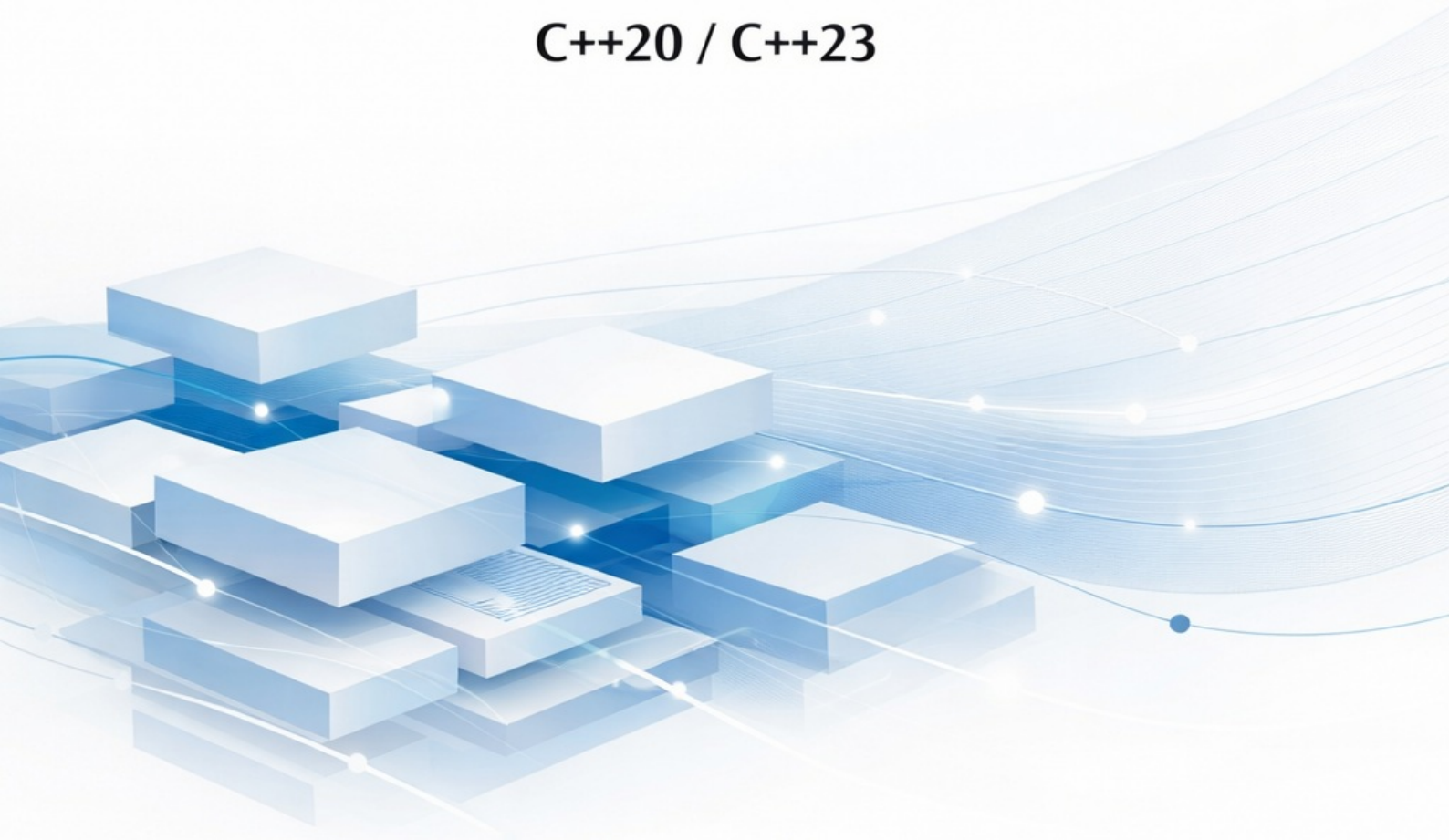
https://simplifycpp.org

Modern C++ Memory Management

Best Practices, Performance, and Advanced Allocation Strategies

C++20 / C++23



Prepared by

Ayman Alheraki

https://simplifycpp.org

Modern C++ Memory Management

Best Practices, Performance, and Advanced Allocation Strategies

C++20 / C++23

Prepared by
Ayman Alheraki

Contents

Chapter 1

Modern C++ Memory Allocation

1.1 Introduction

Memory allocation is one of the most important subjects in C++, not because C++ is merely a language that lets the programmer allocate memory, but because C++ gives the programmer the power to design how ownership, lifetime, performance, and safety interact inside real software.

For many years, memory management in C++ was presented in a narrow and dangerous form: allocate with `new`, release with `delete`, and hope that every control path cleans up correctly. That view is obsolete. In modern C++, the best way to allocate memory is usually not to manually allocate it at all. Instead, the programmer should think in terms of lifetime design, ownership expression, container usage, locality, and abstraction.

C++20 and C++23 continue the long evolution of the language toward safer and more expressive patterns. The language still gives direct control where needed, but the standard style of writing good C++ now favors automatic lifetime management, standard containers, smart pointers, views, and memory-resource-based customization.

This booklet explains the best practical ways to allocate memory in modern C++, when each technique is appropriate, what should be avoided, and how to think like a professional engineer rather than a programmer who merely places objects somewhere in memory.

1.2 A Better Question Than "How Do I Allocate?"

A beginner often asks: how should I allocate memory?

An experienced C++ programmer asks a better question:

- Who owns this object?
- How long should it live?
- Does it need dynamic lifetime?
- Does it need shared ownership?

- Does it need contiguous storage?
- Is performance dominated by allocation cost, cache locality, or fragmentation?
- Can this design avoid allocation entirely?

The real goal is not to allocate memory in a modern-looking way. The real goal is to design a system where lifetime and ownership are obvious, efficient, and hard to misuse.

1.3 The First Rule: Prefer No Explicit Allocation

The best memory allocation is often the one you never write explicitly.

Whenever possible, prefer objects with automatic storage duration, ordinary local variables, and standard containers that manage their own storage. This immediately removes entire categories of bugs such as leaks, double deletes, forgotten cleanup, and exception-related resource loss.

1.3.1 Example: Simple Automatic Object

```
#include <string>
#include <iostream>

struct Logger {
    Logger() { std::cout << "Logger created\n"; }
    ~Logger() { std::cout << "Logger destroyed\n"; }
};

int main() {
    Logger log;
    std::string name = "Modern C++";
}
```

In this example, both objects are cleaned up automatically when the scope ends. No explicit release is needed. This is the simplest and often the best model.

