Gorgo Continuing C++

April 11, 2026

Contents

9. RAII and Resource Management

In *Gorgo Starting C++* you learned that `std::unique_ptr` and `std::shared_ptr` manage memory automatically by freeing it when the pointer goes out of scope. That pattern — acquiring a resource in a constructor and releasing it in a destructor — is one of the most important ideas in C++. It has a name: **RAII** (Resource Acquisition Is Initialization). RAII applies to far more than memory: file handles, network connections, mutex locks, database transactions, and any other resource that must eventually be released. In this chapter you will learn the RAII pattern in depth, exception safety guarantees, scope guards, and how to use custom deleters with smart pointers.

The RAII Pattern

RAII ties the lifetime of a resource to the lifetime of an object:

1. The **constructor** acquires the resource.
2. The **destructor** releases the resource.
3. Because C++ guarantees that destructors run when objects leave scope (even when exceptions are thrown), the resource is always released.

Here is RAII applied to a file handle:

```
#include <cstdio>
#include <stdexcept>
#include <string>

class FileHandle {
public:
    FileHandle(const std::string& path, const char* mode)
        : fp_(std::fopen(path.c_str(), mode))
    {
        if (!fp_) {
            throw std::runtime_error("Cannot open: " + path);
        }
    }

    ~FileHandle()
    {
        if (fp_) {
            std::fclose(fp_);
        }
    }

    // Prevent copying (two objects should not close the same file)
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    // Allow moving
    FileHandle(FileHandle&& other) noexcept : fp_(other.fp_)
    {
        other.fp_ = nullptr;
    }

    FILE* get() const { return fp_; }

private:
```

```
    FILE* fp_;
};
```

With this class, a file is always closed, no matter how the function exits:

```
void process_file(const std::string& path)
{
    FileHandle file(path, "r");
    // ... use file.get() ...
    // If an exception is thrown here, ~FileHandle still runs
}
// ~FileHandle runs when 'file' goes out of scope
```

Without RAII, you would need to remember to call `fclose()` on every exit path — including the ones created by exceptions.

Tip: If you find yourself writing cleanup code in multiple places, you probably need an RAII wrapper. If the resource already has a standard wrapper (like `std::fstream` for files, `std::lock_guard` for mutexes), use that instead of writing your own.

RAII Beyond Memory

RAII works with any resource:

Resource	RAII Wrapper
Heap memory	`std::unique_ptr`, `std::shared_ptr`
Files	`std::fstream`, `std::ofstream`, `std::ifstream`
Mutex locks	`std::lock_guard`, `std::unique_lock`
Database connections	Custom wrapper
Network sockets	Custom wrapper
Temporary files	Custom wrapper (create in ctor, delete in dtor)

Exception Safety Guarantees

When a function throws an exception, what state does it leave the program in? C++ defines three levels of **exception safety**:

Basic Guarantee

The **basic guarantee** promises: - No resources are leaked. - The program is in a valid state (no undefined behavior). - But the state may have changed — partial modifications may be visible.

Most well-written C++ code provides at least the basic guarantee. RAII gives you this almost for free: if every resource is managed by an object, destructors clean up automatically.

Strong Guarantee

The **strong guarantee** promises: - If the function throws, the program state is unchanged — as if the function was never called. - This is "commit or rollback" semantics.

Providing the strong guarantee usually means doing all work on a copy, then swapping:

```
void update_playlist(std::vector<std::string>& playlist, const std::string& song)
{
    std::vector<std::string> temp = playlist;  // copy
```

```
    temp.push_back(song);                    // modify copy
    // If push_back throws (bad_alloc), playlist is untouched
    playlist = std::move(temp);              // commit (noexcept)
}
```

Nothrow Guarantee

The **nothrow guarantee** promises the function never throws. Mark such functions with `noexcept`:

```
void swap(int& a, int& b) noexcept
{
    int temp = a;
    a = b;
    b = temp;
}
```

Destructors are implicitly `noexcept`. Move constructors and move assignment operators should be `noexcept` whenever possible — the standard library containers rely on this for efficiency.

Trap: If a `noexcept` function does throw, `std::terminate` is called and the program crashes. Only use `noexcept` when you are certain the function cannot throw.

