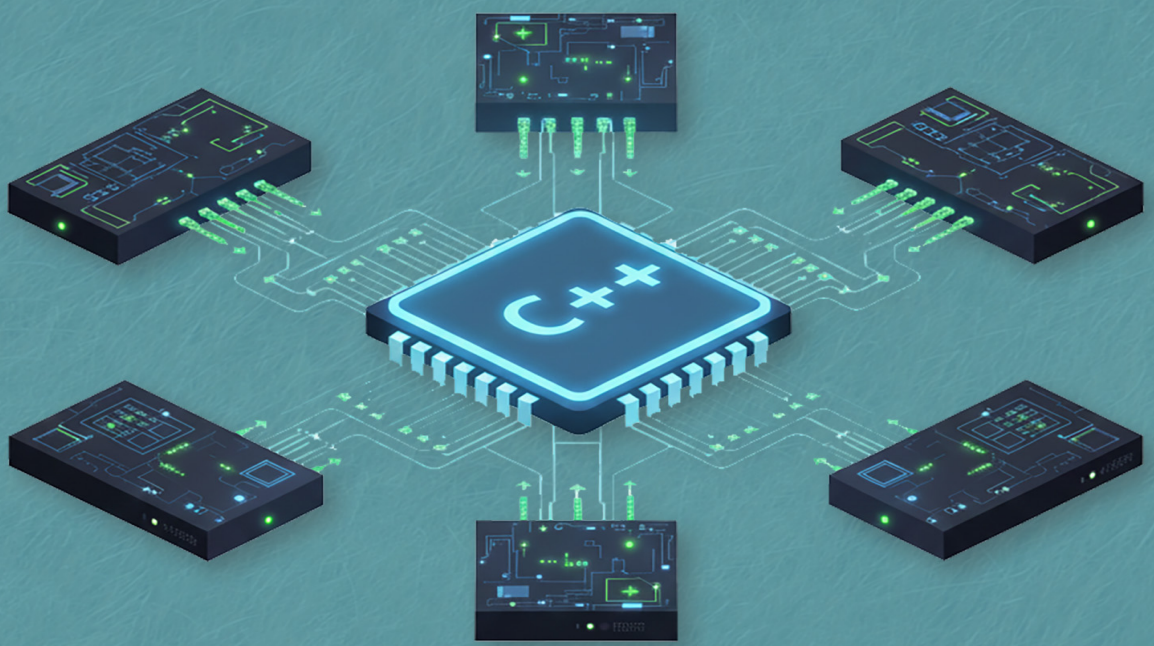
https://simplifycpp.org

Dependency Injection in C++

A Version Designed for Large-Scale Projects



Prepared by Ayman alheraki

Dependency Injection in C++

A Version Designed for Large-Scale Projects

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Author's Introduction

This booklet is written for engineers working on large, long-lived C++ systems—systems where architectural mistakes are far more costly than implementation bugs.
In such systems, complexity rarely comes from algorithms alone. It emerges from uncontrolled dependencies, unclear ownership, and construction logic scattered across the codebase. These issues accumulate quietly, until testing becomes fragile, refactoring becomes risky, and change becomes expensive.
Dependency Injection is often misunderstood in C++. Some dismiss it as unnecessary abstraction. Others attempt to replicate container-heavy designs from managed languages, ignoring C++ fundamentals. Both perspectives miss the core issue.
Dependency Injection in C++ is not about frameworks. It is about architectural control. When grounded in explicit construction, deterministic lifetime, and RAII, Dependency Injection becomes a discipline that restores clarity rather than obscures it. It makes dependencies visible, forces ownership decisions to be honest, and centralizes construction where it belongs, Used thoughtfully, Dependency Injection allows C++ systems to evolve over time without collapsing under their own weight. Used indiscriminately, it becomes another source of complexity.
This booklet argues for the former—Dependency Injection as a deliberate, C++-native design discipline guided by judgment, restraint, and long-term thinking.

Ayman Alheraki

Preface

C++ gives developers complete control over object creation and destruction. This control is one of the language's defining strengths, and also one of its greatest sources of risk.
Unlike many modern programming environments, C++ does not impose an architectural style. It does not require dependency containers, it does not enforce inversion of control, and it does not prevent developers from constructing objects wherever and however they choose.
This freedom enables exceptional performance, precise resource management, and systems that can operate close to hardware. At the same time, it places the full burden of architectural discipline on the engineer.

From Convenience to Architectural Debt

In small programs, direct construction appears harmless. Creating objects locally feels natural, readable, and efficient.
As systems grow, however, the same habit quietly transforms into an architectural trap.
Classes begin to construct their own collaborators. Configuration logic spreads. Hidden assumptions about lifetime and ownership emerge. Dependencies stop being declared and start being discovered.
At this stage:

- Dependencies become implicit

- Testing becomes fragile
- Refactoring becomes dangerous

These problems do not arise suddenly. They accumulate gradually, often unnoticed, until the cost of change exceeds the cost of development itself.

The Reality of C++: Ownership and Lifetime

Unlike garbage-collected environments, C++ does not shield developers from reality.
Every object has:

- A clearly defined lifetime
- A concrete ownership model
- A measurable construction and destruction cost

There is no runtime system that silently cleans up architectural mistakes. There is no implicit ownership transfer. There is no universal allocator that hides cost.
Any design approach in C++ that ignores ownership, lifetime, or construction cost is not merely incomplete — it is fundamentally flawed.
Dependency Injection in C++ must therefore be rooted in these realities, not abstracted away from them.

Why Dependency Injection Is Often Misunderstood in C++

Dependency Injection is frequently introduced to C++ developers through examples and frameworks originating in other ecosystems.
In those contexts, Dependency Injection is often:

- Container-driven
- Runtime-reflective
- Allocation-heavy
- Detached from explicit lifetime control

When these ideas are transplanted directly into C++, the results are predictable: over-engineered systems, unnecessary indirection, and performance concerns that could have been avoided.
This booklet takes a different approach.
Dependency Injection is treated here not as a framework feature, but as a design discipline that emerges naturally from well-understood C++ principles.

Dependency Injection as a Design Discipline

In this booklet, Dependency Injection is presented as a way to enforce clarity, honesty, and responsibility in design.
It is rooted in:

- RAII and deterministic lifetime
- Explicit interfaces
- Constructor-based initialization
- Compile-time reasoning

The goal is not abstraction for its own sake. The goal is not maximum flexibility. The goal is not architectural fashion.
The goal is *controlled flexibility* — flexibility that can be reasoned about, tested, and maintained over years of continuous change.

Scaling Across Teams and Time

Large-scale C++ systems are rarely built by a single individual. They are maintained by teams, often across multiple generations of engineers.
Design decisions made early become constraints later.
A system that hides dependencies forces every new contributor to rediscover architectural intent. A system that exposes dependencies documents itself through its structure.
Dependency Injection, applied with restraint, helps systems scale not only technically, but cognitively.

What This Booklet Intentionally Avoids

This booklet intentionally avoids:

- Promoting a specific Dependency Injection framework
- Presenting Dependency Injection as mandatory
- Treating abstraction as an automatic good

Instead, it emphasizes judgment.
You will find guidance on:

- When Dependency Injection is appropriate
- When it is unnecessary
- When it is actively harmful

Knowing the difference is a hallmark of professional maturity in C++.

How to Read This Booklet

This booklet is not meant to be consumed quickly.
It is meant to be read:

- By architects designing long-lived systems
- By engineers refactoring legacy code
- By teams seeking consistency and clarity

The examples favor realism over brevity. The explanations favor precision over simplicity. Every design choice is motivated by real failure modes observed in large, production C++ systems.

A Final Note Before Beginning

Dependency Injection will not make C++ easier. It will make design choices more visible.
That visibility can feel uncomfortable. It exposes assumptions, forces decisions, and removes convenient shortcuts.
This discomfort is not a weakness. It is the price of building systems that endure.
The chapters that follow are an invitation to embrace that discipline, not as a rule, but as a responsibility.

Chapter 1

Foundations of Dependency Injection in C++

1.1 What Dependency Injection Actually Means

Dependency Injection is the separation of two concerns:

- Object construction
- Object behavior

A class should define what it needs, not how those needs are created.
In C++, this separation is not enforced by the language. It must be enforced by design.

