https://simplifycpp.org

Writing Safe Plugins and DLLs in Modern C++

Prepared by Ayman alheraki

Writing Safe Plugins and DLLs in Modern C++

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Author's Introduction

This booklet was written for experienced C++ engineers who have already discovered that writing *working code* is not the same as writing *survivable code*.

Plugins, shared libraries, and dynamically loaded modules sit at the most fragile boundary in any C++ system. They cross compilation units, toolchains, and often even organizational responsibility. Mistakes made at these boundaries rarely fail immediately. Instead, they accumulate silently and surface later as memory corruption, unexplained crashes, ABI breakage, or production-only failures that resist debugging.

Over the years, many C++ developers have learned plugin development through trial and error, outdated examples, or by copying patterns that appeared to work without understanding why. This approach does not scale. It produces systems that are brittle, unsafe, and expensive to maintain.

This booklet takes a different path.

It treats plugin and DLL development as an *architectural discipline*, not a coding trick. The focus is not on cleverness, but on correctness, predictability, and long-term stability across platforms and compiler versions.

The goal is simple: to help you design plugin systems that remain safe, understandable, and maintainable after years of evolution.

Ayman Alheraki

Preface

C++ offers unmatched power and control, but it does not forgive violations of its fundamental rules. Nowhere is this more evident than at binary boundaries.

Unlike header-only libraries or static linking, plugins and shared libraries force the programmer to confront issues that are usually hidden: Application Binary Interfaces, object ownership across module boundaries, exception propagation, allocator mismatches, and version compatibility.

Modern C++ (C++20 and later) provides better tools than ever before, but it does not remove the underlying constraints. In fact, the language's increasing expressiveness makes it even easier to write code that is *correct in isolation* yet *undefined at the boundary*.

This booklet exists to make those boundaries explicit.

Rather than relying on folklore or platform-specific hacks, we will build a clear mental model of what is safe, what is dangerous, and what is simply undefined behavior. Each design rule is motivated by real failure modes observed in production systems.

No frameworks are introduced. No magic abstractions are used. Every example is minimal, explicit, and intentional.

By the end of this booklet, you should be able to design and implement a professional-grade plugin system that respects the realities of modern C++, across both Windows and Linux.

Chapter 1

What Plugins and DLLs Really Are

At a purely technical level, a plugin is a dynamically loaded binary that exports a known set of symbols. This description is correct, but dangerously incomplete.
It describes *what* a plugin is, not *what it represents* in a C++ system.
In practice, a plugin is not merely code loaded at runtime. It is a contractual boundary between independently compiled worlds.

1.1 Beyond Dynamic Loading

Dynamic loading is often introduced as a convenience feature: the ability to extend an application without recompilation. While this is true, it obscures the deeper implications.
When a plugin is loaded, two independently built binaries are forced to interact inside the same process:

- They may be compiled with different compiler versions
- They may use different standard library implementations
- They may be built with different optimization levels

- They may evolve independently over time

From the compiler's perspective, these binaries were never meant to reason about each other.

1.2 The Disappearance of Compiler Protection

In statically linked code, the compiler enforces:

- Type correctness
- Object layout consistency
- Exception propagation rules
- Lifetime assumptions

At a plugin boundary, these protections disappear.
Once a shared library is loaded, the operating system provides only the most basic guarantees: memory mapping, symbol resolution, and relocation. It does not understand C++ semantics, object lifetimes, or ownership rules.
The responsibility shifts entirely to the developer.

1.3 Operating System Perspective

From the operating system's point of view, plugins are simply shared objects:

- Windows loads a `.dll` using `LoadLibrary`
- Linux loads a `.so` using `dlopen`

The operating system does not enforce:

- ABI compatibility
- Exception safety
- Memory ownership correctness
- Threading discipline

If a mistake is made, the operating system will not prevent it. At best, the process crashes. At worst, corruption remains undetected.

1.4 C++ Perspective: No Such Thing as a Safe Plugin

From the C++ language's perspective, there is no such thing as a "safe plugin."
There are only:

- Defined behavior
- Undefined behavior
- Patterns that avoid undefined behavior

C++ does not provide built-in mechanisms for enforcing safety across binary boundaries.
Instead, it provides powerful tools that must be used with discipline and restraint.
A plugin system is therefore not a language feature. It is an architectural discipline layered on top of C++.

1.5 Why Naive Designs Appear to Work

Many unsafe plugin systems appear to work:

- During early testing

- On a single machine
- With a single compiler version
- Under light load

These systems fail later, often in production, because undefined behavior does not fail deterministically.
The absence of immediate failure is not evidence of correctness.

1.6 The Plugin Boundary as an Architectural Boundary

A plugin boundary must be treated as:

- A trust boundary
- A lifetime boundary
- A versioning boundary
- A failure boundary

Designing a plugin system is therefore closer to designing a distributed system than writing a library.
Assumptions must be explicit. Contracts must be enforced. Failures must be expected.

1.7 Foundation for the Chapters Ahead

This chapter establishes a critical mindset:

> **Plugins are not about dynamic loading. They are about controlling what happens when the compiler can no longer help you.**

Everything that follows in this booklet exists to manage that reality.

The next chapter examines the first and most fundamental constraint at this boundary: the Application Binary Interface (ABI), and why it dominates every design decision in safe plugin systems.

Chapter 2

ABI, Binary Boundaries, and Why They Matter

One of the most common mistakes in plugin and DLL development is assuming that C++ source compatibility implies binary compatibility. It does not.
The Application Binary Interface (ABI) defines how compiled code interacts at runtime. It governs object layout, name mangling, calling conventions, exception handling, RTTI, and memory allocation behavior. Unlike the C++ language itself, the ABI is not fully standardized across platforms or compilers.
When two translation units are statically linked, the compiler can enforce internal consistency. When code is separated into shared libraries, that safety net disappears.

2.1 What the ABI Actually Controls

At the binary level, the ABI defines rules such as:

- How functions are called (calling convention)

- How parameters are passed (registers vs stack)
- How return values are delivered
- How objects are laid out in memory
- How virtual tables are structured
- How names are mangled
- How exceptions are propagated

These rules are not abstract theory. Violating them leads directly to undefined behavior, often without immediate crashes.

2.2 Why C++ Has No Stable Universal ABI

C++ is intentionally designed to allow aggressive optimization and zero-cost abstractions.
This design choice makes a universal, stable ABI extremely difficult.
Even small changes can affect binary layout:

- Adding a virtual function changes the vtable
- Reordering private members changes object size
- Changing compiler flags alters calling conventions
- Switching standard library versions changes type layout

As a result, **C++ ABIs are compiler- and platform-specific**.
This is not a flaw. It is a trade-off.

2.3 The Plugin Boundary as a Trust Boundary

A plugin boundary is a trust boundary.
Once code crosses that boundary, the compiler no longer protects you. The host application must assume that the plugin:

- Was compiled separately
- May evolve independently
- May be replaced without recompiling the host

This means that every exported symbol becomes part of a contract that must be maintained over time.

2.4 Why Header-Only Thinking Fails for Plugins

Many C++ developers are accustomed to header-only or inline-heavy designs. These patterns fail catastrophically at ABI boundaries.
Consider the following class:

```
class Processor {
public:
    virtual int process(int value);
    virtual ~Processor();
};
```

If this class is exported directly from a plugin:

- Its vtable layout becomes part of the ABI
- Any change breaks binary compatibility

- The destructor becomes a cross-module call

Even if the header remains unchanged, recompiling the plugin with a different compiler version may silently break compatibility.

2.5 C vs C++ at Binary Boundaries

The C language has a relatively stable and well-understood ABI across platforms. This is why many successful plugin systems expose C-style interfaces even when implemented in C++.
A C-style function interface:

- Avoids name mangling
- Avoids object layout dependencies
- Avoids exception propagation
- Is easier to version

This does not mean abandoning C++. It means *containing it*.

2.6 The Cost of Ignoring ABI Discipline

Ignoring ABI rules typically leads to:

- Crashes that only appear in production
- Memory corruption detected far from the cause
- Inability to update plugins independently
- Fear-driven development and frozen interfaces

Many teams respond by banning plugins entirely or resorting to unsafe workarounds. Both approaches are avoidable.

2.7 Design Principle: Minimize the ABI Surface

A fundamental rule emerges:

> **Expose the smallest, simplest, and most stable interface possible.**

This principle will guide every design decision in the chapters that follow.
We will:

- Avoid exporting C++ classes directly
- Avoid sharing STL types across boundaries
- Avoid implicit ownership transfer
- Make versioning explicit

In the next chapter, we will examine one of the most dangerous sources of undefined behavior in plugin systems: *memory ownership across binary boundaries*.

