anticipate every possible use case. Instead, it solves one problem exceptionally well and integrates cleanly with surrounding systems.

2.1 Single Responsibility at Library Level

The Single Responsibility Principle applies not only to classes and functions, but also to libraries as architectural units.
A professional library should solve:

- One clearly defined domain problem

- Using one consistent mental model

This mental model must remain stable even as the implementation evolves.

2.1.1 Advanced Example: Domain Leakage

Consider a library originally designed to parse configuration files:

```
namespace config {
    Config load(const Path& path);
}
```

Over time, convenience features are added:

```
namespace config {
    Config load(const Path& path);
    void watch_for_changes(const Path& path);
    void reload_on_signal(int signal);
    void export_to_json(const Config&);
}
```

The library has now leaked into:

- Filesystem monitoring
- Signal handling
- Serialization formats

Each addition introduces new invariants, new failure modes, and new platform-specific behavior.

2.1.2 Professional Decomposition

A disciplined design decomposes responsibilities:

```
namespace config {
    Config load(const Path& path);
}

namespace config_watch {
    Watcher watch(const Path& path);
}

namespace config_export {
    Json to_json(const Config&);
}
```

Each library has a single mental model and a stable scope.

2.1.3 Advanced Anti-Pattern: Semantic Overloading

A common symptom of scope creep is semantic overloading:

```
Result read(const Path& p, Mode m);
```

Where `Mode` may control:

- Text vs binary
- Blocking vs async
- Validation on/off
- Caching behavior

This design multiplies states combinatorially.

2.1.4 Professional Alternative: Semantic Separation

```
TextFile read_text(const Path&);
BinaryFile read_binary(const Path&);
ValidatedFile read_validated(const Path&);
```

Each function has one responsibility and one meaning.

2.1.5 The Cost of Feature Accumulation

Feature accumulation is the primary cause of API decay. Each new feature introduces:

- New states
- New interactions
- New error paths
- New performance characteristics

These costs compound over time.

2.1.6 Advanced Example: Flag Explosion

```
Image load_image(const Path& p,
                 bool cache,
                 bool async,
                 bool validate,
                 bool convert_color);
```

This API is effectively undocumented. Valid combinations are unclear and unenforceable.

2.1.7 Professional Design Using Types

```
Cached<Image> load_cached_image(const Path&);
Async<Image> load_image_async(const Path&);
Validated<Image> load_validated_image(const Path&);
```

Types replace flags. Invalid combinations are unrepresentable.

2.1.8 Focus as a Design Constraint

Professional library authors treat scope as a hard constraint. Features that do not reinforce the core responsibility are rejected, even if they are easy to implement.
This discipline results in:

- Smaller, more stable interfaces
- Clearer invariants
- Easier testing
- Predictable evolution

A focused library earns trust because it behaves consistently.

2.2 Explicit Non-Goals

One of the most overlooked aspects of API design is the explicit declaration of non-goals. Professional APIs state clearly what they will *never* support. This is not a weakness; it is a form of precision.

2.2.1 Advanced Example: Implicit Assumptions

```
auto db = open_database("data.db");
db.query("SELECT * FROM users");
```

Without non-goals, users may assume:

- Thread safety
- Connection pooling
- Automatic retries
- Transaction management

Supporting these assumptions later may be impossible.

2.2.2 Explicit Non-Goals as Part of the Contract

```
// Non-goals:
// - Not thread-safe
// - No connection pooling
// - No implicit transactions
class Database {
public:
    explicit Database(const Path& file);
    Result query(const Sql&);
};
```

The design boundary is now explicit and defensible.

2.2.3 Non-Goals as Stability Mechanisms

Explicit non-goals act as guardrails. They prevent accidental expansion and provide a clear rationale for rejecting feature requests that compromise design integrity.

Chapter 3

Ownership and RAII as API Foundations

Ownership must never be implicit in a C++ API. Any ambiguity regarding ownership and lifetime is a direct path to undefined behavior.
C++ gives developers manual control over resources. Professional APIs encode this control explicitly and safely.

