Gorgo Starting C++

April 11, 2026

Contents

12. Classes

In Chapter 2 you learned that structures group related data together under one name. Since then you have been using structs to bundle fields like `title`, `artist`, and `year` into a single variable. But structs have a problem: all their members are public by default, so any code can reach in and set `year` to -5 or `title` to an empty string. There is no way to enforce rules about valid values, and the functions that operate on a struct live separately from the data they work on. Classes solve this by bundling data *and* behavior into a single unit with control over who can access what. This is the foundation of **object-oriented programming** in C++.

In Chapter 2 you learned that C++ calls any region of memory with a type an **object**. A class is a blueprint — it describes what data and behavior a type has. When you create a variable of that class type, the object you get is called an **instance** of the class. In everyday C++ conversation, "object" and "instance" are used interchangeably when talking about classes. In this chapter you will learn how to define classes, control access to their members, write constructors and destructors, and teach the compiler what `<<` and `==` mean for your own types.

From Structs to Classes

You already know that a `struct` groups related variables together. Here is a simple struct from Chapter 2:

```
struct Song {
    std::string title;
    std::string artist;
    int year;
};
```

You can create a `Song` and access its members with the `.` operator:

```
Song s;
s.title = "Enter Sandman";
s.artist = "Metallica";
s.year = 1991;
```

This works, but as mentioned above, nothing prevents invalid data — anyone can set `year` to -5. Let's fix that.

Access Specifiers

Access specifiers control who can see and use the members of a class. There are three:

- **`public`**: accessible from anywhere.
- **`private`**: accessible only from inside the class itself.
- **`protected`**: accessible from inside the class and from derived classes (classes that extend your class — a topic for a more advanced C++ book).

A **class** is almost identical to a struct, but with one key difference: members are **private by default**, meaning code outside the class cannot access them directly. This lets you control how your data is used.

```
class Song {
public:
    std::string title;
    std::string artist;
    int year;
};
```

Wut: In C++, `struct` and `class` are almost the same thing. The only difference is that members of a `struct` are `public` by default, while members of a `class` are `private` by default. You could use either one, but by convention `class` is used when you want to bundle data with behavior.

```cpp
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    void set_title(const std::string &t) { title = t; }
    std::string get_title() const { return title; }

    void set_year(int y) {
        if (y > 0) {
            year = y;
        }
    }
    int get_year() const { return year; }
};
```

Now `title`, `artist`, and `year` are private. The only way to set the year is through `set_year()`, which checks that the value is positive. This pattern of providing functions to access private data is common in C++.

Tip: Not every private member needs a getter and setter. Only expose what other code actually needs. The whole point of making data private is to keep the interface small and controlled.

Constructors

A **constructor** is a special function that runs automatically when an object is created. Its job is to set up the object so it starts in a valid state.

The constructor has the same name as the class and no return type — not even `void`.

Default Constructor

A **default constructor** takes no arguments:

```cpp
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song() {
        title = "Unknown";
        artist = "Unknown";
        year = 0;
    }
};
```

Now when you create a `Song` without arguments, it starts with sensible defaults:

```cpp
Song s;  // calls the default constructor
// s.title is "Unknown", s.artist is "Unknown", s.year is 0
```

Parameterized Constructor

A **parameterized constructor** takes arguments so you can initialize the object with specific values:

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

public:
    Song() : title("Unknown"), artist("Unknown"), year(0) {}

    Song(const std::string &t, const std::string &a, int y)
        : title(t), artist(a), year(y) {}
};
```

Now you can create songs either way:

```
Song a;                                          // default
Song b("Enter Sandman", "Metallica", 1991);      // parameterized
```

Member Initializer Lists

Notice the `: title(t), artist(a), year(y)` syntax after the constructor's parameter list. This is a **member initializer list**, and it is the preferred way to initialize members in C++.

The member initializer list initializes members directly, rather than default-constructing them and then assigning new values in the body. For simple types like `int`, the difference is negligible. For complex types like `std::string`, the initializer list is more efficient because it avoids constructing a temporary default value that is immediately thrown away.

Tip: Always prefer member initializer lists over assignment in the constructor body. It is more efficient and, for some types like `const` members and references, it is the *only* way to initialize them.

The order of initialization is determined by the order members are *declared* in the class, not the order they appear in the initializer list.

Trap: If you list members in your initializer list in a different order than they are declared, the compiler may warn you. Always match the order of your initializer list to the order of your member declarations.

`explicit` Constructors

A constructor that takes a single argument can be used as an **implicit conversion**. This means the compiler will silently call the constructor to convert one type to another without you asking:

```
class Volume {
public:
    int level;
    Volume(int l) : level(l) {}
};

void play(Volume v) {
    std::cout << "Volume: " << v.level << std::endl;
```

```
}

play(11);   // compiles! the compiler converts 11 to Volume(11)
```

That might be convenient, but it can also cause surprises. The `explicit` keyword prevents this — it forces the caller to construct the object explicitly:

```
class Volume {
public:
    int level;
    explicit Volume(int l) : level(l) {}
};

play(11);           // ERROR: no implicit conversion
play(Volume(11));   // OK: explicit construction
play(Volume{11});   // OK: also explicit
```

Trap: Single-argument constructors without `explicit` are a common source of surprise conversions. The compiler can quietly insert conversions you did not intend, leading to bugs that are hard to spot. As a rule of thumb, mark single-argument constructors `explicit` unless you specifically want implicit conversion.

Constructors with two or more required parameters cannot be called implicitly, so `explicit` matters most on single-argument constructors.

Destructors

A **destructor** is the opposite of a constructor — it runs automatically when an object is destroyed (goes out of scope or is deleted). Its job is to clean up any resources the object owns.

The destructor has the same name as the class, prefixed with ~, and takes no arguments:

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

public:
    Song(const std::string &t, const std::string &a, int y)
        : title(t), artist(a), year(y)
    {
        std::cout << title << " created" << std::endl;
    }

    ~Song() {
        std::cout << title << " destroyed" << std::endl;
    }
};

int main()
{
    Song s("Black Hole Sun", "Soundgarden", 1994);
    std::cout << "doing stuff..." << std::endl;
    return 0;
}
```

Output:

```
Black Hole Sun created
doing stuff...
Black Hole Sun destroyed
```

The destructor is called automatically when `s` goes out of scope at the end of `main()`. You do not call it yourself.

For classes that only contain standard library types like `std::string`, the compiler generates a perfectly good destructor for you. You will see in Chapter 13 that destructors become critical when your class manages its own memory, and in Chapter 14 that the compiler can generate special member functions like copy and move constructors automatically.

Member Functions

Functions declared inside a class are called **member functions** (or methods). They have access to all the class's members, including private ones.

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

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

    void print() const {
        std::cout << title << " by " << artist
                  << " (" << year << ")"
                  << std::endl;
    }

    bool is_90s() const {
        return year >= 1990 && year <= 1999;
    }
};

int main()
{
    Song s("All Star", "Smash Mouth", 1999);
    s.print();

    if (s.is_90s()) {
        std::cout << "Es de los noventa!" << std::endl;
    }

    return 0;
}
```

Output:

```
All Star by Smash Mouth (1999)
Es de los noventa!
```

Notice the `const` after the parameter list in `print()` and `is_90s()`. This tells the compiler that these functions do not modify the object. You should mark every member function that does not change the object as `const`.

Marking member functions `const` matters more than you might think. When you have a `const` reference or a `const` object, the compiler only allows you to call member functions that are marked `const`. This comes up constantly because, as you saw in Chapter 6, passing objects by `const` reference is the standard way to avoid expensive copies.

Consider a function that takes a `Song` by `const` reference:

```
void show(const Song &s)
{
    s.print();   // OK: print() is const
    s.is_90s();  // OK: is_90s() is const
}
```

This compiles because both `print()` and `is_90s()` are marked `const`. The compiler knows they will not modify the object, so calling them through a `const` reference is safe.

Now imagine `print()` was *not* marked `const`:

```
// BAD: forgot const on print()
void print() {
    std::cout << title << " by " << artist
              << " (" << year << ")" << std::endl;
}
```

The `show()` function above would fail to compile with an error like:

```
error: passing 'const Song' as 'this' argument discards qualifiers
```

The compiler is telling you that `print()` *might* modify the object (since it is not `const`), so calling it on a `const Song` is not allowed.

The fix is simple: add `const` after the parameter list. Get into the habit of marking every member function that does not modify the object as `const`. It documents your intent, catches accidental mutations at compile time, and ensures your class works smoothly with `const` references and objects.

Overloading Member Functions

Sometimes you want the same operation to behave differently depending on what information the caller provides. For example, you might want `print()` to work on its own but also accept an optional label. You saw in Chapter 6 that **function overloading** lets you define multiple functions with the same name as long as they have different signatures (recall from Chapter 0 that a signature is the function's name and parameter list). The same technique works with member functions.

Here is a `Song` class with an overloaded `print()`:

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

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

    void print() const {
        std::cout << title << " by " << artist
```

```
                  << " (" << year << ")"
                  << std::endl;
    }

    void print(const std::string &label) const {
        std::cout << label << ": " << title << " by " << artist
                  << " (" << year << ")" << std::endl;
    }
};

Song s("Virtual Insanity", "Jamiroquai", 1996);
s.print();                    // calls print()
s.print("Now playing");       // calls print(const std::string &)
```

Output:

```
Virtual Insanity by Jamiroquai (1996)
Now playing: Virtual Insanity by Jamiroquai (1996)
```

You have already seen overloading without realizing it — the `Song` class has two constructors (the default and the parameterized), and those are overloaded functions with the same name `Song`.

Overloading works well when the number or types of parameters differ more substantially. Here is a `Playlist` class with overloaded `add()` methods — one that appends to the end and one that inserts at a specific position:

```
#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);
    }

    void add(const std::string &song, int position) {
        if (position >= 0 && position <= static_cast<int>(songs.size())) {
            songs.insert(songs.begin() + position, 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;
        }
    }
};

int main()
{
```

```
    Playlist p("90s Jams");
    p.add("Torn");                  // calls add(const std::string &)
    p.add("Kiss from a Rose");      // calls add(const std::string &)
    p.add("Basket Case", 1);        // calls add(const std::string &, int)
    p.print();
    return 0;
}
```

Output:

```
90s Jams:
  1. Torn
  2. Basket Case
  3. Kiss from a Rose
```

Overloading keeps your interface clean: one verb for one concept, regardless of how many ways you can call it. Overloading is not limited to member functions — you saw it with free functions in Chapter 6.

Trap: The compiler chooses an overload based on implicit conversions, which can be surprising. If you have `void play(bool)` and `void play(const std::string &)`, calling `play("Torn")` calls the `bool` version — because `"Torn"` is a `const char*`, which converts to `bool` (a non-null pointer is `true`) rather than constructing a `std::string`. Use `play(std::string("Torn"))` to get the version you intended.

The `this` Pointer

When a member function refers to another member by name, the compiler knows to look for it in the current object — writing `title` inside a `Song` method means "this object's `title`". But sometimes you need a reference to the current object *itself*: to pass it to another function, return it from a method, or compare it against another instance. In English, when we want to refer to ourselves we say `me`. C++ doesn't use `me` — it uses another keyword.

When you call `s.print()`, the `print()` function needs to know *which* object it is operating on. If you have ten `Song` objects, there is only one copy of the `print()` code — it needs a way to find the right `title`, `artist`, and `year`. C++ solves this by automatically passing a hidden pointer to the object into every member function call. That pointer is called `this`. You can think of it as the C++ equivalent of `me`.

Before C++11, raw **pointers** were used constantly in C++. Modern C++ has largely moved away from raw pointers in favor of references and smart pointers (Chapter 13), which is why we have made it this far without discussing them. We are still not going to cover pointers in depth here, but you need to know two things about `this`:

- `this` is a **pointer** to the current object. Use `this->member` to access a member variable or function through the pointer.
- `*this` **dereferences** the pointer, giving you the actual object itself.

You usually do not need `this` because member names are accessible directly — when you write `title` inside a member function, the compiler understands you mean `this->title`. But there are a few situations where `this` is necessary.

Resolving Name Conflicts

When a parameter has the same name as a member, `this->` tells the compiler which one you mean:

```
class Song {
private:
    std::string title;
    int year;
```

```
public:
    Song(const std::string &title, int year)
        : title(title), year(year) {}

    void set_year(int year) {
        // this->year is the member, year is the parameter
        this->year = year;
    }
};
```

Without `this->`, the compiler would think `year = year` is assigning the parameter to itself.