1.3.2 Why This Matters

Automatic lifetime has major advantages:

- cleanup is guaranteed at scope exit
- exception safety becomes much easier
- the code is shorter and clearer
- ownership is obvious

- the compiler can optimize aggressively

A great deal of modern C++ quality comes from learning not to reach for dynamic allocation too early.

1.4 Stack Allocation vs Dynamic Allocation

The phrase "stack allocation" is often used informally to mean automatic local objects, while "heap allocation" usually refers to dynamic allocation obtained through `new`, allocator-backed containers, or memory resources.

The practical distinction is this:

- automatic local objects have lifetime tied to scope
- dynamically allocated objects can outlive the current scope

1.4.1 Automatic Lifetime

```
void work() {
    int value = 42;
    std::array<int, 4> data{1, 2, 3, 4};
}
```

When `work()` ends, both `value` and `data` are destroyed automatically.

1.4.2 Dynamic Lifetime

```
#include <memory>

std::unique_ptr<int> create_value() {
    return std::make_unique<int>(42);
}
```

Here the integer lives beyond the function scope because ownership is returned to the caller.

1.4.3 When Dynamic Allocation Is Actually Needed

Dynamic allocation is justified when one or more of the following is true:

- the object must outlive the current scope
- the object size is large or runtime-defined
- polymorphic lifetime is required

- ownership must be transferred
- the object belongs to a graph, tree, or dynamic structure
- allocation strategy must be customized

The key lesson is that dynamic allocation is a design choice, not a default habit.

1.5 The Old Style: Raw `new` and `delete`

Traditional C++ used explicit `new` and `delete`:

```
int* p = new int(5);
delete p;
```

This is legal C++, but in modern code it is usually the wrong tool.

1.5.1 Why It Is Dangerous

The problem is not that `new` itself is evil. The problem is that raw manual ownership creates fragile code:

- every allocation must be matched correctly
- early returns can skip cleanup
- exceptions can leak resources
- ownership may be unclear
- copying raw pointers does not copy ownership semantics
- arrays require `delete[]` rather than `delete`

1.5.2 Example of Fragile Manual Cleanup

```
#include <stdexcept>

void process(bool fail) {
    int* p = new int(10);

    if (fail) {
        throw std::runtime_error("error");
    }

    delete p;
}
```

If an exception is thrown before `delete`, the memory leaks.

1.5.3 Modern Replacement

```
#include <memory>
#include <stdexcept>

void process(bool fail) {
    auto p = std::make_unique<int>(10);

    if (fail) {
        throw std::runtime_error("error");
    }
}
```

Now cleanup happens automatically regardless of how the function exits.

1.6 RAII: The Central Memory Principle

RAII means Resource Acquisition Is Initialization. In C++, resources are tied to object lifetime. When the object is created, it acquires the resource. When the object is destroyed, it releases the resource.

Memory is only one example. RAII also applies to:

- files
- sockets
- mutexes
- database handles
- temporary state changes

For memory, RAII means you do not write code that "remembers" to free memory later. You build objects whose destructors do that automatically.

1.6.1 A Small RAII Example

```
#include <iostream>

class Buffer {
private:
    int* data;
    std::size_t size;

public:
```

```
    explicit Buffer(std::size_t n)
        : data(new int[n]), size(n) {}

    ~Buffer() {
        delete[] data;
    }

    int& operator[](std::size_t i) {
        return data[i];
    }

    std::size_t length() const {
        return size;
    }
};
```

This class is safer than using raw arrays directly because cleanup is tied to destruction. However, in modern C++, even this class would often be better implemented with `std::vector<int>`.

1.7 The Best Default: Use Standard Library Containers

For most real code, the best way to allocate memory is to let a standard container do it.

1.7.1 `std::vector`

`std::vector` is the most important dynamic storage tool in modern C++.

```
#include <vector>

int main() {
    std::vector<int> values{10, 20, 30};
    values.push_back(40);
}
```

Advantages:

- automatic memory management
- contiguous storage
- amortized growth
- standard algorithms work naturally with it
- good cache locality

- expressive and widely understood

In many programs, if you think you need a dynamic array, you really want `std::vector`.

1.7.2 `std::string`

`std::string` is also a managed dynamic container.

```
#include <string>

std::string message = "Memory is managed for you";
message += " safely";
```

Never use raw `char*` for ordinary owned text unless interfacing with low-level APIs.

1.7.3 `std::array`

For fixed-size storage known at compile time:

```
#include <array>

std::array<int, 5> data{1, 2, 3, 4, 5};
```

This avoids dynamic allocation completely while still giving container semantics.

1.7.4 Container-Centered Thinking

A strong modern rule is:

- if you need many elements, use a container
- if you need one dynamically owned object, use a smart pointer
- if you need a temporary fixed-size object, prefer automatic storage

This alone eliminates much bad memory code.

1.8 Smart Pointers: Controlled Dynamic Lifetime

Smart pointers exist because sometimes dynamic allocation is the right tool, but raw owning pointers are too unsafe.

The three major smart pointers are:

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

1.8.1 `std::unique_ptr`: The Best Default for Dynamic Ownership

A `std::unique_ptr` represents exclusive ownership.

```
#include <memory>

struct Engine {
    void start() {}
};

int main() {
    auto engine = std::make_unique<Engine>();
    engine->start();
}
```

Why it is excellent:

- ownership is explicit
- destruction is automatic
- no reference counting overhead
- move-only semantics prevent accidental copying

In modern C++, if a dynamically allocated object has one clear owner, `std::unique_ptr` is usually the right answer.

1.8.2 Moving a `std::unique_ptr`

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

struct Task {
    int id;
    explicit Task(int v) : id(v) {}
};

int main() {
    std::vector<std::unique_ptr<Task>> tasks;

    auto t = std::make_unique<Task>(1);
    tasks.push_back(std::move(t));
}
```

Ownership is transferred into the vector.

1.8.3 `std::shared_ptr`: Shared Ownership

Use `std::shared_ptr` only when multiple parts of the system truly share lifetime responsibility.

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

struct Config {
    int value = 100;
};

int main() {
    auto cfg = std::make_shared<Config>();
    auto copy = cfg;

    std::cout << cfg.use_count() << '\n';
}
```

It is useful, but often overused.
Costs include:

- reference counting overhead
- less obvious lifetime
- possible cyclic ownership
- extra atomic cost in many implementations

1.8.4 When `std::shared_ptr` Is Appropriate

Use it when:

- no single owner exists
- several long-lived components must co-own the same object
- the design naturally models shared lifetime

Do not use it merely to avoid thinking about ownership.

1.8.5 `std::weak_ptr`: Non-Owning Observation

`std::weak_ptr` is used with `std::shared_ptr` to observe an object without extending its lifetime.

