Gorgo Continuing C++

April 11, 2026

Contents

5. Enums, constexpr, and Compile-Time Programming

Catching mistakes at compile time is always better than catching them at run time. A program that fails to compile is annoying but harmless. A program that compiles and then crashes in production is a disaster. C++ gives you several tools to move work from run time to compile time: **scoped enumerations** give type-safe named constants, **`constexpr`** and **`consteval`** let you compute values during compilation, **`static_assert`** checks conditions before the program ever runs, and **type aliases** make complex types readable. In this chapter you will learn how to use these features to write code that is safer, faster, and clearer.

Scoped Enumerations: `enum class`

You may have seen traditional C-style enums in older code:

```
enum Color { Red, Green, Blue };
enum TrafficLight { Red, Yellow, Green };  // error: Red and Green already defined!
```

The names leak into the surrounding scope and collide. They also implicitly convert to `int`, which can cause subtle bugs.

C++11 introduced **scoped enumerations** (`enum class`) to fix both problems:

```
#include <iostream>

enum class Color { Red, Green, Blue };
enum class TrafficLight { Red, Yellow, Green };

int main()
{
    Color c = Color::Blue;
    TrafficLight t = TrafficLight::Red;

    // int x = c;           // error: no implicit conversion to int
    int x = static_cast<int>(c);  // OK: explicit conversion gives 2

    if (t == TrafficLight::Red) {
        std::cout << "Stop!\n";
    }

    return 0;
}
```

```
Stop!
```

Each enumerator is scoped to its enum, so `Color::Red` and `TrafficLight::Red` do not collide. There is no implicit conversion to `int` — you must use `static_cast` if you need the numeric value.

Underlying Type

By default, the underlying type of an `enum class` is `int`. You can change it:

```
enum class Status : uint8_t { OK = 0, Error = 1, Pending = 2 };
```

This is useful when memory matters (embedded systems, network protocols) or when you need to match an external format.

`using enum` (C++20)

If you get tired of writing the enum name repeatedly, C++20 lets you bring enumerators into scope:

```cpp
#include <iostream>

enum class Direction { North, South, East, West };

void navigate(Direction d)
{
    using enum Direction;  // bring all enumerators into scope

    switch (d) {
    case North: std::cout << "Going north\n"; break;
    case South: std::cout << "Going south\n"; break;
    case East:  std::cout << "Going east\n"; break;
    case West:  std::cout << "Going west\n"; break;
    }
}

int main()
{
    navigate(Direction::North);
    return 0;
}
```

```
Going north
```

Tip: Use `using enum` only in limited scopes (like inside a function or switch statement) to avoid polluting the outer namespace — that would defeat the purpose of scoped enums.

`constexpr`

A `constexpr` variable or function can be evaluated at **compile time**. The compiler computes the result and bakes it into the binary, so there is no run-time cost:

`constexpr` Variables

```cpp
constexpr int max_tracks = 100;
constexpr double pi = 3.14159265358979;
```

A `constexpr` variable must be initialized with a value the compiler can compute. It is implicitly `const`.

`constexpr` Functions

A `constexpr` function **can** be evaluated at compile time if all its arguments are compile-time constants. If called with run-time values, it runs at run time like a normal function:

```cpp
#include <iostream>

constexpr int factorial(int n)
{
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

```cpp
int main()
{
    constexpr int f5 = factorial(5);  // computed at compile time
    std::cout << f5 << "\n";          // 120

    int n = 6;
    int f6 = factorial(n);            // computed at run time (n is not constexpr)
    std::cout << f6 << "\n";          // 720

    return 0;
}
```

```
120
720
```

`constexpr` functions can use loops, conditionals, and local variables. The restrictions are that everything must be evaluable at compile time: no I/O, no dynamic allocation (mostly), and no undefined behavior.

`if constexpr`

`if constexpr` evaluates a condition at compile time and discards the unused branch entirely. This is especially useful in templates:

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
void describe(T value)
{
    if constexpr (std::is_integral_v<T>) {
        std::cout << value << " is an integer\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << value << " is a float\n";
    } else {
        std::cout << value << " is something else\n";
    }
}

int main()
{
    describe(42);
    describe(3.14);
    describe("Complicated");

    return 0;
}
```

```
42 is an integer
3.14 is a float
Complicated is something else
```

Unlike a regular `if`, the discarded branch does not need to compile for the given type. This is what makes `if constexpr` essential for template metaprogramming.