3.1 Forbidden Ownership Patterns

The following pattern is fundamentally flawed:

```
Resource* create();
void destroy(Resource*);
```

This API depends entirely on caller discipline.

3.1.1 Advanced Failure Scenario: Cross-Module Ownership

```
Resource* r = create();
// passed across shared library boundary
```

```
destroy(r); // allocator mismatch
```

This bug is invisible to the compiler and catastrophic at runtime.

3.2 Correct Ownership Encoding

Modern C++ encodes ownership explicitly:

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

Allocator, lifetime, and destruction are unified.

3.2.1 Advanced Example: Ownership Transfer Across Layers

```
std::unique_ptr<Session> open_session();

void run(std::unique_ptr<Session> session);
```

Ownership transfer is explicit, visible, and enforced.

3.2.2 Shared Ownership: Advanced Cautionary Example

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

This design creates reference cycles and memory leaks.

3.2.3 Professional Alternative Using Ownership Hierarchy

```
struct Node {
    std::unique_ptr<Node> next;
    Node* prev;
};
```

Ownership is hierarchical and acyclic.

3.3 Borrowing vs Owning Interfaces

Professional APIs distinguish ownership categories explicitly.

3.3.1 Advanced Borrowing Example

```
void process(std::span<const std::byte> buffer);
```

This function:

- Does not allocate
- Does not own memory
- Does not extend lifetime

3.3.2 Classic Advanced Bug: Returning Borrowed Views

```
std::span<int> values() {
    std::vector<int> v{1,2,3};
    return v; // dangling
}
```

3.3.3 Professional Correction

```
std::vector<int> values();

void use() {
    auto v = values();
    std::span<int> view = v;
}
```

Ownership and borrowing are clearly separated.

3.3.4 Lifetime Clarity as a Design Requirement

Every public API must answer, through types alone:

- Who owns this?
- Who destroys it?
- How long is it valid?

If the answer is not immediate, the API is incomplete. Professional C++ API design treats lifetime clarity as non-negotiable. Correctness begins at the boundary.

Chapter 4

Header Files as Stable Interfaces

Header files define the public contract of a C++ library. They are not merely a technical requirement of the compilation model; they are the *primary interface* through which users understand, reason about, and depend on a library.

In professional C++ systems, header files often remain stable for years, even as implementations are rewritten, optimized, or completely replaced. For this reason, headers must be designed with extreme care. Every symbol exposed in a header becomes part of the long-term API surface and creates an implicit promise to users.

A poorly designed header leaks implementation details, increases compile times, restricts evolution, and makes correctness harder to enforce.

4.1 Rules for Professional Headers

A professional header obeys a small number of strict rules. Violating any of them introduces long-term cost.

4.1.1 Minimal Includes

Headers must include only what is absolutely necessary. Every unnecessary include increases compile time and couples the API to implementation details.

```
// Bad: heavy and unnecessary includes
#include <vector>
#include <map>
#include <iostream>

class Logger {
public:
    void log(const std::string& msg);
};
```

This header exposes implementation dependencies that users do not need.

4.1.2 Correct Minimal Header

```
// Good: minimal includes
#include <string>

class Logger {
public:
    void log(const std::string& msg);
};
```

Implementation-specific headers belong in source files, not in interfaces.

4.1.3 No Exposed Internals

Headers must never expose internal data structures, helper types, or implementation details that are not part of the intended API.

```
// Bad: internal details exposed
class Parser {
public:
    std::vector<Token> tokens;
    void parse();
};
```

This design freezes internal representation and prevents future changes.

4.1.4 Stable Interface with Hidden Internals

```
class Parser {
public:
    Parser();
    void parse();

private:
    struct Impl;
    std::unique_ptr<Impl> impl_;
};
```

This approach preserves freedom to evolve implementation without breaking users.

