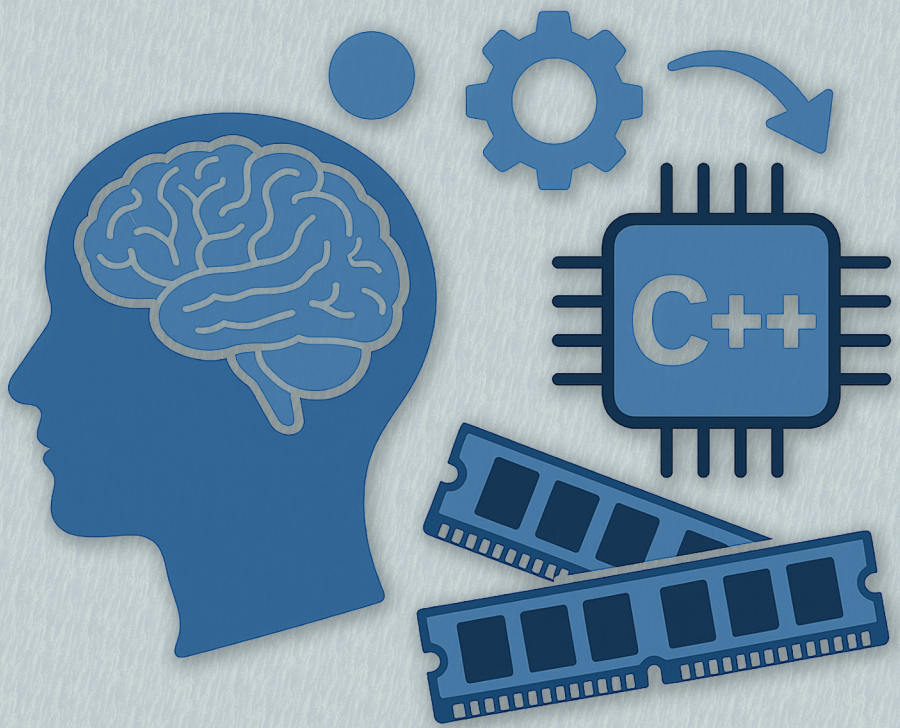
https://simplifycpp.org

Memory Management in Modern C++

A Concise Guide for Professionals



Prepared By Ayman Alheraki

Memory Management in Modern C++

A Concise Guide for Professionals

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Contents **2**

Author's Introduction **4**

1 Foundations of Memory Management in Modern C++ **7**

1.1 Why Memory Still Matters in 2025 7

1.2 From Manual Control to Managed Ownership 8

1.3 Core Design Goals 9

1.4 A Minimal Modern Example 9

2 Object Lifetime and the C++ Memory Model **13**

2.1 Storage Durations and Object Lifetimes 13

2.2 Lifetime and Explicit Lifetime Management (C++23) 14

2.3 Aliasing, Strict Aliasing, and `std::byte` 15

2.4 Alignment and Over-Aligned Types 16

3 Owning and Non-owning Types **18**

3.1 The Ownership Taxonomy 18

3.2 Smart Pointers: `unique_ptr`, `shared_ptr`, `weak_ptr` 18

3.2.1 `std::unique_ptr`: Exclusive Ownership 19

3.2.2 `std::shared_ptr`: Shared Ownership 19
3.2.3 `std::weak_ptr`: Non-owning Handle 20
3.3 Non-owning Views: `span` and `string_view` 20
3.4 Owning vs. Non-owning in Class Design 21

4 Allocators and Polymorphic Memory Resources 24
4.1 Why Allocators Matter 24
4.2 `std::pmr`: Runtime-Polymorphic Memory Resources 25
4.3 Common Standard Memory Resources 25
4.4 Designing with `std::pmr` 26

5 Custom Allocators, Pools, and Short-Lived Objects 29
5.1 Motivations for Custom Allocators 29
5.2 A Simple Fixed-Size Pool Using `std::pmr::memory_resource` 30
5.3 Arena Allocation Patterns 31
5.4 Lifetime and Destruction in Pools 32

6 Concurrency, Tools, and Modern Safety Work 35
6.1 Sharing Memory Across Threads 35
6.1.1 Thread-safe Shared Ownership Example 36
6.2 Memory Safety Tools: Sanitizers and Analyzers 37
6.3 C++ Core Guidelines and Safety Profiles 37
6.4 Patterns for Safer Concurrent Memory Management 38

