Gorgo Starting C++

April 11, 2026

Contents

6. Functions

As your programs grow, stuffing everything into `main` becomes unmanageable. Functions let you break a program into smaller, reusable pieces. Each function has a name, takes some input, does some work, and optionally returns a result.

You have already been using functions without thinking about it — `std::cout <<` calls operator functions behind the scenes, and `main` is itself a function. In this chapter you will learn how to write your own functions, how C++ passes data to and from them, and some powerful features like overloading and function pointers.

Declarations vs. Definitions

A function **declaration** (also called a prototype) tells the compiler the function's name, return type, and parameter types. A function **definition** provides the actual body — the code that runs when the function is called.

```
// Declaration (prototype) --- no body, just a semicolon
int add(int a, int b);

// Definition --- includes the body
int add(int a, int b) {
    return a + b;
}
```

The declaration tells the compiler "this function exists and here is its signature." The definition tells the compiler "here is what the function actually does."

Forward Declarations

The compiler reads your source file from top to bottom. If you call a function before the compiler has seen its definition, it does not know what the function looks like and reports an error. A **forward declaration** solves this by placing the declaration above the call:

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

// Forward declaration
void greet(const std::string &name);

int main() {
    greet("Mack");
    return 0;
}

// Definition comes after main
void greet(const std::string &name) {
    std::cout << "Return of the " << name << "!\n";
}
```

Without the forward declaration, the compiler would not know about `greet` when it encounters the call in `main`.

Tip: You can always avoid forward declarations by placing function definitions above where they are called. However, forward declarations are essential in larger programs where functions call each other, and they are the foundation of header files.

Header Files and the One-Definition Rule

When your program grows beyond a single file, you split it into multiple `.cpp` files. Each `.cpp` file, after all its `#include` directives are expanded, is called a **translation unit**. The compiler compiles each translation unit independently, and the linker combines them into a single program.

C++ enforces the **one-definition rule** (ODR): a function (or variable) can be *defined* only once across all translation units. Declarations can appear as many times as you like, but there must be exactly one definition.

This is why forward declarations exist — they let you declare a function in a header file so multiple `.cpp` files can call it, while the definition lives in exactly one `.cpp` file.

But what if you want to put a small helper function's *definition* directly in a header file? If two `.cpp` files both include that header, the linker sees two definitions and reports an error.

The `inline` keyword solves this:

```
// helpers.h
inline int max_volume() {
    return 11;
}
```

`inline` tells the compiler "this definition may appear in multiple translation units — treat them as the same definition." Now any `.cpp` file can include `helpers.h` without causing a linker error.

Tip: If you define a function in a header file (outside of a class body), mark it `inline` to avoid ODR violations. When you learn about classes in Chapter 12, you will see that member functions defined inside the class body are implicitly `inline`.

Wut: `inline` does not mean "the compiler will inline this function" (i.e., paste its body at each call site). It means "this definition may appear in multiple translation units." Modern compilers decide on their own whether to inline a call, regardless of the keyword.

`static` Functions

Sometimes you want a helper function that is used inside one `.cpp` file but is not meant to be called from anywhere else — it is purely an implementation detail of that file. Imagine two `.cpp` files that each need their own small `clamp` helper, each with a range that makes sense for that file:

```
// audio.cpp
int clamp(int v) {
    return v < 0 ? 0 : (v > 100 ? 100 : v);    // volume: 0..100
}

void set_volume(int v) {
    int safe = clamp(v);
    // ... apply safe volume
}
```

```
// video.cpp
int clamp(int v) {
    return v < -90 ? -90 : (v > 90 ? 90 : v);  // tilt angle: -90..90
}

void tilt_camera(int v) {
    int safe = clamp(v);
```

```
    // ... tilt the camera
}
```

Each file has its own `clamp` with a range that makes sense for *that* file's purpose. The problem is that at link time the linker sees two functions both named `clamp`, both with the signature `int(int)`, and reports a duplicate definition error. Even if the linker somehow picked one, `audio.cpp`'s `clamp` would still be callable from `video.cpp`, which defeats the idea that each `clamp` is a private helper.

The fix is the `static` keyword:

```
// audio.cpp
static int clamp(int v) {
    return v < 0 ? 0 : (v > 100 ? 100 : v);
}