Chapter 3

Memory Ownership Across Module Boundaries

If Application Binary Interfaces define how code *communicates*, memory ownership defines how code *survives*.
Across plugin boundaries, memory ownership is the most frequent source of undefined behavior, silent corruption, and delayed crashes. The reason is simple: C++ does not define how memory allocated in one binary must be released in another.
This chapter establishes strict rules for memory ownership and explains why they are necessary.

3.1 Why Memory Is Not Just Memory

In C++, dynamic memory allocation is not a single global operation. It is a cooperation between:

- The allocator implementation

- The runtime library
- The compiler
- The build configuration

When a plugin and its host are built separately, there is no guarantee that these components match.

Even when both sides use `new` and `delete`, they may not be using the same heap.

3.2 The Classic Failure Pattern

Consider the following scenario:

```
// Plugin side
extern "C" int* create_value() {
    return new int{42};
}
```

```
// Host side
extern "C" int* create_value();

void use_plugin() {
    int* value = create_value();
    delete value; // Undefined behavior
}
```

This code may appear to work during testing. It may even work for years.

Then, one day, it fails.

The failure is not random. It is the inevitable result of mismatched allocation domains.

3.3 Why This Happens

Each binary may:

- Link against a different C++ runtime
- Use a different heap implementation
- Use different allocator debug settings
- Use different build modes (debug vs release)

The C++ standard does not require memory allocated in one module to be safely freed in another.

3.4 Rule 1: The Allocator That Allocates Must Deallocate

The most important rule in plugin design is:

> **The module that allocates memory must also be responsible for freeing it.**

This rule alone eliminates an entire class of failures.

3.5 Returning Ownership Safely

Instead of returning raw pointers that require the host to free memory, return handles with explicit destruction functions:

```
extern "C" int* create_value();
extern "C" void destroy_value(int* ptr);
```

```
void use_plugin() {
    int* value = create_value();
    // use value
    destroy_value(value);
}
```

The plugin retains full control over allocation and deallocation.

3.6 Opaque Handles Over Concrete Types

A more robust approach is to avoid exposing concrete types entirely:

```
struct value_handle;

extern "C" value_handle* create_value();
extern "C" void destroy_value(value_handle*);
extern "C" int get_value(const value_handle*);
```

The host never sees the internal structure. The ABI surface remains stable even if the implementation changes.

3.7 Why STL Types Must Not Cross the Boundary

Standard library types such as `std::string`, `std::vector`, and `std::map` are not ABI-stable.
They:

- Have implementation-defined layouts
- Allocate memory internally
- Depend on the runtime allocator

Passing them across module boundaries is undefined behavior, even when using the same compiler.

3.8 Smart Pointers Do Not Solve This

Smart pointers manage ownership *within* a single binary. They do not solve cross-module allocation issues.
A `std::unique_ptr` that deletes memory in the host that was allocated in the plugin is still undefined behavior.

3.9 Design Principle: Ownership Must Be Explicit

A safe plugin API makes ownership:

- Explicit
- Directional
- Enforced by interface design

If ownership is unclear, the design is already broken.

3.10 Memory Domains as a Mental Model

Think of each module as owning its own memory domain. Data may be accessed across domains, but ownership must never be ambiguous.
This mental model simplifies design decisions and prevents accidental violations.

3.11 Preparing for the Next Layer of Safety

Memory safety is not complete without considering exceptions.
In the next chapter, we will examine why throwing exceptions across plugin boundaries is unsafe, and how to design error handling without relying on cross-module stack unwinding.

Chapter 4

Exception Safety Across DLL and Plugin Boundaries

C++ exceptions are one of the language's most powerful tools for expressing failure. Within a single binary, they integrate naturally with RAII and allow clean separation between normal and error paths.
Across plugin and DLL boundaries, however, exceptions become dangerous.
This chapter explains why exceptions cannot be safely propagated across module boundaries and how to design robust error handling without relying on them.

4.1 How C++ Exceptions Actually Work

When an exception is thrown, the runtime performs stack unwinding:

- It searches for a matching catch handler
- It executes destructors for all in-scope objects
- It relies on compiler-generated metadata

This process depends on:

- Compatible runtime libraries
- Identical exception handling mechanisms
- Consistent stack frame layout

None of these are guaranteed across binary boundaries.

4.2 What Happens at a Plugin Boundary

When a plugin throws an exception that crosses into the host:

- The host may not understand the exception type
- Stack unwinding metadata may be incompatible
- Destructors may not run correctly
- The process may terminate immediately

In many environments, this results in `std::terminate` being called.

4.3 Why Catching at the Boundary Is Not Enough

A common misconception is that catching all exceptions at the boundary makes the system safe:

```
extern "C" int plugin_function() {
    try {
        // risky code
        return 0;
```

```
    } catch (...) {
        return -1;
    }
}
```

While this is better than nothing, it does not solve the full problem.
If any exception escapes the plugin before being caught, the behavior is still undefined.
Furthermore, exception-based control flow obscures ownership and failure semantics.

4.4 Rule 2: Never Let Exceptions Cross the Boundary

The rule is absolute:

> **No C++ exception may cross a plugin or DLL boundary.**

This rule simplifies reasoning and eliminates entire categories of failure.

4.5 Error Reporting Without Exceptions

Instead of throwing exceptions across the boundary, plugins must report errors explicitly.
Common approaches include:

- Return error codes
- Use structured result types
- Provide explicit error query functions

4.6 Using Error Codes

The simplest approach is returning integer error codes:

```
extern "C" int plugin_process(int input, int* output);
```

- Zero indicates success
- Non-zero values indicate failure

This approach is predictable and ABI-safe, but can become verbose.

4.7 Structured Error Results

A more expressive approach uses explicit result structures:

```
struct plugin_result {
    int error_code;
    int value;
};

extern "C" plugin_result plugin_process(int input);
```

This avoids exceptions while still returning meaningful data.

4.8 Error Message Retrieval

To avoid returning dynamically allocated strings, provide an explicit error query function:

```
extern "C" const char* plugin_last_error();
```

The returned pointer must remain valid until the next plugin call or be owned entirely by the plugin.

4.9 Exceptions Inside the Plugin Are Fine

It is important to distinguish internal and external behavior.
Inside the plugin, exceptions may be used freely:

- To simplify internal logic
- To enforce invariants
- To manage complex control flow

The only requirement is that they must be fully handled before crossing the boundary.

4.10 Designing Fail-Fast Boundaries

A professional plugin API is designed to fail fast and fail clearly:

- Invalid inputs are rejected immediately
- Errors are explicit and documented
- Undefined behavior is eliminated by design

Silently continuing after failure is more dangerous than terminating.

4.11 Preparing for Interface Design

With ABI rules, memory ownership, and exception safety established, we are now ready to design robust plugin interfaces.
The next chapter introduces interface design patterns that enforce these rules by construction, rather than relying on discipline alone.

Chapter 5

Designing a Safe Plugin Interface

Safety at plugin boundaries cannot depend on conventions alone. It must be enforced by the structure of the interface itself.
A well-designed plugin interface makes unsafe usage impossible or at least extremely difficult. A poorly designed one relies on documentation and hope.
This chapter presents design patterns that encode ABI stability, memory safety, and exception safety directly into the interface.

5.1 The Purpose of an Interface Boundary

A plugin interface serves two roles:

- It defines what the host is allowed to do
- It defines what the plugin guarantees in return

Anything not explicitly stated in the interface must be considered forbidden.

5.2 Avoiding Direct C++ Class Export

Exporting C++ classes directly from a plugin is one of the most fragile designs possible. Problems include:

- ABI instability due to vtables
- Destructor ownership ambiguity
- Inlining across boundaries
- RTTI incompatibility

Even when it works today, it locks the interface permanently.

5.3 C-Style Interface as the ABI Contract

The safest ABI contract is a C-style interface:

- Functions with C linkage
- Plain data structures
- Explicit lifetime management

Example:

```
extern "C" {

struct plugin_api;

plugin_api* plugin_create();
void plugin_destroy(plugin_api*);
```

```
int plugin_process(plugin_api*, int input, int* output);

}
```

This interface is stable, testable, and ABI-friendly.

5.4 Opaque Handles and Encapsulation

The `plugin_api` type is intentionally incomplete.

- The host cannot access its internals
- The plugin retains full control over layout
- Changes do not affect the ABI

This pattern is known as an *opaque handle*.

5.5 Enforcing Ownership Through Functions

Ownership rules are enforced by providing paired create/destroy functions.

```
plugin_api* plugin_create();
void plugin_destroy(plugin_api*);
```

The host cannot accidentally delete memory incorrectly because it does not know how the object was allocated.