Final Conclusion 40

References 42

Author's Introduction

C++ has always given programmers an unusual degree of control over memory. For decades, this was both its greatest strength and its greatest source of bugs. Manual memory management, raw pointers, and ad-hoc allocation strategies powered the fastest systems on the planet—but also produced some of the most subtle and expensive failures in software history. Modern C++ (from C++11 through C++23 and the upcoming C++26) has radically changed how we can and *should* manage memory. The language and the standard library now offer first-class support for RAII, smart pointers, standardized allocators, polymorphic memory resources, safer views (`std::span`, `std::string_view`), and even explicit lifetime management utilities such as `std::start_lifetime_as` in C++23. At the same time, the C++ Core Guidelines and ongoing work on safety profiles for C++26 push the ecosystem toward more reliable resource and lifetime management.

This mini-booklet is a concise, professional guide to memory management in modern C++. It focuses on:

- Ownership and lifetime as first-class design concepts.
- Modern idioms: RAII, smart pointers, and value semantics.
- Allocators and polymorphic memory resources (`std::pmr`).
- Custom allocation strategies and memory pools.

- The interaction between memory and concurrency.
- Tools, guidelines, and safety profiles relevant after 2020.

The goal is not to re-teach basic C++ syntax, but to unify modern memory management techniques into a compact, practical reference. The examples are written in a modern style (C++20 and C++23 where appropriate), and each chapter ends with exercises and solutions designed for experienced developers who want to sharpen their understanding or update legacy habits.
If you already write C++ professionally, this booklet is intended to:

- Clarify the *why* behind current best practices.
- Highlight what changed since C++11, and especially after 2020.
- Provide concrete patterns you can immediately apply in production.

I assume basic familiarity with templates, RAII, and the standard library. Where necessary, I briefly recall essential ideas, but the focus remains on professional-level design and correct usage in demanding systems.

How to read this booklet. You can read it sequentially, but it is also structured so that each chapter stands as a focused reference:

Ch. 1 Motivation and philosophy.

Ch. 2 Object lifetime and the memory model.

Ch. 3 Owning vs. non-owning types.

Ch. 4 Allocators and `std::pmr`.

Ch. 5 Custom pools and specialized allocators.

Ch. 6 Concurrency, tools, and modern safety work.

Exercises at the end of each chapter range from conceptual questions to small implementation tasks. Solutions immediately follow each exercise section to keep the booklet self-contained and practical.

Version and standard. Unless otherwise specified, assume at least C++20. When C++23 or C++26 features are mentioned, I point them out explicitly so you can map them to your compiler and standard library capabilities.

Let us start with the central idea of modern C++ memory management: *ownership and lifetime are design decisions, not afterthoughts.*

Chapter 1

Foundations of Memory Management in Modern C++

1.1 Why Memory Still Matters in 2025

High-level languages and managed runtimes often promise to "remove" memory management from the programmer's concerns. Yet, modern C++ remains a dominant choice where performance, predictable latency, and precise control over resources are non-negotiable:

- Real-time and embedded systems with strict memory budgets.
- High-frequency trading, game engines, and simulation software.
- Databases, storage engines, and search systems.
- Low-latency networking and high-throughput services.

In such domains, memory layout, allocation patterns, and lifetime management are not micro-optimizations. They are core architectural concerns.

Modern C++ helps by providing:

- Stronger type safety and RAII idioms.
- Safer abstractions over raw memory.
- Standardized tools to express custom allocation strategies.
- Better compiler support for detecting lifetime and aliasing issues.

1.2 From Manual Control to Managed Ownership

Traditional C++ code often centers around:

```
auto* p = new Widget{};
use(p);
delete p;
```

The programmer must ensure that every successful `new` is matched with exactly one `delete`, that no use-after-free occurs, and that exceptions do not leak resources. This model is both powerful and fragile.
Modern C++ replaces this style with RAII types that encode ownership:

```
#include <memory>

void process_widget() {
    auto w = std::make_unique<Widget>();
    use(*w);     // ownership clearly local to this function
}                // automatically destroyed here
```

Key principles:

- **RAII**: Resource Acquisition Is Initialization.

- **Ownership**: who is responsible for releasing the resource?
- **Single Point of Truth**: each resource has exactly one owner.
- **Non-owning views**: let others observe without owning.