1.2 Implicit Dependencies as Architectural Debt

Consider the following design:

```
class Logger {
```

```
public:
    void log(const std::string& message);
};

class OrderService {
    Logger logger;
public:
    void process() {
        logger.log("processing order");
    }
};
```

At first glance, this code is simple. In a large system, it is dangerous.
The service is permanently coupled to a concrete logger. Testing requires the real implementation. Configuration is impossible without modification. Change propagates outward.
This is architectural debt.

1.3 Separating Construction from Behavior

Dependency Injection removes construction responsibility from the class itself:

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

class OrderService {
    Logger& logger;
public:
    explicit OrderService(Logger& l) : logger(l) {}
```

```
    void process() {
        logger.log("processing order");
    }
};
```

The class now depends on an interface, not a concrete type. Construction happens elsewhere. Behavior remains focused.
This distinction is the foundation upon which scalable C++ architectures are built.

1.4 Tight Coupling and Its Long-Term Consequences

In large C++ systems, tight coupling rarely appears as a single, obvious mistake. Instead, it accumulates gradually as convenience-driven decisions are repeated across the codebase.
When classes directly construct their dependencies, several long-term consequences emerge:

- Changes in low-level components propagate unexpectedly
- Unit testing requires complex setup or full system initialization
- Compile times increase due to unnecessary header inclusion
- Architectural boundaries erode over time

These effects are not theoretical. They manifest as slower development velocity, increased defect rates, and resistance to refactoring.
Dependency Injection addresses these issues by making dependencies explicit and by enforcing boundaries through construction.

1.5 Dependency Injection versus Service Locator

A common anti-pattern in C++ systems is the use of a global service locator.

```
class ServiceLocator {
public:
    static Logger& logger();
};
```

Usage appears convenient:

```
class OrderService {
public:
    void process() {
        ServiceLocator::logger().log("processing order");
    }
};
```

While this removes direct construction, it introduces hidden dependencies.
The class no longer declares what it depends on. The dependency graph becomes implicit.
Testing requires global state manipulation. Thread safety becomes difficult to reason about.
From an architectural standpoint, Service Locator replaces explicit coupling with invisible coupling.
Dependency Injection, by contrast, forces dependencies to be visible at construction time.
Visibility is not a weakness. It is a design strength.

1.6 Why Constructor Injection Is Fundamental in C++

C++ provides deterministic object lifetime. Objects are fully initialized before use and destroyed predictably.
Constructor injection aligns perfectly with these guarantees.

```
class PaymentProcessor {
    Logger& logger;
    Database& database;
```

```
public:
    PaymentProcessor(Logger& l, Database& d)
        : logger(l), database(d) {}
};
```

This design communicates several critical facts:

- The processor cannot exist without its dependencies
- Ownership is external and explicit
- The object is always fully initialized

There is no possibility of partially constructed objects. There is no deferred wiring. The compiler enforces correctness.

1.7 Dependency Graphs and System Composition

In large systems, Dependency Injection shifts complexity from individual classes to system composition.
This is intentional.
Rather than hiding complexity inside classes, Dependency Injection centralizes it in well-defined construction code.

```
int main() {
    FileLogger logger{"app.log"};
    SqlDatabase database{"connection-string"};

    PaymentProcessor processor{logger, database};
    processor.run();
}
```

Here, the dependency graph is visible. It can be audited. It can be modified. It can be tested. As systems grow, this composition logic may move to factory functions or dedicated builders, but the principle remains unchanged.

1.8 Cost and Performance Considerations

A frequent concern in C++ is performance overhead.
Proper Dependency Injection introduces:

- No mandatory heap allocation
- No runtime reflection
- No hidden indirection beyond virtual dispatch when chosen

When references or pointers are used appropriately, the generated code is equivalent to direct usage.
In many cases, the compiler can inline calls or devirtualize interfaces when concrete types are known.
Dependency Injection, when applied with care, respects the zero-overhead principle.

1.9 Summary of Chapter One

Dependency Injection in C++ is not a foreign concept. It emerges naturally from disciplined design.
By separating construction from behavior, making dependencies explicit, and aligning with RAII, C++ systems gain:

- Architectural clarity
- Improved testability

- Reduced coupling
- Long-term maintainability

The next chapter explores the most critical aspect of Dependency Injection in C++: ownership, lifetime, and responsibility.

Chapter 2

Ownership, Lifetime, and RAII in Dependency Injection

2.1 Why Ownership Is Central in C++

In C++, Dependency Injection cannot be discussed independently from ownership and lifetime. Unlike managed environments, object lifetime in C++ is explicit, deterministic, and observable.
Every injected dependency raises fundamental questions:

- Who owns this object?
- How long does it live?
- Who is responsible for its destruction?

Ignoring these questions leads to designs that compile, appear to work, and eventually fail in subtle and costly ways.
Dependency Injection does not solve ownership. It exposes it.

2.2 Non-Owning Dependencies and References

The most common and safest form of Dependency Injection in large C++ systems is non-owning injection.

```
class Metrics {
public:
    void record(int value);
};

class Service {
    Metrics& metrics;
public:
    explicit Service(Metrics& m) : metrics(m) {}
};
```

This design communicates intent clearly:

- `Service` does not own `Metrics`
- Lifetime must exceed that of `Service`
- Responsibility is external

References are ideal when:

- The dependency is mandatory
- Lifetime relationships are well-defined
- Null is not a valid state

This approach minimizes complexity and aligns perfectly with RAII.

2.3 Pointer Injection and Optional Dependencies

Sometimes a dependency is optional or its presence varies by configuration.
In such cases, pointers communicate intent better than references.

```
class Cache {
public:
    void store(int key, int value);
};

class Repository {
    Cache* cache;
public:
    explicit Repository(Cache* c) : cache(c) {}

    void save(int key, int value) {
        if (cache) {
            cache->store(key, value);
        }
    }
};
```

Here, the pointer explicitly signals:

- The dependency may be absent
- Ownership is not transferred
- Null handling is required

This explicitness is preferable to hidden defaults or global lookups.

2.4 Owning Dependencies and Smart Pointers

In some designs, a component truly owns its dependency. In such cases, ownership must be explicit and unambiguous.

```
class Worker {
public:
    void execute();
};

class Manager {
    std::unique_ptr<Worker> worker;
public:
    explicit Manager(std::unique_ptr<Worker> w)
        : worker(std::move(w)) {}
};
```

Using `std::unique_ptr` communicates:

- Exclusive ownership
- Clear transfer of responsibility
- Deterministic destruction

Dependency Injection does not prohibit ownership. It requires ownership to be visible.

2.5 Shared Ownership: Use with Caution

Shared ownership introduces complexity that propagates through the system.

```
class EventBus {};
```

```
class Subscriber {
    std::shared_ptr<EventBus> bus;
public:
    explicit Subscriber(std::shared_ptr<EventBus> b)
        : bus(std::move(b)) {}
};
```

While valid in some designs, `std::shared_ptr` should be treated as a last resort.
Shared ownership complicates:

- Lifetime reasoning
- Destruction order
- Performance predictability

In large systems, shared ownership tends to spread unintentionally. Dependency Injection makes this visible, but discipline is still required.

2.6 RAII as the Backbone of Dependency Injection

Resource Acquisition Is Initialization (RAII) is the foundation upon which safe Dependency Injection rests.
Injected dependencies should:

- Be fully initialized before use
- Release resources deterministically
- Never require manual cleanup by dependents

When RAII is respected, Dependency Injection becomes predictable and robust.
When RAII is violated, Dependency Injection amplifies design flaws.

2.7 Construction Order and Safety

Constructor injection guarantees that all dependencies are available before any behavior is executed.