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

int main() {
    auto shared = std::make_shared<int>(42);
    std::weak_ptr<int> weak = shared;

    if (auto locked = weak.lock()) {
        std::cout << *locked << '\n';
    }
}
```

This is especially important for graphs, callback systems, caches, and observer relationships.

1.8.6 Breaking Cycles

A classic misuse of `std::shared_ptr` is creating cycles:

```
#include <memory>

struct B;

struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::shared_ptr<A> a;
};
```

Neither object can be released because each keeps the other alive. One side must become `std::weak_ptr`.

1.9 Use `std::make_unique` and `std::make_shared`

Prefer factory helpers instead of writing raw `new` yourself.

```
auto p1 = std::make_unique<int>(10);
auto p2 = std::make_shared<int>(20);
```

Benefits:

- clearer syntax
- better exception safety
- avoids repeating type names
- often more efficient, especially for `shared_ptr`

1.9.1 Avoid This Style

```
std::unique_ptr<int> p(new int(10));
```

Use:

```
auto p = std::make_unique<int>(10);
```

1.10 Raw Pointers Are Still Useful, But Not For Ownership

Modern C++ does not ban raw pointers. It changes their role.

A raw pointer is often best interpreted as:

- a non-owning reference to an object
- a nullable observer
- an interface to low-level APIs
- a tool for iteration or interoperation

1.10.1 Non-Owning Parameter

```
void print_value(const int* p) {
    if (p) {
        // use *p
    }
}
```

This does not mean the function owns the integer. It merely observes it.

1.10.2 Important Rule

Do not return or store raw pointers as owning handles unless the design clearly documents external ownership and lifetime.

1.11 References vs Pointers

A reference often expresses a stronger contract than a pointer.

Use references when:

- null is not allowed
- the object must exist
- the function only borrows access

```
void increment(int& value) {
    ++value;
}
```

Use pointers when:

- null is meaningful
- rebinding is needed
- pointer-like API compatibility is required

Modern memory design is not only about allocation. It is also about representing intent precisely.

1.12 Views Instead of Ownership: `std::span`

One of the most useful modern tools is `std::span`. It provides a non-owning view of contiguous memory.

```
#include <span>
#include <vector>
#include <array>
#include <iostream>

void print_all(std::span<const int> values) {
    for (int v : values) {
        std::cout << v << ' ';
    }
}

int main() {
    std::vector<int> v{1, 2, 3, 4};
    std::array<int, 3> a{5, 6, 7};

    print_all(v);
    print_all(a);
}
```

1.12.1 Why `std::span` Matters

Before `std::span`, functions often accepted:

- pointer plus length
- raw arrays
- container-specific interfaces

`std::span` gives a clean, standard, safer abstraction.

It does not own memory. It only views memory that must remain valid for the span's lifetime.

1.12.2 A Warning About Lifetime

```
std::span<int> bad() {
    std::vector<int> temp{1, 2, 3};
    return temp; // dangerous: span would refer to dead memory
}
```

A span is never a lifetime manager. It is only a view.

1.13 Avoid Raw Arrays For Ownership

Code like this is outdated for ordinary application design:

```
int* data = new int[100];
delete[] data;
```

Use one of these instead:

- `std::array<T, N>` for fixed-size arrays
- `std::vector<T>` for dynamic-size arrays
- `std::string` for text
- `std::unique_ptr<T[]>` only in rare specialized cases

1.13.1 Example: Fixed Size

```
std::array<int, 4> values{10, 20, 30, 40};
```

1.13.2 Example: Dynamic Size

```
std::vector<int> values(100);
```

1.14 When `std::vector` Is Better Than Manual Allocation

Suppose you need a dynamic buffer.

The old way:

```
std::size_t n = 1024;
char* buffer = new char[n];

// use buffer

delete[] buffer;
```

The modern way:

```
std::size_t n = 1024;
std::vector<char> buffer(n);

// use buffer.data() if raw access is needed
```

Why this is better:

- automatic cleanup
- size is stored with the object
- compatible with algorithms
- less code
- safer in the presence of exceptions

1.15 Reserve Capacity To Reduce Reallocations

A dynamic container may reallocate as it grows. Reallocation can be expensive and can invalidate pointers, references, and iterators.

```
#include <vector>

int main() {
    std::vector<int> values;
    values.reserve(1000);

    for (int i = 0; i < 1000; ++i) {
        values.push_back(i);
    }
}
```

If you already know approximately how many elements will be stored, use `reserve()`. This is not about correctness only. It is a practical performance habit.

1.16 Emplace and In-Place Construction

Modern containers often support in-place construction through `emplace_back()` and related operations.

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

struct User {
    std::string name;
    int age;

    User(std::string n, int a) : name(std::move(n)), age(a) {}
};

int main() {
    std::vector<User> users;
    users.emplace_back("Ayman", 40);
}
```

This can avoid temporary objects and may improve clarity and efficiency.

1.17 Memory Ownership Patterns

A good C++ programmer learns to recognize a few common ownership patterns.

1.17.1 Pattern 1: Local Automatic Object

Use when lifetime is limited to the current scope.

```
FileParser parser;
parser.run();
```

1.17.2 Pattern 2: Owned Container

Use when one object owns many values.

```
std::vector<int> scores;
```

1.17.3 Pattern 3: Exclusive Dynamic Object

Use `std::unique_ptr` when one object owns another object dynamically.

```
std::unique_ptr<Engine> engine = std::make_unique<Engine>();
```

1.17.4 Pattern 4: Shared Dynamic Object

Use `std::shared_ptr` only when ownership is genuinely shared.

```
std::shared_ptr<Config> cfg = std::make_shared<Config>();
```

1.17.5 Pattern 5: Borrowed View

Use reference, pointer, or `std::span` when observing data without owning it.

```
void render(const Mesh& mesh);
void process(std::span<float> samples);
```

Much memory confusion disappears when these patterns are consciously applied.

1.18 Polymorphism and Dynamic Allocation

Polymorphic designs often require dynamic lifetime, because the concrete type may be chosen at runtime.

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

class Shape {
public:
    virtual ~Shape() = default;
    virtual void draw() const = 0;
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Circle\n";
    }
};

int main() {
    std::unique_ptr<Shape> s = std::make_unique<Circle>();
    s->draw();
}
```

This is a very common and correct use of heap allocation in modern C++.

1.18.1 Important Note

Whenever a base class is used polymorphically, it should usually have a virtual destructor.

1.19 PImpl and Reducing Compilation Dependencies

One excellent use of `std::unique_ptr` is the PImpl idiom, where implementation details are hidden behind an incomplete type.

