Gorgo Starting C++

April 11, 2026

Contents

14. Special Members and Friends

In Chapter 12 you learned how to define classes with constructors, destructors, and member functions. Those tools let you create well-behaved types, but two questions remain. First, when your class manages a resource like raw memory, how do you make sure copying, moving, and cleanup all work correctly? Second, how do you give an outside function or class access to private data when making it a member is not an option? This chapter covers both: the special member functions the compiler can generate (and the rules for writing your own), and the `friend` keyword for granting controlled access.

Special Member Functions and the Rule of Five

When you write a class, the compiler can automatically generate up to five special member functions:

1. **Destructor** — cleans up when the object is destroyed
2. **Copy constructor** — creates a new object as a copy of another
3. **Copy assignment operator** — replaces an existing object's contents with a copy of another
4. **Move constructor** — creates a new object by moving from another
5. **Move assignment operator** — replaces an existing object's contents by moving from another

If your class manages a resource (like raw heap memory), and you write any one of these five, you almost certainly need to write *all* five. This is called the **Rule of Five**.

Here is a simplified example of a class that manages its own heap memory. The example uses two C string functions from `<cstring>` whose signatures are:

```
size_t strlen(const char* str);    // returns length of a C string
char*  strcpy(char* dest, const char* src);  // copies src into dest
```

```
#include <iostream>
#include <cstring>

class Lyric {
private:
    char *text;

public:
    // constructor
    Lyric(const char *t) {
        text = new char[std::strlen(t) + 1];
        std::strcpy(text, t);
    }

    // destructor
    ~Lyric() {
        delete[] text;
    }

    // copy constructor
    Lyric(const Lyric &other) {
        text = new char[std::strlen(other.text) + 1];
        std::strcpy(text, other.text);
    }

    // copy assignment
    Lyric &operator=(const Lyric &other) {
        if (this != &other) {
```

```
            delete[] text;
            text = new char[std::strlen(other.text) + 1];
            std::strcpy(text, other.text);
        }
        return *this;
    }

    // move constructor
    Lyric(Lyric &&other) noexcept : text(other.text) {
        other.text = nullptr;
    }

    // move assignment
    Lyric &operator=(Lyric &&other) noexcept {
        if (this != &other) {
            delete[] text;
            text = other.text;
            other.text = nullptr;
        }
        return *this;
    }

    void print() const {
        if (text) {
            std::cout << text << std::endl;
        }
    }
};
```

Notice the `noexcept` keyword on the move constructor and move assignment operator. As you learned in Chapter 11, `noexcept` promises the compiler that these functions will not throw exceptions. Standard library containers like `std::vector` will only use your move operations (instead of slower copies) during reallocation if they are marked `noexcept`.

That is a lot of code just to manage a string. This brings us to better tools for controlling what the compiler generates.

Defaulted and Deleted Functions

The compiler's auto-generation rules create two practical problems. The first is that writing *any* constructor suppresses the default constructor, and writing a destructor or copy operation can suppress the move operations. Sometimes the compiler-generated versions are exactly what you want, but you have to write them by hand just to get them back. The second problem is the opposite: sometimes the compiler happily generates a function that makes no sense for your type. Imagine a class that represents an open connection to a hardware device — copying it would mean two objects fighting over the same device, but the compiler will generate a copy constructor anyway unless you stop it.

Before C++11, the workaround for the second problem was to declare the unwanted function `private` and never define it. That "worked," but anyone who accidentally called it got a confusing linker error instead of a clear explanation.

C++ gives you two tools to solve these problems: `= default` and `= delete`.