Which Guarantee to Aim For

Situation	Recommendation
Destructors	Always nothrow
Move operations	Nothrow whenever possible
Simple operations	Basic guarantee is usually sufficient
Operations that modify shared state	Consider strong guarantee
Swap functions	Nothrow

Scope Guards

Sometimes you need cleanup that does not fit neatly into a class destructor. A **scope guard** is a small RAII object that runs a function when it goes out of scope:

```
#include <functional>
#include <iostream>

class ScopeGuard {
public:
    explicit ScopeGuard(std::function<void()> cleanup)
        : cleanup_(std::move(cleanup)) {}

    ~ScopeGuard()
    {
        if (cleanup_) {
            cleanup_();
        }
    }

    void dismiss() { cleanup_ = nullptr; }
```

```
    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;

private:
    std::function<void()> cleanup_;
};
```

Usage:

```
void process()
{
    auto* raw = acquire_resource();
    ScopeGuard guard([raw]() {
        release_resource(raw);
        std::cout << "Resource released\n";
    });

    // ... do work that might throw ...

    // If we get here successfully, maybe we want to keep the resource:
    // guard.dismiss();  // cancel the cleanup
}
// guard's destructor releases the resource if not dismissed
```

The `dismiss()` method lets you cancel the cleanup if the operation succeeds — useful for commit/rollback patterns.

Tip: The C++ standard library does not have a scope guard yet, but `<experimental/scope>` provides `scope_exit`, `scope_success`, and `scope_fail` in some implementations. Writing your own is straightforward, as shown above.

Custom Deleters with Smart Pointers

`std::unique_ptr` and `std::shared_ptr` call `delete` by default, but you can provide a **custom deleter** for resources that need different cleanup.

`unique_ptr` with Custom Deleter

The deleter is part of the type:

```
#include <cstdio>
#include <iostream>
#include <memory>

int main()
{
    auto file_deleter = [](FILE* fp) {
        if (fp) {
            std::fclose(fp);
            std::cout << "File closed\n";
        }
    };

    std::unique_ptr<FILE, decltype(file_deleter)> file(
        std::fopen("playlist.txt", "w"), file_deleter);
```

```
    if (file) {
        std::fprintf(file.get(), "1. Clocks\n2. Yellow\n");
    }

    return 0;
}
// file_deleter runs here, closing the file
```

```
File closed
```

`shared_ptr` with Custom Deleter

With `shared_ptr`, the deleter is **not** part of the type — you pass it as a constructor argument:

```
#include <cstdlib>
#include <iostream>
#include <memory>

int main()
{
    std::shared_ptr<void> memory(
        std::malloc(1024),
        [](void* ptr) {
            std::free(ptr);
            std::cout << "Memory freed\n";
        }
    );

    // Use memory.get() ...

    return 0;
}
```

```
Memory freed
```

Practical Example: C Library Handles

Many C libraries return opaque handles that must be freed with a specific function. Custom deleters let you manage them with smart pointers:

```
// Example with a hypothetical C library
// Handle create_session();
// void destroy_session(Handle h);

auto deleter = [](Handle* h) { destroy_session(*h); delete h; };
std::unique_ptr<Handle, decltype(deleter)> session(
    new Handle(create_session()), deleter);
```

Tip: When wrapping C library resources, prefer `unique_ptr` with a custom deleter. It has zero overhead compared to manual cleanup and guarantees the resource is released exactly once.

