https://simplifycpp.org

Quick Introduction

to Modern C++ (C++11–C++23)

The Most Important Features in a Practical, Concise Summary

Prepared By Ayman Alheraki

Quick Introduction to Modern C++ (C++11–C++23)

The Most Important Features in a Practical, Concise Summary

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Chapter 1

Preface

Modern C++ has evolved dramatically since C++11. It has become safer, more expressive, faster, and more consistent. This mini-booklet summarizes the most important language and library features introduced between C++11 and C++23, designed for both beginners and experienced developers migrating from older versions of the language.
This guide focuses on:

- essential core language improvements,
- modern memory management,
- concurrency and parallelism,
- template and compile-time enhancements,
- new STL components,
- modules and ranges,
- modern best practices.

The goal is clarity and practical understanding—without unnecessary complexity.

Chapter 2

What Is Modern C++?

Modern C++ refers to the style and capabilities resulting from the big upgrade that started in 2011. It includes:

- safer coding styles,
- deterministic ownership models,
- functional-style tools,
- strong compile-time power,
- clearer syntax,
- better performance,
- a richer standard library.

2.1 The Evolution Timeline

- C++11 — The big revolution

- C++14 — Refinements and simplifications
- C++17 — Practical and widely used improvements
- C++20 — Another major revolution (concepts, coroutines, ranges, modules)
- C++23 — Maturity, polishing, and new utilities

Chapter 3

Core Language Enhancements

3.1 Auto Type Deduction (C++11)

```
auto x = 42;              // int
auto s = std::string("Hello");
auto v = std::vector<int>{1, 2, 3};
```

Improves readability and reduces redundancy.

3.2 Range-based for loops (C++11)

```
for (auto& x : vec) {
    x *= 2;
}
```

Cleaner and safer than manual iterators.

3.3 nullptr

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

Strongly typed null pointer—removes ambiguity from `NULL`.

3.4 Uniform Initialization

```
MyStruct s {1, 2, 3};
std::vector<int> v {1, 2, 3, 4};
```

A single, consistent syntax for initialization.

3.5 Scoped Enumerations (enum class)

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

Prevents name collisions and improves type safety.

Chapter 4

Lambdas and Functional Tools

4.1 Lambda Basics (C++11)

```
auto add = [](int a, int b) { return a + b; };
```

4.2 Generic Lambdas (C++14)

```
auto multiply = [](auto a, auto b) { return a * b; };
```

4.3 Lambda Templates (C++20)

```
auto twice = []<typename T>(T x) { return x * 2; };
```

Lambdas have become one of the most powerful tools in Modern C++.

Chapter 5

Move Semantics and Perfect Forwarding

5.1 Move Semantics

```
std::string s = "Hello";
std::string t = std::move(s);
```

Moves allow transferring resources without expensive copying.

5.2 Move Constructors

```
MyClass(MyClass&& other) noexcept {
    this->data = other.data;
    other.data = nullptr;
}
```

5.3 Perfect Forwarding

```
template <typename T, typename... Args>
```

```
std::unique_ptr<T> make(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

Chapter 6

Smart Pointers and RAII

6.1 unique_ptr

```
auto up = std::make_unique<Foo>();
```

Exclusive ownership; fastest smart pointer.

6.2 shared_ptr

```
auto sp = std::make_shared<Foo>();
```

Reference-counted shared ownership.

6.3 weak_ptr

```
std::weak_ptr<Foo> wp = sp;
```

Non-owning reference; prevents cycles.

Chapter 7

Concurrency and Multithreading

7.1 std::thread

```
std::thread t([]{ work(); });
t.join();
```

7.2 mutex and lock_guard

```
std::lock_guard<std::mutex> lock(mx);
```

7.3 async and futures

```
auto result = std::async(std::launch::async, compute);
```

7.4 Coroutines (C++20)

```
task<int> compute();
```

Lightweight, stackless, suspension-based async operations.

Chapter 8

Compile-time Features

8.1 constexpr (C++11–20)

```
constexpr int sq(int x) { return x*x; }
```

C++20 allows almost everything to be constexpr.

8.2 Variadic Templates

```
template<typename... Args>
void log(Args... args) { }
```

8.3 Concepts (C++20)

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

Stronger compile-time constraints.

Chapter 9

STL and Library Enhancements

9.1 std::optional

```
std::optional<int> f() { return 10; }
```

9.2 std::variant

```
std::variant<int, std::string> v = "Hi";
```

9.3 std::any

9.4 std::array

9.5 std::string_view (C++17)

Lightweight, non-owning string reference.

Chapter 10

Filesystem Library (C++17)

```cpp
namespace fs = std::filesystem;

for (auto &p : fs::directory_iterator(".")) {
    std::cout << p.path() << "\n";
}
```

A must-know feature for real-world applications.

Chapter 11

Modules (C++20)

11.1 Defining a Module

```
export module math;
export int add(int a, int b) { return a + b; }
```

11.2 Importing

```
import math;
std::cout << add(2, 3);
```

Modules replace the header system with a cleaner build model.

Chapter 12

Ranges (C++20)

```
auto evens =
    vec | std::views::filter([](int x){ return x % 2 == 0; });
```

Pipeline-style operations bring C++ closer to functional programming.

Chapter 13

New in C++23

13.1 std::expected

Functional error handling.

```
expected<int, Error> compute();
```

13.2 deducing this

Cleaner member function templates.

13.3 mdspan

A powerful multi-dimensional view for HPC.

13.4 Improved ranges and constexpr support

Chapter 14

Modern C++ Best Practices

- Prefer `unique_ptr` over `shared_ptr`
- Use RAII everywhere possible
- Avoid raw `new` and `delete`
- Use `constexpr` aggressively
- Use `auto` for readability, not for hiding types
- Prefer `enum class`
- Build APIs with clear ownership semantics
- Keep functions small and expressive
- Avoid unnecessary copies
- Use ranges and string_view for modern style

Chapter 15

Summary

Modern C++ is a powerful, expressive, highly efficient systems language. With the improvements from C++11 to C++23, developers now have access to:

- safer memory models
- better abstractions
- strong compile-time guarantees
- modern functional utilities
- better concurrency
- modules and ranges
- richer standard libraries

Mastering these features is essential for writing robust and high-performance software in today's C++ ecosystem.