1.3 Core Design Goals

We can summarize modern C++ memory management goals as:

1. **Safety**: avoid leaks, double-frees, use-after-free, and data races on shared objects.
2. **Performance**: minimize allocations, encourage cache-friendly layouts, and reduce fragmentation.
3. **Predictability**: know when and where memory is allocated and freed, especially in low-latency systems.
4. **Composability**: allocation strategies compose across libraries and components (e.g. `std::pmr`).

These goals are reflected in the C++ Core Guidelines, which emphasize resource safety and lifetime profiles, and in ongoing proposals for C++26 safety profiles that make such guarantees enforceable at the language level.

1.4 A Minimal Modern Example

Consider a simple modern component that owns a buffer with a custom allocation strategy using `std::pmr`:

```cpp
#include <memory_resource>
#include <vector>
#include <cstddef>

class PacketBuffer {
public:
    explicit PacketBuffer(std::pmr::memory_resource* mr =
    ↪  std::pmr::get_default_resource())
        : data_{mr} {}

    void append(const std::byte* src, std::size_t len) {
        data_.insert(data_.end(), src, src + len);
    }

    std::span<const std::byte> bytes() const noexcept { return data_; }

private:
    std::pmr::vector<std::byte> data_;
};
```

Here we already see several modern themes:

- Ownership is clearly modeled by `PacketBuffer`.
- The allocation strategy is configurable via `memory_resource`.
- Users receive a non-owning `std::span` view into the bytes.

Exercises

1.1 Explain why raw `new/delete` are considered low-level implementation details in modern C++, not high-level design tools.

1.2 Identify three real-world situations where predictable allocation and deallocation timing are critical.

1.3 Refactor the following function into a safer modern version:

```
void legacy_process(int n) {
    int* buffer = new int[n];
    for (int i = 0; i < n; ++i) {
        buffer[i] = i * 2;
    }
    // many code paths, early returns, exceptions...
    delete[] buffer;
}
```

Solutions

S1.1 Because they provide no information about ownership, are easy to misuse across multiple control paths and exceptions, and they scale poorly in large code bases compared to RAII wrappers and standard smart pointers.

S1.2 Typical examples include hard real-time control systems, high-frequency trading engines, and audio/video processing pipelines where latency and jitter must be tightly bounded.

S1.3 A modern refactoring using `std::vector`:

```
#include <vector>

void modern_process(int n) {
    std::vector<int> buffer;
    buffer.reserve(n);
    for (int i = 0; i < n; ++i) {
        buffer.push_back(i * 2);
    }
```

```
    // early returns and exceptions are now safe;
    // buffer will be cleaned up automatically.
}
```

Chapter 2

Object Lifetime and the C++ Memory Model

2.1 Storage Durations and Object Lifetimes

C++ distinguishes several storage durations:

- **Static**: objects with program lifetime (global variables, `static` members).
- **Thread**: `thread_local` storage, alive for the lifetime of the thread.
- **Automatic**: local variables on the stack.
- **Dynamic**: objects created via `new`, allocators, or custom memory resources.

Modern C++ code minimizes the use of dynamic storage directly, preferring standard containers and RAII wrappers. However, understanding the memory model remains crucial for:

- Safe interaction with C APIs.

- Custom allocators and memory pools.
- Lock-free and low-level components.

2.2 Lifetime and Explicit Lifetime Management (C++23)

Before C++20, the rules governing when an object's lifetime began and ended were subtle and sometimes underspecified. C++17 introduced `std::launder` to deal with some aliasing and lifetime corner cases. C++23 goes further with *explicit lifetime management* utilities such as `std::start_lifetime_as` and `std::start_lifetime_as_array`.
A simplified example (conceptual, assuming your standard library supports it):

```
#include <memory>
#include <new>
#include <cassert>

struct Point {
    int x;
    int y;
};

void example(unsigned char* buf, std::size_t len) {
    assert(len >= sizeof(Point));

    Point* p = std::start_lifetime_as<Point>(buf);
    p->x = 1;
    p->y = 2;
}
```

These utilities allow you to formally begin the lifetime of an object in raw storage, in a way that is defined by the standard and recognized by optimizing compilers. They are mainly relevant for low-level libraries, serialization frameworks, and memory pools—not for typical

application code. As of 2025, compiler support is still catching up, so check your toolchain status when relying on these features.

2.3 Aliasing, Strict Aliasing, and `std::byte`

Aliasing rules strongly influence how compilers optimize memory access. Violating strict aliasing (for example by reinterpreting one type as another unrelated type) can lead to undefined behavior.
Modern C++ encourages:

- Using `std::byte` for untyped memory buffers.
- Using `std::bit_cast` (C++20) for type-preserving, bit-level reinterpretation.
- Avoiding dubious `reinterpret_cast` idioms that rely on implementation-defined behavior.