Tip: Using member initializer lists (as shown in the constructor above) avoids the `this` ambiguity problem for constructors, as shown in the constructor with `title(title)`. For setter functions, `this->` is a clean way to resolve the conflict.

Returning the Current Object

A member function can return `*this` to give the caller back the object itself. This enables **method chaining** — calling multiple member functions in a single statement:

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

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

    Playlist &add(const std::string &song) {
        songs.push_back(song);
        return *this;
    }

    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;
        }
    }
};

Playlist p("90s Jams");
p.add("Torn").add("Kiss from a Rose").add("Basket Case");
p.print();
```

Output:

```
90s Jams:
  1. Torn
  2. Kiss from a Rose
  3. Basket Case
```

Each call to `add()` returns `*this` — the `Playlist` object itself — so the next `.add()` call operates on the same object. Notice the return type is `Playlist &` (a reference), not `Playlist`, so the object is not copied each time.

Copying the Current Object

You can also use `*this` to make a copy of the current object. This is useful when a member function needs to return a modified version without changing the original:

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

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

    Song with_year(int y) const {
        Song copy = *this;  // copy the current object
        copy.year = y;       // modify only the copy
        return copy;
    }

    void print() const {
        std::cout << title << " by " << artist
                  << " (" << year << ")"
                  << std::endl;
    }
};
```

```
Song original("Torn", "Natalie Imbruglia", 1997);
Song remaster = original.with_year(2023);
original.print();
remaster.print();
```

Output:

```
Torn by Natalie Imbruglia (1997)
Torn by Natalie Imbruglia (2023)
```

`Song copy = *this` creates a full copy of the current `Song`. The function modifies the copy and returns it, leaving the original unchanged. You will see this same pattern in the Operator Overloading section later in this chapter, where `operator+` uses `*this` to make a copy before adding to it.

Default Parameters in Member Functions

Default parameters (introduced in Chapter 6) work in member functions too. The same rules apply: defaults must go at the end of the parameter list, and mixing overloading with defaults can create ambiguity.

Here is the `Playlist` class rewritten to use a default parameter instead of two overloaded `add()` methods:

```
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, int position = -1) {
```

```
        if (position < 0 || position >= static_cast<int>(songs.size())) {
            songs.push_back(song);
        } else {
            songs.insert(songs.begin() + position, 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;
        }
    }
};
```

Now `add("Torn")` appends to the end (using the default `-1`), and `add("Basket Case", 1)` inserts at position 1.

Tip: When you separate a class into a header and source file, put default parameter values in the *declaration* (the header), not in the *definition* (the source file). The compiler needs to see the defaults at the call site.

Separating Declaration from Definition

As your classes grow, putting everything in one file becomes unwieldy. C++ lets you split a class into a **header file** (`.h`) for the declaration and a **source file** (`.cpp`) for the definitions.

song.h — the declaration:

```
#ifndef SONG_H
#define SONG_H

#include <string>

class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song(const std::string &t, const std::string &a, int y);
    void print() const;
    bool is_90s() const;
};

#endif
```

song.cpp — the definitions:

```
#include "song.h"
#include <iostream>

Song::Song(const std::string &t, const std::string &a, int y)
    : title(t), artist(a), year(y) {}
```

```cpp
void Song::print() const {
    std::cout << title << " by " << artist
              << " (" << year << ")"
              << std::endl;
}

bool Song::is_90s() const {
    return year >= 1990 && year <= 1999;
}
```

The `Song::` prefix tells the compiler that these functions belong to the `Song` class. This is the scope resolution operator `::` that you first saw with namespaces in Chapter 1.

The `#ifndef/#define/#endif` pattern is called an **include guard**. It prevents the header from being included more than once in the same compilation, which would cause duplicate definition errors.

Tip: Most compilers support `#pragma once` as a simpler alternative to include guards. It does the same thing and is commonly used, even though it is not part of the C++ standard.

`static` Members

Everything you have seen about classes so far has been about *instance* state: each `Song` has its own `title`, `artist`, and `year`, and each member function operates on one specific object via the hidden `this` pointer. But some things belong to the class as a whole, not to any single instance. How many `Song` objects have been created so far? What is the maximum title length the class allows? What is the default volume to play at when no volume is specified?