```
class System {
    Logger& logger;
    Database& database;
public:
    System(Logger& l, Database& d)
        : logger(l), database(d) {
        logger.log("System initialized");
    }
};
```

There is no intermediate invalid state. The compiler enforces initialization order. Failure is immediate and visible.

2.8 Common Lifetime Mistakes

Even experienced developers make lifetime mistakes:

- Injecting temporaries
- Storing references to short-lived objects
- Mixing ownership and non-ownership inconsistently
- Relying on undocumented lifetime assumptions

Dependency Injection does not eliminate these risks, but it makes them easier to audit and reason about.

2.9 Summary of Chapter Two

In C++, Dependency Injection is inseparable from ownership and lifetime.
Correct designs:

- Make ownership explicit
- Prefer non-owning references
- Use smart pointers deliberately
- Rely on RAII for safety

The next chapter examines how dependencies are injected in practice, focusing on interfaces, abstraction techniques, and injection strategies that scale.

Chapter 3

Injection Techniques and Interface Design

3.1 Why Injection Technique Matters

Dependency Injection is not a single mechanism. It is a family of techniques that trade clarity, flexibility, and safety in different ways.
In large C++ systems, the choice of injection technique directly affects:

- Interface stability
- Testability
- Compile-time dependencies
- Binary compatibility

Poor choices do not fail immediately. They fail years later.

3.2 Constructor Injection Revisited

Constructor injection remains the primary technique for mandatory dependencies.

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

class Engine {
    Storage& storage;
public:
    explicit Engine(Storage& s) : storage(s) {}
};
```

Constructor injection enforces:

- Complete initialization
- Non-null dependencies
- Clear dependency declaration

For large systems, this predictability outweighs any perceived verbosity.

3.3 Setter Injection and Its Risks

Setter injection allows dependencies to be supplied after construction.

```
class Engine {
    Storage* storage = nullptr;
public:
    void set_storage(Storage& s) {
        storage = &s;
    }
```

```
    void run() {
        if (!storage) {
            throw std::logic_error("Storage not set");
        }
        storage->write("running");
    }
};
```

While sometimes necessary, setter injection introduces risk:

- Objects may exist in invalid states
- Order of calls becomes significant
- Invariants are harder to enforce

In large systems, these risks multiply across teams and modules.
Setter injection should be reserved for:

- Truly optional dependencies
- Late-bound configuration
- Rare framework integration points

3.4 Interface Design in C++

Interfaces in C++ are typically expressed as abstract base classes.

```
class Clock {
public:
    virtual ~Clock() = default;
    virtual std::chrono::time_point<std::chrono::system_clock>
    now() const = 0;
};
```

Good interfaces are:

- Small
- Stable
- Focused on behavior
- Independent of implementation details

Dependency Injection amplifies interface quality. Poor interfaces spread quickly. Good interfaces enable long-term stability.

3.5 Avoiding Over-Abstraction

A common mistake is abstracting everything.

```
class IntProvider {
public:
    virtual int get() = 0;
};
```

This level of abstraction adds no value and increases cognitive load.
Abstraction should exist to:

- Enable substitution
- Isolate volatile components
- Protect architectural boundaries

Dependency Injection rewards restraint.

3.6 Type Erasure as an Alternative to Inheritance

Inheritance-based interfaces introduce virtual dispatch and ABI concerns.
Type erasure offers an alternative.

```
class Logger {
    std::function<void(const std::string&)> log_fn;
public:
    explicit Logger(std::function<void(const std::string&)> fn)
        : log_fn(std::move(fn)) {}

    void log(const std::string& msg) {
        log_fn(msg);
    }
};
```

Type erasure:

- Avoids inheritance hierarchies
- Reduces header dependencies
- Can improve ABI stability

However, it trades compile-time checking for runtime flexibility. In large systems, this trade-off must be deliberate.

3.7 Policy-Based and Compile-Time Injection

Not all Dependency Injection is runtime-based.
Policy-based design allows compile-time injection.

```
template<typename LoggerPolicy>
class Engine : private LoggerPolicy {
public:
    void run() {
        this->log("running");
    }
};
```

This approach:

- Eliminates runtime overhead
- Enables aggressive optimization
- Increases template complexity

Compile-time injection is powerful but should be confined to performance-critical paths or libraries with stable policies.

3.8 Header Dependencies and Compilation Impact

Dependency Injection influences build performance.
Abstract interfaces reduce header inclusion. Concrete dependencies increase coupling.
Best practices include:

- Forward declarations
- Pimpl idiom where appropriate
- Minimizing template exposure

In large codebases, build time is an architectural concern.

3.9 Summary of Chapter Three

Injection techniques shape system behavior.
Effective designs:

- Prefer constructor injection
- Use setter injection sparingly
- Design small, stable interfaces
- Choose abstraction mechanisms deliberately

The next chapter examines Dependency Injection at system scale: ABI boundaries, shared libraries, and build-system integration.

Chapter 4

Large-Scale Architecture, ABI Boundaries, and Build Systems

4.1 Dependency Injection Beyond a Single Binary

In large C++ systems, Dependency Injection rarely operates within a single executable. More commonly, it spans:

- Static libraries
- Shared libraries
- Plugins and loadable modules
- Independently versioned components

At this scale, Dependency Injection becomes an architectural coordination mechanism, not merely a design pattern.
The primary constraint is no longer readability alone, but binary compatibility, build isolation, and long-term evolvability.

4.2 Understanding ABI as a Design Constraint

The Application Binary Interface (ABI) defines how compiled components interact. Abstract interfaces used for Dependency Injection often cross ABI boundaries. This introduces strict requirements:

- Stable virtual function layouts
- Consistent calling conventions
- Identical compiler and standard library assumptions

A poorly designed interface can lock a system into a specific compiler or force costly full rebuilds.
Dependency Injection does not remove ABI constraints. It makes them visible.

4.3 Designing ABI-Stable Interfaces

When interfaces cross shared-library boundaries, they must be designed conservatively.

```
class Logger {
public:
    virtual ~Logger() = default;
    virtual void log(const char* message) noexcept = 0;
};
```

Key principles:

- Avoid inline virtual implementations
- Prefer POD-like parameters
- Minimize template exposure

- Mark destructors virtual and noexcept

Dependency Injection encourages interface-based design, but ABI safety requires restraint.

4.4 Factories as ABI Boundaries

A common technique is to expose factories rather than concrete constructors.

```
extern "C" Logger* create_logger();
```

This approach:

- Decouples allocation strategy
- Avoids cross-boundary ownership confusion
- Simplifies plugin loading

The consuming side injects the dependency without knowing its concrete type or construction details.

4.5 Dependency Injection and Plugin Architectures

In plugin-based systems, Dependency Injection becomes essential.
Plugins cannot assume:

- Global state
- Construction order
- Concrete implementations

Instead, dependencies are supplied explicitly by the host application.

```
struct PluginContext {
    Logger& logger;
    Config& config;
};

extern "C" void initialize_plugin(PluginContext& ctx);
```

This design:

- Makes dependencies explicit
- Prevents hidden coupling
- Enables independent plugin evolution

4.6 Static Linking and Large Monorepos

In statically linked environments, Dependency Injection primarily addresses compile-time and organizational concerns.
Without DI:

- Headers accumulate dependencies
- Rebuilds cascade unnecessarily
- Modules become tightly entangled

With disciplined injection:

- Interfaces reduce include pressure
- Translation units become smaller
- Incremental builds improve

Dependency Injection thus contributes directly to developer productivity.

4.7 CMake and Construction Graphs

Build systems already describe dependency graphs. Dependency Injection aligns runtime structure with build-time structure.
In CMake:

- Targets represent architectural components
- Link dependencies express allowed coupling
- DI enforces those boundaries at runtime

A mismatch between build dependencies and injected dependencies is a design smell.
Large systems benefit when build graphs and construction graphs mirror each other.

4.8 C++ Modules and Dependency Control

C++20 modules introduce stronger boundaries than traditional headers.
Dependency Injection complements modules by:

- Reducing module interdependence
- Encouraging interface-based exports
- Limiting transitive exposure

Injected dependencies can often be declared in module interfaces while implementations remain private.
This combination enables:

- Faster builds
- Clear ownership boundaries
- More stable architectures

4.9 Avoiding Container-Centric Architectures

Large systems sometimes adopt centralized Dependency Injection containers.
While containers can simplify wiring, they introduce risks:

- Hidden construction logic
- Global configuration state
- Reduced local reasoning

In C++, container usage should be limited to composition roots, never spread throughout the system.
Explicit construction remains superior for long-term maintenance.

4.10 Summary of Chapter Four

At scale, Dependency Injection is inseparable from architecture, ABI, and build systems.
Robust designs:

- Respect binary boundaries
- Use factories at interfaces
- Align build and runtime dependencies
- Favor explicit composition over global containers

The final chapter focuses on testing, substitution, and ensuring that Dependency Injection delivers long-term value.