4.1.5 Clear Ownership Semantics

Every type and function declared in a header must make ownership rules obvious from the signature alone.

```
// Ambiguous ownership
Resource* get_resource();
```

```
// Explicit ownership
std::unique_ptr<Resource> create_resource();
Resource& borrow_resource();
```

Headers must never rely on comments to explain lifetime.

4.1.6 Stable Surface Area

A professional header avoids unnecessary overloads, flags, and optional parameters that expand the API surface without increasing correctness.
Smaller interfaces are easier to maintain, test, and evolve safely.

Chapter 5

Templates and Concepts as Contracts

Templates are one of the most powerful features in C++, but also one of the most dangerous when used without discipline.
An unconstrained template accepts almost anything. When misused, it fails late, produces unreadable diagnostics, and allows invalid types to propagate deep into the system.
In professional API design, unconstrained templates are defects.

5.1 The Problem with Unconstrained Templates

```
template <typename T>
void serialize(const T& value) {
    value.serialize();
}
```

This function appears generic, but it encodes an undocumented requirement: `T` must provide a `serialize()` member. If it does not, the error will appear far from the call site and be difficult to understand.

5.2 Concept-Constrained Design

Modern C++ introduces concepts to express template requirements explicitly.

```
template <typename T>
concept Serializable = requires(T t) {
    t.serialize();
};
```

This concept defines a clear, checkable contract.

5.2.1 Using Concepts in APIs

```
template <Serializable T>
void serialize(const T& value) {
    value.serialize();
}
```

Now the compiler enforces correctness at the boundary. Invalid types are rejected immediately with clear diagnostics.

5.2.2 Benefits of Concept-Based APIs

Concepts provide:

- Earlier error detection
- Clearer intent
- Better diagnostics
- More maintainable interfaces

Concepts replace informal documentation with executable constraints.

5.2.3 Concepts as Part of the Public Contract

When a concept appears in a public header, it becomes part of the API contract. Users know exactly what is required, and maintainers can evolve implementations without weakening guarantees.

Chapter 6

Performance Without Leaking Complexity

Performance is a first-class concern in C++, but it must be handled with discipline. A professional API is fast by default and predictable in behavior, without forcing users to understand internal optimizations.
Performance tuning should remain an internal concern whenever possible.

6.1 Zero-Overhead Abstractions

The zero-overhead principle states that abstractions should not impose runtime cost when they are not used.
A professional API follows this principle rigorously.

6.1.1 Value Semantics and Move Efficiency

```
class Buffer {
public:
    Buffer(size_t size);
    Buffer(Buffer&&) noexcept;
```

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

This type can be returned by value without performance penalty.

6.1.2 Avoiding Premature Performance Knobs

```
// Bad: exposes internal performance controls
void process(Data& d, bool use_fast_path, bool prefetch);
```

This design forces users to understand internal behavior.

6.1.3 Professional Alternative

```
void process(Data& d);
```

The implementation selects the optimal strategy internally. The API remains simple, stable, and correct.

6.1.4 Predictability Over Micro-Optimization

Professional APIs favor predictable performance over fragile micro-optimizations. Hidden allocations, global caches, and implicit synchronization are avoided unless explicitly documented.

6.2 Performance as Part of the Contract

While implementations may change, performance characteristics must remain stable. Users should be able to reason about complexity and cost without reading source code.
A well-designed API makes performance visible where it matters and invisible where it does not.

Chapter 7

Case Study: A Small Professional C++ Library

This chapter presents a complete, intentionally small C++ library designed according to the principles discussed throughout this booklet. The purpose of this case study is not to demonstrate feature richness, but to show how correctness, clarity, and discipline interact in a real design.

The library demonstrates:

- Explicit ownership encoded in types
- Structured and consistent error handling
- Strong invariants enforced at construction
- Clean, stable public headers
- Simple and predictable build integration

The chosen example is a *configuration file loader*. This domain is simple enough to avoid distraction, yet complex enough to demonstrate real API design challenges.

