Gorgo Continuing C++

April 11, 2026

Contents

8. Namespaces and the Preprocessor

As projects grow, name collisions become inevitable. Two libraries might both define a `log` function, or your `Error` class might clash with one from a dependency. Namespaces solve this by grouping names into distinct scopes. The preprocessor, inherited from C, controls what code the compiler sees — include guards prevent double-inclusion, macros define constants and conditional blocks, and conditional compilation lets you target different platforms. C++20 modules aim to replace much of the preprocessor's job. In this chapter you will learn namespace design, the preprocessor's key features, and a preview of modules.

Namespace Basics

You have used `std::` since the beginning of *Gorgo Starting C++*. `std` is a namespace — a named scope that contains the entire standard library. You can create your own:

```
#include <iostream>

namespace audio {

void play(const char* track)
{
    std::cout << "Playing: " << track << "\n";
}

int volume = 80;

}  // namespace audio

int main()
{
    audio::play("Yellow");
    std::cout << "Volume: " << audio::volume << "\n";

    return 0;
}
```

```
Playing: Yellow
Volume: 80
```

Everything inside `namespace audio { ... }` is accessed with the `audio::` prefix.

Nested Namespaces

Before C++17, nesting namespaces required separate blocks:

```
namespace company {
    namespace audio {
        namespace codec {
            void decode() {}
        }
    }
}
```

C++17 lets you write this in one line:

```
namespace company::audio::codec {
    void decode() {}
}
```

```
company::audio::codec::decode();
```

Anonymous Namespaces

An **anonymous namespace** gives its contents internal linkage — they are visible only within the current file:

```
namespace {
    int helper_count = 0;

    void internal_helper()
    {
        helper_count++;
    }
}
```

This is the modern C++ replacement for the `static` keyword at file scope. Other files cannot see `helper_count` or `internal_helper` even if they try to declare them with `extern`.

Tip: Use anonymous namespaces instead of `static` for file-local functions and variables. The `static` keyword at file scope is a holdover from C and is considered less idiomatic in C++.

Inline Namespaces

Inline namespaces make their contents accessible as if they were in the enclosing namespace. They are primarily used for API versioning:

```
#include <iostream>

namespace mylib {

inline namespace v2 {
    void greet() { std::cout << "Hola from v2!\n"; }
}

namespace v1 {
    void greet() { std::cout << "Hola from v1!\n"; }
}

}  // namespace mylib

int main()
{
    mylib::greet();       // calls v2::greet (inline namespace)
    mylib::v1::greet();   // explicitly calls v1
    mylib::v2::greet();   // explicitly calls v2

    return 0;
}
```

```
Hola from v2!
Hola from v1!
Hola from v2!
```

By marking `v2` as `inline`, users who call `mylib::greet()` automatically get the latest version. Users who need the old version can explicitly qualify `mylib::v1::greet()`.

`using` Declarations vs. `using` Directives

There are two ways to bring namespace members into the current scope.

A **`using` declaration** imports a single name:

```
using std::cout;
using std::string;

cout << "No prefix needed\n";
string s = "clean";
```

A **`using` directive** imports an entire namespace:

```
using namespace std;

cout << "Everything from std is visible\n";
```

Trap: Never put `using namespace std;` (or any `using` directive) in a header file. It pollutes the namespace of every file that includes the header, causing unexpected name collisions. `using namespace` in a `.cpp` file or inside a function is acceptable but use it with care.

The guideline:

Context	Recommendation
Header files	Never use `using namespace`. Use full qualification.
Source files (top level)	`using` declarations for frequently used names
Inside functions	`using namespace` is acceptable for convenience

Include Guards

When a header file is `#included` from multiple places, the compiler can see the same declarations twice. **Include guards** prevent this:

```
// audio.h — traditional include guard
#ifndef AUDIO_H
#define AUDIO_H

void play(const char* track);

#endif  // AUDIO_H
```

The first time `audio.h` is included, `AUDIO_H` is not defined, so the content is processed and `AUDIO_H` gets defined. The second time, `AUDIO_H` is already defined, so the entire content is skipped.