```
#include <memory>

class Widget {
public:
    Widget();
    ~Widget();

    void run();

private:
    struct Impl;
    std::unique_ptr<Impl> impl;
};
```

This can reduce header dependencies, improve build times, and isolate implementation details.

1.20 Custom Deleters

Sometimes memory or resources must be released by a custom function instead of ordinary `delete`.

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

int main() {
    std::unique_ptr<FILE, decltype(&std::fclose)>
        file(std::fopen("data.txt", "r"), &std::fclose);
}
```

This is an excellent example of how smart pointers generalize RAII beyond ordinary memory.

1.21 Object Lifetime Is More Important Than Allocation Syntax

Many bugs that look like allocation bugs are really lifetime bugs.

Examples:

- returning a pointer or reference to a dead local object
- storing pointers into a vector that later reallocates
- using a `std::span` after the source buffer is destroyed
- using iterators after container modification

1.21.1 Classic Lifetime Error

```
const std::string& bad() {
    std::string temp = "danger";
    return temp;
}
```

The issue is not allocation. The issue is that the referred object no longer exists.

A professional memory mindset must always track lifetime, not only storage origin.

1.22 Move Semantics and Memory Efficiency

Modern C++ becomes much more efficient when objects can transfer resources instead of copying them.

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

std::vector<std::string> create_names() {
    std::vector<std::string> names;
    names.push_back("one");
    names.push_back("two");
    return names;
}
```

The return is usually efficient because modern C++ uses move semantics and copy elision.

This means the best memory allocation strategy is often tied to value semantics rather than pointer-heavy design.

1.22.1 Value Types Are Powerful

A common beginner mistake is to dynamically allocate too many things individually. Modern C++ often performs better when objects are stored by value in containers, especially when locality matters.

```
std::vector<User> users;
```

This is often better than:

```
std::vector<std::unique_ptr<User>> users;
```

unless stable addresses, polymorphism, or special lifetime control is required.

1.23 Cache Locality and Allocation Strategy

Memory performance is not only about how fast allocation occurs. It is also about where data is located.

Contiguous data is often much faster to process than scattered heap objects because modern CPUs rely heavily on cache locality.

1.23.1 Better Locality

```
std::vector<Particle> particles;
```

1.23.2 Worse Locality

```
std::vector<std::unique_ptr<Particle>> particles;
```

The second form introduces an extra level of indirection and often scatters objects across memory.

Therefore, one of the best memory practices in modern C++ is to store data densely whenever possible.

1.24 Small Objects and Over-Allocation

It is often a mistake to dynamically allocate tiny objects one by one if they could simply be stored directly.

Bad instinct:

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

Better in many ordinary cases:

```
int value = 5;
```

Not every object deserves independent dynamic lifetime. Heap allocation has cost. It should not be treated as free.

1.25 Memory Fragmentation

Frequent allocation and deallocation of many differently sized objects can fragment memory over time. In ordinary application code, this is often acceptable, but in long-running and performance-critical systems it can become important.

Typical responses include:

- reducing allocation frequency
- reusing objects
- reserving container capacity
- using pools or arenas
- using polymorphic memory resources

This is where advanced allocation strategies begin to matter.

1.26 Pool and Arena Style Allocation

Sometimes many small objects are created and destroyed as a group. In such cases, general-purpose per-object allocation may be unnecessarily expensive.

An arena or pool allocates a larger block and serves smaller allocations from it.

This approach can improve:

- speed
- locality
- fragmentation behavior
- simplified group destruction

1.26.1 Conceptual Use Case

Examples where arena-like thinking is useful:

- parsers
- AST construction
- temporary request processing
- game frame allocations
- short-lived batch operations

1.27 Polymorphic Memory Resources

Modern C++ includes the PMR facilities in `<memory_resource>`. These are powerful tools for customizing allocation behavior without rewriting containers.

1.27.1 A Basic PMR Example

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

int main() {
    std::pmr::monotonic_buffer_resource pool;
    std::pmr::vector<int> values{&pool};

    for (int i = 0; i < 10; ++i) {
        values.push_back(i * 10);
    }

    for (int v : values) {
        std::cout << v << ' ';
    }
}
```

1.27.2 What PMR Gives You

PMR separates the container interface from the allocation strategy. This is valuable because the same logical container can be used with different memory behavior depending on the application.

Important PMR tools include:

- `std::pmr::vector`
- `std::pmr::string`
- `std::pmr::monotonic_buffer_resource`
- `std::pmr::unsynchronized_pool_resource`
- `std::pmr::synchronized_pool_resource`

1.27.3 Monotonic Buffer Resource

A monotonic resource allocates memory in growing chunks and generally releases it all at once when the resource is destroyed. It is very effective when objects have roughly the same lifetime.

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

int main() {
    std::array<std::byte, 4096> buffer;
    std::pmr::monotonic_buffer_resource pool(buffer.data(), buffer.size());

    std::pmr::vector<int> values{&pool};

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

1.27.4 When PMR Is Worth It

Use PMR when:

- allocation patterns are performance-critical
- many short-lived allocations occur
- you want container-level allocation control
- you want to experiment with allocation strategies cleanly

Do not force PMR into simple code where ordinary containers are already sufficient.

1.28 Alignment

Some objects require or benefit from specific alignment. Modern C++ provides alignment support directly.

```cpp
#include <iostream>

struct alignas(64) CacheLineData {
    int values[16];
};

int main() {
    std::cout << alignof(CacheLineData) << '\n';
}
```

Alignment matters in low-level systems code, SIMD-related work, lock-free data structures, and false-sharing reduction.

1.29 Placement New

Placement new constructs an object in already-allocated memory.

```
#include <new>
#include <iostream>

struct Item {
    int x;
    Item(int v) : x(v) {}
};

int main() {
    alignas(Item) unsigned char buffer[sizeof(Item)];
    Item* p = new (buffer) Item(42);

    std::cout << p->x << '\n';

    p->~Item();
}
```

This is an advanced technique. It is useful in allocators, pools, embedded systems, and low-level frameworks. It is not an everyday allocation tool for ordinary application logic.

1.30 C-Style Allocation: `malloc` and `free`

C++ can use `malloc` and `free`, but they are generally inappropriate for ordinary C++ objects because they do not call constructors or destructors.

```
int* p = static_cast<int*>(std::malloc(sizeof(int)));
std::free(p);
```

This may be acceptable for raw byte storage or interoperation, but it is not the normal modern C++ way to manage object lifetime.