Example with `std::bit_cast`:

```
#include <bit>
#include <cstdint>
#include <iostream>

int main() {
    float f = 1.0f;
    std::uint32_t bits = std::bit_cast<std::uint32_t>(f);
    std::cout << bits << '\n';
}
```

This is well-defined and expresses the intent clearly.

2.4 Alignment and Over-Aligned Types

Alignment matters for performance and correctness on many architectures. Modern C++ allows you to define over-aligned types using `alignas`:

```
struct alignas(64) CacheLineAligned {
    int data[16];
};
```

When working with custom allocators or low-level memory pools, you must honor alignment requirements, especially for over-aligned types. Standard allocation functions are required to return memory suitably aligned for any standard type; custom allocators must uphold similar guarantees.

Exercises

2.1 Describe a realistic use case for `std::start_lifetime_as` in a modern C++ library.

2.2 Explain why using `reinterpret_cast<float*>` on an `int*` may be undefined behavior, and how `std::bit_cast` improves the situation.

2.3 Implement a small function that accepts a `std::span<std::byte>` and safely interprets it as a `Point` if the size matches.

Solutions

S2.1 Typical use cases include deserialization libraries that map bytes from a network buffer or file directly into C++ objects, or low-level memory pools that need to construct objects inside pre-allocated raw storage while staying within the rules of the standard.

S2.2 The compiler is allowed to assume that pointers to unrelated types do not alias, which means reinterpreting `int*` as `float*` can break optimizer assumptions and lead to undefined behavior. `std::bit_cast` explicitly expresses that we want to reinterpret the raw bits while preserving value representation rules, giving compilers a well-defined and portable operation.

S2.3 One possible solution:

```cpp
#include <span>
#include <cstddef>
#include <optional>

struct Point {
    int x;
    int y;
};

std::optional<Point> view_as_point(std::span<const std::byte> bytes) {
    if (bytes.size() != sizeof(Point))
        return std::nullopt;

    Point p{};
    std::memcpy(&p, bytes.data(), sizeof(Point));
    return p; // copy; lifetime and aliasing are well-defined
}
```

Chapter 3

Owning and Non-owning Types

3.1 The Ownership Taxonomy

Modern C++ encourages you to categorize pointers and references explicitly:

- **Owning types**: responsible for releasing resources. Examples: `std::unique_ptr`, `std::shared_ptr`, containers (e.g. `std::vector`).
- **Non-owning types**: observe but do not own. Examples: references, raw pointers, `std::span`, `std::string_view`.

A simplified rule of thumb:

- Use owning types for member variables that manage resources.
- Use non-owning references or spans in function parameters.

3.2 Smart Pointers: `unique_ptr`, `shared_ptr`, `weak_ptr`

3.2.1 `std::unique_ptr`: Exclusive Ownership

`std::unique_ptr` encodes exclusive ownership. It is the first choice for managing heap-allocated objects:

```
#include <memory>

class Manager {
public:
    explicit Manager(std::unique_ptr<Widget> w)
        : widget_{std::move(w)} {}

private:
    std::unique_ptr<Widget> widget_;
};
```

Key properties:

- Non-copyable; movable.
- Custom deleters are supported.
- Ideal for PIMPL and unique resources.

3.2.2 `std::shared_ptr`: Shared Ownership

`std::shared_ptr` manages a reference-counted object:

```
#include <memory>
#include <vector>

std::shared_ptr<Config> global_config;

void init_config() {
```

```
    global_config = std::make_shared<Config>();
}
```

Modern advice:

- Use `shared_ptr` when ownership is genuinely shared.
- Avoid forming cycles; use `weak_ptr` to break them.
- Be aware of atomic reference count updates and their cost, especially in multithreaded code.

