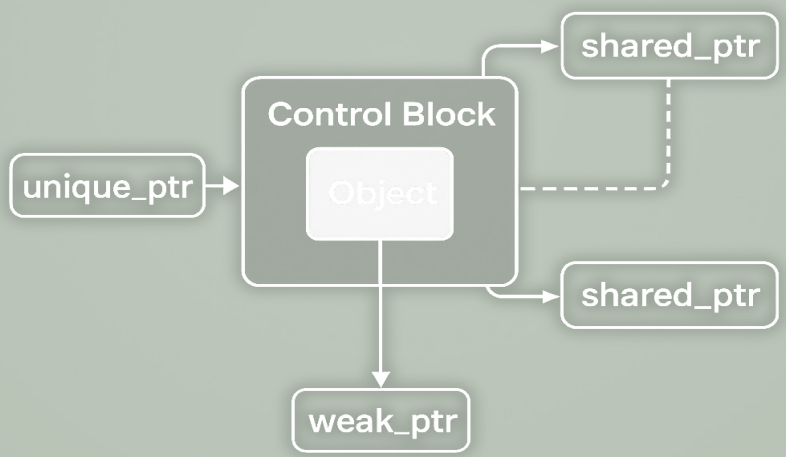
https://simplifycpp.org

Smart Pointers in Modern C++

The Complete Concise Guide



Prepared by Ayman Alheraki

Smart Pointers in Modern C++
The Complete Concise Guide

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Preface

Smart pointers are among the most transformative additions introduced in Modern C++ from C++11 onward. They replaced decades of unsafe practices built around raw pointers and manual memory management, providing an ownership model that is deterministic, self-documenting, safe, and efficient.
This booklet is intentionally concise yet deep. It targets professional programmers and C++ developers who want a clean, practical, modern understanding of the three core smart pointers:

- **std::unique_ptr**
- **std::shared_ptr**
- **std::weak_ptr**

In addition, we explore deleters, control blocks, reference counting internals, best practices, and real-world examples. The goal is to give you a solid operational grasp of smart pointers—in less than 30 pages—but without sacrificing quality, depth, or clarity.

Chapter 1

Introduction to Smart Pointer Philosophy

1.1 Why Raw Pointers Are Dangerous

Raw pointers provide no ownership semantics. They represent an address only, not an ownership relationship. This leads to several well-known problems:

- Memory leaks due to missing `delete`
- Double deletion due to multiple owners
- Dangling pointers after deletion
- Ownership confusion in large codebases
- Exception-unsafe code paths

Consider the following example illustrating classical mistakes:

```
void bad() {
    int* p = new int(10);
```

```
    if (some_condition()) return;  // memory leak!
    delete p;
}
```

Smart pointers solve these issues using RAII.

1.2 The RAII Ownership Model

RAII binds a resource's lifetime to object scope:

```
{
    std::unique_ptr<int> p = std::make_unique<int>(42);
    // object is alive
}
// p goes out of scope → memory freed automatically
```

This is exception-safe, deterministic, and eliminates vast classes of bugs.

Chapter 2

std::unique_ptr — Exclusive Ownership

2.1 Overview

`unique_ptr` models **exclusive ownership**. Only one pointer is allowed to own the resource at a time.

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

Copying a `unique_ptr` is forbidden:

```
// std::unique_ptr<int> q = p;  // ERROR: copy constructor deleted
```

2.2 Move Semantics

Ownership can be transferred using `std::move`:

```
auto a = std::make_unique<int>(100);
auto b = std::move(a);   // 'b' owns the object now
```

2.3 Custom Deleters

A powerful feature:

```
struct Closer {
    void operator()(FILE* f) const {
        if (f) fclose(f);
    }
};

std::unique_ptr<FILE, Closer> file(fopen("data.txt", "r"));
```

2.4 unique_ptr for Arrays

```
auto arr = std::make_unique<int[]>(50);
```

2.5 Real-World Use Cases

- Allocating objects whose ownership is localized in a function
- Managing resources in RAII wrappers
- Preventing accidental copying in APIs
- Ensuring deterministic destruction of temporary objects

Chapter 3

std::shared_ptr — Shared Ownership

3.1 Reference Counting Basics

A shared pointer maintains a **control block** containing:

- The reference count (owners)
- The weak reference count
- The deleter
- The pointer to the managed object

Example:

```
auto p1 = std::make_shared<int>(100);
auto p2 = p1;

std::cout << p1.use_count();   // 2
```

3.2 How Control Blocks Work (Internally)

Shared pointers allocate two memory blocks:

1. One for the control block
2. One for the object (unless make_shared is used)

3.2.1 Why make_shared is Faster

```
auto x = std::make_shared<BigStruct>(); // one allocation only
```

3.3 Shared Ownership Anti-Patterns

- Never use shared ownership when unique ownership is clear.
- Avoid passing shared_ptr by value unless ownership is intended.
- Avoid keeping shared_ptr inside performance-critical containers.
- Avoid cycles (discussed next).