Chapter 5

Testing, Substitution, and Long-Term Maintainability

5.1 Why Dependency Injection Changes Testing Fundamentally

In large C++ systems, testing difficulties are rarely caused by lack of test frameworks. They are caused by design decisions that hard-wire dependencies into components.
When dependencies are constructed internally, tests must either:

- Initialize large portions of the system
- Rely on fragile global state
- Avoid testing altogether

Dependency Injection changes this by making dependencies explicit, movable, and replaceable. Testing becomes a natural consequence of design, not a special activity.

5.2 Designing for Substitution

A component designed for Dependency Injection implicitly supports substitution.

```
class Clock {
public:
    virtual ~Clock() = default;
    virtual std::chrono::time_point<std::chrono::system_clock>
    now() const = 0;
};
```

Production code may use a real clock, while tests inject a deterministic one.

```
class FakeClock : public Clock {
    std::chrono::time_point<std::chrono::system_clock> t;
public:
    explicit FakeClock(decltype(t) start) : t(start) {}
    auto now() const -> decltype(t) override {
        return t;
    }
};
```

No conditional compilation. No runtime switches. Only substitution.

5.3 Mocking Without Heavy Frameworks

C++ does not require mocking frameworks to test behavior.

Simple hand-written mocks are often clearer and more robust.

```
class TestLogger : public Logger {
public:
    std::vector<std::string> messages;
```

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

This approach:

- Avoids macro-heavy frameworks
- Keeps tests readable
- Encourages interface discipline

Dependency Injection enables such designs naturally.

5.4 Controlling Side Effects

Large systems suffer when side effects are scattered throughout the codebase.
Injected dependencies allow side effects to be centralized and controlled.
Typical side-effecting components include:

- Logging
- File I/O
- Networking
- Time and randomness

By injecting these dependencies, core logic becomes testable and deterministic, while side effects are isolated at the edges.

5.5 Testing Construction Graphs

Dependency Injection shifts complexity to construction logic. That logic should be tested.

```
TEST(SystemComposition, WiringIsValid) {
    FileLogger logger{"test.log"};
    InMemoryDatabase db;

    System system{logger, db};
    system.run();
}
```

These tests verify:

- Correct wiring
- Lifetime compatibility
- Absence of hidden dependencies

Such tests are inexpensive and prevent architectural regressions.

5.6 Refactoring and Long-Term Evolution

The true value of Dependency Injection appears over time.
Well-designed systems:

- Allow components to be replaced
- Support gradual architectural change
- Enable parallel team development

Poorly designed systems resist change until replacement becomes the only option.
Dependency Injection is not a guarantee of success, but it significantly increases the odds.

5.7 Avoiding Dependency Injection Fatigue

Overuse of Dependency Injection can be as harmful as underuse.
Symptoms include:

- Excessive indirection
- Deep constructor parameter lists
- Interfaces that mirror implementations

Good design balances:

- Explicitness
- Simplicity
- Pragmatism

Dependency Injection should serve the system, not dominate it.

5.8 Summary of Chapter Five

Dependency Injection delivers its greatest benefits in testing and long-term maintenance.
When applied with discipline, it enables:

- Deterministic tests
- Clear substitution
- Architectural resilience
- Sustainable evolution

The final sections conclude the booklet and summarize its core principles.

Conclusion

This booklet has deliberately avoided presenting Dependency Injection in C++ as a borrowed practice or a fashionable technique.
Dependency Injection in C++ is not about copying practices from other languages, frameworks, or ecosystems. It is about embracing what makes C++ distinct: explicit construction, deterministic lifetime, and zero-overhead abstractions.
Any approach to Dependency Injection that ignores these characteristics is fundamentally incompatible with the language.

Dependency Injection as a C++ Discipline

C++ does not impose architectural patterns. It provides primitives.
Those primitives—constructors, destructors, references, value semantics, and RAII— are powerful but unforgiving.
Dependency Injection in C++ is therefore not a feature to be enabled, but a discipline to be practiced.
It requires engineers to:

- Think deliberately about construction
- Make ownership explicit

- Accept responsibility for lifetime management

This discipline is demanding, but it is also what enables C++ systems to remain robust under scale and time.

Visibility as an Architectural Virtue

Large-scale systems fail quietly. They do not collapse suddenly; they erode.
Hidden dependencies, implicit construction, and undocumented assumptions accumulate until change becomes dangerous.
This booklet has emphasized visibility as a core architectural value.
When dependencies are visible:

- Systems become easier to reason about
- Testing becomes straightforward
- Refactoring becomes safer

Dependency Injection enforces this visibility by design, but only when applied with intention.

Ownership, Lifetime, and Responsibility

C++ does not abstract away responsibility.
Every object exists for a reason, lives for a duration, and must be destroyed correctly.
Dependency Injection does not remove the burden of ownership. It exposes it.
This exposure is not a weakness. It is a strength.
Systems that model ownership honestly are systems that fail predictably, recover cleanly, and evolve safely.

Architecture That Resists Entropy

Entropy is the natural state of large systems.
Without deliberate structure:

- Dependencies sprawl
- Responsibilities blur
- Architectural boundaries collapse

Dependency Injection, applied with restraint, creates resistance to this entropy.
By separating construction from behavior, centralizing composition, and isolating volatile components, systems gain a form of architectural inertia that favors stability over decay.

Frameworks Are Optional — Discipline Is Not

Throughout this booklet, frameworks have been intentionally de-emphasized.
This is not a rejection of tooling, but a reminder of priority.
Frameworks may assist composition, but they do not define architecture. They do not understand ownership. They do not reason about lifetime. They do not bear responsibility.
Dependency Injection remains effective with or without frameworks, as long as discipline is present.

Designing for Time, Not Just Functionality

Large-scale C++ systems are rarely written once. They are lived with.
They must support:

- New requirements

- New engineers
- New platforms
- New constraints

Design choices that appear neutral today often become liabilities years later.
Dependency Injection, when aligned with C++ principles, is a design choice that favors time.
It makes change possible without making it reckless.

Restraint as a Measure of Maturity

Perhaps the most important lesson is not when to use Dependency Injection, but when not to.
Mature design is characterized by restraint:

- Abstraction only where it earns its cost
- Indirection only where it provides clarity
- Flexibility only where it protects stability

Dependency Injection is not weakened by restraint. It is strengthened by it.

Final Perspective

C++ rewards engineers who think long-term, who value clarity over cleverness, and who treat architecture as a responsibility rather than an afterthought.
Dependency Injection is not a framework. It is not a mandate. It is not a shortcut.
It is a discipline.
Used thoughtfully, it allows C++ systems to grow without losing control.
Used indiscriminately, it becomes just another source of complexity.
The difference is not technical. It is architectural. It is intentional. It is human.

Appendices

Appendix A — Practical Dependency Injection Checklist

This appendix provides a concise, practical checklist to evaluate Dependency Injection usage in large-scale C++ systems. It is intended for code reviews, architectural audits, and onboarding new contributors.

Construction and Visibility

Before approving a design, verify the following:

- All mandatory dependencies are supplied via constructors
- No class silently constructs its own collaborators
- No hidden global access is used for core services
- Construction logic is centralized and readable

If a dependency is required for correct behavior, it must be visible in the constructor signature.

Ownership and Lifetime

Check ownership semantics explicitly:

- References indicate non-owning, mandatory dependencies
- Raw pointers indicate optional, non-owning dependencies
- `std::unique_ptr` indicates exclusive ownership
- `std::shared_ptr` usage is rare and justified

Every injected dependency should answer the question: *Who destroys this object, and when?*

Interface Quality

Evaluate injected interfaces carefully:

- Interfaces are small and behavior-focused
- No interface mirrors a concrete implementation
- No unnecessary getters or data exposure
- Virtual destructors are present where required

If an interface exists only to satisfy Dependency Injection, it is likely unnecessary.

Build and Dependency Hygiene

At system scale, verify:

- Headers avoid including concrete implementations
- Forward declarations are used where possible
- Template-based injection is limited to performance-critical paths
- ABI-sensitive interfaces are stable and conservative

Dependency Injection should reduce build pressure, not increase it.