5.6 Versioning the Interface

Every plugin interface must be versioned explicitly.

```
extern "C" int plugin_api_version();
```

The host can refuse to load incompatible plugins before invoking any function.

5.7 Function Tables as Stable Interfaces

For more complex APIs, function tables provide controlled extensibility:

```
struct plugin_v1 {
    int (*process)(void*, int, int*);
    void (*destroy)(void*);
};
```

- New versions can extend the structure
- Older fields remain stable
- Binary compatibility is preserved

5.8 Separating ABI from C++ Convenience

The ABI interface should be minimal and conservative. C++ convenience wrappers may exist on the host side.

```
class Plugin {
public:
    explicit Plugin(plugin_api* api) : api_(api) {}
    int process(int input);
```

```
private:
    plugin_api* api_;
};
```

This allows modern C++ usage without exposing it across the boundary.

5.9 Design Principle: Make Illegal States Unrepresentable

A well-designed plugin interface:

- Prevents misuse by construction
- Makes ownership explicit
- Avoids implicit assumptions

If the interface allows unsafe usage, it will eventually be used unsafely.

5.10 Preparing for Dynamic Loading

Designing the interface is only half the problem. The next chapter will focus on safely loading plugins at runtime and validating their interfaces before use.
Dynamic loading introduces a new class of failure modes that must be handled explicitly.

Chapter 6

Safe Plugin Loading on Windows and Linux

Designing a safe plugin interface is meaningless if the host loads plugins recklessly. Dynamic loading introduces failure modes that do not exist in statically linked programs: missing symbols, incompatible binaries, partial initialization, and runtime environment mismatches.
This chapter focuses on defensive plugin loading on both Windows and Linux.

6.1 Dynamic Loading as a Trust Boundary

Loading a plugin is an act of trust.
Once a shared library is loaded:

- Its initialization code executes immediately
- Global constructors may run
- Static state may be modified

A robust host treats plugin loading as a potentially hostile operation.

6.2 Platform Overview

- Windows uses `LoadLibrary` / `GetProcAddress`
- Linux uses `dlopen` / `dlsym`

Despite API differences, the design principles are identical.

6.3 Step 1: Load the Binary Carefully

On Windows:

```
#include <windows.h>

HMODULE handle = LoadLibraryA("plugin.dll");
if (!handle) {
    // loading failed
}
```

On Linux:

```
#include <dlfcn.h>

void* handle = dlopen("./plugin.so", RTLD_NOW);
if (!handle) {
    const char* error = dlerror();
}
```

Failure must be handled immediately and explicitly.

6.4 Step 2: Resolve Required Symbols

Never assume symbols exist.

```
// Windows
auto create_fn = reinterpret_cast<plugin_create_fn>(
    GetProcAddress(handle, "plugin_create")
);

// Linux
auto create_fn = reinterpret_cast<plugin_create_fn>(
    dlsym(handle, "plugin_create")
);
```

If any required symbol is missing, the plugin must be rejected.

6.5 Step 3: Validate the Interface Version

Before invoking any function, validate the plugin's API version:

```
using version_fn = int (*)();

auto version = reinterpret_cast<version_fn>(
    dlsym(handle, "plugin_api_version")
);

if (!version || version() != EXPECTED_VERSION) {
    // incompatible plugin
}
```

This step prevents undefined behavior caused by mismatched expectations.

6.6 Step 4: Controlled Initialization

Plugin initialization should be explicit.

```
plugin_api* api = create_fn();
if (!api) {
    // initialization failed
}
```

Avoid relying on global state or implicit initialization order.

6.7 Failure Recovery and Cleanup

If any step fails:

- Destroy partially created objects
- Unload the library
- Report the failure clearly

On Windows:

```
FreeLibrary(handle);
```

On Linux:

```
dlclose(handle);
```

Leaking a failed plugin load is a design error.

6.8 Avoiding Static Initialization Hazards

Plugins with heavy global initialization are dangerous.

- Initialization order is undefined
- Failures are hard to recover from
- Debugging becomes difficult

Prefer explicit initialization functions over global constructors.

6.9 Design Principle: Fail Early, Fail Loud

A professional plugin host:

- Validates everything
- Assumes nothing
- Refuses to proceed on uncertainty

Silent fallback behavior hides errors and multiplies risk.

6.10 Preparing for Real-World Usage

With safe loading in place, we are ready to address how plugins behave in real systems:

- Logging
- Diagnostics
- Error reporting

The next chapter introduces techniques for observability and controlled failure.

Chapter 7

Logging, Diagnostics, and Fail-Fast Design

A plugin system that cannot explain its failures is not production-ready.
Crashes, misbehavior, and performance degradation are inevitable in long-lived systems.
What distinguishes professional systems is not the absence of failure, but the ability to detect, diagnose, and respond to it predictably.
This chapter focuses on designing observability into plugin systems without compromising safety or ABI stability.

7.1 Why Logging Across Plugin Boundaries Is Difficult

Logging appears simple until it crosses a binary boundary.
Common mistakes include:

- Sharing logging objects across modules
- Passing `std::string` messages across the boundary
- Assuming thread-safe logging behavior

These approaches violate the same ABI and ownership rules discussed earlier.

7.2 Rule 3: Logging Must Respect Module Boundaries

Each module must remain responsible for its own internal logging mechanisms.
Communication across the boundary must be explicit and minimal.

7.3 Callback-Based Logging

A safe approach is for the host to provide a logging callback during plugin initialization.

```
using log_fn = void (*)(int level, const char* message);

struct plugin_host_api {
    log_fn log;
};
```

The plugin does not own the logger. It only invokes the callback.

7.4 Registering the Logger

```
extern "C" void plugin_initialize(plugin_api*, const plugin_host_api*);
```

The host controls logging policy while the plugin remains decoupled.

7.5 Log Levels and Stability

Log levels should be represented using stable, plain values:

```
enum log_level {
    LOG_INFO = 0,
    LOG_WARNING = 1,
    LOG_ERROR = 2
};
```

Avoid using scoped enums or C++ types across the boundary.

7.6 Diagnostics Without Shared State

Diagnostic data should be:

- Queried explicitly
- Owned by the plugin
- Returned as simple values

Example:

```
extern "C" int plugin_get_status(plugin_api*);
```

7.7 Fail-Fast vs Silent Recovery

Fail-fast design prioritizes correctness over availability.
When invariants are violated:

- Abort the operation immediately
- Log the failure clearly
- Avoid partial recovery that hides errors

Silent recovery often causes long-term data corruption.

7.8 Asserting Invariants Inside Plugins

Assertions are valuable inside plugins but must not cross boundaries.

```
void internal_function(int value) {
    assert(value >= 0);
}
```

If an assertion fails, the plugin should treat it as a fatal internal error.

7.9 Handling Plugin Misbehavior

A robust host must assume plugins can misbehave:

- Invalid outputs
- Unexpected return codes
- Performance degradation

The host should isolate and disable faulty plugins when possible.

7.10 Design Principle: Make Failures Obvious

A system that hides its failures is more dangerous than one that crashes predictably. Clear diagnostics illustrates intent, reduces debugging time, and supports long-term maintenance.

7.11 Preparing for Concurrency

Logging and diagnostics become significantly more complex in multithreaded systems.
The next chapter explores concurrency, thread safety, and the rules governing multithreaded plugins.

Chapter 8

Threading and Concurrency Across Plugin Boundaries

Concurrency magnifies every design mistake.
A plugin system that appears correct in single-threaded tests may fail catastrophically when invoked concurrently. Undefined behavior that occurs once per million calls will eventually occur in production.
This chapter defines strict rules for thread safety across plugin boundaries.

8.1 The Illusion of Thread Safety

C++ provides powerful concurrency primitives, but none of them guarantee safety across shared library boundaries.
Common incorrect assumptions include:

- Believing that `std::mutex` is safe across modules
- Assuming global state is shared predictably

- Treating plugin functions as reentrant by default

None of these assumptions are safe.

8.2 Rule 4: Never Share Synchronization Primitives Across the Boundary

Synchronization primitives are implementation-defined.
Passing objects such as:

- `std::mutex`
- `std::condition_variable`
- `std::atomic`

across plugin boundaries is undefined behavior.
Each module must manage its own synchronization internally.

8.3 Reentrancy Must Be Explicit

The plugin interface must clearly state whether functions:

- Are thread-safe
- Are reentrant
- Require external synchronization

If this is not stated explicitly, the function must be assumed unsafe.