`consteval`

`consteval` (C++20) is stricter than `constexpr`. A `consteval` function **must** be evaluated at compile time. If you try to call it with run-time values, the compiler rejects the code:

```
consteval int square(int n)
{
    return n * n;
}

constexpr int x = square(5);  // OK: compile time
// int y = 6; square(y);      // error: y is not a compile-time constant
```

Use `consteval` when a function only makes sense at compile time — like computing array sizes, lookup table entries, or hash values for compile-time string matching.

Tip: Use `constexpr` when a function *can* run at compile time. Use `consteval` when it *must* run at compile time.

`constinit`

`constinit` (C++20) ensures that a variable with **static storage duration** (globals, file-scope variables, `static` locals) is initialized at compile time, avoiding the "static initialization order fiasco":

```
constinit int global_max = 100;  // guaranteed compile-time initialization
```

Unlike `constexpr`, `constinit` does not make the variable `const` — you can modify it after initialization:

```
constinit int counter = 0;  // initialized at compile time

void increment()
{
    counter++;  // OK: counter is not const
}
```

Wut: `constinit` only affects initialization, not the entire lifetime of the variable. A `constinit` variable is mutable after initialization (unlike `constexpr`, which is always `const`).

`static_assert`

`static_assert` checks a condition at compile time. If the condition is false, compilation fails with the message you provide:

```
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
static_assert(sizeof(void*) == 8, "this code assumes 64-bit pointers");
```

It is useful for documenting and enforcing assumptions about the platform, and for checking template parameters:

```
template<typename T>
class NumericBuffer {
    static_assert(std::is_arithmetic_v<T>, "T must be a numeric type");
    // ...
};
```

```
NumericBuffer<int> ok;            // compiles
// NumericBuffer<std::string> bad; // error: T must be a numeric type
```

`static_assert` has zero run-time cost — it exists only during compilation.

Type Aliases

Type aliases give a new name to an existing type, making complex types readable and easier to change later.

`using` vs. `typedef`

C++ has two ways to create type aliases. The modern `using` syntax is preferred:

```
// Modern (preferred)
using Playlist = std::vector<std::string>;
using SongMap = std::map<std::string, int>;

// Old-style typedef (still works, same effect)
typedef std::vector<std::string> Playlist;
typedef std::map<std::string, int> SongMap;
```

`using` is easier to read, especially for function pointer types:

```
// using
using Callback = void(*)(int);

// typedef — harder to parse
typedef void(*Callback)(int);
```

Alias Templates

`using` can also create templated type aliases, which `typedef` cannot:

```
template<typename T>
using Vec = std::vector<T>;

Vec<int> numbers = {1, 2, 3};
Vec<std::string> words = {"Estoy", "aqui"};
```

This is especially useful for simplifying nested template types:

```
template<typename K, typename V>
using HashMap = std::unordered_map<K, V>;

HashMap<std::string, int> scores;
```

Try It: Compile-Time Playground

Here is a program that exercises the compile-time features from this chapter. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <array>
#include <iostream>
#include <type_traits>

enum class Note { C = 0, D = 2, E = 4, F = 5, G = 7, A = 9, B = 11 };

constexpr int note_to_midi(Note n, int octave)
```

```
{
    return (octave + 1) * 12 + static_cast<int>(n);
}

consteval int bpm_to_ms(int bpm)
{
    return 60'000 / bpm;
}

template<typename T>
using Grid = std::array<std::array<T, 4>, 4>;

int main()
{
    // constexpr
    constexpr int middle_c = note_to_midi(Note::C, 4);
    std::cout << "Middle C MIDI: " << middle_c << "\n";

    // consteval — must be compile time
    constexpr int beat_ms = bpm_to_ms(120);
    std::cout << "120 BPM = " << beat_ms << " ms per beat\n";