Appendix A — Practical Dependency Injection Checklist

This appendix documents recurring mistakes observed in large C++ systems that attempted to adopt Dependency Injection.

The Container Everywhere Anti-Pattern

A centralized Dependency Injection container used throughout the system leads to:

- Hidden dependencies
- Runtime configuration complexity
- Loss of local reasoning

Containers, if used at all, must be confined to composition roots.

Interface Explosion

Creating interfaces for every class introduces unnecessary indirection.
Symptoms include:

- One-to-one interface-to-implementation mapping
- Excessive virtual dispatch
- Increased cognitive load

Abstraction must serve a purpose: substitution, stability, or isolation.

Lifetime Mismatch Bugs

Common lifetime errors include:

- Injecting stack-allocated objects into longer-lived components
- Storing references to temporary objects
- Sharing ownership without clear responsibility

Dependency Injection exposes these bugs early but does not prevent them automatically.

Configuration Leakage

When configuration data is injected too deeply, business logic becomes coupled to environment concerns.
Prefer:

- High-level configuration objects
- Narrow, purpose-driven parameters
- Avoiding configuration access in low-level components

Appendix C — A Minimal Composition Root Example

This appendix presents a deliberately small but complete *composition root* example. Its purpose is not to demonstrate advanced techniques, but to show the essential structure that scales correctly as systems grow in size and complexity.
In Dependency Injection, the composition root is the location where the object graph of the application is assembled. It is the only place where concrete implementations are chosen, configured, and connected.
Every other part of the system operates exclusively on abstractions and injected dependencies.

What This Example Is — and Is Not

This example is:

- Explicit
- Deterministic
- Testable
- Free of frameworks

This example is *not*:

- A full application architecture
- A container-based solution
- A framework replacement

Its value lies in clarity rather than completeness.

The Complete Composition Root

The entire object graph is assembled in `main`:

```
int main() {
    FileLogger logger{"app.log"};
    SqlDatabase database{"db.sqlite"};
    SystemClock clock;

    OrderService service{logger, database, clock};
    service.process();
}
```

This code is intentionally simple. That simplicity is a feature, not a limitation.

Explicit Construction and Ordering

Construction order is visible and meaningful:

- Infrastructure components are created first
- Domain services are constructed afterward
- Execution begins only after the graph is complete

There is no hidden initialization phase, no deferred wiring, and no implicit dependency resolution.
Every object is fully initialized before it is used.

Explicit Dependency Declaration

The `OrderService` constructor reveals its full set of requirements:

```
class OrderService {
    Logger& logger;
    Database& database;
    Clock& clock;
public:
    OrderService(Logger& l, Database& d, Clock& c)
        : logger(l), database(d), clock(c) {}

    void process();
};
```

From this signature alone, a reader can immediately infer:

- The service depends on logging, persistence, and time
- The service does not own these dependencies

- The service cannot exist in an invalid state

This level of transparency is impossible when dependencies are hidden behind globals, singletons, or service locators.

No Global State, No Hidden Wiring

The absence of global access is intentional.
There is:

- No static registry
- No singleton accessor
- No thread-local fallback

Every dependency travels through the system via explicit constructor parameters.
As a result:

- The dependency graph is inspectable
- Tests can substitute components freely
- Reasoning about behavior remains local

Isolation of Behavior from Construction

The service performs no construction logic:

```
void OrderService::process() {
    logger.log("processing order");
    database.save(/* ... */);
}
```

The service:

- Does not know where dependencies come from
- Does not know how they are configured
- Does not control their lifetime

This separation ensures that business logic remains unaffected by infrastructure changes.

Test Substitution with No Code Changes

Because dependencies are injected, testing requires no modification of production code:

```
FakeLogger logger;
InMemoryDatabase database;
FakeClock clock;

OrderService service{logger, database, clock};
service.process();
```

There are:

- No conditional compilation flags
- No testing hooks
- No runtime switches

The same constructor works in all environments.

Scaling the Composition Root

As systems grow, this structure scales naturally.
The composition root may evolve into:

- A factory function
- A dedicated bootstrap module
- A platform-specific entry point

For example:

```
std::unique_ptr<OrderService> build_service() {
    static FileLogger logger{"app.log"};
    static SqlDatabase database{"db.sqlite"};
    static SystemClock clock;

    return std::make_unique<OrderService>(
        logger, database, clock
    );
}
```

The principle remains unchanged: *construction is centralized, behavior is distributed*.

Relationship to Frameworks

Frameworks may automate composition, but they do not replace the responsibility of design.
Even when frameworks are introduced:

- The composition root still exists
- Dependency boundaries still matter
- Ownership and lifetime remain explicit concerns

This example demonstrates the baseline that frameworks must respect, not bypass.

Key Takeaways from This Appendix

This minimal composition root demonstrates that:

- Dependency Injection does not require containers
- Explicit construction is readable and scalable
- Testability emerges naturally from design
- Architecture improves without runtime cost

The simplicity shown here is not accidental. It reflects the core philosophy of Dependency Injection in C++:

> *Make dependencies visible, construction explicit, and behavior independent of wiring.*

When these principles are followed, large C++ systems remain understandable, testable, and maintainable over time.

Appendix D — When Not to Use Dependency Injection

Dependency Injection is a powerful architectural technique, but it is not a universal solution. In C++, indiscriminate application of Dependency Injection can degrade clarity, performance, and maintainability.
This appendix exists to counterbalance earlier chapters by defining clear, defensible situations where Dependency Injection should *not* be applied.
Good engineering is not measured by how many patterns are used, but by how well design choices match actual constraints.

Avoiding Pattern Dogmatism

One of the most common mistakes in large teams is turning architectural techniques into rules. Dependency Injection becomes harmful when:

- It is applied mechanically
- It is used to signal sophistication rather than solve problems
- It obscures simple relationships

Patterns exist to serve systems, not to dominate them.

Trivial and Stable Components

Not every component benefits from indirection.
Avoid Dependency Injection when a component is:

- Small
- Self-contained
- Unlikely to change
- Free of external side effects

For example:

```
class Angle {
    double radians;
public:
    explicit Angle(double r) : radians(r) {}
    double value() const { return radians; }
};
```

Introducing abstraction here adds no value. It increases complexity without enabling substitution, testability, or architectural flexibility.

Value Types and Pure Functions

Dependency Injection is fundamentally about managing *collaboration between objects*.
Value types and pure functions do not collaborate in the same way.

```
double normalize(double value, double max) {
    return value / max;
}
```

Injecting such behavior:

- Obscures intent
- Reduces readability
- Provides no meaningful benefit

Prefer direct usage for:

- Mathematical functions
- Stateless algorithms
- Domain value objects

Performance-Critical Inner Loops

In performance-sensitive code, even minimal indirection may be unacceptable.
Examples include:

- Numerical kernels
- Signal processing pipelines
- Real-time simulation loops

- High-frequency trading components

In such contexts, clarity of ownership and data flow often matters more than substitutability. Compile-time configuration, inline functions, and policy-based design are often superior alternatives.

Tight Lifetime Coupling

Some objects are naturally and inseparably coupled.

```
class FileBuffer {
    std::ifstream file;
    std::vector<char> buffer;
public:
    explicit FileBuffer(const std::string& path)
        : file(path), buffer(4096) {}
};
```

Here:

- Ownership is local
- Lifetime is tightly bound
- Separation adds no clarity

Injecting such dependencies would obscure responsibility rather than improve design.

Over-Abstraction and Interface Proliferation

A frequent failure mode is creating interfaces purely to enable Dependency Injection. Symptoms include:

- One interface per concrete class

- Interfaces that mirror implementations
- Constructor signatures bloated with abstractions

This form of abstraction:

- Increases cognitive load
- Slows development
- Provides no real decoupling

Abstraction must protect volatility, not satisfy ideology.

Local Construction with Clear Boundaries

Local construction is often correct when scope and responsibility are clear.

```
class Controller {
    Timer timer;
public:
    void run() {
        timer.start();
        // ...
    }
};
```

If:

- The dependency is purely internal
- No substitution is expected
- Testing is unaffected

then Dependency Injection is unnecessary.