8.4 Single-Threaded Plugin Model

The safest default model is single-threaded access:

- One thread owns the plugin instance
- All calls are serialized by the host
- The plugin requires no internal locking

This model is simple, predictable, and often sufficient.

8.5 Thread-Safe Plugin Model

If thread-safe access is required, the responsibility lies with the plugin.

```
struct plugin_api {
    int (*process)(void*, int, int*);
};
```

The plugin must internally protect shared state.
The host must not attempt to synchronize plugin internals.

8.6 Avoiding Hidden Global State

Global state inside plugins is particularly dangerous:

- Initialization order is undefined
- Lifetime is difficult to manage
- Testing becomes unreliable

Prefer instance-based state tied to explicit lifetimes.

8.7 Thread Ownership and Call Context

The interface must specify:

- Which thread may call which function
- Whether callbacks may occur on plugin threads
- Whether blocking is permitted

Ambiguity leads to deadlocks and race conditions.

8.8 Callbacks and Concurrency

Callbacks from plugin to host introduce bidirectional control flow.
Rules must be explicit:

- Callbacks must not re-enter plugin code unless documented
- Lock ordering must be well-defined
- Long-running callbacks must be avoided

8.9 Performance vs Safety Trade-offs

Concurrency introduces trade-offs:

- Fine-grained locking improves throughput but increases complexity
- Coarse-grained locking simplifies design but reduces scalability

Plugin systems should favor predictability over peak performance.

8.10 Design Principle: Make Concurrency a First-Class Concern

Concurrency cannot be added later without breaking compatibility.
Threading guarantees must be:

- Designed upfront
- Documented in the interface
- Enforced by implementation

8.11 Preparing for Lifetime Management

Concurrency complicates lifetime management.
The next chapter examines how to define clear lifetimes for plugins, instances, and resources to prevent leaks and use-after-free errors.

Chapter 9

Lifetime Management and Resource Control

Lifetime management is where most plugin systems quietly fail.
Memory ownership rules may be correct, threading rules may be documented, and interfaces may be stable—yet the system can still collapse if object lifetimes are unclear or unenforced.
This chapter establishes strict lifetime models for plugins, plugin instances, and the resources they manage.

9.1 Why Lifetime Errors Are So Dangerous

Lifetime errors rarely cause immediate crashes.
Instead, they manifest as:

- Use-after-free bugs
- Dangling callbacks
- Random crashes during shutdown

- Deadlocks during unloading

These failures are notoriously difficult to reproduce and debug.

9.2 Defining Lifetime Layers

A professional plugin system defines clear lifetime layers:

- **Process lifetime** — the host application
- **Plugin lifetime** — the loaded shared library
- **Instance lifetime** — objects created by the plugin
- **Call lifetime** — individual function invocations

Each layer must be explicitly managed and never implicitly assumed.

9.3 Rule 5: No Outstanding Objects at Plugin Unload

Before unloading a plugin:

- All plugin-created objects must be destroyed
- No threads may still be executing plugin code
- No callbacks into the plugin may remain

Violating this rule results in immediate undefined behavior.

9.4 Explicit Shutdown Protocols

Plugins must support an explicit shutdown sequence.

```
extern "C" void plugin_shutdown(plugin_api*);
```

This function allows the plugin to:

- Stop background threads
- Flush internal state
- Release resources deterministically

Implicit cleanup during `dlclose` or `FreeLibrary` is unsafe.

9.5 Instance-Based Lifetime Control

Avoid global plugin state whenever possible.
Instead, tie state to explicit instances:

```
plugin_api* plugin_create();
void plugin_destroy(plugin_api*);
```

This approach:

- Enables multiple independent instances
- Simplifies testing
- Makes ownership explicit

9.6 Callbacks and Lifetime Hazards

Callbacks introduce hidden lifetime dependencies.
If a plugin stores a host callback:

- The callback must remain valid for the plugin's lifetime
- The plugin must not call it after shutdown begins

These constraints must be documented and enforced.

9.7 Thread Shutdown Discipline

Threads created by a plugin must:

- Be joined or stopped during shutdown
- Never outlive the plugin instance
- Never execute after unload

Detached threads are almost always a design error in plugin systems.

9.8 Avoiding Destruction Order Traps

Destruction order matters.
The correct order is:

1. Stop external interaction
2. Stop background work
3. Destroy instances

4. Unload the plugin

Reversing this order leads to undefined behavior.

9.9 RAII Is Necessary but Not Sufficient

RAII simplifies internal resource management, but it does not solve cross-boundary lifetime issues.
RAII must be combined with:

- Explicit shutdown APIs
- Clear ownership rules
- Host-side enforcement

9.10 Design Principle: Lifetimes Must Be Boring

A correct lifetime model is intentionally uninteresting.
If you need to reason deeply about when something is destroyed, the design is already too complex.

9.11 Preparing for Real-World Evolution

With lifetime management in place, the system is now stable enough to evolve.
The next chapter explores versioning strategies that allow plugins and hosts to change independently without breaking compatibility.

Chapter 10

Versioning, Compatibility, and Evolution

A plugin system that cannot evolve is already obsolete.
Real-world software changes continuously: bugs are fixed, features are added, performance is improved, and security issues are addressed. A professional plugin architecture must support this evolution without forcing simultaneous recompilation of the host and all plugins.
This chapter defines strict versioning rules that preserve compatibility over time.

10.1 Why Versioning Is Hard in C++

In C++, even small changes can have large binary consequences:

- Adding a function may change structure layout
- Reordering fields may break ABI
- Changing error semantics may invalidate assumptions

Unlike source-level APIs, binary interfaces cannot rely on recompilation to restore correctness.

10.2 Rule 6: Version the ABI, Not the Implementation

Versioning must apply to the *binary contract*, not the internal code.
If the ABI changes, the version must change. If the ABI does not change, the version must not change.
Violating this rule leads to silent incompatibility.

10.3 Explicit Interface Version Query

Every plugin must expose an explicit version function:

```
extern "C" int plugin_api_version();
```

The host must refuse to load plugins with incompatible versions before calling any other function.

10.4 Semantic Meaning of Versions

Versions must have clear semantics:

- Major version changes break compatibility
- Minor version changes extend compatibility
- Patch versions fix behavior without changing the ABI

Only the major version is typically checked at runtime.

10.5 Extending Interfaces Safely

To extend an interface without breaking compatibility:

- Append new functions
- Never remove or reorder existing ones
- Preserve existing behavior

This rule applies to both function tables and exported symbols.

10.6 Versioned Function Tables

Function tables enable controlled evolution:

```
struct plugin_v1 {
    int (*process)(void*, int, int*);
    void (*destroy)(void*);
};

struct plugin_v2 {
    plugin_v1 base;
    int (*process_ex)(void*, int, int*, int flags);
};
```

Older hosts can safely ignore newer fields.

10.7 Capability-Based Design

Instead of relying on version numbers alone, plugins may expose capabilities:

```
extern "C" int plugin_has_feature(int feature_id);
```

This allows flexible negotiation between host and plugin.

10.8 Backward Compatibility Obligations

Once an ABI is published, it becomes a long-term obligation.
Removing or altering behavior breaks trust and increases maintenance costs.
Professional systems treat ABI stability as a contract.

10.9 Forward Compatibility Limits

Forward compatibility is inherently limited.
Older hosts cannot understand new semantics unless explicitly designed to do so. Design must prioritize backward compatibility.

10.10 Deprecation Strategy

Deprecation must be explicit and slow:

- Mark features as deprecated
- Maintain them across multiple versions
- Remove them only with a major version bump

Abrupt removal is a design failure.

10.11 Design Principle: Change Is Inevitable, Chaos Is Optional

A stable plugin system does not resist change. It channels change through explicit, predictable mechanisms.

With versioning in place, we can now examine advanced operational topics that appear in long-running systems.

The next chapter explores allocators and memory domains in more depth.

Chapter 11

Allocators, Memory Domains, and Custom Heaps

In long-running systems, memory correctness is not enough. Memory behavior must also be predictable.
Plugins that leak memory, fragment heaps, or assume allocator compatibility often degrade system performance slowly and invisibly. By the time the problem is detected, diagnosis is difficult and costly.
This chapter introduces allocator discipline as a first-class design concern in plugin architectures.

11.1 Why Default Allocation Is Not Neutral

Using `new` and `delete` appears simple, but it hides complexity.
Default allocation behavior depends on:

- The C++ runtime library

- Platform-specific heap implementations
- Debug vs release configuration
- Environment variables and OS policies

Across plugin boundaries, these assumptions no longer hold.