1.30.1 Why It Is Usually Wrong For Objects

- constructors are skipped
- destructors are skipped
- mixing `new/delete` with `malloc/free` is invalid

1.31 Avoid Mixing Allocation Families

Never do this:

```
int* p = new int;
std::free(p); // wrong
```

And never do this:

```
int* p = static_cast<int*>(std::malloc(sizeof(int)));
delete p; // wrong
```

Each family has its own matching deallocation rules.

1.32 Exception Safety and Memory

A major reason modern C++ prefers RAII and containers is exception safety.

If an exception occurs in code with manual cleanup, memory can leak. If objects own their resources properly, unwinding automatically releases them.

1.32.1 Safe Design Pattern

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

void process() {
    std::vector<std::string> lines;
    lines.push_back("one");
    lines.push_back("two");
}
```

No matter how the function exits, the vector cleans up correctly.

1.33 Common Memory Mistakes In C++

1.33.1 Mistake 1: Using `new` For Ordinary Objects

```
auto name = std::make_unique<std::string>("hello");
```

This is technically fine, but often unnecessary. If dynamic lifetime is not needed, just write:

```
std::string name = "hello";
```

1.33.2 Mistake 2: Overusing `std::shared_ptr`

Many programmers use `std::shared_ptr` because it feels safer than thinking. But it often hides poor ownership design.

1.33.3 Mistake 3: Returning Pointers To Internal Temporary Data

```
const char* bad() {
    std::string s = "temporary";
    return s.c_str();
}
```

The pointer becomes invalid after the string dies.

1.33.4 Mistake 4: Storing Dangling Views

A `std::span`, reference, or raw pointer is only valid while the underlying object is alive.

1.33.5 Mistake 5: Ignoring Reallocation Invalidation

```
std::vector<int> v{1, 2, 3};
int* p = v.data();

v.push_back(4); // may reallocate, p may now be invalid
```

1.34 A Practical Decision Guide

When deciding how to allocate memory, the following guide is useful.

1.34.1 Case 1: One Ordinary Local Object

Use a normal local variable.

```
Widget w;
```

1.34.2 Case 2: Fixed Number of Elements

Use `std::array`.

```
std::array<int, 8> data;
```

1.34.3 Case 3: Dynamic Number of Elements

Use `std::vector`.

```
std::vector<int> data;
```

1.34.4 Case 4: One Dynamically Owned Object

Use `std::unique_ptr`.

```
auto obj = std::make_unique<Widget>();
```

1.34.5 Case 5: Truly Shared Lifetime

Use `std::shared_ptr`.

```
auto obj = std::make_shared<Widget>();
```

1.34.6 Case 6: Borrowing Data Without Owning

Use reference, pointer, or `std::span`.

```
void use(const Widget& w);
void use_many(std::span<const Widget> ws);
```

1.34.7 Case 7: Performance-Critical Specialized Allocation

Consider PMR, pooling, or custom allocators.

1.35 Real Example: A Better Design For A Dynamic System

Consider a small job system.

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

class Job {
public:
    virtual ~Job() = default;
    virtual void run() = 0;
};

class PrintJob : public Job {
private:
    std::string message;

public:
    explicit PrintJob(std::string m) : message(std::move(m)) {}
```

```
    void run() override {
        std::cout << message << '\n';
    }
};

class JobQueue {
private:
    std::vector<std::unique_ptr<Job>> jobs;

public:
    template<typename T, typename... Args>
    void add_job(Args&&... args) {
        jobs.push_back(std::make_unique<T>(std::forward<Args>(args)...));
    }

    void run_all() {
        for (auto& job : jobs) {
            job->run();
        }
    }
};

int main() {
    JobQueue queue;
    queue.add_job<PrintJob>("Task 1");
    queue.add_job<PrintJob>("Task 2");
    queue.run_all();
}
```

Why this design is good:

- the queue owns all jobs
- ownership is exclusive and clear
- polymorphism is supported
- cleanup is automatic
- there is no manual `delete`

1.36 Real Example: Borrowing A Buffer Safely

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

```cpp
#include <iostream>

void normalize(std::span<double> values) {
    if (values.empty()) return;

    double first = values.front();
    for (double& v : values) {
        v -= first;
    }
}

int main() {
    std::vector<double> samples{10.5, 12.0, 15.5};
    normalize(samples);

    for (double v : samples) {
        std::cout << v << ' ';
    }
}
```

This is a good modern pattern because the function works with many contiguous sources without taking ownership.

1.37 Real Example: PMR For Temporary Parsing Data

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

int main() {
    std::pmr::monotonic_buffer_resource pool;
    std::pmr::vector<std::pmr::string> tokens{&pool};

    tokens.emplace_back("let", &pool);
    tokens.emplace_back("value", &pool);
    tokens.emplace_back("=", &pool);
    tokens.emplace_back("42", &pool);

    for (const auto& t : tokens) {
        std::cout << t << '\n';
    }
}
```

This style is attractive when many temporary small strings and containers are created and destroyed together.

1.38 What Changed In Modern C++ Thinking

Older C++ culture often admired manual management as a sign of skill. Modern C++ recognizes something deeper: the best engineer is not the one who manually frees memory everywhere, but the one who designs software where ownership is obvious and cleanup is automatic.

This is not about reducing power. It is about using power correctly.

Modern C++ style prefers:

- value semantics when possible
- containers over manual arrays
- `std::unique_ptr` over raw owning pointers
- explicit borrowing through references, pointers, and spans
- custom allocation only where it truly matters

1.39 Recommended Memory Allocation Hierarchy

A useful hierarchy for everyday design is:

1. Prefer ordinary local objects
2. Prefer `std::array` for fixed-size groups
3. Prefer `std::vector` and other standard containers for dynamic groups
4. Use `std::unique_ptr` for exclusive dynamic ownership
5. Use `std::shared_ptr` only when shared lifetime is truly required
6. Use `std::span` or references for borrowing
7. Use PMR or custom allocators only for advanced performance cases
8. Use raw `new/delete` only in rare low-level implementation work

This ordering is practical, modern, and highly effective.

1.40 Short Best Practices Checklist

- Prefer value semantics
- Avoid explicit `new/delete` in application code
- Use `std::make_unique` by default for dynamic single ownership
- Use `std::make_shared` only where shared ownership is real
- Prefer `std::vector` over dynamic raw arrays
- Reserve capacity when growth size is known
- Use `std::span` for non-owning contiguous access
- Watch iterator, pointer, and reference invalidation
- Design lifetime first, allocation second
- Optimize allocation strategy only after identifying real need

1.41 Final Thoughts

The best way to allocate memory in C++20 and C++23 is not a single syntax rule. It is a hierarchy of good design decisions.

If an object can live naturally as a local value, do that.

If a group of objects belongs together, use a standard container.

If dynamic ownership is necessary, use `std::unique_ptr` by default.

If ownership is shared for real architectural reasons, use `std::shared_ptr` carefully.

If you only need to view memory, borrow it with references, pointers, or `std::span`.

And if performance demands specialized control, modern C++ gives you advanced tools such as PMR, custom allocators, alignment facilities, and placement construction.

The true spirit of modern C++ memory management is this: keep control, but express it through design, not through repetitive manual cleanup.

That is the best allocation strategy in modern C++.

Chapter 2

Deep Practical Examples and Real-World Memory Design

2.1 Introduction

Understanding memory allocation in modern C++ is not about memorizing tools such as `std::unique_ptr` or `std::vector`. It is about understanding how real systems behave, how lifetime flows through a program, and how small design decisions affect correctness, performance, and maintainability.

This chapter focuses on practical patterns, common mistakes, performance considerations, and realistic system design.

2.2 Case Study 1: Configuration System Design

A configuration object is often shared across many components.

2.2.1 Bad Design: Global Raw Pointer