Try It: RAII in Action

Here is a program that demonstrates RAII with different resource types. Type it in, compile with `g++ -std=c++23`, and experiment:

```cpp
#include <cstdio>
#include <functional>
#include <iostream>
#include <memory>
#include <stdexcept>
#include <string>

// Simple scope guard
class ScopeGuard {
public:
    explicit ScopeGuard(std::function<void()> fn) : fn_(std::move(fn)) {}
    ~ScopeGuard() { if (fn_) fn_(); }
    void dismiss() { fn_ = nullptr; }
    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;
private:
    std::function<void()> fn_;
};

// RAII file wrapper using unique_ptr with custom deleter
using FilePtr = std::unique_ptr<FILE, decltype([](FILE* f) { std::fclose(f); })>;

FilePtr open_file(const std::string& path, const char* mode)
{
    FILE* fp = std::fopen(path.c_str(), mode);
    if (!fp) throw std::runtime_error("Cannot open: " + path);
    return FilePtr(fp);
}

int main()
{
    // RAII file handle
    try {
        auto file = open_file("/tmp/raii_test.txt", "w");
        std::fprintf(file.get(), "Somebody That I Used to Know\n");
        std::cout << "Wrote to file\n";
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << "\n";
    }
    // file is automatically closed here

    // Scope guard
    std::cout << "Starting operation...\n";
    {
        ScopeGuard guard([]() {
            std::cout << "Cleanup complete\n";
        });

        std::cout << "Doing work...\n";
        // guard.dismiss();  // uncomment to skip cleanup
    }
    // guard runs cleanup here

    // shared_ptr with custom deleter
```

```
    {
        auto ptr = std::shared_ptr<int>(
            new int(42),
            [](int* p) {
                std::cout << "Custom delete: " << *p << "\n";
                delete p;
            }
        );
        std::cout << "Value: " << *ptr << "\n";
    }

    return 0;
}
```

Try adding a function that throws an exception after opening a file and verify the file is still closed. Try the scope guard with `dismiss()` to see the difference.

Key Points

- **RAII** (Resource Acquisition Is Initialization) ties resource lifetime to object lifetime. The constructor acquires, the destructor releases.
- C++ guarantees destructors run when objects leave scope, even during exception unwinding. This makes RAII the foundation of exception-safe code.
- The **basic guarantee** promises no leaks and valid state but allows partial modifications.
- The **strong guarantee** promises rollback on failure (copy, modify, swap).
- The **nothrow guarantee** (`noexcept`) promises no exceptions. Use it for destructors, moves, and swaps.
- **Scope guards** are lightweight RAII objects that run a cleanup function at scope exit. `dismiss()` can cancel the cleanup for commit/rollback patterns.
- **Custom deleters** let `unique_ptr` and `shared_ptr` manage non-memory resources (file handles, C library objects).
- `unique_ptr` custom deleters are part of the type; `shared_ptr` custom deleters are not.
- When wrapping C resources, prefer `unique_ptr` with a custom deleter — zero overhead, guaranteed cleanup.

Exercises

1. **Think about it:** Why is RAII considered one of the most important patterns in C++? How does it compare to try/finally in languages like Java and Python?
2. **What happens here?**

   ```
   void risky()
   {
       FILE* fp = fopen("data.txt", "r");
       process(fp);  // might throw
       fclose(fp);
   }
   ```

   What goes wrong if `process` throws an exception? How would you fix it with RAII?
3. **Think about it:** Why should move constructors and move assignment operators be `noexcept`? What happens if they are not?
4. **Where is the bug?**

   ```
   class Connection {
   public:
   ```

```
    Connection() { connect(); }
    ~Connection() { disconnect(); }
};

void transfer()
{
    Connection c1;
    Connection c2 = c1;  // copy
    // ... work ...
}
```

5. **Calculation:** How many times is `fclose` called?

```
{
    auto d = [](FILE* f) { fclose(f); };
    std::unique_ptr<FILE, decltype(d)> f1(fopen("a.txt", "r"), d);
    auto f2 = std::move(f1);
}
```

6. **Think about it:** What is the difference between the basic guarantee and the strong guarantee? Give an example where the basic guarantee is sufficient and one where you would want the strong guarantee.

7. **Where is the bug?**

```
void process()
{
    auto ptr = std::make_unique<int[]>(100);
    // ... do work ...
    ptr.release();  // "release" the memory
}
```

8. **What does this print?**

```
{
    ScopeGuard g1([]() { std::cout << "A "; });
    ScopeGuard g2([]() { std::cout << "B "; });
    ScopeGuard g3([]() { std::cout << "C "; });
}
```

   (Using the ScopeGuard class from this chapter.)

9. **Think about it:** Why does `shared_ptr` not include the deleter in its type, while `unique_ptr` does? What design trade-off does this represent?

10. **Write a program** that wraps `malloc`/`free` in a `unique_ptr` with a custom deleter. Allocate an array of 10 integers, fill them with values, print them, and verify the memory is freed by printing a message in the deleter.