11.2 Memory Domains as Isolation Units

A memory domain is a conceptual boundary that owns its allocations.
In a plugin system:

- The host owns its memory domain
- Each plugin owns its own memory domain
- Memory must not freely cross domains

This model prevents allocator mismatch and simplifies reasoning.

11.3 Rule 7: Never Assume Allocator Compatibility

Even if both host and plugin use the same compiler, allocator compatibility is not guaranteed.
Therefore:

- Memory allocated in one module must not be freed in another
- STL containers must not cross the boundary
- Allocator state must remain internal

11.4 Custom Allocators Inside Plugins

Plugins are free to use custom allocators internally.

```
class plugin_allocator {
public:
    void* allocate(std::size_t size);
    void deallocate(void* ptr);
};
```

These allocators:

- Improve locality
- Reduce fragmentation
- Enable instrumentation

As long as allocation and deallocation remain within the plugin, safety is preserved.

11.5 Exposing Allocation Services Safely

If a plugin must expose allocation-related functionality, it should do so via explicit APIs:

```
extern "C" void* plugin_allocate(std::size_t size);
extern "C" void plugin_deallocate(void* ptr);
```

The ownership rules remain explicit and enforceable.

11.6 Allocator Injection from the Host

In some designs, the host may provide allocator callbacks:

```
struct host_allocator {
    void* (*allocate)(std::size_t);
    void (*deallocate)(void*);
};
```

This approach:

- Centralizes memory tracking
- Enables unified diagnostics
- Requires strict lifetime guarantees

Allocator injection must be negotiated during initialization.

11.7 Avoiding Global Allocator Overrides

Overriding global operators such as `operator new` inside a plugin is dangerous.
It may:

- Affect unrelated code
- Interfere with the host allocator
- Break third-party libraries

Prefer local allocators with explicit scope.

11.8 Memory Diagnostics and Instrumentation

Custom allocators enable powerful diagnostics:

- Leak detection
- Allocation tracing
- Lifetime analysis

These tools are invaluable in long-running plugin-based systems.

11.9 Performance Considerations

Allocator strategy affects:

- Throughput
- Latency
- Cache behavior

However, performance must never come at the cost of correctness or safety.

11.10 Design Principle: Memory Control Is Architecture

Memory allocation is not an implementation detail in plugin systems. It is a core architectural concern that must be designed, documented, and enforced.

11.11 Preparing for Dynamic Behavior

With allocator discipline established, we can now address one of the most requested but most dangerous features in plugin systems: hot reloading.
The next chapter examines what is possible, what is risky, and what should be avoided entirely.

Chapter 12

Hot Reloading: Possibilities, Risks, and Reality

Hot reloading is often presented as a productivity feature. In reality, it is a high-risk operation that pushes plugin systems to the edge of what is safely possible in C++.
This chapter explains what hot reloading really means, what can be done safely, and where the hard limits lie.

12.1 What Hot Reloading Actually Is

Hot reloading means replacing a plugin binary at runtime without restarting the host process. This typically involves:

- Unloading a shared library
- Replacing the binary on disk
- Loading the new version

- Rebinding symbols

Each of these steps carries risk.

12.2 Why Hot Reloading Is Dangerous

C++ was not designed with hot code replacement in mind.
Major hazards include:

- Dangling function pointers
- Stale vtables
- In-flight calls during unload
- Static state persistence
- Thread execution in unloaded code

Many systems attempt hot reload and quietly accumulate undefined behavior.

12.3 Rule 8: Hot Reloading Is Optional, Safety Is Not

Hot reloading must never be a foundational requirement.
A system that depends on hot reload correctness is fragile by design.
Hot reloading should be treated as:

- A best-effort optimization
- A development-time feature
- A strictly controlled operation

12.4 The Only Safe Reload Boundary

The only reliably safe hot reload boundary is between plugin instances, not inside them. This requires:

- No active plugin calls
- No background threads running plugin code
- No outstanding callbacks
- All instances destroyed

If these conditions cannot be guaranteed, hot reload must be refused.

12.5 Two-Phase Reload Protocol

A safe reload protocol follows two phases:

Phase 1: Quiescence

- Stop issuing new calls to the plugin
- Wait for in-flight calls to complete
- Shut down plugin-managed threads
- Destroy all plugin instances

Phase 2: Replacement

- Unload the plugin
- Replace the binary
- Load the new plugin
- Reinitialize explicitly

Skipping any step results in undefined behavior.

12.6 State Migration Is Not Hot Reloading

Migrating state from an old plugin version to a new one is a separate problem.
It requires:

- Explicit serialization formats
- Versioned state schemas
- Compatibility guarantees

Implicit state reuse is unsafe and unreliable.

12.7 Development-Time vs Production Reloading

Hot reloading is most useful during development:

- Faster iteration
- Reduced restart time
- Interactive debugging

In production systems, restart-based deployment is often safer and more predictable.

12.8 Operating System Limitations

Neither Windows nor Linux guarantee immediate code removal after unloading:

- Memory pages may remain mapped
- Deferred cleanup may occur
- Debuggers may interfere

Relying on precise unload timing is a mistake.

12.9 Alternatives to Hot Reloading

Safer alternatives include:

- Process isolation
- Worker subprocesses
- Restartable services
- Message-based boundaries

These approaches trade convenience for robustness.

12.10 Design Principle: Prefer Restartability Over Reloadability

A system that can restart cleanly is easier to reason about than one that tries to replace itself while running.
Hot reloading should be treated as an optimization, never as a requirement.

12.11 Preparing for a Complete Example

With all architectural pieces in place, we are now ready to assemble a complete, minimal, professional plugin system.
The next chapters present a full case study: host application, plugin implementation, failure handling, and analysis.

Chapter 13

Case Study Overview: A Minimal Professional Plugin System

After establishing architectural rules and safety constraints, it is time to assemble them into a complete, working system.
This case study presents a minimal yet professional plugin architecture that respects ABI stability, memory ownership, exception safety, concurrency rules, and lifetime management across Windows and Linux.
The goal is not to build a feature-rich framework, but to demonstrate a *correct* foundation that can be extended safely.

13.1 Design Goals of the Case Study

The system is designed with the following goals:

- Cross-platform compatibility (Windows and Linux)
- Strict ABI stability

- Explicit ownership and lifetimes
- No exception propagation across boundaries
- No STL types at the boundary
- Predictable loading and unloading

Every design decision can be traced back to principles established in earlier chapters.

13.2 System Components

The system consists of three clearly separated components:

1. The host application
2. The plugin binary
3. The shared ABI interface

Each component has a single responsibility and a well-defined role.

13.3 The ABI Interface Layer

The ABI interface is the only contract shared between host and plugin.
It is defined in a single header that contains:

- Opaque types
- Function declarations with C linkage
- Version identifiers
- Error codes

This header must remain stable over time.

13.4 Why the ABI Header Is Isolated

The ABI header:

- Avoids including standard library headers
- Avoids templates and inline functions
- Avoids compiler-specific extensions

It is intentionally boring.
Boring interfaces are stable interfaces.

13.5 Host Responsibilities

The host application is responsible for:

- Loading and unloading the plugin
- Validating the ABI version
- Managing plugin lifetimes
- Serializing access when required
- Handling errors and logging

The host never assumes internal plugin behavior.

13.6 Plugin Responsibilities

The plugin is responsible for:

- Implementing the ABI contract
- Managing its internal memory and threads
- Handling errors internally
- Exposing only stable functionality

The plugin never assumes host implementation details.

13.7 Error Handling Strategy

All errors are reported explicitly using:

- Integer error codes
- Null pointers for creation failures
- Explicit error query functions

Exceptions are fully contained inside the plugin.

13.8 Concurrency Model

The case study uses a conservative concurrency model:

- Plugin instances are not reentrant
- The host serializes calls
- No background threads are started implicitly

This model simplifies reasoning and testing.

13.9 Lifetime Model

The lifetime model follows strict ordering:

1. Load plugin
2. Query version
3. Create plugin instance
4. Use plugin
5. Destroy instance
6. Unload plugin

No step is skipped or reordered.

13.10 Why This System Is Intentionally Minimal

Many plugin examples hide complexity behind frameworks or macros.
This case study does the opposite:

- Every rule is explicit
- Every responsibility is visible
- Every failure mode is handled

The result is a system that can be trusted and evolved.

13.11 What Comes Next

The following chapters will implement this system step by step:

- The shared ABI header
- The plugin implementation
- The host loader
- Failure analysis and testing

Each part will be minimal, complete, and correct.

Chapter 14

The Shared ABI Header

The shared ABI header is the foundation of the entire plugin system.
It is the only file that both the host application and the plugin are allowed to include. Every symbol declared in this header becomes part of a long-term binary contract.
If this header is unstable, unclear, or careless, no amount of defensive coding elsewhere can compensate.