Here is what the "how many `Songs` have been created" question looks like *without* static members:

```cpp
// song_count.h
extern int song_count;

// song_count.cpp
int song_count = 0;

// song.h
#include <string>
#include "song_count.h"

class Song {
    std::string title;
public:
    Song(const std::string &t) : title(t) { ++song_count; }
    ~Song() { --song_count; }
};
```

The counter is conceptually part of the `Song` class — it only makes sense in relation to `Song` — but it lives outside the class as a global variable. That is awkward for several reasons:

- It needs an `extern` declaration in its own header and a definition in a `.cpp` file, just like any other shared global (you saw that pattern in Chapter 6).
- Anyone can modify `song_count` from anywhere in the program, defeating the encapsulation that classes are supposed to provide.
- The name `song_count` pollutes the global namespace.
- If you add a second piece of `Song`-wide state, you get yet another global with all the same problems.

`static` Data Members

The `static` keyword inside a class declaration gives you a variable that belongs to the *class* rather than to any instance:

```
// song.h
#include <string>

class Song {
    std::string title;
    static int count;           // declaration --- shared by all Songs
public:
    Song(const std::string &t) : title(t) { ++count; }
    ~Song() { --count; }

    static int instance_count() { return count; }
};
```

There is exactly one `count` for the whole program, no matter how many `Song` objects exist. Every `Song` constructor and destructor sees the same `count` and updates the same underlying storage.

There is one small wrinkle: declaring a `static` data member inside the class body is not enough — you also have to provide its *definition* somewhere, because the linker needs a place to put the actual storage. The convention is to declare it in the header and define it in the matching `.cpp` file:

```
// song.cpp
#include "song.h"

// definition --- exactly one, in one .cpp file
int Song::count = 0;
```

Notice the `Song::` prefix that attaches the definition to the class, and notice that the `static` keyword does **not** appear in the definition — it only appears in the declaration. Repeating `static` in the definition is a compile error.

If two `.cpp` files both define `Song::count`, the linker will complain about a duplicate definition for the same reason a plain global would. Static data members still obey the one-definition rule — the whole point is that they act like a well-behaved global that is scoped to the class instead of to the whole program.

Tip: For `static const` or `static constexpr` data members with a simple value, C++17 lets you define the value *inline* inside the class body and skip the separate `.cpp` definition entirely:

```
class Song {
    static constexpr int max_title_length = 200;
};
```

No matching `int Song::max_title_length = 200;` in a `.cpp` file is needed. This is the preferred style for class-wide constants.

`static` Member Functions

A `static` member function belongs to the class rather than to any instance. You call it by qualifying it with the class name and the scope resolution operator:

```
int n = Song::instance_count();
```

You do not need a `Song` object to call it — in fact, you do not need *any* `Songs` to exist at all.

A `static` member function has no `this` pointer, so it cannot touch non-static member variables or call non-static member functions directly. It can only see other `static` members. The `instance_count()` function

above works because `count` is also `static`.

Wut: The `static` keyword has two completely *unrelated* meanings depending on where you use it:

- **On a free (non-member) function** (Chapter 6): `static` gives the function **internal linkage**, meaning it is private to the `.cpp` file it lives in and invisible to the linker outside that file.
- **Inside a class declaration** (this chapter): `static` means the member belongs to the *class* rather than to an instance. It has nothing to do with linkage — a `static` member function is still visible to the linker like any other member function, and code in other translation units can call `Song::instance_count()` perfectly well.

Same keyword, completely different jobs. The C++ committee has been apologizing for this ever since.

When to Use `static` Members

Static members are the right tool when the data or behavior really is class-wide, not per-instance:

- **Class-wide constants and configuration** — maximum length, default values, version numbers. Prefer `static constexpr` when the value is known at compile time.
- **Class-wide counters or caches** — "how many of these have been created," "the most recently created one," a shared lookup table.
- **Factory functions** — static member functions that create and return new instances of the class, often as an alternative to constructors when the construction process is more involved than just initializing members.
- **Sentinel values** — a well-known constant that callers can compare against, like "not found."