7.1 Library Goals and Non-Goals

7.1.1 Goals

- Load configuration data from a file
- Provide read-only access to configuration values
- Fail explicitly and safely on errors
- Avoid global state
- Be deterministic and testable

7.1.2 Non-Goals

- No hot reloading
- No implicit file watching
- No runtime mutation
- No global configuration singleton
- No thread-safety guarantees

These non-goals are intentional. They reduce complexity and protect long-term correctness.

7.2 Public Header Design

The public header defines the entire contract of the library. No implementation details are exposed.

```
// config.hpp
#pragma once

#include <string>
#include <string_view>
#include <unordered_map>
#include <expected>

namespace config {

enum class Error {
    FileNotFound,
    ParseError,
    InvalidFormat
};

class Config {
public:
    Config(const Config&) = delete;
    Config& operator=(const Config&) = delete;

    Config(Config&&) noexcept;
    Config& operator=(Config&&) noexcept;

    std::string_view get(std::string_view key) const;

private:
    explicit Config(std::unordered_map<std::string, std::string> data);

    std::unordered_map<std::string, std::string> data_;
};

std::expected<Config, Error> load(const std::string& path);
```

```
} // namespace config
```

7.2.1 Design Analysis

This header encodes several important decisions:

- Ownership of configuration data is explicit and internal
- Configuration objects are movable but not copyable
- Errors are returned explicitly using `std::expected`
- The API surface is intentionally small

Users can understand the full behavior of the library by reading this header alone.

7.3 Enforcing Invariants

The `Config` object guarantees that:

- It always contains valid key-value data
- Keys are unique
- Values are immutable after construction

Invalid states cannot be observed because construction is private.

7.4 Implementation Overview

Implementation details remain fully hidden in the source file.

```
// config.cpp
#include "config.hpp"
#include <fstream>

namespace config {

Config::Config(std::unordered_map<std::string, std::string> data)
    : data_(std::move(data)) {}

Config::Config(Config&&) noexcept = default;
Config& Config::operator=(Config&&) noexcept = default;

std::string_view Config::get(std::string_view key) const {
    auto it = data_.find(std::string(key));
    return it != data_.end() ? it->second : std::string_view{};
}

std::expected<Config, Error> load(const std::string& path) {
    std::ifstream file(path);
    if (!file) {
        return std::unexpected(Error::FileNotFound);
    }

    std::unordered_map<std::string, std::string> data;
    std::string line;

    while (std::getline(file, line)) {
        auto pos = line.find('=');
        if (pos == std::string::npos) {
```

```
            return std::unexpected(Error::ParseError);
        }
        data.emplace(line.substr(0, pos), line.substr(pos + 1));
    }

    return Config{std::move(data)};
}

} // namespace config
```

7.4.1 Error Handling Rationale

Errors are:

- Explicit
- Typed
- Impossible to ignore accidentally

The API does not throw exceptions and does not return sentinel values. Failure must be handled deliberately by the caller.

7.5 Usage Example

```
auto cfg = config::load("app.cfg");
if (!cfg) {
    // handle error explicitly
    return;
}

std::string_view host = cfg->get("host");
```

The usage is simple, predictable, and misuse-resistant.

7.6 Why This Library Is Small

This library deliberately avoids:

- Dynamic reconfiguration
- Multi-threaded access
- Plugin systems
- Implicit global access

Each avoided feature reduces surface area and long-term maintenance cost.
Correctness is preserved by design rather than by convention.

Chapter 8

Professional API Design Checklist

Before publishing a C++ API, a professional library author should verify the following checklist rigorously.

8.1 Ownership and Lifetime

- Is ownership explicit in every public function?
- Are raw owning pointers avoided?
- Are borrowing interfaces clearly distinguished?

8.2 Error Handling

- Is a single error-handling strategy used consistently?
- Are failures explicit and unignorable?
- Are error types meaningful and scoped?

8.3 Invariants and Correctness

- Are invalid states unrepresentable?
- Are invariants enforced at construction?
- Are partially initialized objects impossible?