14.1 Purpose of the ABI Header

The ABI header defines:

- What the plugin exposes
- How the host may interact with it
- Which guarantees are provided
- Which responsibilities are enforced

Anything not explicitly declared here must be considered inaccessible.

14.2 Strict Design Constraints

A correct ABI header must obey strict constraints:

- No C++ standard library headers
- No templates
- No inline functions
- No exceptions
- No RTTI
- No compiler-specific extensions

These constraints are non-negotiable.

14.3 C Linkage and Symbol Stability

All exported functions use C linkage to avoid name mangling:

```
#ifdef __cplusplus
extern "C" {
#endif

// declarations

#ifdef __cplusplus
}
#endif
```

This guarantees predictable symbol names across compilers.

14.4 Opaque Types

All plugin-owned objects are opaque to the host:

```
struct plugin_api;
```

The host cannot:

- Inspect object layout
- Allocate instances
- Destroy instances incorrectly

This enforces ownership by construction.

14.5 Version Declaration

The ABI version is defined explicitly:

```
#define PLUGIN_API_VERSION 1

int plugin_api_version();
```

The host checks this value before calling any other function.

14.6 Error Codes

Error handling is explicit and stable:

```
enum plugin_error {
    PLUGIN_OK = 0,
    PLUGIN_ERROR_INVALID_ARGUMENT = 1,
    PLUGIN_ERROR_INTERNAL = 2
};
```

Error codes must never change meaning within the same major version.

14.7 Lifecycle Functions

Object lifetime is controlled through explicit functions:

```
plugin_api* plugin_create();
void plugin_destroy(plugin_api*);
```

No implicit construction or destruction is permitted.

14.8 Core Functional Interface

The functional surface of the plugin is minimal:

```
int plugin_process(plugin_api*, int input, int* output);
```

All inputs and outputs use plain data types.

14.9 Optional Initialization Hooks

Optional hooks are explicit:

```
void plugin_initialize(plugin_api*, const void* host_api);
void plugin_shutdown(plugin_api*);
```

These functions must be safe to call exactly once.

14.10 Complete Example ABI Header

The following is a complete, minimal ABI header:

```
/* plugin_api.h */
#ifndef PLUGIN_API_H
#define PLUGIN_API_H

#ifdef __cplusplus
extern "C" {
#endif

#define PLUGIN_API_VERSION 1

struct plugin_api;

enum plugin_error {
    PLUGIN_OK = 0,
    PLUGIN_ERROR_INVALID_ARGUMENT = 1,
    PLUGIN_ERROR_INTERNAL = 2
};

int plugin_api_version(void);

plugin_api* plugin_create(void);
void plugin_destroy(plugin_api*);

int plugin_process(plugin_api*, int input, int* output);

void plugin_shutdown(plugin_api*);

#ifdef __cplusplus
}
```

```
#endif

#endif /* PLUGIN_API_H */
```

14.11 Why This Header Is Intentionally Small

A small ABI surface:

- Is easier to version
- Is harder to misuse
- Survives change

Most plugin systems fail because their ABI grows without discipline.

14.12 Design Principle: Treat the ABI Header as Law

Once published, the ABI header becomes immutable.
Changes require:

- Careful versioning
- Backward compatibility planning
- Explicit migration paths

Breaking the ABI is not a refactor. It is a breaking change.

14.13 What Comes Next

With the ABI defined, we can now implement the plugin itself.

The next chapter builds a complete plugin implementation that obeys every rule established so far.

Chapter 15

Implementing the Plugin

With the ABI contract defined, we can now implement the plugin itself. This implementation must satisfy two competing goals:

- Internally, it should use modern C++ idioms and structure.
- Externally, it must present a minimal, stable, C-compatible interface.

This separation is essential. A professional plugin is modern on the inside and conservative on the outside.

15.1 Internal Plugin Structure

Internally, the plugin is free to use:

- Classes and RAII
- Exceptions
- Standard library containers

- Modern language features

None of these must escape the plugin boundary.

```
// plugin_internal.h
#pragma once

class PluginImpl {
public:
    int process(int input);

private:
    int internal_state_ = 0;
};
```

This class is never visible to the host.

15.2 Internal Logic and Exception Use

Exceptions are allowed internally to simplify logic:

```
// plugin_internal.cpp
#include "plugin_internal.h"
#include <stdexcept>

int PluginImpl::process(int input) {
    if (input < 0) {
        throw std::invalid_argument("negative input");
    }

    internal_state_ += input;
    return internal_state_;
}
```

Exception-based control flow remains confined within the plugin.

15.3 Bridging ABI and Implementation

The ABI layer acts as a thin wrapper around the internal implementation.

```
// plugin_api_impl.cpp
#include "plugin_api.h"
#include "plugin_internal.h"
#include <new>

struct plugin_api {
    PluginImpl impl;
};
```

The opaque `plugin_api` structure now contains the real implementation.

15.4 Plugin Creation

```
extern "C" plugin_api* plugin_create(void) {
    try {
        return new plugin_api{};
    } catch (...) {
        return nullptr;
    }
}
```

All exceptions are caught and translated into failure.

15.5 Plugin Destruction

```
extern "C" void plugin_destroy(plugin_api* api) {
    delete api;
}
```

Allocation and deallocation occur in the same module.

15.6 Implementing the Core Function

```
extern "C" int plugin_process(plugin_api* api, int input, int* output) {
    if (!api || !output) {
        return PLUGIN_ERROR_INVALID_ARGUMENT;
    }

    try {
        *output = api->impl.process(input);
        return PLUGIN_OK;
    } catch (...) {
        return PLUGIN_ERROR_INTERNAL;
    }
}
```

No exception crosses the boundary. All failures are explicit.

15.7 Version Reporting

```
extern "C" int plugin_api_version(void) {
    return PLUGIN_API_VERSION;
}
```

This function must never fail.

15.8 Optional Initialization and Shutdown

```
extern "C" void plugin_initialize(plugin_api*, const void*) {
```

```
    // optional: register host services
}

extern "C" void plugin_shutdown(plugin_api*) {
    // optional: release resources
}
```

Initialization and shutdown are explicit and predictable.

15.9 Threading Considerations in This Implementation

This implementation assumes:

- Single-threaded access per plugin instance
- No background threads
- Host-side serialization

These assumptions must be documented and enforced.

15.10 Why This Implementation Is Safe

This plugin implementation:

- Contains all modern C++ features internally
- Exposes only C-compatible functions
- Controls memory ownership strictly
- Prevents exception leakage
- Respects ABI stability

It is minimal, but it is correct.

15.11 What Comes Next

With the plugin implemented, the final missing piece is the host application.
The next chapter implements a robust host loader that validates, loads, uses, and unloads the plugin safely on both Windows and Linux.

Chapter 16

Implementing the Host Loader

The host application is the final authority in a plugin system. It decides which plugins are loaded, when they are invoked, and when they are unloaded.
A careless host can violate every safety rule established so far. A disciplined host enforces them automatically.
This chapter implements a robust, cross-platform host loader that treats plugin interaction as a controlled and potentially failing operation.

16.1 Host Responsibilities Revisited

The host is responsible for:

- Loading the shared library
- Resolving required symbols
- Validating ABI compatibility
- Managing plugin lifetimes

- Serializing access when required
- Handling all failures explicitly

The host must assume that plugins can fail.

16.2 Platform Abstraction

To keep the host portable, dynamic loading is abstracted behind a small wrapper.

```
// dynamic_library.h
#pragma once

class DynamicLibrary {
public:
    DynamicLibrary() = default;
    ~DynamicLibrary();

    bool load(const char* path);
    void unload();

    void* symbol(const char* name) const;

private:
    void* handle_ = nullptr;
};
```

This abstraction hides platform-specific APIs.