Chapter 4

std::weak_ptr — Observing Without Owning

4.1 Why weak_ptr Exists

Weak pointers prevent cycles:

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

4.2 Locking weak_ptr

```
if (auto p = weak.lock()) {
    std::cout << *p;
}
```

4.2.1 Expired State

```
if (weak.expired()) {
    // object destroyed
}
```

Chapter 5

Advanced Smart Pointer Mechanics

5.1 Alias Constructor

Allows sharing the control block but pointing to a subobject:

```
auto vec = std::make_shared<std::vector<int>>(100);
std::shared_ptr<int> first(vec, &vec->front());
```

5.2 Custom Deleters for shared_ptr

```
auto fp = std::shared_ptr<FILE>(fopen("a.txt","r"), fclose);
```

5.3 Resetting Ownership

```
std::shared_ptr<int> p = std::make_shared<int>(7);
p.reset();
```

Chapter 6

Smart Pointers and Performance

6.1 Cost Analysis

6.1.1 unique_ptr:

- Nearly zero overhead
- Best choice for most objects

6.1.2 shared_ptr:

- Atomic reference counts $\rightarrow$ expensive in multithreaded contexts
- Extra memory allocation unless `make_shared` is used

6.2 When shared_ptr is Dangerous

- Cycles

- Passing by value carelessly
- Inviting unnecessary ownership semantics

6.3 When smart pointers are Slower Than Raw Pointers

- Deeply embedded, extremely performance-critical inner loops
- Hot paths inside game engines / physics engines
- Micro-level optimization where object lifetime is already controlled manually

Chapter 7

Practical Coding Techniques

7.1 Returning Smart Pointers

```
std::unique_ptr<Foo> createFoo() {
    return std::make_unique<Foo>();
}
```

7.2 Passing Smart Pointers to Functions

7.2.1 Case 1: Ownership Transfer

```
void take(std::unique_ptr<X> x);
```

7.2.2 Case 2: Shared Ownership

```
void addObserver(std::shared_ptr<Observer> obs);
```

7.2.3 Case 3: Borrowing Without Owning

```
void use(const Foo* f);       // fastest
void use(Foo& f);
```

7.3 Factory Patterns

```
class Factory {
public:
    static std::unique_ptr<Foo> create() {
        return std::make_unique<Foo>();
    }
};
```

Chapter 8

Smart Pointers and Multithreading

8.1 Thread Safety of shared_ptr

`shared_ptr` operations that modify the reference count are thread-safe.
But the managed object itself is **not** automatically thread-safe.

8.2 Atomic Shared Pointers

```
std::atomic<std::shared_ptr<Foo>> ap;
```

8.3 Hazard Pointers vs shared_ptr

In high-performance lock-free systems, shared_ptr may not be ideal.

Chapter 9

Patterns, Anti-Patterns, and Best Practices

9.1 Golden Rules

- Prefer `unique_ptr` unless sharing is mandatory.
- Avoid raw `new` and `delete`.
- Pass shared_ptr by const reference unless sharing is intended.
- Use weak_ptr to break cycles.
- Use make_unique / make_shared.
- Never store smart pointers in tight compute loops.

9.2 Common Mistakes

- Passing shared_ptr by value accidentally
- Creating unnecessary shared ownership

- Using shared_ptr where unique_ptr is enough
- Misusing weak_ptr without checking expired()

9.3 Idiomatic Use in Modern C++

```
auto engine = std::make_unique<Engine>();
auto scene  = std::make_shared<Scene>();

scene->addObject(std::make_shared<Model>());
```

Chapter 10

Case Studies and Real-World Applications

Smart pointers are not merely theoretical constructs; their value becomes most visible in real-world C++ systems where memory ownership models directly influence performance, correctness, and maintainability. In this chapter, we present a set of detailed and practical case studies that demonstrate how smart pointers solve real ownership, lifetime, and resource problems in modern C++ applications.
Each case is designed to teach a specific lesson—an ownership pattern, an anti-pattern, a best practice, or a subtle danger. Together, they form a practical reference that helps software engineers design safer and more expressive architectures.

10.1 Case Study 1: Managing a Rendering Pipeline

Rendering engines manage thousands of objects: models, textures, shaders, materials, buffers, and resources that must be loaded and unloaded at precise times.
A naive approach might use raw pointers:

```
Shader* loadShader(const std::string& path) {
    Shader* s = new Shader();
```

```
    s->load(path);
    return s;
}
```

If any failure occurs before the caller deletes the shader, memory leaks occur.
Using `unique_ptr` eliminates the hazard:

```
std::unique_ptr<Shader> loadShader(const std::string& path) {
    auto s = std::make_unique<Shader>();
    s->load(path);
    return s;  // moved
}
```

The caller now owns the resource exclusively. If loading fails, the function exits and memory is automatically freed.