3.2.3 `std::weak_ptr`: Non-owning Handle

`std::weak_ptr` observes a `shared_ptr` without contributing to the reference count:

```
#include <memory>

std::weak_ptr<Config> config_cache;

std::shared_ptr<Config> get_config() {
    if (auto cached = config_cache.lock()) {
        return cached;
    }
    auto cfg = std::make_shared<Config>();
    config_cache = cfg;
    return cfg;
}
```

3.3 Non-owning Views: `span` and `string_view`

C++20 introduced `std::span`, a non-owning view over a contiguous sequence:

```cpp
#include <span>
#include <vector>

void process(std::span<const int> data);

void example() {
    std::vector<int> v{1, 2, 3};
    process(v); // implicit conversion to span
}
```

Similarly, `std::string_view` is a non-owning view over a string:

```cpp
#include <string>
#include <string_view>

void log_error(std::string_view message);

void example() {
    std::string msg = "error!";
    log_error(msg);              // ok
    log_error("temporary text"); // OK: temporary lifetime is extended
}
```

When using non-owning views, you must ensure the viewed data outlives the view; otherwise, you create dangling references.

3.4 Owning vs. Non-owning in Class Design

Consider a high-level design choice:

- A GUI widget that *owns* its internal buffer.
- A renderer that *borrows* a view of that buffer.

```
class WidgetBuffer {
public:
    explicit WidgetBuffer(std::size_t size)
        : data_(size) {}

    std::span<const std::byte> bytes() const noexcept { return data_; }
    std::span<std::byte> bytes() noexcept { return data_; }

private:
    std::vector<std::byte> data_;
};

class Renderer {
public:
    void draw(std::span<const std::byte> buffer);
};
```

The ownership is clearly localized, and dependencies use non-owning views.

Exercises

3.1 Describe a scenario where using `std::shared_ptr` instead of `std::unique_ptr` introduces unnecessary overhead.

3.2 Write a small class that owns a `std::vector<int>` and exposes it via a non-owning `std::span<int>`.

3.3 Identify potential lifetime bugs in this function:

```
std::string_view make_view() {
    std::string s = "hello";
    return s;
}
```

Solutions

S3.1 If an object has a single logical owner and is never shared across threads, using `shared_ptr` adds reference counting overhead (possibly atomic operations) without any benefit. For example, a local cache object used only inside one function should be `unique_ptr` or a plain object, not `shared_ptr`.

S3.2 Example:

```
#include <vector>
#include <span>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n)
        : data_(n, 0) {}

    std::span<int> view() noexcept { return data_; }
    std::span<const int> view() const noexcept { return data_; }

private:
    std::vector<int> data_;
};
```

S3.3 The function returns a `std::string_view` referring to a `std::string` that is destroyed when the function returns, leaving a dangling view. The caller will observe undefined behavior when using the result. The fix is to ensure the viewed string outlives the view, or to return a `std::string` instead.

Chapter 4

Allocators and Polymorphic Memory Resources

4.1 Why Allocators Matter

Allocators control where and how memory is obtained and released. For high-performance or constrained systems, choosing or customizing the allocation strategy can:

- Reduce fragmentation and cache misses.
- Avoid expensive system calls.
- Enforce that all allocations come from specific memory regions (e.g. shared memory, NUMA nodes, embedded memory).

The traditional `Allocator` model in C++ is compile-time polymorphic: the allocator type is part of the container type. This limits interoperability and makes runtime configuration awkward.

4.2 `std::pmr`: Runtime-Polymorphic Memory Resources

C++17 introduced `std::pmr` (polymorphic memory resources), allowing containers to use a runtime-selected memory resource:

```
#include <memory_resource>
#include <vector>

void example_pmr() {
    std::byte buffer[1024];
    std::pmr::monotonic_buffer_resource pool{buffer, sizeof(buffer)};
    std::pmr::vector<int> v{&pool};

    v.reserve(100);
    for (int i = 0; i < 100; ++i) {
        v.push_back(i);
    }
}
```

Key ideas:

- `std::pmr::memory_resource` is an abstract base class.
- `std::pmr::polymorphic_allocator` forwards to a memory resource at runtime.
- `std::pmr::vector`, `std::pmr::string`, etc. are aliases that use polymorphic allocators.

4.3 Common Standard Memory Resources

The standard provides several memory resource types:

`new_delete_resource` Uses global `operator new/delete`.

`null_memory_resource` Always fails (useful for testing).

`monotonic_buffer_resource` Allocates from an underlying buffer with simple monotonic growth.

`synchronized_pool_resource` / `unsynchronized_pool_resource` Pool allocators for small allocations.

