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

ZephyrAmmor

October 2025

Contents

Module 6: Modern C++ Features (C++11–C++23)

Purpose

Understand the evolution of C++ into its modern form — focusing on features that enhance safety, readability, performance, and abstraction. Learn how modern idioms replace traditional boilerplate code.

6.1 Type Deduction and `auto`

Overview

- Introduced in C++11, refined through later standards.
- Automatically deduces variable types from initialization.

Example

```cpp
auto x = 42;         // int
auto y = 3.14;       // double
auto z = "Hello";    // const char*
```

Uses

- Reduces verbosity.
- Ensures consistency with return types in templates.

Note

`auto` deduces *by value*. Use `auto&` or `const auto&` when needed.

6.2 Range-Based for Loops

Example

```cpp
std::vector<int> nums = {1, 2, 3, 4};
for (auto n : nums) {
    std::cout << n << " ";
}
```

Modern Use Case

Use when iterating over STL containers or arrays.

6.3 Lambda Expressions

Syntax

```cpp
[capture](parameters) -> return_type {
    // body
};
```

Example

```cpp
auto add = [](int a, int b) { return a + b; };
std::cout << add(2, 3);  // 5
```

Capture Modes

- `[ ]` — captures nothing
- `[=]` — captures all by value
- `[&]` — captures all by reference
- `[=, &x]` — all by value except x by reference

6.4 Smart Pointers (C++11)

Types

Smart Pointer	Ownership Model	Header
`unique_ptr`	Sole ownership	`<memory>`
`shared_ptr`	Reference-counted ownership	`<memory>`
`weak_ptr`	Non-owning observer	`<memory>`

 Example

```cpp
#include <memory>

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

6.5 Move Semantics and Rvalue References

 Motivation

Optimize performance by *moving* resources instead of copying them.

 Example

```cpp
std::string s1 = "Hello";
std::string s2 = std::move(s1);  // moves content, avoids deep copy
```

 Use in Classes

```cpp
class Example {
public:
    Example(Example&& other) noexcept {
        data = std::move(other.data);
    }
};
```

6.6 `nullptr`, `enum class`, and `constexpr`

 `nullptr`

- Type-safe null pointer.
- Replaces `NULL` and `0`.

 Example

```cpp
int* p = nullptr;
```

 `enum class`

- Scoped enumerations prevent name clashes.

```cpp
enum class Color { Red, Green, Blue };
Color c = Color::Red;
```

 `constexpr`

- Compile-time constant evaluation.

```cpp
constexpr int square(int n) { return n * n; }
```

6.7 Structured Bindings (C++17)

Example

```cpp
auto [x, y] = std::make_pair(10, 20);
std::cout << x << ", " << y;
```

6.8 `std::optional`, `std::variant`, and `std::any`

`std::optional`

Represents an optional value.

```cpp
std::optional<int> value = 42;
if (value) std::cout << *value;
```

`std::variant`

Type-safe union.

```cpp
std::variant<int, std::string> data = 10;
data = "Hello";
```

`std::any`

Stores value of any type.

```cpp
std::any a = 42;
std::cout << std::any_cast<int>(a);
```

6.9 Ranges, Concepts, and Coroutines (C++20–C++23)

Ranges

Simplify working with collections.

```cpp
#include <ranges>
for (int n : std::views::iota(1, 6)) std::cout << n << " ";
```

Concepts

Constraint-based template programming.

```cpp
template <typename T>
requires std::integral<T>
T add(T a, T b) { return a + b; }
```

Coroutines

Simplify async and generator functions.

```cpp
#include <coroutine>
// Example omitted for brevity — advanced topic
```

6.10 Summary

Category	Key Feature	Benefit
Type System	`auto`, `constexpr`, `decltype`	Cleaner, safer code
Memory	Smart Pointers, Move Semantics	Safer, faster resource management
Functions	Lambdas, Ranges, Coroutines	Modern expressiveness
Templates	Concepts	Safer generic code