// video.cpp
static int clamp(int v) {
    return v < -90 ? -90 : (v > 90 ? 90 : v);
}
```

A `static` function has **internal linkage**: the linker only makes the function visible inside the translation unit where it is defined. Each `.cpp` file gets its own private copy of `clamp`, so the two no longer collide, and nothing outside `audio.cpp` can accidentally call `audio.cpp`'s version — that version does not exist as far as the linker is concerned.

Use `static` on any file-scope function that is a helper for the current `.cpp` file and should not be part of the file's public interface. It is the C++ way of saying "private to this file."

Wut: The `static` keyword has another, completely unrelated meaning when applied to a variable *inside* a function: it makes that variable persist across calls instead of being recreated each time. That is local `static`, and it has nothing to do with linkage — it is just a name collision in the grammar.

Tip: Modern C++ offers a second way to get the same "private to this file" effect — put the helper inside an **anonymous namespace**:

```
namespace {
    int clamp(int v) { /* ... */ }
}
```

Anonymous namespaces apply to functions, variables, and types, while file-scope `static` only applies to functions and variables. For a single helper function, either style is fine.

`extern` Declarations

The mirror image of `static` is `extern`: it tells the compiler "the thing I am naming here is *defined* in another translation unit — trust me that the linker will find it."

You have already been using an implicit form of `extern` without knowing it. Every **function declaration** (the forward-declaration syntax you saw earlier in this chapter) is already `extern` by default:

```
int max_volume();          // declaration; implicitly extern
extern int max_volume();   // exact same thing, extern just made explicit
```

For functions you almost never write `extern` explicitly — just use the plain forward declaration.

Where `extern` actually matters is **global variables**. Suppose you want a global `master_volume` that every file in the program can read:

```cpp
// audio.cpp
int master_volume = 5;

// video.cpp
int master_volume;   // BUG: this DEFINES a second master_volume
```

Writing `int master_volume;` in `video.cpp` does not reference the one from `audio.cpp` — it creates a brand-new variable with its own storage inside `video.cpp`. The linker now sees two `master_volumes`, an ODR violation, and refuses to link.

`extern` fixes the problem by turning that line into a *declaration* instead of a definition:

```cpp
// video.cpp
extern int master_volume;   // declaration only; storage lives in audio.cpp
```

Now `video.cpp` can read and write `master_volume`, the actual storage lives in `audio.cpp`, and there is exactly one `master_volume` across the whole program. The usual pattern is to put the `extern` declaration in a header that every file includes, with the real definition in exactly one `.cpp` file:

```cpp
// globals.h
extern int master_volume;

// audio.cpp
#include "globals.h"
int master_volume = 5;

// video.cpp
#include "globals.h"
// master_volume is now visible --- no new definition created
```

When you need `extern`:

- To share a global variable across `.cpp` files. Put the `extern` declaration in a header, and put the real definition (with an initializer) in exactly one `.cpp` file.

When you do not need `extern`:

- For function declarations — they are already `extern` by default, so `int clamp(int v);` and `extern int clamp(int v);` are identical. Just write the plain forward declaration.
- For global variables that are only used inside one `.cpp` file. Mark those `static` (or put them in an anonymous namespace) so they stay private to that file.
- For local variables inside a function. They live on the stack and have no linkage to begin with.

Trap: Forgetting `extern` on a shared-global declaration in a header is the classic way to accidentally define a fresh copy of the variable in every translation unit that includes that header. Everything compiles cleanly, but at link time you get a duplicate definition error — or, even worse, the program links and every file silently operates on its own private copy of what was supposed to be a shared global. Always write `extern` on global-variable declarations in headers.

Parameters and Return Values

A function can take zero or more parameters and return a single value. The return type appears before the function name:

```cpp
int multiply(int x, int y) {
    return x * y;
}
```

You call the function by passing arguments that match the parameter types:

```cpp
int result = multiply(6, 7);
std::cout << result << "\n";   // 42
```

`void` Functions

If a function does not return a value, its return type is `void`:

```cpp
void print_chorus() {
    std::cout << "I want it that way\n";
}
```

You can still use `return;` (with no value) inside a `void` function to exit early, but you are not required to.