Lesson

Use unique_ptr as the default for loading engine resources. It minimizes leaks and expresses exclusive ownership clearly.

10.2 Case Study 2: Entity–Component Systems (ECS)

Game engines and simulation frameworks often use an ECS architecture where many entities share components (renderable data, physics bodies, input handlers, etc.).
A common pattern:

```
class Entity {
public:
    std::shared_ptr<Transform> transform;
    std::shared_ptr<Mesh> mesh;
};
```

This works well because components are shared across systems. However, using shared_ptr everywhere can cause accidental cycles:

```
class Transform {
public:
    std::shared_ptr<Entity> owner;  // BAD: creates cycle!
};
```

Correct solution:

```
class Transform {
public:
    std::weak_ptr<Entity> owner;    // break the cycle
};
```

Lesson

Use shared_ptr only for shared resources. Use weak_ptr for back-references to break cycles.

10.3 Case Study 3: Building an Observer Pattern

The observer pattern requires observers to subscribe to a subject. A common bug is circular lifetime:

```
class Subject {
    std::vector<std::shared_ptr<Observer>> observers;
};
```

Observers then hold:

```
class Observer {
    std::shared_ptr<Subject> subject;  // cycle
};
```

Fix:

```
class Observer {
    std::weak_ptr<Subject> subject;    // no cycle
};
```

Additional Safety

Use RAII tokens for automatic unsubscribe:

```
struct Subscription {
    std::function<void()> onDestroy;
    ~Subscription() { if (onDestroy) onDestroy(); }
};
```

Lesson

Observer patterns require weak_ptr for references back to the subject. Otherwise, memory will never be freed.

10.4 Case Study 4: Thread Pools and Task Ownership

In a thread pool, tasks must be shared among worker threads. A common design:

```
using Task = std::function<void()>;

std::shared_ptr<Task> nextTask();
```

Workers receive shared ownership of tasks. However, tasks sometimes depend on external data that must not outlive them.
Correct approach:
Use unique_ptr for tasks that must be consumed:

```
std::unique_ptr<Task> nextTask();
```

Use shared_ptr only when genuine sharing exists:

```
auto job = std::make_shared<JobContext>();
auto handle = job;   // safe sharing between threads
```

Lesson

Thread pools require a clear ownership model: unique_ptr for one-time tasks, shared_ptr for shared state.

10.5 Case Study 5: Managing File Handles and OS Resources

Smart pointers can manage system resources—not just memory.

Example:

```
struct FileCloser {
    void operator()(FILE* f) const {
        if (f) fclose(f);
    }
};
```

Usage:

```
std::unique_ptr<FILE, FileCloser> file(
    fopen("log.txt","w")
);
```

Adding Logging Deleters

```
auto file = std::unique_ptr<FILE, decltype(&fclose)>{
    fopen("data.bin", "rb"),
    fclose
};
```

Lesson

Smart pointers can manage any resource with a destructor-like action.

10.6 Case Study 6: Polymorphism and Smart Pointers

A frequent mistake is returning raw pointers from factory methods.

Correct approach:

```
std::unique_ptr<Base> createObject(int type) {
    if (type == 1) return std::make_unique<DerivedA>();
    return std::make_unique<DerivedB>();
}
```

Storing Polymorphic Objects

```
std::vector<std::unique_ptr<Base>> objects;
objects.push_back(std::make_unique<DerivedA>());
objects.push_back(std::make_unique<DerivedB>());
```

Lesson

Polymorphic ownership belongs to unique_ptr by default.

10.7 Case Study 7: Building a Cache System

Caches often use shared ownership when items are reused by several subsystems.
Example:

```
class TextureCache {
    std::unordered_map<std::string, std::weak_ptr<Texture>> pool;
};
```

Flow:

1. User requests texture
2. Cache looks up weak_ptr
3. If expired → load new texture
4. Store shared_ptr in map as weak_ptr

Lesson

Caches benefit from weak_ptr to avoid keeping unused resources alive.

10.8 Case Study 8: Lifecycle Independence Using shared_ptr

Sometimes independent subsystems must share a long-lived object:

```
auto database = std::make_shared<Database>();
```

Conclusion

Smart pointers are not merely a modern convenience—they are the backbone of safe, maintainable, professional C++ development. They enforce ownership semantics, improve readability, reduce bugs, and integrate seamlessly with RAII.
Mastering them is essential for anyone writing Modern C++ today.