Example using a pool resource:

```
#include <memory_resource>
#include <list>

void example_pool() {
    std::pmr::unsynchronized_pool_resource pool;
    std::pmr::list<std::string> names{&pool};

    names.emplace_back("Alice");
    names.emplace_back("Bob");
}
```

4.4 Designing with `std::pmr`

A common modern pattern is to let library components accept a `memory_resource*` in their constructor:

```
class LogBuffer {
public:
    explicit LogBuffer(std::pmr::memory_resource* mr =
    ↪  std::pmr::get_default_resource())
        : entries_{mr} {}
```

```
    void add(std::string_view msg) {
        entries_.emplace_back(msg);
    }

private:
    std::pmr::vector<std::pmr::string> entries_;
};
```

This approach makes your component allocation-aware and testable:

- In production, use the default resource or a tuned pool.
- In tests, inject a counting or debugging resource.

Exercises

4.1 Explain why the traditional compile-time allocator model makes `std::vector<int, MyAllocator<int>>` incompatible with `std::vector<int>` by default.

4.2 Write a small function that creates a `std::pmr::vector<int>` backed by a local `monotonic_buffer_resource`.

4.3 Design an interface for a logging component that can be configured with different `std::pmr::memory_resource` implementations.

Solutions

S4.1 Because the allocator type is part of the container's type, the two vectors have different types and cannot be assigned or passed to interfaces that expect the other type without a conversion. Polymorphic allocators decouple the container type from the concrete allocation strategy.

S4.2 Example:

```
#include <memory_resource>
#include <vector>

std::pmr::vector<int> make_local_vector() {
    static thread_local std::byte buffer[1024];
    static thread_local std::pmr::monotonic_buffer_resource pool{buffer,
    ↪  sizeof(buffer)};
    std::pmr::vector<int> v{&pool};
    v.reserve(100);
    return v; // copies allocator state by pointer
}
```

S4.3 One possible interface:

```
#include <memory_resource>
#include <string>
#include <vector>

class PmrLogger {
public:
    explicit PmrLogger(std::pmr::memory_resource* mr =
    ↪  std::pmr::get_default_resource())
        : entries_{mr} {}

    void log(std::string_view msg);

    std::span<const std::pmr::string> entries() const noexcept { return
    ↪  entries_; }

private:
    std::pmr::vector<std::pmr::string> entries_;
};
```

Chapter 5

Custom Allocators, Pools, and Short-Lived Objects

5.1 Motivations for Custom Allocators

Despite the power of `std::pmr`, there are still cases where custom allocators are beneficial:

- Embedded systems with strict memory regions.
- High-frequency allocation/deallocation of fixed-size objects.
- Shared memory segments across processes.

You can implement custom allocators either via the classic `Allocator` interface or by implementing your own `memory_resource` for `std::pmr`.

5.2 A Simple Fixed-Size Pool Using `std::pmr::memory_resource`

Conceptually, a pool allocator manages a fixed-size block and serves objects of a known size quickly:

```
#include <memory_resource>
#include <cstddef>
#include <vector>

class FixedBlockResource : public std::pmr::memory_resource {
public:
    FixedBlockResource(void* buffer, std::size_t size)
        : start_{static_cast<std::byte*>(buffer)},
          end_{start_ + size},
          current_{start_} {}

private:
    void* do_allocate(std::size_t bytes, std::size_t alignment) override {
        auto aligned = align_up(current_, alignment);
        if (aligned + bytes > end_) {
            throw std::bad_alloc{};
        }
        current_ = aligned + bytes;
        return aligned;
    }

    void do_deallocate(void*, std::size_t, std::size_t) override {
        // no-op for simple monotonic pool
    }
```

```
    bool do_is_equal(const std::pmr::memory_resource& other) const noexcept
    ↪  override {
        return this == &other;
    }

    static std::byte* align_up(std::byte* ptr, std::size_t alignment) {
        auto addr = reinterpret_cast<std::uintptr_t>(ptr);
        auto aligned = (addr + alignment - 1) & ~(alignment - 1);
        return reinterpret_cast<std::byte*>(aligned);
    }

    std::byte* start_;
    std::byte* end_;
    std::byte* current_;
};
```

This simplified resource never deallocates; it is suitable for short-lived arenas where all allocations are released at once by destroying the resource.

5.3 Arena Allocation Patterns

Arena (region) allocation is a powerful pattern:

- Many allocations from a pool.
- No individual deallocation; free everything at once.
- Great for request-scoped data in servers, or frame-scoped data in game engines.