`#pragma once` is a simpler alternative supported by all major compilers:

```
// audio.h — pragma once
#pragma once

void play(const char* track);
```

Feature	Include guards	`#pragma once`
Standard	Yes (part of the language)	Not standard, but universally supported
Syntax	Verbose (3 lines)	One line
Edge cases	Works everywhere	May fail with symlinks or network drives

Tip: Either approach works. `#pragma once` is simpler and is the de facto standard in modern codebases. Use traditional include guards if your build environment has exotic filesystem issues.

Macros and Conditional Compilation

The C preprocessor runs before the compiler sees your code. Macros are text substitutions — they replace one sequence of tokens with another.

`#define`

```
#define MAX_TRACKS 100
#define PI 3.14159

int tracks[MAX_TRACKS];
```

The preprocessor replaces every occurrence of `MAX_TRACKS` with `100` before compilation.

Tip: Prefer `constexpr` variables over `#define` for constants. `constexpr` is type-safe and respects scopes; macros do not:

```
constexpr int max_tracks = 100;  // preferred
#define MAX_TRACKS 100            // avoid when possible
```

Function-Like Macros

```
#define SQUARE(x) ((x) * (x))

int a = SQUARE(5);    // expands to ((5) * (5)) = 25
int b = SQUARE(2+3);  // expands to ((2+3) * (2+3)) = 25
```

The extra parentheses are critical — without them, `SQUARE(2+3)` would expand to `2+3 * 2+3 = 11`.

Trap: Macros are pure text replacement. They do not understand types, scopes, or expressions. Prefer `constexpr` functions or templates over function-like macros.

Conditional Compilation

Conditional compilation lets you include or exclude code based on compile-time conditions:

```
#ifdef _WIN32
    #include <windows.h>
#elif defined(__linux__)
    #include <unistd.h>
#elif defined(__APPLE__)
    #include <mach/mach.h>
#endif
```

Common predefined macros:

Macro	Meaning
`_WIN32`	Windows
`__linux__`	Linux
`__APPLE__`	macOS / iOS
`__cplusplus`	C++ standard version (e.g., `202302L` for C++23)
`NDEBUG`	Release mode (disables `assert`)

```
#if __cplusplus >= 202002L
    // C++20 or later
    #include <ranges>
#else
    // Fallback for older compilers
#endif
```

When to Use Macros

Macros still have legitimate uses: - Include guards - Platform-specific conditional compilation - `assert()` (needs to capture `__FILE__` and `__LINE__`) - Compile-time feature detection

For everything else — constants, inline functions, type-safe generics — use `constexpr`, `inline`, and templates.

Modules Preview (C++20)

C++20 introduced **modules** as a modern replacement for the `#include` / header-file model. Modules solve several long-standing problems: - Headers are processed every time they are included, slowing compilation - Include order can matter (macros leak across headers) - Include guards / `#pragma once` are workarounds, not solutions

Basic Syntax

A module is defined with `export module`:

```
// greeting.cppm (module interface file)
export module greeting;

import <string>;

export std::string greet(const std::string& name)
{
    return "Hola, " + name + "!";
}
```

And consumed with `import`:

```
// main.cpp
import greeting;
import <iostream>;

int main()
{
    std::cout << greet("Mundo") << "\n";

    return 0;
}
```

Current State

Module support is improving across compilers, but as of this writing: - MSVC has the most mature support - GCC and Clang support modules but with some limitations - Build system support (CMake, etc.) is still evolving

Tip: Modules are the future of C++ code organization, but the ecosystem is not fully there yet. Learn the concepts now, and start using them when your toolchain supports them well. For now, headers with `#pragma once` remain the practical choice for most projects.