Embedded and Hard Real-Time Constraints

In embedded systems, dynamic allocation and indirection may be prohibited entirely.
Static construction and compile-time binding are often mandatory.
Dependency Injection can still exist, but in a constrained form:

- Static object graphs
- Template-based policies
- No runtime configuration

Applying general-purpose DI techniques without adaptation can violate system constraints.

The Cost of Indirection

Every layer of indirection has a cost:

- Additional mental overhead
- Reduced locality of reasoning
- Potential performance impact

While often negligible, these costs accumulate in large systems.
Dependency Injection must earn its place through tangible benefits.

A Balanced Decision Framework

Before introducing Dependency Injection, ask the following questions:

- Does this dependency vary across environments?
- Does it involve side effects?

- Will it benefit from substitution or testing?
- Does abstraction improve clarity?

If the answer is consistently “no,” direct construction is likely the correct choice.

Final Thoughts on Restraint

Dependency Injection is most effective when applied selectively.
The most maintainable C++ systems are not those with the most abstraction, but those with the *right amount*.
Knowing when *not* to use Dependency Injection is a mark of architectural maturity, not a failure of design.
Dependency Injection is a tool. Engineering judgment determines when it is used.

Appendix E — Final Design Principles

This appendix consolidates the architectural principles presented throughout this booklet into a coherent set of design guidelines.
These principles are not rules. They are decision-making tools derived from long-term experience with large, evolving C++ systems. Each principle exists to counter a specific failure mode commonly observed in real-world codebases.

Principle 1 — Make Dependencies Explicit

Hidden dependencies are the primary source of architectural decay.
Whenever a class depends on another component, that relationship must be visible at construction time.
Explicit dependencies:

- Improve readability
- Enable accurate reasoning
- Prevent accidental coupling

Constructor signatures serve as architectural documentation. If a dependency matters, it must be visible.

Principle 2 — Align Dependency Injection with RAII

RAII is not optional in C++. It is the foundation of correctness.
Dependency Injection must respect:

- Deterministic construction
- Deterministic destruction
- Clear ownership semantics

Injected dependencies should always be:

- Fully initialized
- Valid for the lifetime of the dependent object
- Released automatically

Any injection strategy that undermines RAII introduces hidden failure modes.

Principle 3 — Prefer Construction-Time Guarantees

The earlier correctness is enforced, the cheaper it is.
Constructor-based Dependency Injection provides:

- Compile-time validation
- Elimination of partially initialized objects
- Clear object invariants

Late binding, setter injection, and post-construction wiring should be treated as exceptions, not defaults.
In C++, construction-time guarantees are one of the language's greatest strengths.

Principle 4 — Respect ABI and Build Boundaries

Large C++ systems are constrained not only by source code, but by binaries, toolchains, and build graphs.
Dependency Injection must account for:

- ABI stability
- Shared library boundaries
- Compilation dependencies

Well-designed interfaces:

- Minimize exposure
- Remain stable over time
- Avoid template leakage across boundaries

Dependency Injection should reinforce architectural boundaries, not blur them.

Principle 5 — Design for Testing and Evolution

Testability is not a separate concern. It is a direct consequence of design.
When dependencies are injectable:

- Side effects can be isolated
- Behavior can be tested deterministically
- Systems become easier to evolve

Designs that resist testing inevitably resist change.
Dependency Injection enables evolution by decoupling behavior from environment.

Principle 6 — Avoid Unnecessary Abstraction

Abstraction is not free.
Every abstraction introduces:

- Indirection
- Cognitive overhead
- Maintenance cost

Dependency Injection should be applied only where it provides tangible benefit:

- Volatility isolation
- Substitution
- Architectural stability

Abstracting stable, local, or trivial components weakens rather than strengthens design.

Principle 7 — Centralize Construction, Distribute Behavior

A healthy C++ system exhibits a clear separation:

- Construction is centralized
- Behavior is distributed

The composition root defines:

- Which implementations are used
- How lifetimes are managed
- How subsystems are connected

The rest of the system should not care. This separation is a hallmark of scalable architecture.

Principle 8 — Favor Local Reasoning

Engineers should be able to understand a component by reading its interface and constructor. Dependency Injection supports local reasoning by:

- Eliminating hidden dependencies
- Avoiding global state
- Making assumptions explicit

Systems that require global context to understand do not scale cognitively.

Principle 9 — Let the Language Work for You

C++ already provides:

- Strong type systems
- Deterministic lifetime
- Zero-overhead abstractions

Dependency Injection should leverage these features, not work around them.
Designs that fight the language inevitably accumulate technical debt.

Principle 10 — Maturity Over Mechanism

Ultimately, Dependency Injection is not the goal. Good design is.
Mature C++ systems are characterized by:

- Restraint
- Clarity
- Long-term thinking

Dependency Injection, used wisely, supports these qualities. Used indiscriminately, it undermines them.

Closing Perspective

C++ rewards engineers who design for clarity and longevity.
Dependency Injection is most effective when guided by judgment rather than fashion, and by engineering discipline rather than ideology.
When dependencies are explicit, ownership is honest, and construction is deliberate, large C++ systems remain understandable, testable, and resilient over time.

Appendix F — Case Study: Evolving a Legacy Subsystem

This appendix presents a realistic evolution scenario commonly encountered in long-lived C++ systems.

Initial State

A legacy subsystem directly constructs its collaborators:

```
class ReportGenerator {
    FileLogger logger{"report.log"};
    SqlDatabase db{"reports.db"};
public:
    void generate() {
        logger.log("start");
        db.query("SELECT * FROM data");
        logger.log("end");
    }
};
```

Problems:

- Hard-coded configuration
- No test isolation
- Tight coupling to I/O and storage

Step 1 — Extract Interfaces

```
class Logger {
public:
    virtual ~Logger() = default;
```

```
    virtual void log(const std::string&) = 0;
};

class Database {
public:
    virtual ~Database() = default;
    virtual void query(const std::string&) = 0;
};
```

Step 2 — Constructor Injection

```
class ReportGenerator {
    Logger& logger;
    Database& db;
public:
    ReportGenerator(Logger& l, Database& d)
        : logger(l), db(d) {}

    void generate() {
        logger.log("start");
        db.query("SELECT * FROM data");
        logger.log("end");
    }
};
```

Results

- Configuration moved to composition root
- Testability improved immediately
- No behavior changes required

- Refactoring localized and safe

This evolution pattern scales well and minimizes risk when modernizing legacy code.

Appendix G — Frequently Asked Questions

Is Dependency Injection Too Verbose in C++?

C++ prioritizes explicitness. Verbosity often reflects clarity rather than overhead.
Constructor signatures communicate architectural intent. This is a feature, not a flaw.

Should I Use a Dependency Injection Framework?

In most C++ systems, no.
Frameworks can:

- Obscure construction logic
- Introduce hidden global state
- Complicate debugging

Explicit construction is usually superior.

What About Performance?

Well-designed Dependency Injection:

- Adds no mandatory heap allocation
- Introduces no reflection
- Allows compiler optimization

Performance issues typically arise from misuse, not from the concept itself.

How Deep Should Constructor Injection Go?

Inject dependencies until:

- Core logic is isolated
- Side effects are pushed to edges
- Responsibilities are clear

Avoid injecting trivial utilities that do not vary or affect testability.

Appendix H — Glossary of Key Terms

This glossary defines key terms as they are used *specifically within the context of this booklet*. The definitions emphasize architectural intent, ownership semantics, and long-term maintainability rather than generic or language-agnostic meanings.
Each term reflects practical usage in real-world, large-scale C++ systems.

Dependency Injection A design technique in which a component receives its dependencies from external code rather than constructing them internally.

In C++, Dependency Injection is primarily concerned with:

- Separating object construction from object behavior
- Making dependencies explicit and visible
- Enabling substitution without modifying core logic

Unlike managed languages, Dependency Injection in C++ must respect deterministic lifetime, ownership, and zero-overhead abstractions. It is a discipline, not a framework feature.

Composition Root The location in a program where the complete object graph is constructed and all concrete implementations are selected.

In a well-designed C++ system:

- The composition root is small and centralized
- It depends on concrete types
- The rest of the system depends only on abstractions

Typical composition roots include:

- `main()`
- Application bootstrap functions
- Platform-specific entry points

The composition root defines architectural boundaries and enforces construction discipline.