Examples of good use from the standard library:

- `std::numeric_limits<int>::max()` — a `static constexpr` function returning the largest value an `int` can hold. A compile-time class-wide constant that depends only on the type, so attaching it to the type is exactly right.
- `std::string::npos` — a `static const` data member that `std::string::find()` returns to mean "not found." A sentinel value that is conceptually tied to `std::string` and nothing else.
- `std::chrono::system_clock::now()` — a `static` factory function that returns the current time as a `time_point`. The class *is* the clock; `now()` is a class-wide operation that manufactures an instance.

When Not to Use `static` Members

Static members get tempting as a way to bolt loose functions or global variables onto an existing class, and that temptation usually leads to bad design.

- **Do not create a class just to hold a bunch of static functions.** If none of the functions touch instance state, they are really free functions and should live in a namespace instead. A "utility class" that is never instantiated is almost always a namespace wearing a costume.
- **Do not use a static data member as a stealth global variable.** If the data is not conceptually tied to the class as a whole — if it just happens to be used by some of the class's methods — it is really a global, and all the usual problems with globals still apply (hidden coupling, hard-to-test code, surprising initialization order across translation units).
- **Do not mark a function `static` just because its current implementation happens not to touch `this`.** If the function logically operates on an instance, leave it as a normal member function. Otherwise, you pin callers to the `Class::foo()` syntax, and having to switch back to `obj.foo()` later when the implementation starts needing `this` is an annoying breaking change.

An example from the standard library that illustrates the temptation: `std::thread::hardware_concurrency()` reports how many hardware threads the system can run concurrently. It lives on `std::thread` as a static

method, but it has nothing to do with any particular thread — it is a report about the machine. You could equally well imagine it as a free function in `<thread>` alongside the other threading utilities. It is a reasonable design and not really a bug, but it also reads like the kind of function that ends up as a `static` member more by association than by necessity. When you are choosing between "static member function on `X`" and "free function in a namespace near `X`," lean toward the free function unless the operation is genuinely tied to the class.

Operator Overloading

C++ lets you define what operators like `==`, `+`, and others mean for your own types. This is called **operator overloading**. You write an operator overload as a member function named `operator` followed by the symbol.

The + Operator

You can overload `+` to make it do something meaningful for your class. Here is a `Playlist` where `+` adds a song and returns a new playlist:

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

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

    Playlist operator+(const std::string &song) const {
        Playlist result = *this;
        result.songs.push_back(song);
        return result;
    }

    Playlist &operator+=(const std::string &song) {
        songs.push_back(song);
        return *this;
    }

    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;
        }
    }
};

Playlist p("90s Jams");
p = p + "Torn";
p += "Kiss from a Rose";
p += "Basket Case";
p.print();
```

Output:

```
90s Jams:
  1. Torn
  2. Kiss from a Rose
  3. Basket Case
```

The `operator+` member function takes a song title, makes a copy of the current playlist with `*this`, appends the song to the copy, and returns it as a new `Playlist`. The original playlist is not modified — just like `3 + 2` does not change the value of `3`.

The `operator+=` member function modifies the playlist in place and returns a reference to `*this`. Notice that `operator+` is `const` but `operator+=` is not. `operator+` does not modify the object it is called on — it creates a new playlist with the extra song — so it can be `const`. `operator+=` modifies the object itself by appending directly to its `songs` vector, so it cannot be `const`. This mirrors how built-in types work: `x + 1` does not change `x`, but `x += 1` does.

The == Operator for Comparison

You can also define what it means for two objects to be equal:

```
class Playlist {
    // ... (same members as above)

public:
    // ... (same constructor and methods as above)

    bool operator==(const Playlist &other) const {
        return name == other.name && songs == other.songs;
    }
};
```

```
Playlist a("Mix");
a = a + "Torn";
Playlist b("Mix");
b = b + "Torn";
Playlist c("Mix");
c = c + "Basket Case";

std::cout << (a == b) << std::endl;  // 1 (true)
std::cout << (a == c) << std::endl;  // 0 (false)
```

Notice that `operator==` has access to private members of `other` because `other` is the same class.

Wut: You can overload the function-call operator `()` as a member function. This makes an object *callable*, as if it were a function:

```
struct Greeter {
    std::string greeting;

    void operator()(const std::string &name) const {
        std::cout << greeting << ", " << name << "!" << std::endl;
    }
};