= default

When you write `= default`, you are saying "generate the default version of this function for me." This solves the first problem: adding one special member function suppresses the compiler's auto-generation of others, but you still want the default behavior:

```
class Song {
private:
    std::string title;
    std::string artist;

public:
    Song(const std::string &t, const std::string &a) : title(t), artist(a) {}

    // Writing a custom constructor suppresses the default constructor.
    // Bring it back:
    Song() = default;

    // The compiler-generated versions are fine for these:
    Song(const Song &) = default;
    Song &operator=(const Song &) = default;
    ~Song() = default;
};
```

`= default` also documents your intent: it tells anyone reading the code "I thought about this and the compiler's version is correct."

= delete

`= delete` solves the second problem. When you write `= delete`, the function exists but calling it is a compile-time error. This lets you prevent operations that do not make sense for your type:

```
class AudioStream {
private:
    int device_id;

public:
    AudioStream(int id) : device_id(id) {}

    // Copying an active audio stream would cause two objects
    // to fight over the same hardware device.
    AudioStream(const AudioStream &) = delete;
    AudioStream &operator=(const AudioStream &) = delete;

    // Moving is fine --- ownership transfers cleanly.
    AudioStream(AudioStream &&other) noexcept : device_id(other.device_id) {
        other.device_id = -1;
    }

    AudioStream &operator=(AudioStream &&other) noexcept {
        if (this != &other) {
            device_id = other.device_id;
            other.device_id = -1;
        }
        return *this;
    }
```

```
};

AudioStream a(1);
AudioStream b = a;              // ERROR: copy constructor is deleted
AudioStream c = std::move(a);   // OK: move constructor is available
```

You can delete any function, not just special members. A common use is preventing implicit conversions:

```
void set_volume(int v);
void set_volume(double) = delete;  // prevent set_volume(3.14)
```

Tip: `= delete` gives a clear compiler error with a message like "use of deleted function." This is much better than making a function private and leaving it undefined, which was the pre-C++11 workaround and produced cryptic linker errors instead.

The Rule of Zero

The **Rule of Zero** says: if your class does not manage a resource directly, do not write any of the five special member functions. Let the compiler generate them for you.

This is closely related to **RAII** (Resource Acquisition Is Initialization), a fundamental C++ pattern where you acquire resources in the constructor and release them in the destructor. When you follow the Rule of Zero, you rely on types that already implement RAII (like `std::string`, `std::vector`, and `std::unique_ptr`) so your class does not need to.

How do you avoid managing resources directly? Use smart pointers and standard library types that already handle their own memory:

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

class Lyric {
private:
    std::string text;  // std::string manages its own memory

public:
    Lyric(const std::string &t) : text(t) {}

    void print() const {
        std::cout << text << std::endl;
    }
};
```

This version does the same thing as the Rule of Five version above, but in a fraction of the code. The compiler-generated copy constructor, move constructor, and destructor all do the right thing because `std::string` already knows how to copy, move, and clean up after itself.

Tip: Follow the Rule of Zero whenever you can. Use `std::string` instead of `char*`, `std::vector` instead of raw arrays, and `std::unique_ptr` instead of raw `new`/`delete`. If all your members manage themselves, you do not need to write any special member functions.

Friends

So far, only a class's own member functions can access its private data. But sometimes an outside function or another class genuinely needs that access, and making it a member function is inconvenient or impossible.

Consider printing a `Playlist` with `std::cout`. You would like to write `std::cout << my_playlist`, but `operator<<` cannot be a member function of `Playlist` — the left operand of `<<` is a `std::ostream`, not a `Playlist`, so the compiler would need to add the overload to `std::ostream`, which you do not own. It has to be a free function, and a free function cannot access private members.

Or consider a class that manages another class's internals — like a `DJ` that manipulates a `Playlist`'s track order. You could make `DJ` a subclass or merge the two classes together, but neither makes sense: a DJ is not a kind of playlist, and a playlist does not need DJ behavior baked in.

C++ solves both problems with the `friend` keyword. A class can declare specific functions or classes as **friends**, granting them access to its private and protected members.

Friend Functions

To make a free function a friend, declare it with the `friend` keyword inside the class:

```cpp
#include <iostream>
#include <string>
#include <vector>

class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

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

    friend std::ostream &operator<<(std::ostream &os, const Playlist &p);
};

std::ostream &operator<<(std::ostream &os, const Playlist &p) {
    os << p.name << ":" << std::endl;
    for (size_t i = 0; i < p.songs.size(); ++i) {
        os << "  " << i + 1 << ". " << p.songs[i] << std::endl;
    }
    return os;
}

Playlist p("90s Jams");
p.add("I'll Be There for You");
p.add("Torn");
std::cout << p;
```

Output:

```
90s Jams:
  1. I'll Be There for You
  2. Torn
```

The `friend` declaration inside `Playlist` tells the compiler that `operator<<` may access `name` and `songs` directly, even though it is not a member function. The function itself is defined outside the class, just like any free function.

Notice that `operator<<` returns `std::ostream &` so that calls can be chained: `std::cout << a << b`. This is the same pattern the standard library uses for `std::cout << "hello" << std::endl` — each `<<` returns the stream, ready for the next one.

Friend Classes

You can also make an entire class a friend. Every member function of the friend class then has access to the private members:

```
class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

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

    void print() const {
        std::cout << name << ":" << std::endl;
        for (size_t i = 0; i < songs.size(); ++i) {
            std::cout << "  " << i + 1 << ". " << songs[i] << std::endl;
        }
    }

    friend class DJ;
};

class DJ {
private:
    std::string name;

public:
    DJ(const std::string &n) : name(n) {}

    void intro(const Playlist &p) const {
        std::cout << name << ": up next, " << p.songs.size()
                  << " tracks from " << p.name << "!" << std::endl;
    }

    void swap_first_last(Playlist &p) const {
        if (p.songs.size() > 1) {
            std::string temp = p.songs.front();
            p.songs.front() = p.songs.back();
            p.songs.back() = temp;
        }
    }
};

Playlist p("90s Mix");
p.add("Torn");
p.add("Kiss from a Rose");
```

```
p.add("I'll Be There for You");

DJ dj("DJ Jazzy Jeff");
dj.intro(p);
p.print();
```

Output:

```
DJ Jazzy Jeff: up next, 3 tracks from 90s Mix!
90s Mix:
  1. Torn
  2. Kiss from a Rose
  3. I'll Be There for You
```

The `DJ` class can read and modify `Playlist`'s private `songs` and `name` because `Playlist` declared `DJ` as a friend. Notice that the friendship is one-directional — `DJ` can access `Playlist`'s private members, but `Playlist` cannot access `DJ`'s private members.

Rules of Friendship

Friendship has a few important rules:

- **Friendship is granted, not taken.** A class must declare its own friends inside its definition. You cannot claim friendship from outside.
- **Friendship is not mutual.** If `Playlist` declares `DJ` as a friend, `DJ` can access `Playlist`'s private members, but `Playlist` cannot access `DJ`'s private members unless `DJ` also declares `Playlist` as a friend.
- **Friendship is not inherited.** If `DJ` is a friend of `Playlist`, a class derived from `DJ` does not automatically get that friendship.
- **Friendship is not transitive.** If `A` is a friend of `B`, and `B` is a friend of `C`, `A` is *not* automatically a friend of `C`.

Tip: Use `friend` sparingly. Every friend is a piece of outside code that depends on your class's internal representation. If you change how the class stores its data, you have to update every friend too. Prefer member functions or public interfaces when possible, and reserve `friend` for cases like `operator<<` where there is no alternative.

Trap: Declaring too many friends defeats the purpose of making members private in the first place. If you find yourself adding friends frequently, consider whether the class's public interface is missing something.

Key Points

- The compiler can generate five special member functions: destructor, copy constructor, copy assignment, move constructor, and move assignment.
- The **Rule of Five**: if you write any one of the five, write all five.
- `= default` explicitly requests the compiler-generated version; `= delete` prevents a function from being called.
- Use `= delete` to make a type non-copyable, non-movable, or to prevent implicit conversions.
- The **Rule of Zero**: prefer types that manage themselves so you do not need to write any special member functions.
- **RAII** ties resource lifetimes to object lifetimes — acquire in the constructor, release in the destructor.
- The `friend` keyword grants a specific function or class access to private members.
- Use `friend` for operators like `<<` where the left operand is not your class.
- Friendship is granted, not taken; it is not mutual, inherited, or transitive.

Exercises

1. Explain the difference between the Rule of Five and the Rule of Zero. Which one should you prefer and why?

2. A coworker writes a class with a move constructor but `std::vector` keeps copying objects instead of moving them during reallocation. What is wrong with the move constructor?