Using a monotonic resource, we can implement an arena wrapper:

```
class Arena {
public:
```

```
    explicit Arena(std::size_t size)
        : buffer_(size),
          resource_(buffer_.data(), buffer_.size()),
          allocator_{&resource_} {}

    template <class T, class... Args>
    T* create(Args&&... args) {
        return
        ↪  std::allocator_traits<decltype(allocator_)>::allocate(allocator_,
        ↪  1),
               std::construct_at(

                   ↪  std::allocator_traits<decltype(allocator_)>::allocate(alloc
                   ↪  1),
                  std::forward<Args>(args)...);
    }

    std::pmr::memory_resource* resource() noexcept { return &resource_; }

private:
    std::vector<std::byte> buffer_;
    std::pmr::monotonic_buffer_resource resource_;
    std::pmr::polymorphic_allocator<std::byte> allocator_;
};
```

(You may want to refine this implementation to avoid double allocation; the code here illustrates the idea rather than providing production-ready details.)

5.4 Lifetime and Destruction in Pools

When you manage objects in an arena, you must ensure that:

- Objects are destroyed before the storage is reused or released.

- If you rely on `std::destroy` or manual loops, you apply them consistently.
- You avoid storing pointers outside the arena that outlive it.

For performance-critical code, you might accept a “bulk destruction” model where you explicitly destroy all objects at once when the arena goes out of scope.

Exercises

5.1 Describe a scenario where using an arena-style allocator improves performance compared to the default allocator.

5.2 Modify the `FixedBlockResource` to support a reset operation that rewinds the pool to its beginning.

5.3 Discuss the trade-offs between a monotonic pool and a pool that supports individual deallocation.

Solutions

S5.1 Typical scenarios include HTTP request handling, where each request allocates many small objects, all of which become unreachable at the end of the request. An arena can allocate these cheaply and release them all at once when the request finishes.

S5.2 One possible extension:

```
class FixedBlockResource : public std::pmr::memory_resource {
public:
    FixedBlockResource(void* buffer, std::size_t size)
        : start_{static_cast<std::byte*>(buffer)},
          end_{start_ + size},
```

```
            current_{start_} {}

    void reset() noexcept { current_ = start_; }

    // ... same as before
};
```

S5.3 A monotonic pool is simple and extremely fast, but can waste memory if allocations persist across phases or if there is fragmentation inside the arena. A pool that supports individual deallocation is more flexible and memory-efficient but must maintain free lists, perform more bookkeeping, and often has higher per-allocation overhead.

Chapter 6

Concurrency, Tools, and Modern Safety Work

6.1 Sharing Memory Across Threads

In multithreaded programs, memory management interacts with the C++ memory model and synchronization primitives. Core rules:

- Every shared mutable object must be protected by a synchronization mechanism (mutex, atomics, etc.) or be immutable.
- `std::shared_ptr` is thread-safe for its control block (reference counts), but not for the pointed-to object.
- Use `std::atomic` where lock-free semantics are necessary, but be aware of complexity and subtle bugs.

6.1.1 Thread-safe Shared Ownership Example

```
#include <memory>
#include <thread>
#include <vector>

std::shared_ptr<Config> config;

void reader() {
    auto local = config; // increase refcount safely
    if (local) {
        use_config(*local);
    }
}

void writer() {
    auto new_cfg = std::make_shared<Config>();
    // Safely publish new config
    config = std::move(new_cfg);
}

int main() {
    config = std::make_shared<Config>();
    std::thread t1(reader), t2(writer);
    t1.join();
    t2.join();
}
```

This example relies on the guarantee that assignments to `std::shared_ptr` are thread-safe with respect to its control block; depending on your exact concurrency patterns, you may still need additional synchronization for the pointed-to data.

6.2 Memory Safety Tools: Sanitizers and Analyzers

Modern toolchains provide powerful dynamic and static tools:

- **AddressSanitizer (ASan)**: detects out-of-bounds accesses, use-after-free, and related issues.
- **UndefinedBehaviorSanitizer (UBSan)**: detects various forms of undefined behavior (such as invalid casts, null dereferences).
- **MemorySanitizer (MSan)**: detects use of uninitialized memory.
- **Static analyzers** (Clang-Tidy, MSVC analyzers, etc.) with checks derived from the C++ Core Guidelines.

Running your tests with sanitizers enabled is now considered best practice for serious C++ projects.

6.3 C++ Core Guidelines and Safety Profiles

The C++ Core Guidelines provide a set of rules about resource and lifetime safety, with dedicated profiles for:

- **Type safety**.
- **Bounds safety**.
- **Lifetime safety**.