```
Config* global_config = nullptr;

void init() {
    global_config = new Config();
}
```

Problems:

- unclear ownership
- no clear destruction point
- potential leak
- difficult testing

2.2.2 Better Design: Shared Ownership Only If Necessary

```
#include <memory>

class Config {
public:
    int value = 42;
};

std::shared_ptr<Config> create_config() {
    return std::make_shared<Config>();
}
```

2.2.3 Best Design: Explicit Ownership Passing

```
class System {
private:
    const Config& config;

public:
    explicit System(const Config& cfg) : config(cfg) {}

    void run() {
        // use config safely
    }
};
```

Key insight:

- avoid global ownership
- prefer passing references when lifetime is guaranteed

2.3 Case Study 2: Large Data Processing Pipeline

Suppose you are processing millions of records.

2.3.1 Bad Design: Fragmented Allocation

```
std::vector<std::unique_ptr<Record>> records;
```

Problems:

- poor cache locality
- many heap allocations
- pointer indirection

2.3.2 Better Design: Contiguous Storage

```
std::vector<Record> records;
```

Advantages:

- contiguous memory
- fewer allocations
- better CPU cache usage

2.3.3 When Pointer Storage Is Justified

Use pointer-based storage only when:

- polymorphism is required
- object size is extremely large
- object lifetime is independent

2.4 Case Study 3: Object Graph (Tree Structure)

2.4.1 Correct Ownership Model

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

struct Node {
    int value;
    std::vector<std::unique_ptr<Node>> children;
};
```

Here:

- parent owns children
- destruction is recursive and automatic

2.4.2 Avoid This Design

```
struct Node {
    std::vector<Node*> children;
};
```

Problems:

- unclear ownership
- memory leaks likely

2.5 Case Study 4: Observer Pattern Without Leaks

2.5.1 Dangerous Version

```
std::vector<std::shared_ptr<Listener>> listeners;
```

If listeners also reference the source, cycles occur.

2.5.2 Correct Version

```
std::vector<std::weak_ptr<Listener>> listeners;
```

```
for (auto& w : listeners) {
    if (auto s = w.lock()) {
        s->on_event();
    }
}
```

2.6 Case Study 5: High-Frequency Allocation System

Suppose a system allocates thousands of small objects per frame.

2.6.1 Naive Approach

```
for (...) {
    auto obj = std::make_unique<Item>();
}
```

Problems:

- heavy allocation overhead
- fragmentation risk

2.6.2 Better Approach: Object Pool Concept

Reuse objects instead of allocating repeatedly.

2.6.3 Best Approach: PMR Arena

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

std::pmr::monotonic_buffer_resource pool;
std::pmr::vector<int> values{&pool};
```

All allocations are grouped and released together.

2.7 Lifetime Bugs That Professionals Watch For

2.7.1 Dangling Pointer

```
int* bad() {
    int x = 10;
    return &x;
}
```

2.7.2 Iterator Invalidation

```
std::vector<int> v{1,2,3};
int* p = v.data();

v.push_back(4); // invalidates p
```

2.7.3 Temporary Object Lifetime

```
const std::string& ref = std::string("temp");
```

The reference becomes invalid.

2.8 Performance Insight: Allocation Is Not Free

Even in modern systems:

- allocation may lock internal structures
- deallocation may be expensive
- memory locality matters more than allocation speed

2.8.1 Better Strategy

- reduce number of allocations
- reuse memory
- prefer contiguous structures

2.9 False Sharing and Alignment

In multithreaded systems, two threads writing to adjacent memory can cause performance degradation.

```
struct alignas(64) Counter {
    int value;
};
```

This prevents cache line sharing.

2.10 Designing APIs With Memory in Mind

Bad API:

```
Widget* create_widget();
```

Better API:

```
std::unique_ptr<Widget> create_widget();
```

Best API when ownership is external:

```
void process(const Widget& w);
```

APIs should express ownership clearly.

2.11 When Not To Use Dynamic Allocation

Avoid dynamic allocation when:

- object is small
- lifetime is local
- no polymorphism is needed

Example:

```
int x = 5; // better than unique_ptr<int>
```

2.12 Advanced Pattern: Stable Addresses

Sometimes objects must not move in memory.
Example:

- storing pointers into objects
- interfacing with C APIs

Solution:

```
std::vector<std::unique_ptr<Object>> objects;
```

Here, objects are stable even if vector resizes.

2.13 Advanced Pattern: Custom Allocation Strategy

```
#include <memory_resource>

std::pmr::unsynchronized_pool_resource pool;
std::pmr::vector<int> data{&pool};
```

Use when:

- allocation patterns are predictable
- performance is critical

2.14 Real-World Anti-Patterns

2.14.1 Using shared_ptr Everywhere

This hides design problems.

2.14.2 Returning Raw Owning Pointers

Unclear ownership leads to bugs.

2.14.3 Over-Allocating Small Objects

Too many allocations reduce performance.

2.15 A Professional Memory Mindset

A strong C++ engineer:

- designs ownership first
- minimizes allocation count
- prefers locality over fragmentation
- uses abstraction without losing control
- understands lifetime deeply

2.16 Final Practical Checklist

- Is this allocation necessary?
- Who owns this object?
- Can I use a container instead?
- Can I avoid allocation entirely?
- Will this design scale?
- Is lifetime obvious?

2.17 Final Thoughts

Memory management in modern C++ is no longer about manual control versus safety. It is about structured control.

The best systems are not those that allocate cleverly, but those that allocate rarely, predictably, and correctly.

When you reach that level, memory management stops being a source of bugs and becomes a silent strength of your design.

Chapter 3

Advanced Memory Engineering

3.1 Introduction

At an advanced level, memory management is no longer about choosing between `std::vector` and `std::unique_ptr`. It becomes a systems problem involving allocation strategies, cache behavior, threading, fragmentation, and interaction with the operating system.

This chapter explores how memory actually behaves beneath modern C++ abstractions and how expert developers design systems that scale efficiently.

3.2 What Happens When You Call `new`

The expression:

```
auto p = new int(42);
```

does more than allocate memory.

The process typically includes:

1. calling a global or class-specific allocation function
2. requesting memory from the runtime allocator
3. possibly interacting with OS-level memory managers
4. returning a suitably aligned memory block
5. invoking the constructor

Similarly, `delete`:

1. calls the destructor
2. returns memory to the allocator

This means allocation cost includes:

- allocator overhead
- possible locking
- fragmentation management
- cache effects

3.3 Global Allocators and Their Behavior

Most systems use a general-purpose allocator such as:

- glibc malloc
- jemalloc
- tcmalloc

These allocators attempt to balance:

- speed
- fragmentation control
- multi-thread scalability

However, no general-purpose allocator is optimal for all workloads.

3.3.1 Key Observation

If allocation patterns are predictable, a custom allocator or pool can outperform general-purpose allocators significantly.

3.4 Allocation Cost Model

Allocation is not constant time in practice.

Factors include:

- size class lookup
- free list traversal
- lock contention
- cache misses
- system calls for large allocations

Therefore:

- many small allocations are expensive
- allocation frequency matters more than allocation size

3.5 Fragmentation

Fragmentation occurs when free memory exists but is not usable efficiently.

3.5.1 Types of Fragmentation

- internal fragmentation: unused space inside allocated blocks
- external fragmentation: scattered free regions

3.5.2 Impact

- increased memory usage
- reduced cache efficiency
- unpredictable performance

3.6 Cache Hierarchy and Memory Layout

Modern CPUs rely heavily on cache:

- L1 cache (very fast, small)
- L2 cache
- L3 cache (shared)
- main memory (slow)

Performance is often dominated by memory access patterns rather than computation.

3.6.1 Contiguous Data Advantage

```
std::vector<int> data;
```

Sequential access benefits from:

- spatial locality
- cache line prefetching

3.6.2 Pointer-Based Layout Disadvantage

```
std::vector<std::unique_ptr<int>> data;
```

This introduces:

- pointer indirection
- scattered memory
- cache misses

3.7 False Sharing

False sharing occurs when multiple threads modify data located on the same cache line.

```
struct Counter {
    int value;
};
```

If multiple threads update adjacent counters, performance degrades.

3.7.1 Solution: Alignment

```
struct alignas(64) Counter {
    int value;
};
```

This ensures each instance occupies its own cache line.

3.8 NUMA Awareness

In multi-socket systems, memory is not uniformly accessible.

- local memory is faster
- remote memory is slower

Advanced systems may:

- allocate memory per thread
- bind threads to cores
- use NUMA-aware allocators

This level of optimization is critical in high-performance computing and large-scale servers.

3.9 Custom Allocators in C++

C++ allows custom allocators through allocator-aware containers.

```
template<typename T>
struct MyAllocator {
    using value_type = T;

    T* allocate(std::size_t n) {
        return static_cast<T*>(::operator new(n * sizeof(T)));
    }

    void deallocate(T* p, std::size_t) {
        ::operator delete(p);
    }
};
```

3.9.1 Usage

```
std::vector<int, MyAllocator<int>> data;
```

3.9.2 When Useful

- tracking allocations
- custom memory pools
- performance tuning

3.10 Object Pools

Object pools reuse memory instead of allocating repeatedly.

3.10.1 Simple Concept

- allocate a block
- reuse objects from it
- return objects to pool instead of deleting

3.10.2 Example Skeleton

```
#include <vector>

template<typename T>
class ObjectPool {
private:
    std::vector<T*> free_list;

public:
    T* acquire() {
        if (free_list.empty()) {
            return new T();
        }
        T* obj = free_list.back();
        free_list.pop_back();
        return obj;
    }

    void release(T* obj) {
        free_list.push_back(obj);
    }
};
```

3.10.3 Advantages

- reduces allocation overhead
- improves locality

3.11 Arena Allocation

Arena allocators allocate large blocks and free them all at once.

3.11.1 Use Case

- parsing
- AST construction
- temporary computation phases

3.11.2 Key Benefit

- near-zero deallocation cost

3.12 PMR Deep Dive

Polymorphic memory resources allow runtime selection of allocation strategy.

3.12.1 Example

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

std::pmr::unsynchronized_pool_resource pool;
std::pmr::vector<int> data{&pool};
```

3.12.2 Important Resources

- monotonic_buffer_resource
- unsynchronized_pool_resource
- synchronized_pool_resource

3.12.3 Design Insight

PMR separates:

- container logic
- memory strategy

This is one of the most powerful modern C++ features.

3.13 Lock-Free Allocation Concepts

High-performance systems sometimes avoid locks in allocation.

Techniques include:

- thread-local storage
- per-thread pools
- lock-free free lists

These designs reduce contention but increase complexity.

3.14 Overriding Global `new/delete`

You can override global allocation:

```
void* operator new(std::size_t size) {
    // custom allocation
    return std::malloc(size);
}
```

3.14.1 Use Cases

- profiling
- debugging
- custom memory tracking

3.15 Memory Tracking and Debugging

Advanced systems track memory usage:

- allocation counts
- peak memory
- leak detection

Techniques include:

- custom allocators
- instrumentation
- logging allocation events

3.16 When Optimization Is Worth It

Not all code requires advanced memory tuning.

Optimize when:

- profiling shows allocation bottlenecks
- system is latency-sensitive
- memory usage is high

Do not optimize prematurely.

3.17 Design Philosophy at Expert Level

At this level, memory is not a detail. It is part of architecture.

A strong engineer:

- minimizes allocation frequency
- maximizes locality
- understands hardware behavior
- chooses allocation strategies intentionally

3.18 Final Thoughts

Modern C++ gives you full control over memory, but also the responsibility to use it wisely.

The progression is:

- beginner: learns `new/delete`
- intermediate: uses smart pointers and containers
- advanced: designs ownership correctly
- expert: controls memory behavior at system level

At the expert level, memory allocation is no longer a coding detail. It is a performance, scalability, and architectural concern.

Mastering it is one of the defining skills of a true C++ engineer.