8.4 Headers and Interfaces

- Are headers minimal and stable?
- Are implementation details fully hidden?
- Is the public surface area intentionally small?

8.5 Misuse Resistance

- Is incorrect usage difficult or impossible?
- Are dangerous operations explicit?
- Does the type system guide correct usage?

8.6 Evolution and Longevity

- Can the implementation evolve without breaking users?
- Are non-goals documented and enforced?
- Is backward compatibility considered?

A professional API is not defined by how easy it is to write, but by how hard it is to misuse. This checklist is not optional. It is the minimum standard for long-lived, production-quality C++ libraries.

Chapter 9

Conclusion

API design is not a secondary concern. It is not a cosmetic layer applied after implementation, nor is it a matter of personal style, naming preference, or aesthetic taste. In C++, API design is a *core engineering activity* that determines the long-term correctness, safety, and survivability of a software system.

Once an API is published, it becomes a dependency. It is copied into documentation, embedded in build scripts, depended upon by other teams, and relied on in production environments. At that point, changing it is no longer a purely technical decision. It becomes a financial, organizational, and sometimes political problem.

For this reason, APIs must be designed deliberately, conservatively, and with a long time horizon in mind.

In C++, APIs define fundamental properties of a system, including:

- How resources are acquired, owned, and released
- How errors are represented, propagated, and handled
- What states are valid, invalid, or impossible

- How software evolves without breaking existing users

These decisions are not isolated. They interact deeply with performance, safety, testability, and maintainability. A weak decision at the API level propagates throughout the entire system. Modern C++ provides powerful tools to address these challenges directly. RAII enables deterministic and exception-safe resource management. Strong types allow domain concepts to be expressed precisely and enforced by the compiler. Concepts make template requirements explicit and diagnosable. Deterministic destruction and value semantics allow predictable behavior even in complex systems.

These tools fundamentally change what is possible in API design. Correctness no longer depends primarily on discipline, documentation, or convention. It can be enforced mechanically.

This shift has an important implication: the responsibility of an API designer is not to make usage easy. Ease of use is subjective and often short-lived. Instead, the responsibility is to make misuse difficult or impossible.

A professional API guides its users toward correct usage. It prevents invalid states from being represented. It makes ownership and lifetime unambiguous. It forces errors to be handled explicitly. It exposes only what must be exposed and hides everything else.

Small, well-designed libraries often outlive larger and more feature-rich alternatives. They survive refactoring, personnel changes, and evolving requirements because their interfaces are stable and their guarantees are clear. Their value lies not in the number of functions they expose, but in the strength of the promises they make and keep.

In this sense, API design is not only a technical activity, but an ethical one. Publishing an API is a promise to future users, including engineers who may never meet the original author. It is a commitment that correctness, clarity, and stability were treated as first-class goals rather than afterthoughts.

A professional C++ API does not merely work. It earns trust over time.

References

The principles and practices discussed in this booklet are informed by the following authoritative sources, standards, and professional works:

- ISO/IEC JTC1/SC22/WG21 *Programming Languages — C++ (C++20, C++23 draft papers and proposals)*
- Bjarne Stroustrup *A Tour of C++ (Second Edition)*, Addison-Wesley, 2020
- Nicolai M. Josuttis *C++20 – The Complete Guide*, No Starch Press, 2022
- Herb Sutter *C++ Core Guidelines*, continuously updated professional guidelines
- Andrei Alexandrescu *Modern C++ Design (Revisited Concepts and Practices)*, conference materials and updated talks
- LLVM Project Documentation *C++ API Design and ABI Stability Practices*
- Microsoft C++ Team *Guidelines for Library Design and ABI Compatibility*
- ISO C++ Proposal Papers *P0323 (std::expected)*, *P0732 (Class Template Argument Deduction)*, *P0896 (Concepts)*
- Industrial C++ Codebases *Design practices derived from large-scale systems in compilers, operating systems, and performance-critical libraries*