```cpp
void check_age(int age) {
    if (age < 0) {
        std::cout << "Invalid age\n";
        return;   // exit early
    }
    std::cout << "Age: " << age << "\n";
}
```

Pass-by-Value

By default, C++ passes arguments **by value**. This means the function receives a *copy* of the argument, not the original. Modifying the parameter inside the function does not affect the caller's variable:

```cpp
#include <iostream>

void try_to_change(int x) {
    x = 999;   // modifies the local copy only
}

int main() {
    int num = 42;
    try_to_change(num);
    std::cout << num << "\n";   // still 42
    return 0;
}
```

The function `try_to_change` modified its own copy of `x`, but `num` in `main` was never touched.

Pass-by-Reference

If you want a function to modify the caller's variable, pass it **by reference** using &:

```cpp
#include <iostream>

void make_it_louder(int &volume) {
    volume = 11;   // modifies the original
}

int main() {
    int vol = 5;
    make_it_louder(vol);
    std::cout << "Volume: " << vol << "\n";   // 11
    return 0;
}
```

The `&` after the type means `volume` is a reference — an alias for the caller's variable, not a copy. Any changes to `volume` are changes to `vol`.

A classic use of pass-by-reference is a swap function:

```
#include <iostream>

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);
    std::cout << x << " " << y << "\n";   // 20 10
    return 0;
}
```

Tip: Use pass-by-value for small types like `int`, `char`, and `double`. Use pass-by-reference for larger types like `std::string` and structures to avoid the cost of copying.

`const` Parameters

Sometimes you want to pass by reference for efficiency (to avoid copying) but you do not want the function to modify the argument. Mark the parameter `const`:

```
void print_song(const std::string &title) {
    std::cout << "Now playing: " << title << "\n";
    // title = "something else";   // ERROR: title is const
}
```

The `const` reference gives you the best of both worlds: no copy is made, and the compiler guarantees the function cannot modify the original.

Tip: A good rule of thumb: if a function does not need to modify a parameter, make it `const`. For small types like `int`, pass by value. For larger types like `std::string`, pass by `const` reference. This communicates your intent clearly and lets the compiler catch accidental modifications.

Wut: A `const` reference can bind to a temporary value, but a non-const reference cannot:

```
void print_song(const std::string &title);
void modify_song(std::string &title);

print_song("Semi-Charmed Life");   // OK: const ref binds to temporary
// ERROR: non-const ref cannot bind to temporary
modify_song("Semi-Charmed Life");
```

This is why `const` references are so useful for function parameters that only read their argument.

Structures and Pass-by-Value

When you pass a structure by value, the entire structure is copied. For small structures this is fine, but for large ones the copy can be expensive:

```cpp
struct Album {
    std::string title;
    std::string artist;
    int year;
    int tracks;
};

// BAD: copies the entire Album struct
void print_album_bad(Album a) {
    std::cout << a.title << " by " << a.artist << "\n";
}

// GOOD: passes a const reference --- no copy, no modification
void print_album(const Album &a) {
    std::cout << a.title << " by " << a.artist << "\n";
}
```

Tip: Pass structures by `const` reference unless the function needs its own copy to modify. Copying a structure that contains strings or vectors copies all that data, which can be surprisingly slow.

Default Parameters

You can give function parameters default values. If the caller does not provide an argument for that parameter, the default is used:

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

void play(const std::string &song, int volume = 5) {
    std::cout << "Playing " << song << " at volume " << volume << "\n";
}

int main() {
    play("Return of the Mack");         // volume defaults to 5
    play("Return of the Mack", 11);     // volume is 11
    return 0;
}
```

Default parameters must appear at the end of the parameter list. You cannot put a defaulted parameter before a non-defaulted one:

```cpp
// OK
void play(const std::string &song, int volume = 5);

// ERROR: default parameter not at the end
void play(int volume = 5, const std::string &song);
```

Trap: Default values are specified in the declaration, not the definition (if they are separate). Specifying defaults in both places is an error:

```
// declaration with default
void play(const std::string &song, int volume = 5);

// definition --- no default here
void play(const std::string &song, int volume) {
    std::cout << song << " at " << volume << "\n";
}
```

Function Overloading

C++ lets you define multiple functions with the same name as long as their parameter lists differ. This is called **function overloading**:

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

void display(int value) {
    std::cout << "Integer: " << value << "\n";
}

void display(const std::string &value) {
    std::cout << "String: " << value << "\n";
}

void display(double value) {
    std::cout << "Double: " << value << "\n";
}

int main() {
    display(42);
    display("I want it that way");
    display(3.14);
    return 0;
}
```

The compiler decides which version to call based on the types of the arguments you pass. The functions must differ in the number or types of parameters — you cannot overload based on return type alone.

Recursive Functions

A function that calls itself is called a **recursive function**. Recursion is a powerful technique for problems that can be broken down into smaller versions of themselves.

The classic example is computing a factorial. The factorial of `n` (written `n!`) is `n * (n-1) * (n-2) * ... * 1`:

```
#include <iostream>

int factorial(int n) {
    if (n <= 1) {
        return 1;         // base case
    }
    return n * factorial(n - 1);  // recursive case
```

```
}

int main() {
    std::cout << "5! = " << factorial(5) << "\n";   // 120
    return 0;
}
```

Every recursive function needs two things:

1. A **base case** — a condition that stops the recursion.
2. A **recursive case** — where the function calls itself with a "smaller" problem.

Without a base case, the function calls itself forever until the program crashes with a stack overflow.

Trap: Forgetting the base case is the most common recursion mistake. Always ask yourself: "Under what condition does this function *not* call itself?" If you cannot answer that clearly, you have an infinite recursion.

Let's trace through `factorial(3)` to see how it works:

```
factorial(3)
  → 3 * factorial(2)
        → 2 * factorial(1)
              → returns 1     (base case)
        → returns 2 * 1 = 2
  → returns 3 * 2 = 6
```

Function Pointers

In C++, functions have addresses in memory just like variables do. A **function pointer** stores the address of a function so you can call it indirectly.

Here is the basic syntax:

```
#include <iostream>

int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

int main() {
    // Declare a function pointer
    int (*operation)(int, int);

    operation = add;
    std::cout << operation(10, 3) << "\n";   // 13

    operation = subtract;
    std::cout << operation(10, 3) << "\n";   // 7

    return 0;
}
```

The declaration `int (*operation)(int, int)` means: `operation` is a pointer to a function that takes two `int` parameters and returns an `int`. The parentheses around `*operation` are required — without them, you would be declaring a function that returns an `int *`.

Simplifying with `using`

The function pointer syntax is admittedly ugly. You can clean it up with a `using` alias:

```
using MathOp = int (*)(int, int);

MathOp operation = add;
std::cout << operation(10, 3) << "\n";
```

Callbacks

Function pointers are commonly used as **callbacks** — you pass a function to another function, which calls it at the right time:

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

void process_songs(const std::string songs[], int count,
                   void (*action)(const std::string &)) {
    for (int i = 0; i < count; i++) {
        action(songs[i]);
    }
}

void announce(const std::string &song) {
    std::cout << "Now playing: " << song << "\n";
}

void shout(const std::string &song) {
    std::cout << ">> " << song << "!! <<\n";
}

int main() {
    std::string playlist[] = {
        "I Want It That Way",
        "Return of the Mack",
        "Semi-Charmed Life"
    };

    std::cout << "--- Chill mode ---\n";
    process_songs(playlist, 3, announce);

    std::cout << "--- Fiesta mode ---\n";
    process_songs(playlist, 3, shout);

    return 0;
}
```

The function `process_songs` does not know or care what `action` does — it just calls it for each song. This lets you change the behavior by passing a different function, without modifying `process_songs` itself.

Tip: Function pointers are the C++ mechanism for callbacks, but modern C++ often uses lambdas and `std::function` instead. You will encounter function pointers in older code and C libraries, so it is important to understand them.

`[[nodiscard]]`

Sometimes a function's return value is the whole point of calling it — ignoring it is almost certainly a bug. The `[[nodiscard]]` attribute tells the compiler to warn the caller if they discard the return value:

```
[[nodiscard]] int find_track(const std::string &playlist);
```

If someone calls `find_track("90s Jams")` without using the result, the compiler produces a warning:

```
find_track("90s Jams");           // warning: ignoring return value of
                                   // function declared with 'nodiscard'
int pos = find_track("90s Jams"); // OK --- return value is used
```

C++20 lets you add a reason that appears in the warning:

```
[[nodiscard("track index needed for playback")]]
int find_track(const std::string &playlist);
```

Tip: Use `[[nodiscard]]` on functions where ignoring the return value is almost certainly a bug — error codes, computed results, or newly allocated resources.

Operator Functions

Whew, there is a lot to functions in C++, right? Well, there is more: operator functions. To some they are an abomination, but to others, especially those writing core C++ APIs, they are fun ways to expand the semantics of language operators in new and unimagined ways.

You have actually been using operator overloading since Chapter 1. The `<<` and `>>` operators are built into C++ for shifting the bits of an integer left and right (you saw this in Chapter 4). But visually they look like they are pushing data in a direction — and the iostream library used that intuition to overload them for stream I/O. The compiler only knows how to use `<<` and `>>` with integers. The iostream library defined operator functions that taught the compiler how to apply `<<` and `>>` to `std::ostream`, `std::istream`, and other types. Every time you write `std::cout << "hello"`, you are calling an operator function.