Appendices

Appendix A — Memory Allocation Cheat Sheet

Quick Decision Table

Situation	Recommended Tool
Single local object	Plain variable
Fixed-size array	std::array
Dynamic-size array	std::vector
Single owned dynamic object	std::unique_ptr
Shared ownership	std::shared_ptr
Non-owning access	reference / pointer / std::span
Temporary bulk allocation	PMR (monotonic_buffer_resource)
High-performance allocation	pool / custom allocator
Polymorphic objects	std::unique_ptr¡Base¿

Golden Rules

- Prefer value semantics
- Avoid raw `new/delete`
- Use `std::unique_ptr` by default for ownership
- Use `std::shared_ptr` only when necessary
- Prefer contiguous memory structures
- Design lifetime before allocation

Common Mistakes Summary

- Dangling pointers and references
- Overuse of shared_ptr
- Ignoring container reallocation

- Excessive small allocations

Appendix B — Memory Design Decision Tree

Step-by-Step Decision Guide

1. Is dynamic allocation required?
    - No → Use local variable or std::array
    - Yes → Continue
2. Is it a collection?
    - Yes → Use std::vector or container
    - No → Continue
3. Is ownership exclusive?
    - Yes → std::unique_ptr
    - No → Continue
4. Is ownership shared?
    - Yes → std::shared_ptr
    - No → Use raw pointer/reference (non-owning)
5. Is allocation performance critical?
    - Yes → PMR / pool allocator
    - No → standard tools are sufficient

Mental Shortcut

- Default: value or vector
- Ownership: unique_ptr
- Sharing: shared_ptr (rare)
- Viewing: span/reference

Appendix C — Comparison With Rust Memory Model

Philosophical Difference

C++	Rust
Flexible ownership model	Strict ownership rules enforced by compiler
Manual + RAII-based safety	Compile-time safety guarantees
Freedom with responsibility	Safety with restrictions

Basic Comparison

Concept	C++	Rust
Single ownership	std::unique_ptr	Box¡T¿
Shared ownership	std::shared_ptr	Rc¡T¿ / Arc¡T¿
Borrowing	reference / pointer	&T / &mut T
Views	std::span	slices

Key Insight

- C++ gives control and flexibility
- Rust enforces safety at compile time
- Both can produce high-performance systems

When C++ Excels

- legacy systems
- fine-grained control
- custom memory strategies

When Rust Excels

- safety-critical systems
- reducing memory bugs
- concurrency safety

Appendix D — Real Benchmark Scenarios

Scenario 1: Vector vs Pointer Storage

```
std::vector<int> data; // contiguous
std::vector<std::unique_ptr<int>> data_ptr;
```

Expected:

- vector¡int¿ is faster due to locality
- pointer version suffers cache misses

Scenario 2: Reserve vs No Reserve

```
std::vector<int> v;
v.reserve(1000000);
```

Result:

- fewer reallocations
- faster insertion

Scenario 3: Pool vs Standard Allocation

```
// standard allocation
for (...) {
    auto obj = std::make_unique<Item>();
}

// pool allocation
// reuse objects
```

Expected:

- pool is faster under heavy allocation load

Scenario 4: PMR vs Standard Vector

```
std::pmr::monotonic_buffer_resource pool;
std::pmr::vector<int> data{&pool};
```

Result:

- improved performance for short-lived allocations

Scenario 5: False Sharing

```
struct Counter { int value; };
struct alignas(64) CounterAligned { int value; };
```

Result:

- aligned version avoids cache contention

Appendix E — Final Professional Guidelines

Engineering-Level Rules

- Memory is part of architecture, not just implementation
- Allocation frequency matters more than allocation size
- Locality often dominates raw allocation speed
- Ownership must be explicit and simple

What Senior Engineers Do Differently

- avoid unnecessary allocation
- design for cache efficiency
- minimize fragmentation
- use profiling before optimization

Final Message

The evolution of C++ memory management is not about removing control. It is about structuring control.

The strongest systems are not those that use the most advanced allocators, but those that require the least allocation in the first place.

References

Primary Standards and Normative References

1. ISO/IEC 14882, *Programming Languages — C++*. This is the formal language and library standard for C++, including the core memory model, object lifetime rules, dynamic allocation, allocators, smart pointers, and polymorphic memory resources.

2. *Draft C++ Standard* (browser rendering of a recent working draft). Especially relevant clauses for this booklet include the general memory management library, the `<memory>` header, smart pointers, allocator requirements, and polymorphic allocator facilities.

3. *C++ Core Guidelines*, edited by Bjarne Stroustrup and Herb Sutter. This is one of the most important practical references for modern C++ style, especially for RAII, ownership design, smart pointers, resource safety, and avoiding raw owning pointers.

Library References Used Throughout the Booklet

1. `<memory>` — general memory management facilities, smart pointers, allocators, uninitialized memory algorithms, and related utilities.

2. `std::unique_ptr` — the preferred standard tool for exclusive dynamic ownership in modern C++.

3. `std::shared_ptr` and `std::weak_ptr` — used when shared ownership is genuinely required, with `weak_ptr` used to observe without extending lifetime and to break ownership cycles.

4. `std::make_unique` and related construction helpers — preferred factory helpers for safer and clearer dynamic object construction.

5. `std::span` — the standard non-owning view over contiguous memory, useful for expressing borrowing rather than ownership.

6. `std::pmr::polymorphic_allocator` and the polymorphic memory resource facilities — central to advanced allocation strategies where container logic is separated from allocation policy.

7. Allocator requirements and allocator-aware containers — essential background for understanding custom allocation strategies and container integration.

Practical Secondary Reference

1. *cppreference.com*. While not the language standard itself, it is an extremely valuable practical reference for day-to-day C++ work, examples, feature status, library summaries, and quick lookup of standard library facilities.

Recommended Professional Reading

1. Bjarne Stroustrup, *A Tour of C++*. A strong high-level reference for modern C++ design and idioms.

2. Bjarne Stroustrup, *The C++ Programming Language*. A deeper and broader reference for the language and standard library.

3. Scott Meyers, *Effective Modern C++*. Useful for understanding modern ownership, move semantics, initialization, and best practices in post-C++11 C++.

4. Herb Sutter and Andrei Alexandrescu, *C++ Coding Standards*. A practical engineering-oriented reference on writing robust C++ systems.

Reference Notes

This booklet is primarily grounded in the ISO C++ standard library memory facilities, the C++ Core Guidelines, and practical standard-library references. The strongest conceptual foundations for the material presented here are RAII, explicit ownership, allocator-aware design, locality-conscious data structures, and the disciplined use of modern standard abstractions instead of manual raw-memory management.

Closing Note

Modern C++ memory management is not about avoiding complexity. It is about organizing it.

The strongest engineers are not those who control memory manually everywhere, but those who design systems where memory behaves predictably, efficiently, and safely.