    // static_assert
    static_assert(note_to_midi(Note::C, 4) == 60, "Middle C should be MIDI 60");
    static_assert(bpm_to_ms(120) == 500, "120 BPM should be 500 ms");

    // enum class with using enum
    using enum Note;
    constexpr auto a440 = note_to_midi(A, 4);
    std::cout << "A440 MIDI: " << a440 << "\n";

    // Type alias template
    Grid<int> pattern = {{
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {1, 0, 1, 0},
        {0, 1, 0, 1}
    }};

    std::cout << "Pattern[0][2]: " << pattern[0][2] << "\n";

    return 0;
}
```

```
Middle C MIDI: 60
120 BPM = 500 ms per beat
A440 MIDI: 69
Pattern[0][2]: 1
```

Try changing the `bpm_to_ms` call to use a non-constexpr variable and see the error. Add more notes and experiment with `static_assert` to verify your MIDI calculations.

Key Points

- **`enum class`** creates scoped enumerations with no implicit conversion to `int` and no name leakage. Use `static_cast` for explicit conversion.
- You can specify the underlying type: `enum class Foo : uint8_t`.
- **`using enum`** (C++20) brings enumerators into scope within a limited region.
- **`constexpr`** variables are compile-time constants. `constexpr` functions can run at compile time or run time depending on their arguments.
- **`consteval`** (C++20) functions must run at compile time — calling them with run-time values is a compile error.
- **`constinit`** (C++20) guarantees compile-time initialization for static-duration variables without making them `const`.
- **`if constexpr`** evaluates conditions at compile time and discards the unused branch, which is essential for templates.
- **`static_assert`** checks conditions at compile time with zero run-time cost.
- **Type aliases** with `using` are preferred over `typedef`. `using` supports alias templates; `typedef` does not.

Exercises

1. **Think about it:** Why does `enum class` require `static_cast` to convert to `int`, when the old `enum` converted implicitly? What bugs does this prevent?

2. **What does this print?**

```cpp
enum class Suit : int { Hearts = 0, Diamonds, Clubs, Spades };
int x = static_cast<int>(Suit::Spades);
std::cout << x << "\n";
```

3. **Where is the bug?**

```cpp
enum class Priority { Low, Medium, High };

void handle(Priority p)
{
    if (p == 2) {
        std::cout << "High priority!\n";
    }
}
```

4. **Calculation:** What is the value of `result`?

```cpp
constexpr int power(int base, int exp)
{
    int result = 1;
    for (int i = 0; i < exp; ++i) {
        result *= base;
    }
    return result;
}

constexpr int result = power(2, 10);
```

5. **Think about it:** What is the practical difference between `constexpr` and `consteval`? When would you use one over the other?

6. **Where is the bug?**

```cpp
consteval int compute(int x) { return x * x; }
```

```cpp
int main()
{
    int n;
    std::cin >> n;
    int result = compute(n);
    std::cout << result << "\n";
    return 0;
}
```

7. **What does this print?**

```cpp
template<typename T>
void check(T value)
{
    if constexpr (std::is_integral_v<T>) {
        std::cout << value * 2 << "\n";
    } else {
        std::cout << value << "\n";
    }
}

check(5);
check(3.14);
```

8. **Think about it:** Why does `constinit` exist as a separate keyword from `constexpr`? What problem does it solve that `constexpr` does not?

9. **Where is the bug?**

```cpp
using StringPair = std::pair<std::string, std::string>;

StringPair get_pair()
{
    return {"Hola", "mundo"};
}

auto [a, b] = get_pair();
std::cout << a << " " << b << "\n";
```

(Trick question — is there actually a bug?)

10. **Write a program** that defines a `constexpr` function to convert Fahrenheit to Celsius (`(f - 32) * 5 / 9.0`). Use `static_assert` to verify that `212` F is `100.0` C and `32` F is `0.0` C. Then define an `enum class Season { Spring, Summer, Fall, Winter }` and a `constexpr` function that returns a typical temperature for each season. Print all four seasons and their temperatures.