Try It: Namespace Organization

Here is a program that demonstrates namespace design. Type it in, compile with `g++ -std=c++23`, and experiment:

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

namespace studio {

namespace audio {
    void play(const std::string& track)
    {
        std::cout << "Playing: " << track << "\n";
    }
}

namespace video {
    void play(const std::string& clip)
    {
        std::cout << "Showing: " << clip << "\n";
    }
}

inline namespace v2 {
    std::string format_title(const std::string& title)
    {
        return "[" + title + "]";
    }
}

namespace v1 {
    std::string format_title(const std::string& title)
    {
        return title;
    }
}

}  // namespace studio

namespace {
    int internal_counter = 0;
    void tick() { internal_counter++; }
}
```

```cpp
int main()
{
    // Nested namespaces
    studio::audio::play("Speed of Sound");
    studio::video::play("music video");

    // Inline namespace (v2 is default)
    std::cout << studio::format_title("Harder Better Faster Stronger") << "\n";
    std::cout << studio::v1::format_title("Harder Better Faster Stronger") << "\n";

    // Anonymous namespace
    tick();
    tick();
    std::cout << "Counter: " << internal_counter << "\n";

    // using declaration
    using studio::audio::play;
    play("Around the World");

    // Conditional compilation
    #ifdef NDEBUG
        std::cout << "Release build\n";
    #else
        std::cout << "Debug build\n";
    #endif

    return 0;
}
```

Try renaming the `play` functions to the same name in different namespaces and see how the compiler resolves them. Try removing the `inline` from `v2` and see what happens when you call `studio::format_title`.

Key Points

- **Namespaces** group names to avoid collisions. Use `::` to access members.
- **Nested namespaces** can be declared with `namespace a::b::c { }` (C++17).
- **Anonymous namespaces** give contents internal linkage (file-local visibility), replacing `static` at file scope.
- **Inline namespaces** make their contents accessible as if they were in the enclosing namespace, useful for API versioning.
- A **`using` declaration** imports a single name; a **`using` directive** imports an entire namespace. Never use `using namespace` in header files.
- **Include guards** (`#ifndef`/`#define`/`#endif`) or **`#pragma once`** prevent double-inclusion of headers.
- **Macros** are text substitution. Prefer `constexpr` for constants and templates/`inline` for function-like macros.
- **Conditional compilation** (`#ifdef`, `#if`) is useful for platform-specific code and feature detection.
- **Modules** (C++20) are the modern alternative to headers, offering faster compilation and better isolation. Ecosystem support is still maturing.

Exercises

1. **Think about it:** Why is `using namespace std;` in a header file dangerous? Give a specific example of a name collision it could cause.

2. **What does this print?**

```cpp
namespace a {
    int x = 1;
    namespace b {
        int x = 2;
    }
}

std::cout << a::x << " " << a::b::x << "\n";
```

3. **Where is the bug?**

```cpp
// file1.cpp
namespace { int count = 0; }

// file2.cpp
extern int count;
void increment() { count++; }
```

4. **Think about it:** When would you use an inline namespace? Describe a real-world scenario where it would be useful.

5. **What does this print?**

```cpp
#define DOUBLE(x) x * 2

int result = DOUBLE(3 + 4);
std::cout << result << "\n";
```

6. **Where is the bug?** (And what is the fix?)

```cpp
// utils.h
using namespace std;

string format(int x);
```

7. **Think about it:** What advantages do C++20 modules have over traditional headers? Why hasn't the industry fully adopted them yet?

8. **What does this print?**

```cpp
namespace outer {
    inline namespace inner {
        int value = 42;
    }
}

std::cout << outer::value << "\n";
std::cout << outer::inner::value << "\n";
```

9. **Calculation:** Given the macro:

```cpp
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

What is the value of `MAX(3+1, 2+3)`?

10. **Write a program** that defines a namespace `music` with sub-namespaces `rock` and `pop`, each containing a function `top_song()` that returns a different string. Use a `using` declaration to bring one into scope and call both. Add an anonymous namespace with a helper function used by both sub-namespaces.