Gorgo C for C++ Programmers

April 11, 2026

7. Functions

C functions look a lot like C++ functions — same return types, same curly braces, same `return` statement. But several features you rely on in C++ are simply not available in C. There is no function overloading, no default arguments, no references, and no `auto` return type deduction. Every parameter is pass by value. If you want a function to modify a caller's variable, you pass a pointer.

Despite these restrictions, C functions are straightforward. Once you understand the handful of differences from C++, you will find them easy to work with.

Function Declarations and Definitions

In C++, you can often define a function before it is called and skip the separate declaration. In C, you should always declare a function (provide its **prototype**) before you call it. A prototype tells the compiler the function's return type, name, and parameter types — without the body:

```
int add(int a, int b);        // declaration (prototype)
```

The **definition** provides the body:

```
int add(int a, int b) {       // definition
    return a + b;
}
```

Prototypes typically go in header files (`.h`) so that other `.c` files can call the function. The definition lives in exactly one `.c` file.

Tip: In C++, `int foo()` means "takes no parameters." In C, `int foo()` means "takes an *unspecified* number of parameters" — the compiler will not check your arguments at all. To declare a function that truly takes no parameters in C, write `int foo(void)`. Always use `void` in empty parameter lists.

Here is the difference in action:

```
int get_score(void);    // takes no parameters — compiler enforces this
int get_score();        // unspecified parameters — compiler won't check
```

You will see `int main(void)` throughout this book for exactly this reason.

No Overloading

In C++, you can have multiple functions with the same name but different parameter types:

```
int max(int a, int b);
double max(double a, double b);   // C++ overload — fine
```

C does not support this. Every function must have a unique name. If you need a `max` for both `int` and `double`, you name them differently:

```
int max_int(int a, int b);
double max_double(double a, double b);
```

No Default Arguments

C++ lets you provide default values for parameters:

```
void greet(const char *name, int times = 1);   // C++ — valid
```

C does not support default arguments. Every argument must be provided at every call site:

```
void greet(const char *name, int times);   // C — no defaults allowed

greet("Iron Man", 3);   // must pass both arguments
```

Pass by Value

As you saw in the Pointers chapter, all function parameters in C are pass by value — the function receives a copy of each argument. To modify a caller's variable, pass a pointer. There are no `&` reference parameters in C; every "output" parameter is a pointer.

`const` Parameters

When a function takes a pointer parameter, the caller cannot tell from the call site whether the function will modify the data. The `const` keyword solves this by making a promise: "I will not modify what this pointer points to."

```
#include <stdio.h>

void print_name(const char *name) {
    printf("I am %s\n", name);
    // name[0] = 'X';   // ERROR — name points to const data
}

int main(void) {
    print_name("Iron Man");
```

```
    return 0;
}
```

Using `const` serves two purposes. First, it documents intent — anyone reading the prototype knows the function will not modify the data. Second, it catches bugs at compile time. If you accidentally try to write through a `const` pointer, the compiler will stop you.

Tip: Make every pointer parameter `const` unless the function genuinely needs to modify the pointed-to data. This is one of the cheapest and most effective ways to prevent bugs in C.

You will see `const char *` everywhere in C — it is the standard way to accept a string that the function will read but not modify. Functions like `printf`, `strlen`, and `strcmp` all take `const char *` parameters.

Passing Structures

Structures can be passed by value just like any other type. The function receives a complete copy of the struct:

```
#include <stdio.h>

struct hero {
    char name[40];
    int power;
};

void print_hero(struct hero h) {
    printf("%s (power: %d)\n", h.name, h.power);
}

int main(void) {
    struct hero tony = {"Iron Man", 100};
    print_hero(tony);   // passes a COPY of tony
    return 0;
}
```

This works, but copying a large struct every time you call a function is wasteful. If `struct hero` had thousands of bytes of data, every call to `print_hero` would copy all of it onto the stack.

The solution is to pass a pointer to the struct instead. Since the function only needs to read the data, use `const`:

```
void print_hero(const struct hero *h) {
    printf("%s (power: %d)\n", h->name, h->power);
}

int main(void) {
    struct hero tony = {"Iron Man", 100};
    print_hero(&tony);   // passes only an 8-byte pointer
    return 0;
}
```

Tip: For small structs (a few bytes), passing by value is fine and sometimes clearer. For anything larger, prefer `const struct type *param`. When the function needs to modify the struct, drop the `const`.

