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

ZephyrAmmor

October 2025

Contents

Module 10: Best Practices and Design Patterns in C++

Purpose

This module distills *decades of hard-learned lessons* from real-world C++ engineering — showing not just how to write working code, but how to write clean, efficient, and maintainable systems. Learn design philosophies, coding idioms, and patterns that separate amateurs from professionals.

10.1 Core Principles

Principle	Description	Example
RAII (Resource Acquisition Is Initialization)	Manage resources via object lifetime.	`std::lock_guard<std::mutex> lock(mtx);`
Single Responsibility	A class should do one thing well.	Split file reader vs. parser.
DRY (Don't Repeat Yourself)	Abstract reusable logic.	Use templates or helper functions.
KISS (Keep It Simple, Stupid)	Avoid overengineering.	Prefer clear over clever.
YAGNI (You Aren't Gonna Need It)	Don't write features you "might" need later.	Trim future speculation.
Rule of Zero/Three/Five	Define special members only when necessary.	Use smart pointers for automatic cleanup.

10.2 Coding Best Practices

Naming Conventions

- Use `PascalCase` for classes and structs.
- Use `camelCase` for variables and functions.
- Use `ALL_CAPS` for constants or macros.

Code Clarity

- Avoid magic numbers → use named constants.
- Prefer `enum class` over old-style enums.
- Use meaningful variable names.

Safety Practices

- Use `nullptr` (not `NULL`).
- Always initialize variables.
- Prefer `std::array` or `std::vector` over raw arrays.
- Prefer `unique_ptr` or `shared_ptr` over raw `new/delete`.

10.3 Performance Practices

Memory Management

- Avoid unnecessary copies → use `std::move()` when appropriate.
- Pass large objects by reference-to-const.
- Use `reserve()` for containers when possible.

Efficient Code

- Inline short functions when beneficial.
- Use algorithms (`std::sort`, `std::find`) instead of manual loops.
- Profile before optimizing — avoid premature optimization.

10.4 Error Handling

Strategy

- Use exceptions for recoverable errors.
- Use assertions for internal logic checks.
- Avoid returning raw error codes unless interfacing with C.

Example

```
try {
    processFile("data.txt");
} catch (const std::runtime_error& e) {
    std::cerr << "Error: " << e.what();
}
```

10.5 Design Patterns (Core)

Pattern	Type	Use Case	Example
Singleton	Creational	Global shared instance.	Logger, Config Manager.
Factory Method	Creational	Defer object creation to subclasses.	GUI widgets.
Builder	Creational	Complex object construction step-by-step.	JSON builders.
Observer	Behavioral	Notify multiple objects of changes.	Event systems.

Pattern	Type	Use Case	Example
Strategy	Behavioral	Select algorithm at runtime.	Sorting strategies.
Adapter	Structural	Bridge incompatible interfaces.	Legacy integration.
Decorator	Structural	Add behavior dynamically.	I/O streams.
Command	Behavioral	Encapsulate actions.	Undo/redo system.

10.6 Modern C++ Idioms

Idiom	Description	Example
Pimpl (Pointer to Implementation)	Hide implementation details.	Reduces compile-time dependencies.
SFINAE / Concepts	Constrain templates elegantly.	`requires std::integral<T>`
Non-copyable base	Prevent copying.	`class NonCopyable { NonCopyable(const NonCopyable&) = delete; };`
CRTP (Curiously Recurring Template Pattern)	Static polymorphism.	`template<class D> class Base {}`
RAII Wrappers	Manage any resource automatically.	`std::unique_ptr<File>`

10.7 Code Architecture Best Practices

- Prefer composition over inheritance.
- Keep classes small, cohesive, and loosely coupled.
- Group related code in namespaces.
- Separate interface (`.h`) from implementation (`.cpp`).
- Apply dependency inversion — high-level modules shouldn't depend on low-level details.

10.8 Concurrency Best Practices

- Avoid shared mutable state.
- Use `std::async`, not raw threads, when task-based concurrency fits.
- Use synchronization primitives (`std::mutex`, `std::lock_guard`) safely.
- Keep thread lifetimes short and scoped.

10.9 Safety & Maintainability

- Use static analysis tools (clang-tidy, cppcheck).
- Document all public APIs.
- Write unit tests (GoogleTest, Catch2, doctest).
- Keep functions small and focused.
- Prefer immutability where possible.

10.10 Summary — Professional C++ Mindset

Domain	Guideline	Goal
Code Style	Simple, readable, minimal boilerplate	Maintainability
Memory	Smart pointers, RAII	Safety
Design	Patterns & SOLID principles	Scalability
Performance	Measure before optimizing	Efficiency

Domain	Guideline	Goal
Concurrency	Data safety first	Reliability

Next Module → Module 11: Building Real-World C++ Projects (Architecture, Integration, Testing, Deployment)