Ownership Responsibility for managing an object's lifetime, including its destruction and resource release.

In C++, ownership is explicit and non-negotiable. It is commonly expressed through:

- Automatic storage duration
- `std::unique_ptr` for exclusive ownership
- `std::shared_ptr` for shared ownership (used sparingly)

Dependency Injection does not remove ownership concerns; it exposes them. Correct design requires that ownership be clear, documented, and enforced by types.

Lifetime The period during which an object exists and may be safely accessed.

Lifetime relationships are critical in Dependency Injection, particularly when:

- Dependencies outlive their consumers
- Objects are shared across threads
- Systems span dynamic libraries or plugins

Incorrect lifetime assumptions are a primary source of undefined behavior in C++. Dependency Injection improves lifetime reasoning by making object relationships explicit.

RAII Resource Acquisition Is Initialization; a fundamental C++ idiom in which resource management is tied directly to object lifetime.

RAII guarantees that:

- Resources are acquired during construction
- Resources are released during destruction
- Cleanup is deterministic and exception-safe

Dependency Injection must align with RAII. Injected dependencies should always be valid, fully initialized objects that manage their own resources.

ABI Application Binary Interface; the low-level contract that governs how compiled components interact at the binary level.

The ABI defines:

- Function calling conventions
- Object layout and alignment
- Name mangling
- Exception propagation rules

When Dependency Injection crosses shared-library boundaries, ABI stability becomes a primary design concern. Interfaces must be conservative, stable, and carefully designed.

Interface An abstraction that defines behavior without specifying implementation.

In C++, interfaces are typically expressed as:

- Abstract base classes
- Type-erased wrappers
- Policy-based templates (compile-time interfaces)

Interfaces used for Dependency Injection should be:

- Small
- Behavior-focused
- Stable over time

Poor interface design amplifies coupling rather than reducing it.

Service Locator An architectural anti-pattern in which dependencies are retrieved from a global registry rather than being explicitly supplied.

Service Locator is harmful because it:

- Hides dependencies
- Encourages global state
- Breaks local reasoning
- Complicates testing

While it may resemble Dependency Injection superficially, Service Locator inverts responsibility incorrectly and undermines architectural clarity.

Constructor Injection A form of Dependency Injection where dependencies are supplied through a class constructor.

Constructor injection:

- Guarantees fully initialized objects
- Makes dependencies visible
- Prevents invalid intermediate states

In C++, constructor injection is the preferred and safest injection technique.

Setter Injection A form of Dependency Injection where dependencies are supplied after object construction, typically via setter functions.

Setter injection:

- Allows optional dependencies
- Introduces potential invalid states
- Requires defensive programming

In large C++ systems, setter injection should be used sparingly and only when constructor injection is impractical.

Composition The act of building complex behavior by combining simpler components.

Dependency Injection supports composition by:

- Externalizing wiring logic
- Allowing flexible substitution
- Preserving clear responsibility boundaries

Composition is favored over inheritance for scalable C++ design.

Zero-Overhead Abstraction A design principle stating that abstractions should not impose runtime cost beyond what a hand-written equivalent would incur.

Properly applied Dependency Injection in C++:

- Avoids mandatory heap allocation
- Avoids runtime reflection
- Allows compiler optimization

When abstraction introduces measurable overhead, its benefit must justify the cost.

This glossary reinforces the central theme of this booklet: precise language leads to precise design. Clear terminology enables clear reasoning, which is essential for building large, long-lived C++ systems.

Appendix I — Final Checklist for Large-Scale Adoption

This appendix provides a comprehensive checklist for teams considering the broad adoption of Dependency Injection in large-scale C++ systems.
It is intended to be used as:

- A pre-adoption readiness assessment
- A design review guide
- A refactoring validation tool
- A long-term architectural audit reference

The checklist emphasizes organizational readiness as much as technical correctness. Dependency Injection succeeds only when both are aligned.

1. Team Understanding of Ownership and Lifetime

Before introducing Dependency Injection at scale, the entire development team must share a clear understanding of ownership semantics.
Verify that:

- Engineers understand the difference between owning and non-owning relationships
- References, raw pointers, and smart pointers are used intentionally
- Lifetime assumptions are documented and enforced by types

If ownership is unclear, Dependency Injection will amplify confusion rather than reduce it. A team that cannot reason about object lifetime is not ready for large-scale Dependency Injection.

2. Consistent Use of RAII

RAII must be consistently applied across the codebase.
Confirm that:

- Resources are acquired during construction
- Cleanup is deterministic and exception-safe
- Injected dependencies are always fully initialized objects

Dependency Injection relies on predictable lifetime. Violations of RAII undermine its guarantees and introduce subtle failure modes.

3. Interface Stability and Quality

Interfaces are the primary surface through which Dependency Injection operates.
Ensure that:

- Interfaces are small and behavior-focused
- Interfaces isolate volatility rather than mirror implementations
- Virtual destructors are present where required
- ABI-sensitive interfaces are conservative and stable

Unstable or overly broad interfaces create long-term coupling that is difficult to reverse.

4. Explicit Construction and Composition Roots

Large-scale adoption requires clear and consistent construction practices.
Verify that:

- Each executable or subsystem has a well-defined composition root
- Object graphs are assembled in one place
- Construction logic is not scattered across the system

If object creation is implicit or distributed, Dependency Injection loses its architectural value.

5. Build System Alignment

The build system must support the architectural boundaries introduced by Dependency Injection.
Confirm that:

- Build targets reflect architectural components
- Interfaces reduce header inclusion pressure
- Dependency direction is enforced by the build graph

A mismatch between build-time dependencies and runtime injection relationships is a warning sign of architectural drift.

6. ABI and Binary Boundary Awareness

For systems spanning shared libraries or plugins, ABI considerations are critical.
Ensure that:

- Injected interfaces crossing binary boundaries are ABI-safe
- Allocation and deallocation responsibilities are clear
- Factories are used where appropriate to isolate ABI concerns

Dependency Injection does not eliminate ABI constraints; it makes them visible and unavoidable.

7. Testing Strategy and Substitution Benefits

Adoption should measurably improve testability.
Verify that:

- Core logic can be tested without infrastructure dependencies
- Test doubles can be injected without modifying production code
- Side effects are isolated at system boundaries

If testing remains difficult after introducing Dependency Injection, the design is likely flawed.

8. Avoidance of Global State and Service Locators

Check explicitly for anti-patterns.
Ensure that:

- No global registries are used to retrieve dependencies
- No hidden fallback mechanisms exist
- All core dependencies are visible at construction time

Service Locator patterns undermine the transparency that Dependency Injection is meant to provide.

9. Controlled Use of Dependency Injection Containers

If containers are introduced, their scope must be strictly limited.
Confirm that:

- Containers are confined to composition roots
- Business logic does not depend on container APIs
- Construction remains understandable without container knowledge

In C++, containers should assist composition, not replace explicit design.

10. Performance and Complexity Assessment

Before broad adoption, evaluate performance and complexity impact.
Ask:

- Does indirection affect critical paths?

- Are abstractions justified by real variability?
- Is compile-time or runtime cost acceptable?

Dependency Injection must earn its cost through demonstrable architectural benefit.

11. Incremental Adoption Strategy

Large systems should not adopt Dependency Injection all at once.
Ensure that:

- Adoption is incremental and localized
- Legacy code is modernized safely
- Each step produces measurable improvement

Forced, large-scale rewrites often introduce more risk than benefit.

12. Documentation and Architectural Communication

Finally, Dependency Injection must be documented.
Verify that:

- Ownership and lifetime rules are written down
- Composition root responsibilities are documented
- Architectural intent is communicated to new team members

Unwritten rules do not scale.

Final Assessment

Dependency Injection is ready for large-scale adoption only when it supports existing engineering discipline rather than attempting to replace it.
When:

- Dependencies are explicit
- Ownership is honest
- Construction is deliberate
- Testing improves measurably

Dependency Injection becomes a stabilizing force in long-lived C++ systems.
When these conditions are absent, it becomes another source of complexity.
Use this checklist not as a gate, but as a guide toward architectural maturity.

