Gorgo Continuing C++

April 11, 2026

Contents

12. Best Practices and Common Idioms

You now have a solid foundation in C++: classes, templates, the standard library, concurrency, and more. Knowing the language features is necessary, but knowing how to use them well is what separates a competent programmer from a great one. This chapter covers the coding standards, idioms, and patterns that experienced C++ developers rely on, and closes with a look at what is coming in C++26.

Coding Standards and Style

Naming Conventions

There is no single naming convention in C++. The standard library uses `snake_case` for everything. Google style uses `CamelCase` for types and `snake_case` for functions. The most important rule is **consistency within your project**:

Element	Common styles
Types / classes	`PascalCase` or `snake_case`
Functions	`snake_case` or `camelCase`
Variables	`snake_case`
Constants	`kConstantName` or `UPPER_CASE`
Private members	`name_` (trailing underscore)
Macros	`UPPER_CASE` (the one universal convention)

Const-Correctness

Mark everything `const` that should not change. This communicates intent, catches bugs at compile time, and enables optimizations:

```
// Parameters
void process(const std::string& name);   // does not modify name
void modify(std::string& name);           // may modify name

// Member functions
class Playlist {
public:
    int size() const;                     // does not modify the object
    void add(const std::string& song);    // modifies the object
};

// Variables
const int max_tracks = 100;
```

Tip: Make everything `const` by default. Only remove `const` when you have a reason to mutate. This is the opposite of what most beginners do, and it prevents a large class of bugs.

The Rule of Zero

If your class does not manage resources directly (it uses `std::string`, `std::vector`, smart pointers, etc.), you should not write any special member functions — the compiler-generated defaults do the right thing:

```
class Song {
public:
    Song(std::string title, std::string artist)
```

```
        : title_(std::move(title)), artist_(std::move(artist)) {}

    // No destructor, no copy/move constructors, no assignment operators.
    // The compiler generates correct versions automatically.

private:
    std::string title_;
    std::string artist_;
};
```

This is the **Rule of Zero**: let the compiler do the work.

The Rule of Five

If your class manages a resource directly (raw pointers, file handles, etc.), you must define all five special member functions:

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. Move constructor
5. Move assignment operator

```
#include <cstring>

class Buffer {
public:
    Buffer(size_t size) : data_(new char[size]), size_(size) {}

    ~Buffer() { delete[] data_; }

    Buffer(const Buffer& other) : data_(new char[other.size_]), size_(other.size_)
    {
        std::memcpy(data_, other.data_, size_);
    }

    Buffer& operator=(const Buffer& other)
    {
        if (this != &other) {
            delete[] data_;
            data_ = new char[other.size_];
            size_ = other.size_;
            std::memcpy(data_, other.data_, size_);
        }
        return *this;
    }

    Buffer(Buffer&& other) noexcept : data_(other.data_), size_(other.size_)
    {
        other.data_ = nullptr;
        other.size_ = 0;
    }

    Buffer& operator=(Buffer&& other) noexcept
    {
        if (this != &other) {
```

```
            delete[] data_;
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
        }
        return *this;
    }

private:
    char* data_;
    size_t size_;
};
```

Tip: If you find yourself writing the Rule of Five, ask whether a standard type (like `std::vector<char>`) could manage the resource for you instead, bringing you back to the Rule of Zero.

When to Use `auto`

`auto` deduces the type from the initializer. Use it when the type is obvious or verbose, not when it obscures meaning:

```
// Good: type is clear from context
auto it = my_map.find("key");
auto [name, score] = get_result();
auto ptr = std::make_unique<Widget>();

// Bad: what type is this?
auto x = compute();          // unclear — reader must find compute()
auto result = process(data); // unclear
```

Common C++ Idioms

PIMPL (Pointer to Implementation)

The **PIMPL** idiom hides implementation details behind a pointer, reducing compile-time dependencies. Changes to the implementation do not require recompiling users of the class:

```
// playlist.h
#include <memory>
#include <string>

class Playlist {
public:
    Playlist();
    ~Playlist();
    void add(const std::string& song);
    int size() const;

private:
    struct Impl;                // forward declaration
    std::unique_ptr<Impl> pimpl_;
};
```

```cpp
// playlist.cpp
#include "playlist.h"
#include <vector>

struct Playlist::Impl {
    std::vector<std::string> songs;
};

Playlist::Playlist() : pimpl_(std::make_unique<Impl>()) {}
Playlist::~Playlist() = default;

void Playlist::add(const std::string& song)
{
    pimpl_->songs.push_back(song);
}

int Playlist::size() const
{
    return static_cast<int>(pimpl_->songs.size());
}
```

The header does not include `<vector>`, so files that include `playlist.h` do not depend on `<vector>`. This can significantly reduce build times in large projects.

Wut: The destructor must be defined in the `.cpp` file (even as `= default`) because `unique_ptr` needs the complete type to destroy it. If you let the compiler generate the destructor in the header, it will fail because `Impl` is incomplete there.

CRTP (Curiously Recurring Template Pattern)

The **CRTP** provides **static polymorphism** — polymorphic behavior resolved at compile time instead of run time:

```cpp
#include <iostream>

template<typename Derived>
class Player {
public:
    void play()
    {
        static_cast<Derived*>(this)->play_impl();
    }
};

class MP3Player : public Player<MP3Player> {
public:
    void play_impl()
    {
        std::cout << "Playing MP3\n";
    }
};

class WAVPlayer : public Player<WAVPlayer> {
public:
    void play_impl()
```

```
    {
        std::cout << "Playing WAV\n";
    }
};

template<typename T>
void start_playback(Player<T>& player)
{
    player.play();
}

int main()
{
    MP3Player mp3;
    WAVPlayer wav;
    start_playback(mp3);  // Playing MP3
    start_playback(wav);  // Playing WAV

    return 0;
}
```

```
Playing MP3
Playing WAV
```

Unlike virtual functions, CRTP has no vtable overhead. The compiler inlines the call because it knows the exact type at compile time.