16.3 Platform-Specific Implementation

Windows Implementation

```
// dynamic_library_win.cpp
#include "dynamic_library.h"
#include <windows.h>

bool DynamicLibrary::load(const char* path) {
    handle_ = LoadLibraryA(path);
    return handle_ != nullptr;
}

void DynamicLibrary::unload() {
    if (handle_) {
        FreeLibrary(static_cast<HMODULE>(handle_));
        handle_ = nullptr;
    }
}

void* DynamicLibrary::symbol(const char* name) const {
    return handle_ ? reinterpret_cast<void*>(
        GetProcAddress(static_cast<HMODULE>(handle_), name)) : nullptr;
}

DynamicLibrary::~DynamicLibrary() {
    unload();
}
```

Linux Implementation

```
// dynamic_library_linux.cpp
#include "dynamic_library.h"
#include <dlfcn.h>
```

```
bool DynamicLibrary::load(const char* path) {
    handle_ = dlopen(path, RTLD_NOW);
    return handle_ != nullptr;
}

void DynamicLibrary::unload() {
    if (handle_) {
        dlclose(handle_);
        handle_ = nullptr;
    }
}

void* DynamicLibrary::symbol(const char* name) const {
    return handle_ ? dlsym(handle_, name) : nullptr;
}

DynamicLibrary::~DynamicLibrary() {
    unload();
}
```

16.4 Defining Required Plugin Symbols

The host defines exact function signatures matching the ABI header:

```
using version_fn = int (*)();
using create_fn  = plugin_api* (*)();
using destroy_fn = void (*)(plugin_api*);
using process_fn = int (*)(plugin_api*, int, int*);
```

These types must never drift from the ABI definition.

16.5 Loading and Validating the Plugin

```
bool load_plugin(const char* path) {
    DynamicLibrary lib;
    if (!lib.load(path)) {
        return false;
    }

    auto version = reinterpret_cast<version_fn>(
        lib.symbol("plugin_api_version"));

    if (!version || version() != PLUGIN_API_VERSION) {
        return false;
    }

    return true;
}
```

No plugin code is executed before version validation.

16.6 Creating and Using a Plugin Instance

```
plugin_api* api = create();
if (!api) {
    // creation failed
}

int result = 0;
int err = process(api, 10, &result);
if (err != PLUGIN_OK) {
    // handle error
}
```

All return values are checked. Nothing is assumed.

16.7 Destruction and Unloading

```
destroy(api);
lib.unload();
```

The destruction order is enforced explicitly.

16.8 Serializing Access

If the plugin is not documented as thread-safe, the host must serialize calls:

```
#include <mutex>

std::mutex plugin_mutex;

void safe_call(plugin_api* api) {
    std::lock_guard<std::mutex> lock(plugin_mutex);
    // call plugin functions
}
```

The host never relies on undocumented thread safety.

16.9 Failure Handling Strategy

Every failure results in one of:

- Plugin rejection
- Plugin disablement

- Explicit error reporting

Silent fallback behavior is avoided.

16.10 Why This Host Is Robust

This host loader:

- Validates everything
- Enforces lifetime rules
- Contains platform-specific behavior
- Prevents undefined behavior by design

The plugin system is now complete.

16.11 What Comes Next

With both plugin and host implemented, we can analyze failure modes, testing strategies, and common mistakes observed in real systems.
The next chapter provides a structured failure analysis of the system.

Chapter 17

Failure Analysis and Common Mistakes

Even well-designed plugin systems fail when assumptions drift, discipline erodes, or new contributors unknowingly violate architectural rules.
This chapter catalogs the most common failure modes observed in real-world plugin-based C++ systems and explains how the design presented in this booklet prevents them.

17.1 Why Failure Analysis Matters

Most failures in plugin systems:

- Do not appear immediately
- Cannot be reproduced reliably
- Are blamed on unrelated components

Failure analysis provides defensive knowledge that prevents repeating the same mistakes.

17.2 Mistake 1: Passing STL Types Across the Boundary

This is the most frequent and dangerous mistake.
Symptoms include:

- Random crashes during destruction
- Memory corruption detected far from the cause
- Inconsistent behavior across machines

Root cause:

- STL layout and allocators are not ABI-stable

Prevention:

- Use plain data types at the boundary
- Convert to STL types internally

17.3 Mistake 2: Letting Exceptions Escape

Allowing exceptions to cross the plugin boundary results in:

- Immediate process termination
- Corrupted stack unwinding
- Undefined behavior during cleanup

Prevention:

- Catch all exceptions at the boundary
- Translate failures into error codes

17.4 Mistake 3: Implicit Ownership Assumptions

Implicit ownership leads to:

- Double deletion
- Use-after-free
- Leaks during error paths

Prevention:

- Explicit create/destroy functions
- Opaque handles
- Clear lifetime documentation

17.5 Mistake 4: Ignoring Versioning Early

Delaying versioning decisions results in:

- Frozen interfaces
- Forced simultaneous updates
- Emergency ABI breaks

Prevention:

- Version the ABI from day one
- Enforce version checks at load time

17.6 Mistake 5: Assuming Thread Safety

Assuming plugins are thread-safe by default leads to:

- Data races
- Deadlocks
- Heisenbugs

Prevention:

- Document threading guarantees
- Serialize access unless proven safe

17.7 Mistake 6: Relying on Global State

Global state inside plugins causes:

- Uncontrolled initialization order
- Difficult teardown
- Unpredictable interactions

Prevention:

- Use instance-based design
- Tie state to explicit lifetimes

17.8 Mistake 7: Unsafe Hot Reloading

Unsafe reload attempts result in:

- Dangling function pointers
- Execution of unloaded code
- Silent corruption

Prevention:

- Require full quiescence
- Prefer restart-based deployment

17.9 Mistake 8: Incomplete Failure Handling

Ignoring partial failures leads to:

- Resource leaks
- Inconsistent state
- Accumulated technical debt

Prevention:

- Fail early
- Clean up deterministically

17.10 Mistake 9: Treating the ABI as an Implementation Detail

When the ABI is treated casually:

- Changes slip in unnoticed
- Compatibility breaks silently
- Trust is lost

Prevention:

- Treat the ABI header as immutable
- Review changes with extreme care

17.11 Mistake 10: Overengineering the Interface

Excessive abstraction leads to:

- Bloated ABI surfaces
- Increased maintenance burden
- Reduced stability

Prevention:

- Keep the ABI minimal
- Build abstractions on the host side

17.12 Design Principle: Most Failures Are Predictable

Very few plugin failures are novel. Most are well-known consequences of violating fundamental rules.
A disciplined architecture does not eliminate bugs, but it prevents entire categories of failure.

17.13 What Comes Next

With failure modes understood, the final chapters will focus on:

- Testing strategies
- Maintenance guidelines
- Final conclusions and appendices

These sections complete the booklet as a long-term professional reference.

Chapter 18

Testing Plugin Systems Safely

Testing plugin systems requires more discipline than testing ordinary C++ libraries. Dynamic loading, binary boundaries, and independent compilation introduce failure modes that unit tests alone cannot detect.
This chapter presents a structured testing strategy that validates correctness, safety, and long-term stability.

18.1 Why Plugin Testing Is Different

Plugin systems differ from statically linked code because:

- Binaries are built independently
- Interfaces are validated at runtime
- Errors may appear only under specific loading orders
- ABI mismatches cannot be detected by the compiler

Testing must therefore include runtime and integration layers.

18.2 Layered Testing Strategy

A professional plugin system is tested at multiple layers:

1. Internal unit tests (inside the plugin)
2. ABI-level tests
3. Host-plugin integration tests
4. Failure and stress tests

Each layer detects different classes of defects.

18.3 Internal Plugin Unit Tests

Inside the plugin, normal C++ unit testing applies.

- Test internal classes directly
- Use exceptions freely
- Validate edge cases aggressively

These tests never involve dynamic loading.

18.4 ABI-Level Contract Tests

ABI-level tests treat the plugin as a black box.

- Load the plugin dynamically
- Resolve symbols explicitly

- Call functions through function pointers
- Validate error codes and outputs

This verifies the binary contract directly.

18.5 Example ABI Test Harness

```
bool test_plugin_contract(const char* path) {
    DynamicLibrary lib;
    if (!lib.load(path)) return false;

    auto version = reinterpret_cast<version_fn>(
        lib.symbol("plugin_api_version"));
    if (!version || version() != PLUGIN_API_VERSION) return false;

    auto create = reinterpret_cast<create_fn>(
        lib.symbol("plugin_create"));
    auto destroy = reinterpret_cast<destroy_fn>(
        lib.symbol("plugin_destroy"));

    if (!create || !destroy) return false;

    plugin_api* api = create();
    if (!api) return false;

    destroy(api);
    return true;
}
```

This test detects missing symbols and basic incompatibility.

18.6 Host-Plugin Integration Tests

Integration tests exercise real usage patterns:

- Load plugins in different orders
- Invoke functions repeatedly
- Simulate failures
- Unload and reload plugins

These tests often reveal lifetime and shutdown bugs.

18.7 Testing Failure Paths

Failure paths are as important as success paths.
Test scenarios include:

- Invalid arguments
- Null pointers
- Partial initialization
- Rejected versions

Every failure must be deterministic and recoverable.