An **operator function** lets you do the same thing for your own types — define what an operator like `+`, `==`, or `<<` does when applied to them. You write it as a regular function named `operator` followed by the symbol.

Here is a `Score` struct with overloaded `+`, `==`, and `>`:

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

struct Score {
    std::string player;
    int points;
};

Score operator+(const Score &a, const Score &b) {
    return Score{a.player + " & " + b.player, a.points + b.points};
}

bool operator==(const Score &a, const Score &b) {
    return a.points == b.points;
}

bool operator>(const Score &a, const Score &b) {
```

```
    return a.points > b.points;
}
```

Now you can use these operators naturally:

```
int main()
{
    Score a{"Fly", 95};
    Score b{"Intergalactic", 88};

    Score combined = a + b;
    std::cout << combined.player << ": " << combined.points << std::endl;

    if (a > b) {
        std::cout << a.player << " wins" << std::endl;
    }

    return 0;
}
```

Output:

```
Fly & Intergalactic: 183
Fly wins
```

The compiler sees `a + b` and looks for an `operator+` that takes two `Score` parameters. It finds yours and calls it like any other function.

Rules

There are a few rules that the compiler enforces:

- **You cannot create new operators.** You can overload `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `<<`, `>>`, `()`, `[]`, and many others, but you cannot invent `operator@` or `operator**`.
- **You cannot change arity** (the number of operands an operator takes). Binary operators stay binary (two operands), unary operators stay unary (one operand). You cannot make `+` take three operands.
- **You cannot change precedence or associativity.** Even if you overload `+` and `*`, the compiler still evaluates `*` before `+` just like it does with built-in types.
- **At least one operand must be a user-defined type.** You cannot redefine what `+` means for two `int`s.
- **Some operators cannot be overloaded at all:** `::`, `.`, `.*`, `?:`, and `sizeof`.

The << Operator for Output

You can overload `<<` so that `std::cout` works with your types. The left operand is an `std::ostream &` and the right operand is your type:

```
std::ostream &operator<<(std::ostream &os, const Score &s) {
    os << s.player << ": " << s.points;
    return os;
}
```

Returning the `os` reference is what lets you chain calls: `std::cout << a << " vs " << b`.

```
Score a{"Fly", 95};
std::cout << a << std::endl;  // prints: Fly: 95
```

Tip: Make your operators behave the way people expect. `+` should combine things, `==` should compare them, `<<` should print them. If `a + b` deleted `b`, your coworkers would not be amused.

Trap: Do not overload `&&`, `||`, or `,`. The built-in versions of `&&` and `||` use **short-circuit evaluation** — the right side is only evaluated if the left side does not already determine the result. When you overload them, both sides are *always* evaluated because the compiler treats them as regular function calls. This can break logic that depends on short-circuiting, like `ptr != nullptr && ptr->valid()`.

In Chapter 12 you will learn how to write operator functions as **member functions** of a class, which gives them access to private data. The operator functions shown here are free functions that work with structs whose members are public.

Try It: Functions Starter

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

// Forward declaration
void play(const std::string &song, int volume = 5);

// Pass by value vs pass by reference
void double_value(int x) {
    x *= 2;   // only modifies the copy
}

void double_ref(int &x) {
    x *= 2;   // modifies the original
}

// Recursive countdown
void countdown(int n) {
    if (n <= 0) {
        std::cout << "Vamos!\n";
        return;
    }
    std::cout << n << "... ";
    countdown(n - 1);
}

// Function pointer
using Formatter = void (*)(const std::string &);

void upper_announce(const std::string &s) {
    std::cout << ">> " << s << " <<\n";
}

void quiet_announce(const std::string &s) {
    std::cout << "(" << s << ")\n";
}

// Definition of play
```

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

int main() {
    // Default parameters
    play("Semi-Charmed Life");
    play("Semi-Charmed Life", 11);

    // Pass by value vs reference
    int num = 10;
    double_value(num);
    std::cout << "After double_value: " << num << "\n";   // 10
    double_ref(num);
    std::cout << "After double_ref: " << num << "\n";     // 20

    // Recursion
    countdown(5);