Greeter hola{"Hola"};
hola("amigo");       // prints: Hola, amigo!
```

An object that overloads `()` is called a **functor**. You will see in *Gorgo Continuing C++* that a lambda is really just a functor the compiler generates for you.

Conversion Operators

Just as a constructor can convert *to* your type, a **conversion operator** lets your type convert *to* another type. You write it as a member function named `operator` followed by the target type, with no explicit return type:

```
class Volume {
private:
    int level;

public:
    explicit Volume(int l) : level(l) {}

    operator int() const {
        return level;
    }
};
```

Now a `Volume` can be used anywhere an `int` is expected:

```
Volume v(11);
int n = v;                      // implicit conversion: n is 11
std::cout << v + 1 << std::endl; // 12
```

This is convenient, but implicit conversions can cause the same surprises as non-explicit constructors. The fix is the same — add `explicit`:

```
explicit operator int() const {
    return level;
}
```

Now the conversion only happens when you ask for it:

```
Volume v(11);
// int n = v;                    // ERROR: no implicit conversion
int n = static_cast<int>(v);     // OK: explicit cast
```

The most common use of `explicit` conversion operators is `explicit operator bool()`. This is the pattern the standard library uses for streams — it is why `if (std::cin)` works (a boolean context like `if` allows explicit conversions) but `int x = std::cin` does not:

```
class Connection {
private:
    int fd;

public:
    explicit Connection(int f) : fd(f) {}

    explicit operator bool() const {
        return fd >= 0;
    }
};

Connection conn(3);
// OK: bool context allows explicit conversion
if (conn) {
    std::cout << "connected" << std::endl;
}
// bool b = conn;                 // ERROR: not a bool context
```

Tip: Prefer `explicit` on conversion operators. An implicit `operator bool()` would let code like `int x = conn + 5;` compile silently — `conn` would convert to `bool` (which is `true`, i.e., `1`), then `1 + 5` gives `6`. That is almost never what you want.

Putting It All Together

Here is a complete program that uses everything from this chapter:

```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);
    }

    Playlist operator+(const std::string &song) const {
        Playlist result = *this;
        result.songs.push_back(song);
        return result;
    }

    Playlist &operator+=(const std::string &song) {
        songs.push_back(song);
        return *this;
    }

    bool operator==(const Playlist &other) const {
        return name == other.name && songs == other.songs;
    }

    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;
        }
    }
};

int main()
{
    Playlist a("90s Jams");
    a.add("Torn");
    a += "Kiss from a Rose";

    Playlist b = a + "Basket Case";

    a.print();
    std::cout << std::endl;
    b.print();
    std::cout << std::endl;
```

```
    if (a == b) {
        std::cout << "Same playlist" << std::endl;
    } else {
        std::cout << "Different playlists" << std::endl;
    }

    return 0;
}
```

Output:

```
90s Jams:
  1. Torn
  2. Kiss from a Rose

90s Jams:
  1. Torn
  2. Kiss from a Rose
  3. Basket Case

Different playlists
```

Key Points

- A class bundles data and the functions that operate on that data together.
- Members are `private` by default in a `class` and `public` by default in a `struct`.
- Access specifiers (`public`, `private`, `protected`) control who can access members.
- Constructors initialize objects; use member initializer lists for efficiency.
- Mark single-argument constructors `explicit` to prevent implicit conversions.
- Destructors clean up when an object is destroyed.
- Mark member functions `const` when they do not modify the object.
- The `this` pointer refers to the current object inside a member function.
- Function overloading lets you define multiple functions with the same name but different parameter lists.
- Default parameters let callers omit trailing arguments by providing default values.
- Do not mix overloading and default parameters in ways that create ambiguous calls.
- Split large classes into a header file (`.h`) for declarations and a source file (`.cpp`) for definitions.
- Operator overloading lets you define the behavior of operators for your own types.
- Conversion operators (`operator T()`) let a class convert to another type; mark them `explicit` to prevent surprises.
- The `friend` keyword grants outside functions or classes access to private members — see Chapter 14.

Exercises

1. What is the difference between a `struct` and a `class` in C++? Why would you choose one over the other?
2. What does the following program print?

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

class Band {
private:
    std::string name;
```

```
    int formed;

public:
    Band(const std::string &n, int y) : name(n), formed(y) {
        std::cout << name << " arrives" << std::endl;
    }

    ~Band() {
        std::cout << name << " exits" << std::endl;
    }
};

int main()
{
    Band a("Metallica", 1981);
    Band b("Soundgarden", 1984);
    std::cout << "show time" << std::endl;
    return 0;
}
```

3. What is wrong with the following class?

```
class Counter {
private:
    int count;

public:
    Counter() : count(0) {}

    void increment() const {
        count++;
    }

    int get_count() const { return count; }
};
```

4. Why should you prefer member initializer lists over assignment in the constructor body? Give an example of a situation where the initializer list is *required*.