18.8 Stress and Longevity Testing

Long-running tests reveal issues that short tests miss:

- Memory leaks
- Heap fragmentation
- Resource exhaustion
- Rare race conditions

Plugins should be loaded, used, and unloaded repeatedly over long periods.

18.9 Threading Tests

If the plugin supports concurrency:

- Test concurrent access explicitly
- Use high iteration counts
- Validate documented guarantees

If concurrency is not supported, verify that misuse fails safely.

18.10 Testing Version Compatibility

Maintain test suites for:

- Old plugins with new hosts
- New plugins with old hosts
- Mixed plugin versions

These tests protect backward compatibility.

18.11 Automating Plugin Tests

Automation is essential:

- Build plugins independently
- Run tests on clean environments
- Test multiple compiler versions
- Test multiple platforms

Manual testing is insufficient for ABI safety.

18.12 Design Principle: Test the Boundary, Not Just the Code

Most plugin failures occur at boundaries.
Testing must reflect that reality by exercising:

- Binary loading
- Interface contracts
- Lifetime transitions

18.13 What Comes Next

With testing strategies defined, we can now conclude the booklet and provide long-term maintenance guidance.
The next chapter presents the final conclusion, followed by appendices and references that summarize and reinforce the material.

Conclusion

Writing safe plugins and DLLs in Modern C++ is not about clever techniques or esoteric language tricks. It is about discipline, restraint, and respect for binary reality.
C++ gives developers extraordinary expressive power, but that power comes with non-negotiable constraints. At binary boundaries, the language no longer protects you. The compiler cannot enforce contracts across independently built modules, and undefined behavior becomes easy to introduce and difficult to detect.
This booklet has shown that safe plugin systems are achievable when they are designed intentionally.
The core lessons can be summarized as follows:

- Treat the ABI as a long-term contract, not an implementation detail.
- Never allow ownership, exceptions, or STL types to cross plugin boundaries.
- Make lifetimes explicit and enforceable.
- Assume failure and design for it explicitly.
- Prefer simplicity and predictability over cleverness.

Modern C++ features remain invaluable inside plugin implementations. RAII, exceptions, strong types, and standard library containers enable clarity and correctness. The key is containment: modern features belong inside modules, not across binary boundaries.

A professional plugin architecture is one that ages gracefully. It allows plugins and hosts to evolve independently, failures to be diagnosed clearly, and changes to be introduced without fear of destabilizing the system.
If this booklet has one central message, it is this:

> **Safe plugin systems are not accidental. They are designed.**

By respecting the rules outlined here, you can build systems that remain correct, maintainable, and trustworthy long after their initial release.

Appendices

ABI Safety Checklist

This checklist summarizes the non-negotiable rules for safe plugin and DLL development. Any violation of these rules introduces undefined behavior.

ABI Boundary Rules

- Do not expose C++ classes across the boundary
- Do not expose templates or inline functions
- Do not expose virtual functions or vtables
- Use C linkage for all exported symbols

Type Rules

- Use only plain data types at the boundary
- Never pass STL containers or `std::string`
- Avoid compiler-specific types and extensions

Exception Rules

- Never allow exceptions to cross the boundary
- Catch all exceptions at the ABI layer
- Translate failures into explicit error codes

Memory Rules

- The allocator that allocates must deallocate
- Do not share ownership implicitly
- Use explicit create/destroy functions

Threading Rules

- Do not share synchronization primitives
- Document threading guarantees explicitly
- Serialize access unless thread safety is guaranteed

Do and Don’t Tables

Interface Design

Table 13-1: Plugin Interface Design: Do and Don't Guidelines

Do	Don't
Use opaque handles	Export C++ classes
Version the ABI	Assume compatibility
Keep interfaces small	Expose internal details
Design for explicit ownership	Rely on implicit lifetimes
Validate versions at load time	Load plugins blindly
Fail fast on incompatibility	Attempt silent fallback
Document threading guarantees	Assume thread safety
Keep the ABI header minimal	Add convenience helpers to the ABI

Memory Management

Table 13-2: Memory Ownership: Do and Don't Guidelines

Do	Don't
Pair allocation and deallocation	Delete across module boundaries
Keep ownership explicit	Share raw pointers freely
Use instance-based lifetimes	Rely on global state
Destroy objects before unloading	Unload plugins with live objects
Keep allocation inside the owner	Assume allocator compatibility
Use explicit create/destroy APIs	Mix `new` and `delete` across DLLs

Error Handling

Table 13-3: Error Handling and Diagnostics: Do and Don't Guidelines

Do	Don't
Return explicit error codes	Throw exceptions across boundaries
Fail fast on invalid states	Hide or ignore failures
Log errors explicitly	Assume success by default
Validate inputs aggressively	Trust external callers
Report failure deterministically	Rely on undefined behavior
Abort unsafe operations early	Attempt partial recovery silently

Common Undefined Behavior Traps

This appendix lists common traps that appear safe but are not.

Trap: Matching Compilers Means Matching ABI

Using the same compiler version does not guarantee ABI compatibility. Build flags, runtime libraries, and standard library versions still matter.

Trap: Smart Pointers Solve Ownership

Smart pointers manage ownership only within a single binary. They do not solve cross-module allocation problems.

Trap: Catching Exceptions at the Top Level Is Enough

If an exception escapes even once, behavior is already undefined. Exception safety must be enforced by interface design.

Trap: Hot Reloading Is Just Unloading and Reloading

Without strict quiescence guarantees, hot reloading executes unloaded code.

Plugin Design Templates

This appendix provides minimal templates that can be reused safely.

Minimal ABI Header Template

```
#ifndef PLUGIN_API_H
#define PLUGIN_API_H

#ifdef __cplusplus
extern "C" {
#endif

#define PLUGIN_API_VERSION 1

struct plugin_api;

int plugin_api_version(void);

plugin_api* plugin_create(void);
void plugin_destroy(plugin_api*);

#ifdef __cplusplus
```

```
}
#endif

#endif
```

Minimal Host Load Sequence

```
load_library();
resolve_symbols();
check_version();
create_instance();
use_plugin();
destroy_instance();
unload_library();
```

Minimal Plugin Lifecycle

```
initialize();
process();
shutdown();
```

Maintenance and Evolution Guidelines

Long-Term Maintenance Rules

- Never change the ABI without a major version bump
- Test old plugins with new hosts continuously
- Treat plugin bugs as architectural signals

Code Review Focus Areas

- Boundary type usage
- Ownership clarity
- Lifetime enforcement
- Failure handling paths

When to Redesign

A redesign is justified when:

- The ABI surface has grown uncontrollably
- Compatibility hacks dominate the code
- Correctness depends on undocumented behavior

Redesign early rather than accumulate technical debt.

References

The following references informed the principles, rules, and examples presented in this booklet. They were selected for their long-term relevance, technical accuracy, and authoritative status.

C++ Language and Standards

- ISO/IEC 14882:2020 — Programming Languages — C++ (C++20)
- ISO/IEC 14882:2023 — Programming Languages — C++ (C++23)
- ISO C++ Core Guidelines

Application Binary Interface (ABI)

- System V Application Binary Interface Specification
- Microsoft x64 Software Conventions
- Microsoft C++ ABI and Runtime Library Documentation
- Itanium C++ ABI (for name mangling and exception handling concepts)

Operating Systems and Dynamic Loading

- Windows Dynamic-Link Library (DLL) Architecture
- Windows Loader and Module Lifetime Semantics
- POSIX Dynamic Linking Interfaces (`dlopen`, `dlsym`, `dlclose`)
- ELF Object File Format and Loader Behavior

Memory Management and Allocation

- C++ Object Lifetime and Storage Duration Rules
- Runtime Heap and Allocator Design in Modern Systems
- Memory Fragmentation and Long-Running Process Behavior

Concurrency and Threading

- C++ Memory Model and Happens-Before Relationships
- Platform Threading Models (Windows and POSIX)
- Deadlock and Race Condition Analysis in Modular Systems

Software Architecture and Engineering Practice

- Large-Scale C++ System Design Case Studies
- Defensive Programming and Fail-Fast Design Principles

- Long-Term Maintenance Strategies for Modular Systems

Testing and Reliability

- Binary Compatibility Testing Techniques
- Integration and Longevity Testing for Plugin-Based Systems
- Regression Testing Strategies for ABI Stability

General Philosophy

- Zero-Overhead Principle in C++
- Explicit Ownership and Deterministic Lifetime Design
- Simplicity as a Reliability Strategy

These references represent stable foundations rather than transient implementations. While tools and frameworks evolve, the underlying principles governing binary safety, ownership, and modular design remain constant.
This booklet is intended to serve as a durable reference grounded in those principles.