    // Function pointer
    Formatter fmt = upper_announce;
    fmt("Return of the Mack");
    fmt = quiet_announce;
    fmt("Return of the Mack");

    return 0;
}
```

Key Points

- A function declaration tells the compiler about a function's signature. A definition provides the body.
- Forward declarations let you call a function before its definition appears in the file.
- The one-definition rule (ODR) requires exactly one definition of each function across all translation units. Use `inline` for functions defined in header files.
- C++ passes arguments by value by default — the function gets a copy.
- Use `&` to pass by reference when you need the function to modify the caller's variable or to avoid copying large objects.
- Mark parameters `const` when the function should not modify them. Use `const` references for large types you only need to read.
- Default parameters provide fallback values and must appear at the end of the parameter list.
- Function overloading lets you define multiple functions with the same name but different parameter lists.
- Recursive functions call themselves. Always ensure there is a base case to stop the recursion.
- Function pointers store the address of a function and enable callbacks — passing behavior as a parameter.
- Operator functions let you define what operators like `+`, `==`, and `<<` mean for your own types.
- You cannot create new operators, change arity, or change precedence.
- Do not overload `&&`, `||`, or `,` — it breaks short-circuit evaluation.
- `[[nodiscard]]` warns the caller if they ignore a function's return value.

Exercises

1. **Think about it:** Why does C++ pass arguments by value by default instead of by reference? What advantage does this give you in terms of reasoning about your code?

2. **What does this print?**

```
void mystery(int a, int &b) {
    a = a + 10;
    b = b + 10;
}

int main() {
    int x = 5, y = 5;
    mystery(x, y);
    std::cout << x << " " << y << "\n";
    return 0;
}
```

3. **Calculation:** What does `factorial(6)` return, using the recursive factorial function shown in this chapter?

4. **Where is the bug?**

```
int countdown(int n) {
    return n + countdown(n - 1);
}
```

5. **What does this print?**

```
void greet(const std::string &name) {
    std::cout << "Hola, " << name << "\n";
}

void greet(const std::string &name, int times) {
    for (int i = 0; i < times; i++) {
        std::cout << "Hola, " << name << "! ";
    }
    std::cout << "\n";
}

int main() {
    greet("Mack");
    greet("Mack", 3);
    return 0;
}
```

6. **Where is the bug?**

```
void set_volume(int volume = 5, const std::string &song) {
    std::cout << song << " at " << volume << "\n";
}
```

7. **What does this print?**

```
int apply(int (*func)(int, int), int a, int b) {
    return func(a, b);
}

int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }

int main() {
    std::cout << apply(add, 3, 4) << "\n";
```

```
    std::cout << apply(mul, 3, 4) << "\n";
    return 0;
}
```

8. **What does this print?**

```
struct Volume {
    int level;
};

Volume operator+(const Volume &a, const Volume &b) {
    return Volume{a.level + b.level};
}

bool operator>(const Volume &a, const Volume &b) {
    return a.level > b.level;
}

int main() {
    Volume a{5};
    Volume b{6};
    Volume c = a + b;
    std::cout << c.level << std::endl;
    std::cout << (a > b) << std::endl;
    return 0;
}
```

9. **Think about it:** Why should you not overload `&&` and `||`? What behavior do the built-in versions have that overloaded versions lose?

10. **Write a program** that defines a function `is_even` that returns `true` if a number is even and `false` otherwise. Write a second function `count_if` that takes an array of integers, its size, and a function pointer to a predicate (a function that takes an `int` and returns `bool`). `count_if` should return how many elements satisfy the predicate. Test it by counting the even numbers in an array.

11. **Where is the bug?** A coworker has a `helpers.h` header file with the following function definition. Two `.cpp` files both `#include "helpers.h"`. The program compiles, but the linker reports a "multiple definition" error. What is the fix?

```
// helpers.h
int double_it(int n) {
    return n * 2;
}
```

12. **What does the compiler do** with the following code?

```
[[nodiscard]] int compute(int a, int b) {
    return a * b;
}

int main() {
    compute(6, 7);
    return 0;
}
```

13. **Think about it / where is the bug?** Two functions take the same `Album` struct, which has a `std::string title`, a `std::string artist`, and an `int year`. Both functions only need to *read* the album.

```cpp
void print_album(Album a) {
    std::cout << a.title << " by " << a.artist << "\n";
}

void print_album_ref(const Album &a) {
    std::cout << a.title << " by " << a.artist << "\n";
}
```

Both compile and produce the same output. Why is the second version preferred for a struct like `Album`? Would the same answer apply to a function that takes a single `int`? Why or why not?