Appendix J — Dependency Injection and Concurrency

Concurrency as a Design Pressure

In large-scale C++ systems, concurrency is not optional. Threading, task systems, and asynchronous execution place additional pressure on architectural design.
Dependency Injection interacts with concurrency primarily through:

- Ownership and lifetime across threads
- Thread-safety guarantees of injected components
- Isolation of synchronization concerns

Poorly designed injection amplifies race conditions. Well-designed injection localizes them.

Injecting Thread-Safe Components

Thread-safe dependencies should advertise their guarantees explicitly.

```
class ThreadSafeQueue {
public:
    void push(int value);
    bool try_pop(int& value);
};
```

Injecting such components by reference is safe when their internal synchronization is correct.

```
class Worker {
    ThreadSafeQueue& queue;
public:
    explicit Worker(ThreadSafeQueue& q) : queue(q) {}
};
```

Dependency Injection clarifies responsibility: the queue manages synchronization; the worker does not.

Avoiding Implicit Shared State

Concurrency bugs often arise from implicitly shared global objects.
Injected dependencies make sharing explicit.

```
class MetricsCollector {
public:
    void record(int value);
};
```

If a shared instance is injected intentionally, its shared nature is visible at construction time. If isolation is desired, each worker receives its own instance.
This visibility prevents accidental contention.

Task-Based Systems and Dependency Injection

Modern C++ systems often use task schedulers rather than raw threads.
Dependencies should be injected into task contexts explicitly.

```
struct TaskContext {
    Logger& logger;
    MetricsCollector& metrics;
};

void process_task(TaskContext& ctx) {
    ctx.logger.log("task started");
    ctx.metrics.record(1);
}
```

This approach:

- Avoids hidden thread-local state
- Simplifies reasoning about execution
- Improves testability of concurrent code

Lifetime Boundaries in Concurrent Systems

In concurrent systems, lifetime errors are amplified.
Dependency Injection enforces discipline:

- Long-lived components own synchronization primitives
- Short-lived tasks hold non-owning references
- Destruction order is explicit and deterministic

RAII remains the primary defense against concurrency bugs.

Appendix K — Dependency Injection in Embedded and Real-Time Systems

Constraints of Embedded Environments

Embedded and real-time systems impose constraints often absent in desktop or server environments:

- Limited memory
- Strict timing requirements
- No dynamic allocation at runtime

Dependency Injection must adapt to these constraints.

Static Construction and Injection

In embedded systems, dependencies are often constructed statically.

```
static UartLogger logger;
static FlashStorage storage;

static Controller controller{logger, storage};
```

This form of injection:

- Avoids dynamic allocation
- Preserves explicit dependencies
- Maintains deterministic startup

Dependency Injection does not require dynamic memory.

Compile-Time Configuration

Policy-based design is common in embedded systems.

```
template<typename LoggerPolicy>
class Controller : private LoggerPolicy {
public:
    void run() {
        this->log("running");
    }
};
```

Compile-time injection:

- Eliminates runtime cost
- Enables aggressive optimization
- Increases compile-time complexity

The trade-off is often acceptable in constrained environments.

Testing Embedded Code with Dependency Injection

Injected dependencies enable simulation.

```
class FakeUart {
public:
    void write(const char* msg);
};
```

Production and test environments differ only in injected implementations.
This dramatically improves test coverage without hardware dependence.

Appendix L — Final Remarks on Design Discipline

This booklet has deliberately avoided presenting Dependency Injection as a universal solution. That omission is intentional.
Dependency Injection is not a silver bullet. It does not replace understanding of C++, it does not compensate for weak architecture, and it does not absolve engineers from reasoning carefully about systems they build.
Its value lies elsewhere.

Discipline Over Mechanism

Modern software engineering is rich in mechanisms: patterns, frameworks, libraries, and tools. Discipline, however, is rare.
Discipline is the ability to:

- Say no to unnecessary abstraction
- Choose clarity over cleverness
- Favor long-term stability over short-term convenience

Dependency Injection is powerful only when it operates within such discipline. Without it, the technique degenerates into ceremony without substance.

The Discipline to Make Dependencies Visible

Hidden dependencies are a form of technical dishonesty.
They allow code to appear simple while relying on invisible assumptions.
Design discipline requires that:

- Dependencies are explicit

- Assumptions are visible
- Construction logic is honest

Dependency Injection enforces this visibility, but only if it is applied deliberately. Visibility is not free; it must be chosen.

The Discipline to Model Ownership Honestly

C++ does not forgive ambiguity about ownership.
Every object has a lifetime. Every resource must be released. Every abstraction must respect these facts.
Design discipline requires:

- Clear ownership semantics
- Deterministic lifetime management
- Alignment with RAII principles

Dependency Injection exposes ownership decisions that might otherwise remain implicit. This exposure is uncomfortable, but it is essential.

The Discipline to Resist Unnecessary Abstraction

Abstraction is a double-edged tool.
While it enables flexibility and substitution, it also introduces:

- Indirection
- Cognitive overhead
- Maintenance cost

Mature design resists abstraction until it is clearly justified.
Dependency Injection should be applied only where it isolates volatility, supports testing, or protects architectural boundaries.
Abstraction for its own sake is not design; it is avoidance.

Designing for People, Not Just Code

Large-scale C++ systems outlive individual contributors.
They are read, modified, and extended by engineers who were not present during their original design.
Design discipline acknowledges this reality.
Explicit construction, clear dependency boundaries, and honest ownership models are acts of communication as much as they are technical choices.

Longevity as the Ultimate Metric

Performance benchmarks are measurable. Feature counts are visible. Longevity is harder to quantify, but it is the most meaningful success metric.
Systems that endure:

- Are understandable
- Are adaptable
- Fail predictably rather than catastrophically

Dependency Injection, used with discipline, contributes directly to these qualities.

Final Perspective

C++ is a language that rewards responsibility.

It offers immense power, but demands precision, honesty, and long-term thinking.
Dependency Injection does not make C++ easier. It makes design choices more visible.
That visibility is its greatest strength.
Large-scale C++ systems reward engineers who embrace discipline with longevity, stability, and clarity.
This is not an accident. It is the natural outcome of deliberate, restrained design.

References

The material in this booklet is based on established practices, standards, and long-term experience in large-scale C++ systems. The following references represent authoritative sources that informed the concepts, examples, and design principles discussed.

Language Standards and Core Guidance

- ISO/IEC 14882 — Programming Languages — C++ (C++17, C++20, C++23 editions)
- ISO C++ Core Guidelines — Design, Resource Management, and Interfaces
- C++ Standard Library Specification and Rationale Documents

Large-Scale C++ Design and Architecture

- Industrial case studies on large-scale C++ system evolution
- Long-lived C++ codebase refactoring and modernization reports
- Architectural patterns for native systems with explicit ownership
- Dependency management strategies in monolithic and modular C++ systems

Dependency Injection and Software Design Principles

- Object-oriented design principles applied to native languages
- Constructor-based design and explicit dependency modeling
- Comparison studies of Dependency Injection and Service Locator patterns
- Design-for-testability methodologies in systems programming

Memory Management and RAII

- Resource Acquisition Is Initialization (RAII) idiom documentation
- Smart pointer design and ownership semantics
- Deterministic lifetime management in C++
- Exception safety guarantees and construction correctness

ABI, Toolchains, and Build Systems

- Application Binary Interface (ABI) specifications
- Compiler documentation for GCC, Clang, and MSVC
- Binary compatibility and shared library design guidelines
- CMake target-based architecture and dependency modeling
- Build-time versus runtime dependency alignment studies

Testing, Mocking, and Verification

- Unit testing strategies for native systems
- Test doubles, fakes, and manual mocking techniques in C++
- Isolation of side effects for deterministic testing
- Construction-graph testing and composition validation

Concurrency and Systems Programming

- Modern C++ concurrency model and memory ordering
- Thread-safety design guidelines for shared components
- Task-based parallelism and execution models
- Dependency management in concurrent and asynchronous systems

Embedded and Real-Time Systems

- Static construction and initialization patterns
- Compile-time configuration using policy-based design
- Dependency Injection in memory- and time-constrained environments
- Testing embedded software through injected abstractions

Professional Practice and Engineering Experience

- Long-term maintenance reports from enterprise C++ projects
- Post-mortem analyses of architectural failure modes
- Best practices derived from multi-decade native codebases
- Engineering guidelines for sustainable software evolution