Recent work (in the context of C++26) aims at standardizing safety profiles that can be enforced by compilers. The idea is to make rules such as “no dangling pointers” and “no

out-of-bounds access" part of an opt-in profile, where violations are diagnosable and often forbidden at compile time.
For professional C++ developers, understanding and enabling these profiles (or equivalent static analysis configurations) is becoming an important part of modern memory management practice.

6.4 Patterns for Safer Concurrent Memory Management

Recommended patterns include:

- Prefer message-passing and immutable data to shared mutable state when possible.
- Use `std::shared_ptr` + `std::weak_ptr` carefully to manage shared state that may outlive some consumers.
- Avoid ad-hoc lock-free structures unless you have a strong reason and deep expertise; use library-provided concurrency primitives.

Exercises

6.1 Explain why using `new/delete` inside a tight, multithreaded loop can harm performance and scalability.

6.2 Describe how AddressSanitizer can help validate a new memory pool implementation.

6.3 Propose a simple design for a thread-safe Logger that uses `std::pmr` internally but exposes a minimal, safe API to callers.

Solutions

S6.1 `new/delete` typically involve global locks and system calls; they are not designed for fine-grained, high-frequency allocations in many threads. Contention on the allocator's internal structures can severely limit scalability and increase latency.

S6.2 AddressSanitizer can detect out-of-bounds accesses, use-after-free, and double-free errors when you test your pool implementation. Running focused tests with ASan enabled on your custom allocator provides strong evidence that it handles lifetime and bounds correctly.

S6.3 One design is a Logger that:

- Accepts log messages via a thread-safe enqueue function (protected by a mutex or a lock-free queue).
- Uses a `std::pmr::vector<std::pmr::string>` or a `std::pmr::deque` internally for buffering.
- Manages its `std::pmr::memory_resource` privately so callers never see raw pointers or allocator details.

Final Conclusion

Memory management is not a solved problem that you can ignore in modern C++. It is, however, a problem that the language and ecosystem now help you handle in far safer and more expressive ways than in the past.
The key messages of this booklet are:

- **Ownership and lifetime are design concepts.** Choose owning vs. non-owning types deliberately.
- **Prefer RAII and standard abstractions.** Use `unique_ptr`, `shared_ptr`, standard containers, `span`, and `string_view`.
- **Use allocators and `std::pmr` to control allocation.** For performance-critical systems, the ability to configure allocation strategies is essential.
- **Leverage tools and guidelines.** Sanitizers, static analyzers, and the C++ Core Guidelines are not optional in professional contexts.
- **Stay current with the standard.** C++20, C++23, and upcoming C++26 features (such as explicit lifetime management and safety profiles) significantly improve how we can reason about memory.

Used correctly, modern C++ gives you control close to the hardware with much stronger safety guarantees than older styles of C++ programming. The techniques presented here—

from smart pointers to polymorphic allocators and safety tools—are now part of the core skill set of professional C++ developers.

The next step is practice: apply these techniques in real projects, audit existing code for outdated memory patterns, and adopt static and dynamic analysis as a normal part of your development process.

References

- ISO/IEC 14882:2020, *Programming Languages — C++*, International Organization for Standardization, 2020.
- ISO/IEC 14882:2024 (C++23 Standard), *Programming Languages — C++*, International Organization for Standardization, 2024.
- B. Stroustrup et al., *C++ Core Guidelines*, Standard C++ Foundation, continuously updated, accessed 2025. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- J. Lakos, V. Romeo, R. Guidi, *Embracing Modern C++ Safely*, Addison-Wesley, 2021.
- P. Halpern, "Polymorphic Memory Resources," WG21 Paper N3916, 2014.
- R. Kaiser, "C++17 Polymorphic Memory Resources (pmr) and STL for Embedded Applications," EMBO++ Tutorial Slides, 2021.
- T. Doumler, "Fixing `std::start_lifetime_as` for Arrays," WG21 Paper P2679R0, 2022.
- J. Müller, "An (In-)Complete Guide to C++ Object Lifetimes," C++ on Sea Conference, Talk and Slides, 2024.
- A. Drescher et al., "Core Safety Profiles for C++26," WG21 Paper P3081R1, 2025.

- B. Zagrodzki, “C++23 Library Features and Reference Cards,” *C++ Stories*, 2024.
- F. Sarrocco, “Memory Management in C++: Best Practices and Common Pitfalls,” Technical Article, 2023.
- PVS-Studio Team, “How frivolous use of polymorphic allocators can imbitter your life,” PVS-Studio C++ Blog, 2025.