```
class Track {
private:
    std::string title;
    std::vector<int> samples;

public:
    Track(const std::string &t) : title(t) {}

    Track(Track &&other)
        : title(std::move(other.title)),
          samples(std::move(other.samples)) {}
};
```

3. What does `= default` do when applied to a special member function? Why would you write `Song() = default;` instead of just omitting the default constructor?

4. What does the following code do, and why is it useful?

```
class Connection {
public:
    Connection(int fd) : fd_(fd) {}
    Connection(const Connection &) = delete;
    Connection &operator=(const Connection &) = delete;
private:
    int fd_;
};
```

5. Why does `operator<<` for output have to be a free function (or a friend) rather than a member function of your class?

6. What does the following program print?

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

class Vault {
private:
    std::string secret;

public:
    Vault(const std::string &s) : secret(s) {}

    friend void peek(const Vault &v);
};

void peek(const Vault &v) {
    std::cout << v.secret << std::endl;
}

int main()
```

```
{
    Vault v("Vogue");
    peek(v);
    return 0;
}
```

7. If class `A` declares class `B` as a friend, and class `B` declares class `C` as a friend, can `C` access `A`'s private members? Why or why not?

8. A class has a `std::string name`, a `std::vector<int> scores` with 3 elements, and an `int id`. How many of the five special member functions do you need to write if all members are standard library types or built-in types?

9. Write a class called `Album` with private members for `title` (string), `artist` (string), and `track_count` (int). Give it a parameterized constructor, a `const` member function that prints the album info, an overloaded `==` operator that compares all three fields, and a friend `operator<<` for output. Test it in `main()` by creating two albums, printing them with `<<`, and comparing them.

10. **Write a program** that defines a class `Buffer` that owns a heap-allocated `char *` and a `size_t` length. Implement **all five** special member functions explicitly:

    - destructor
    - copy constructor
    - copy assignment operator
    - move constructor (mark `noexcept`)
    - move assignment operator (mark `noexcept`)

    Add a small helper that prints which special member is running ("copy ctor", "move ctor", and so on) so you can watch them fire. In `main`, create one `Buffer`, copy it into another, move-construct a third, and let all of them go out of scope. Trace the output by hand before you run it and confirm it matches.

11. **Where is the bug?**

```
class Buffer {
    char *data;
    std::size_t len;
public:
    Buffer(std::size_t n) : data(new char[n]), len(n) {}
    ~Buffer() { delete[] data; }

    Buffer &operator=(const Buffer &other) {
        delete[] data;
        len = other.len;
        data = new char[len];
        for (std::size_t i = 0; i < len; ++i) data[i] = other.data[i];
        return *this;
    }
};

int main()
{
    Buffer b(100);
    b = b;            // assign to itself
    return 0;
}
```

    Walk through what happens inside `operator=` when the right-hand side *is* the left-hand side. Why is the result undefined behavior, and what is the standard fix?

12. **Think about it:** Why does `std::vector` insist that the move constructor and move assignment operator be marked `noexcept` before it will use them? What does the vector do *instead* if your move operations are not `noexcept`, and what is the performance cost?