The trade-off: you cannot store different `Player` types in the same container (no common base class), so CRTP does not replace virtual functions when you need run-time polymorphism (Chapter 1).

Tag Dispatch

Tag dispatch selects a function overload at compile time using empty tag types:

```
#include <iostream>

struct fast_tag {};
struct safe_tag {};

void process(int x, fast_tag)
{
    std::cout << "Fast path: " << x << "\n";
}

void process(int x, safe_tag)
{
    if (x < 0) {
        std::cout << "Error: negative\n";
        return;
    }
    std::cout << "Safe path: " << x << "\n";
}

int main()
{
    process(42, fast_tag{});
```

```
    process(-1, safe_tag{});

    return 0;
}
```

```
Fast path: 42
Error: negative
```

Tag dispatch was more common before C++20. Today, `if constexpr` and concepts (Chapter 2) often achieve the same result more cleanly.

Code Review Checklist

When reviewing your own or others' C++ code, check for:

Correctness

- ☐ Does the code handle all edge cases?
- ☐ Are there potential off-by-one errors?
- ☐ Is error handling complete (exceptions, error codes)?

Resource Management

- ☐ Are all resources managed by RAII (Chapter 9)?
- ☐ Is the Rule of Zero or Rule of Five followed?
- ☐ Are smart pointers used instead of raw `new`/`delete`?

Safety

- ☐ Is `const` used wherever possible?
- ☐ Are `override` and `final` used on virtual functions?
- ☐ Are data races avoided with mutexes or atomics (Chapter 10)?

Performance

- ☐ Are large objects passed by `const&` or moved?
- ☐ Are unnecessary copies avoided?
- ☐ Is `reserve()` called on vectors when the size is known?

Style

- ☐ Is naming consistent with the project convention?
- ☐ Are comments explaining *why*, not *what*?
- ☐ Is the code simple and readable?

What's Next: C++26 Preview

C++ continues to evolve. Here are some features expected in C++26 and beyond:

`std::execution` (Senders/Receivers)

A structured framework for asynchronous and parallel programming, replacing ad-hoc thread management with composable operations.

Pattern Matching (post-C++26)

A cleaner alternative to `if`/`else` chains and `std::visit`, still under active development:

```
// Proposed syntax (not final)
inspect (value) {
    0 => std::cout << "zero\n";
```

```
    int i if i > 0 => std::cout << "positive\n";
    _ => std::cout << "other\n";
};
```

Reflection

The ability to inspect types at compile time — querying member names, types, and attributes programmatically. This will enable automatic serialization, ORM generation, and much more.

Contracts

Preconditions and postconditions as part of function declarations:

```
int sqrt_of(int x)
    pre (x >= 0)
    post (r: r >= 0)
{
    // ...
}
```

Tip: C++ evolves roughly every three years. Keeping up with the latest standards ensures you can write cleaner, safer code. Follow the ISO C++ committee's progress and experiment with new features as your compiler supports them.

Key Points

- **Naming conventions** vary; consistency within a project matters most.
- **Const-correctness**: make everything `const` by default.
- **Rule of Zero**: if your class does not manage resources directly, do not write special member functions.
- **Rule of Five**: if you manage a resource directly, define all five special members.
- Use `auto` when the type is clear or verbose, not when it obscures meaning.
- **PIMPL** hides implementation details behind a pointer, reducing compile dependencies and build times. The destructor must be in the `.cpp` file.
- **CRTP** provides static polymorphism with no vtable overhead, but does not support run-time polymorphism.
- **Tag dispatch** selects overloads at compile time; largely superseded by `if constexpr` and concepts.
- A good code review checks correctness, resource management, safety, performance, and style.
- **C++26** will bring `std::execution`, pattern matching, reflection, and contracts.

Exercises

1. **Think about it:** Why does the text recommend starting with `const` and removing it only when needed, rather than the other way around?
2. **Think about it:** When should you follow the Rule of Zero vs. the Rule of Five? How do you decide?
3. **Where is the bug?**

   ```
   class Data {
   public:
       Data(int n) : ptr_(new int[n]), size_(n) {}
       ~Data() { delete[] ptr_; }
   private:
       int* ptr_;
       int size_;
   };
   ```

```
Data a(10);
Data b = a;
```

4. **Think about it:** What are the trade-offs between CRTP (static polymorphism) and virtual functions (dynamic polymorphism)? When would you choose each?

5. **What does this print?**

```
auto x = 42;
auto y = 3.14;
auto z = x + y;
std::cout << z << "\n";
```

6. **Think about it:** Why must the PIMPL destructor be defined in the `.cpp` file? What error do you get if you let the compiler generate it in the header?

7. **Where is the problem?**

```
void add_to_vector(std::vector<int> v, int x)
{
    v.push_back(x);
}

std::vector<int> data = {1, 2, 3};
add_to_vector(data, 4);
std::cout << data.size() << "\n";
```

8. **Think about it:** The code review checklist mentions checking that comments explain "why" not "what." Why is this distinction important? Give an example of a bad comment and a good one for the same code.

9. **Calculation:** A class has a `std::vector<std::string>` member and a `std::unique_ptr<Widget>` member. How many special member functions should you write for this class?

10. **Write a program** that uses the CRTP to create a `Printable<Derived>` base class with a `print()` method that calls `Derived::to_string()`. Create two derived classes (e.g., `Song` and `Album`) that each implement `to_string()`. Demonstrate calling `print()` on instances of each.