Smart Pointers, Dumb Mistakes

Khushboo Verma

2025

Smart Pointers, Dumb Mistakes

Khushboo Verma

Platform Engineer
Appwrite

Spot the bug! 🐛

```cpp
// What's wrong with this code?
std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
std::unique_ptr<int> ptr(arr.get() + 5); // DANGER!
```

Double Delete.

arr will call delete[] on the whole array when it is destroyed.
ptr will call delete (not delete[]) on the 6th element.
So now two unique_ptrs think they each own part of the same allocation. This is a violation of unique_ptr's core rule: *only one owner*.

‼️ Always keep ownership clear and unique.

Why this talk matters

01

The Reality Check

- Google reported 70% of security bugs are memory safety issues
- Microsoft: Same statistic for Windows vulnerabilities
- Even with smart pointers, memory bugs persist
- "Smart" doesn't mean "foolproof"

02

What We'll Cover

- Why raw pointers still matter in 2025
- The three musketeers: unique_ptr, shared_ptr, weak_ptr
- Real-world gotchas that cost companies millions
- Tools to catch these issues early

Raw Pointers - The Foundation

```cpp
void processData() {
    int* data = new int[1000];
    // ... complex processing ...
    if (errorCondition) {
        return;  // LEAK! Forgot delete[]
    }
    delete[] data;
}
```

Why Start Here?

- Smart pointers are built on raw pointers
- Understanding ownership semantics is crucial
- Performance baseline for comparison

RAII:

Resource Acquisition Is Initialization

RAII Ensures:
✅ Automatic cleanup
✅ Exception safety
✅ No memory/resource leaks

Common RAII classes in C++:
`std::unique_ptr`, `std::shared_ptr` – smart pointers
`std::vector`, `std::string` – manage dynamic memory
`std::ifstream`, `std::ofstream` – manage file handles
`std::lock_guard`, `std::scoped_lock` – manage locks

A resource (like memory, file handles, sockets, locks, etc.) is acquired in a constructor and released in the destructor.

unique_ptr - Exclusive Ownership

The Promise

```
auto ptr = std::make_unique<Resource>();
```

- 👉🏼 Automatic cleanup
- 👉🏼 Exception safe
- 👉🏼 Move semantics
- 👉🏼 Zero overhead

Basic Usage

```
std::unique_ptr<int> ptr =
std::make_unique<int>(42);

std::unique_ptr<int[]> arr =
std::make_unique<int[]>(10); // Transfer
ownership

auto ptr2 = std::move(ptr); // ptr is
now null
```

unique_ptr

Gotcha #1 - Double Delete

```
auto ptr = std::make_unique<int>(42);
delete ptr.get(); // ❌ UNDEFINED BEHAVIOR!
// ptr's destructor will delete again


// Similarly dangerous:
int* raw = ptr.get();
std::unique_ptr<int> ptr2(raw); // ❌ Two owners!
```

Rule: Never delete the result of get()!

unique_ptr

Gotcha #2 - Array vs Single Object

```cpp
// ❌ Wrong - undefined behavior
std::unique_ptr<int> arr(new int[10]);
// Uses delete instead of delete[]

// ✅ Correct
std::unique_ptr<int[]> arr(new int[10]);

// ✅ Best practice
auto arr = std::make_unique<int[]>(10);
```

unique_ptr

Gotcha #3 - Custom Deleters Gone Wrong

```
void dangerousDeleter() {
   int cleanup_count = 0;
   auto deleter = [&cleanup_count](int* p) { // ❌ Captures
by reference!
      cleanup_count++; // cleanup_count might be destroyed!
      delete p;
  };

  std::unique_ptr<int, decltype(deleter)> ptr(new int(42),
deleter);

  // If ptr outlives this function, deleter references dead
memory! }
```

shared_ptr - Shared Ownership

The Promise

```
auto ptr1 = std::make_shared<Resource>();

auto ptr2 = ptr1; // Reference count = 2


// Object destroyed when last shared_ptr is destroyed
```

Key Concepts

- 👉 Tracks how many shared_ptrs point to object
- 👉 Stores reference count and other metadata
- 👉 Reference counting uses atomic operations
- 👉 Object destroyed when count reaches zero

shared_ptr - The Control Block Mystery

The Promise

```
// Two allocations: object + control block
std::shared_ptr<int> ptr1(new int(42));

// One allocation: object + control block together
auto ptr2 = std::make_shared<int>(42);
```

Control Block Contains:

- 👉 Reference count
- 👉 Weak reference count
- 👉 Custom deleter (if any)
- 👉 Allocator (if any)

shared_ptr

Gotcha #1 - The Circular Reference Death Trap

```
struct Parent {
    std::shared_ptr<Child> child;
};

struct Child {
    std::shared_ptr<Parent> parent; // ❌ CYCLE!
};

void createLeak() {
    auto parent = std::make_shared<Parent>();
    auto child = std::make_shared<Child>();

    parent->child = child;    // parent holds child
    child->parent = parent;   // child holds parent

    // Memory leak! Neither can be destroyed
  }
```

weak_ptr

The cycle breaker

```cpp
struct Parent {
  std::shared_ptr<Child> child;
};
struct Child {
  std::weak_ptr<Parent> parent; // ✅ Breaks the
cycle!
};

// Safe usage
if (auto parent_ptr = child->parent.lock()) {
  // Safe to use parent_ptr
  std::cout << "Parent is alive!\n";
} else {
  std::cout << "Parent is gone!\n";
}
```

weak_ptr Usage Patterns

Observer Pattern

```cpp
class Subject {
    std::vector<std::weak_ptr<Observer>> observers_;
public:
    void notify() {
        for (auto it = observers_.begin(); it != observers_.end();) {
            if (auto obs = it->lock()) {
                obs->update();
                ++it;
            } else {
                it = observers_.erase(it); // Remove dead observers
            }
        }
    }
};
```

weak_ptr

Lifetime Tracking

```
void weakPointerLifecycle() {
    std::weak_ptr<int> weak;
    {
        std::shared_ptr<int> shared = std::make_shared<int>(42);
        weak = shared;  // weak_ptr observes the shared_ptr

        if (auto locked = weak.lock()) {
            std::cout << "Inside scope: weak_ptr is alive, value = " <<
*locked << "\n";
        } else {
            std::cout << "Inside scope: weak_ptr is expired\n";
        }
    } // shared_ptr goes out of scope, memory is deallocated

    if (weak.expired()) {
        std::cout << "Outside scope: weak_ptr is expired\n";
    } else {
        std::cout << "Outside scope: weak_ptr is still alive\n";
    }
}
```

shared_ptr

Gotcha #2 - The Aliasing Constructor Trap

```
struct BigData {
    std::vector<int> huge_vector; // 1MB of data
    int small_value;
 };
auto big_data = std::make_shared<BigData>();

// ❌ This keeps the entire BigData alive just for one int!
std::shared_ptr<int> small_ptr(big_data, &big_data->small_value);

// BigData can't be destroyed while small_ptr exists
```

shared_ptr

Gotcha #3 - Thread Safety Misconceptions

```
std::shared_ptr<int> global_ptr;

// Thread 1
global_ptr = std::make_shared<int>(42);

 // Thread 2
auto local = global_ptr; // ❌ RACE CONDITION!

// The shared_ptr object itself isn't thread-safe!
// Only the reference counting is thread-safe
```

shared_ptr

Gotcha #4 - Exception Safety

```
// ❌ DANGEROUS - exception safety violation
process(std::shared_ptr<A>(new A), std::shared_ptr<B>(new B));

// If B's constructor throws, A leaks!
// Order of evaluation is unspecified

// ✅ SAFE - use make_shared
auto a = std::make_shared<A>();
auto b = std::make_shared<B>();
process(a, b);
```

Possible flow:

1. new A ✅ succeeds → returns raw pointer
2. new B ❌ throws → no shared_ptr<B> constructed
3. shared_ptr<A>(new A) ❌ never runs → raw pointer leaked

Overhead Comparison (typical 64-bit system)

01 Pointer Type	02 Size	03 Overhead	04 Notes
Raw pointer	8 bytes	None	Baseline
unique_ptr	8 bytes	None*	*in optimized builds
shared_ptr	16 bytes	Control block	+atomic operations

Tools for Detection and Prevention

Static Analysis

- **Clang Static Analyzer**: Built into Clang, catches many smart pointer issues
- **PVS-Studio**: Commercial tool with smart pointer specific checks
- **Cppcheck**: Open source, basic smart pointer validation

Runtime Detection

```
# Address Sanitizer (best for
use-after-free, double-delete)
g++ -fsanitize=address -g program.cpp

# Valgrind (comprehensive but slower)
valgrind --tool=memcheck --leak-check=full
./program

# Thread Sanitizer (for race conditions)
g++ -fsanitize=thread -g program.cpp
```

Best Practices Checklist

01

The Golden Rules ✅

- **Prefer make_unique/make_shared** over new
- **Use weak_ptr** to break cycles
- **Avoid get()** unless interfacing with C APIs
- **Be explicit** about custom deleters
- **Test with sanitizers** in CI/CD
- **Profile** before optimizing reference counting
- **Understand ownership** semantics clearly

02

Code Review Red Flags ❌

- Raw new/delete mixed with smart pointers
- get() followed by delete
- shared_ptr for clearly single-owner scenarios
- Missing weak_ptr in potential parent-child relationships

Migration Strategies for Legacy Code

01

Incremental Approach

- **Start with leaf functions** (functions that don't call others)
- **Use static analysis** to identify problem areas
- **Migrate one module at a time**
- **Add tests** for each migrated component
- **Use tools** to verify no regressions

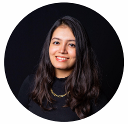
02

Common Patterns

```
// Before
void processData(Data* data) {
    // ... work with raw pointer
}

// After
void processData(const std::shared_ptr<Data>&
data) {
    // ... same work, but ownership is clear
}
```

Thank you!



Questions? Contact

*X: https://x.com/khushbooverma_*