The C++ Master Companion — Syntax, Insight & Practice

ZephyrAmmor

October 2025

Contents

Module 3: Memory and Pointers

> "Memory is where your program truly lives — manage it wisely, and it will serve you faithfully. Mismanage it, and it will haunt you."

□ Overview

This module covers one of the most fundamental yet error-prone aspects of C++ — memory management. Mastering pointers, references, and ownership models separates a good programmer from a great one. This is where you learn to think *like the compiler* and *like the CPU*.

□ 3.1 Stack vs Heap

Concept Overview

- Stack: Automatically managed memory for local variables. Fast, but limited in size.
- Heap: Dynamically allocated memory controlled manually by the programmer.

Visual

```
Memory Layout:
+---------------------+  High Address
|     Heap (dynamic)  |  ← grows upward
|---------------------┤
|        Stack        |  ← grows downward
|---------------------┤
|   Code / Data Segs  |
+---------------------+  Low Address
```

Example

```cpp
void example() {
    int a = 10;            // Stored on stack
    int* b = new int(20); // Stored on heap

    std::cout << a << ", " << *b;
    delete b;             // Manual cleanup
}
```

Pitfall

□ Memory Leak: Forgetting to `delete` heap memory results in wasted memory that never returns to the OS.

Insight

□ Stack memory is automatically reclaimed when the function exits, while heap memory must be explicitly freed.

□ 3.2 Pointers and References

Concept Overview

Pointers store addresses, not values. References are aliases — safer and simpler.

Syntax Block

```cpp
int x = 5;
int* ptr = &x;    // pointer to x
int& ref = x;     // reference to x
```

Example

```cpp
void update(int* p, int& r) {
    *p += 10;
    r += 20;
}
```

```cpp
int main() {
    int a = 5;
    update(&a, a);
    std::cout << a; // Output: 35
}
```

Pitfalls

- Dangling Pointer: Using a pointer to deleted memory.
- Null Pointer Dereference: Dereferencing `nullptr` causes a crash.

Insight

□ References cannot be reseated — once bound, they refer to the same object forever.

□ 3.3 Dynamic Memory Allocation

Concept Overview

C++ lets you allocate memory dynamically with `new` and release it with `delete`.

Syntax

```cpp
int* ptr = new int(42);
delete ptr; // Always match new/delete

int* arr = new int[5];
delete[] arr; // Always match new[]/delete[]
```

Best Practice

□ Prefer smart pointers (see below) to avoid manual memory management.

□ 3.4 Smart Pointers

Concept Overview

Smart pointers automate memory management using RAII — Resource Acquisition Is Initialization.

Common Types

Smart Pointer	Header	Ownership Model
`std::unique_ptr`	`<memory>`	Sole ownership
`std::shared_ptr`	`<memory>`	Shared ownership
`std::weak_ptr`	`<memory>`	Non-owning observer

Example

```cpp
#include <memory>

void smart_example() {
    auto p1 = std::make_unique<int>(10);  // unique ownership
    auto p2 = std::make_shared<int>(20);  // shared ownership
    auto p3 = p2;                         // shared ownership count++

    std::weak_ptr<int> wp = p2;           // non-owning reference
}
```

Under the Hood

 Smart pointers use reference counting (for `shared_ptr`) and move semantics (for `unique_ptr`) to ensure deterministic cleanup.

Pitfall

 Cyclic References: Two `shared_ptr` pointing to each other never free memory — use `weak_ptr` to break cycles.

3.5 RAII (Resource Acquisition Is Initialization)

Concept Overview

RAII ensures that resources are acquired and released automatically when objects go in/out of scope.

Example

```cpp
#include <fstream>

void readFile() {
    std::ifstream file("data.txt"); // Opens file
    if (!file) return;
    // File automatically closes when function exits
}
```

Insight

 Constructors acquire resources, destructors release them — no explicit cleanup needed.

3.6 Move Semantics and Ownership

Concept Overview

Introduced in C++11, move semantics transfer ownership of resources instead of copying.

Example

```cpp
#include <utility>
#include <vector>

std::vector<int> makeVector() {
    std::vector<int> v = {1, 2, 3};
    return v; // Moved, not copied
}

int main() {
    auto data = makeVector();
}
```

Under the Hood

 When returning local objects, the compiler uses Return Value Optimization (RVO) or move constructors to avoid deep copies.

Best Practice

 Implement move constructors and assignment operators if your class manages resources manually.

3.7 Common Memory Bugs

Bug Type	Description	Example
Dangling Pointer	Pointer to deallocated memory	`int* p = new int(5); delete p; *p = 10;`
Memory Leak	Memory not released	`new int(5); // no delete`
Double Free	Deleting same pointer twice	`delete p; delete p;`
Uninitialized Pointer	Using pointer before initialization	`int* p; *p = 10;`

Tip

 Use Valgrind, ASan, or Visual Studio Address Sanitizer to detect memory issues.

3.8 Under the Hood: How Allocation Works

- **new** requests memory from the heap allocator (usually `malloc` internally).
- The allocator maintains free lists of available blocks.
- Smart pointers wrap these allocations with destructors that automatically free memory when no longer referenced.

Visualization

```
std::shared_ptr<int> a = std::make_shared<int>(42);
      │
      ├──> Heap: [42]
      └──> Control Block: [ref_count = 1]
```

Insight

 Every dynamic allocation costs time — avoid excessive small allocations in performance-critical code.

3.9 Best Practices Summary

- Prefer stack over heap unless dynamic lifetime is required.
- Use smart pointers instead of raw pointers.
- Follow RAII: wrap resources in objects.
- Avoid manual delete — let destructors or smart pointers handle cleanup.
- Use move semantics for efficiency.
- Regularly test with memory sanitizers.

Availability: Core since C++98; Smart Pointers and Move Semantics added in C++11.

> Mentor's Note: If you truly understand memory and pointers, you understand C++. Everything else builds upon this foundation.