5. If a class has three `int` members and a `std::string` member, how many bytes *minimum* does an object of that class occupy on a system where `int` is 32 bits and `std::string` is 32 bytes? (Ignore padding for this question.)

6. What does the following code output?

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

class Song {
private:
    std::string title;
public:
    Song(const std::string &t) : title(t) {}

    bool operator==(const Song &other) const {
        return title == other.title;
    }
```

```
};

int main()
{
    Song a("All Star");
    Song b("All Star");
    Song c("Enter Sandman");

    std::cout << (a == b) << std::endl;
    std::cout << (a == c) << std::endl;
    return 0;
}
```

7. What is the bug in this code?

```
class Player {
private:
    std::string name;
    int score;

public:
    Player(const std::string &name, int score) {
        name = name;
        score = score;
    }
};
```

8. What does the following program print?

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

class Radio {
public:
    void play(const std::string &song) {
        std::cout << "Playing: " << song << std::endl;
    }

    void play(const std::string &song, int volume) {
        std::cout << "Playing: " << song << " at volume " << volume << std::endl;
    }

    void play(int station) {
        std::cout << "Tuned to station " << station << std::endl;
    }
};

int main()
{
    Radio r;
    r.play("Torn");
    r.play(98);
    r.play("Basket Case", 11);
    return 0;
}
```

9. What is wrong with the following code?

```
class Speaker {
public:
    void set_volume(int v) {
        volume = v;
    }

    void set_volume(int v, int max = 100) {
        volume = (v > max) ? max : v;
    }

private:
    int volume;
};
```

10. Why must default parameters appear at the end of the parameter list? What happens if you try to put a default parameter before a non-default one?

11. What is wrong with this code?

```
class TrackNumber {
public:
    int number;
    TrackNumber(int n) : number(n) {}
};

void play(TrackNumber t) {
    std::cout << "Track " << t.number << std::endl;
}

int main() {
    play(7);
    return 0;
}
```

12. What does `explicit operator bool()` allow that a non-explicit `operator bool()` also allows? What does it prevent?

13. Write a class called `Counter` with a private `int count` that starts at 0. Give it an `increment()` method, a `reset()` method, and a `const` method `value()` that returns the current count. Overload `operator==` to compare two counters by their count. Test it in `main()` by incrementing, printing, resetting, and comparing two counters.

14. **Where is the bug?**

```
class Playlist {
    std::vector<std::string> tracks;
public:
    int size() const { return tracks.size(); }
};

int main()
{
    Playlist p;
    p.tracks.push_back("Wonderwall");
    std::cout << p.size() << "\n";
```

```
    return 0;
}
```

What does the compiler say, and which design rule does it enforce? What is the smallest change you can make to `Playlist` so the program compiles, and what is the *better* change?

15. **Write a program** that defines a class `Builder` whose `add(int)` method returns `*this` by reference so calls can be chained:

    ```
    Builder b;
    b.add(1).add(2).add(3).add(4);
    ```

    The class should keep a `std::vector<int>` internally and have a `print()` method that prints all of the values added so far. Why does `add` return `*this` and not just a fresh `Builder`?

16. **Think about it:** A class has this conversion operator:

    ```
    class Volume {
        int level;
    public:
        explicit Volume(int v) : level(v) {}
        explicit operator bool() const { return level > 0; }
    };
    ```

    Why is `operator bool` marked `explicit`? Which of the following calls compile, and which do not?

    ```
    Volume v(3);
    if (v) { /* ... */ }                    // (a)
    bool b = v;                             // (b)
    bool b2 = static_cast<bool>(v);         // (c)
    int  n  = v;                            // (d)
    ```