If the function needs to modify the struct, you pass a non-`const` pointer:

```c
void level_up(struct hero *h) {
    h->power += 10;
}
```

This pattern — `const` pointer for read-only access, non-`const` pointer for modification — is the C equivalent of const references and non-const references in C++.

Recursive Functions

C supports recursion just like C++. A function can call itself, and each call gets its own set of local variables on the stack.

Factorial

```c
#include <stdio.h>

long factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}

int main(void) {
    printf("5! = %ld\n", factorial(5));    // 5! = 120
    printf("10! = %ld\n", factorial(10));  // 10! = 3628800
    return 0;
}
```

Fibonacci

```c
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(void) {
    for (int i = 0; i < 10; i++) {
        printf("%d ", fibonacci(i));
    }
    printf("\n");
    // 0 1 1 2 3 5 8 13 21 34
    return 0;
}
```

Trap: This Fibonacci implementation has exponential time complexity because it recomputes the same values over and over. It works for small inputs, but try `fibonacci(50)` and you will be waiting a long time. In practice, you would use an iterative approach or memoization.

Stack Depth

Every function call pushes a new frame onto the call stack. Recursive functions can exhaust the stack if the recursion goes too deep. There is no built-in protection — C has no exception mechanism, so there is no stack

overflow exception to catch. The program will simply crash (usually with a segmentation fault). Keep your recursion depth reasonable, or convert deep recursion to iteration.

Function Pointers

In C++, you might use `std::function`, lambdas, or template parameters to pass behavior around. C has none of those — but it does have **function pointers**. A function pointer is a variable that holds the address of a function.

Declaring Function Pointers

The syntax takes some getting used to. A pointer to a function that takes two `int` parameters and returns an `int` looks like this:

```
int (*fp)(int, int);
```

Read it from the inside out: `fp` is a pointer (`*fp`) to a function that takes `(int, int)` and returns `int`.

Wut: The parentheses around `*fp` are critical. Without them, `int *fp(int, int)` declares a *function* that returns an `int *` — a completely different thing. The parentheses force `fp` to be a pointer first.

Using Function Pointers

You can assign a function's name to a function pointer (the function name decays to a pointer, just like array names do) and then call the function through the pointer:

```
#include <stdio.h>

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

int main(void) {
    int (*op)(int, int);

    op = add;
    printf("3 + 4 = %d\n", op(3, 4));       // 7

    op = multiply;
    printf("3 * 4 = %d\n", op(3, 4));       // 12

    return 0;
}
```

Simplifying with `typedef`

The raw function pointer syntax is noisy. A `typedef` gives it a clean name:

```
#include <stdio.h>

typedef int (*binop_fn)(int, int);

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

void apply(binop_fn fn, int x, int y) {
```

```
    printf("result = %d\n", fn(x, y));
}

int main(void) {
    apply(add, 10, 3);        // result = 13
    apply(subtract, 10, 3);   // result = 7
    return 0;
}
```

Now `binop_fn` is a type that means "pointer to a function taking two `int` parameters and returning `int`." You can use it for parameters, local variables, struct members, and arrays of function pointers.

Callbacks

Function pointers are often used as **callbacks** — you pass a function pointer to another function, which calls it at the right moment. This is the C equivalent of passing a lambda or functor in C++. The Odds and Ends chapter shows a practical example using `qsort`, the standard library's sorting function that takes a comparison callback.

Try It: Functions Starter

```
#include <stdio.h>

// Prototype with const pointer parameter
void shout(const char *msg);

// Struct and a function that takes a const pointer to it
struct song {
    char title[50];
    int year;
};

void print_song(const struct song *s) {
    printf("\"%s\" (%d)\n", s->title, s->year);
}

// Recursive factorial
long factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

// Function pointer demo
int sumar(int a, int b) { return a + b; }
int restar(int a, int b) { return a - b; }

typedef int (*math_fn)(int, int);

void compute(math_fn fn, int x, int y) {
    printf("  result = %d\n", fn(x, y));
}

// shout definition
void shout(const char *msg) {
    printf(">>> %s <<<\n", msg);
```

```c
}

int main(void) {
    // const parameter
    shout("I Am Iron Man");

    // Passing a struct by pointer
    struct song fav = {"Iron Man", 1970};
    print_song(&fav);

    // Recursion
    printf("7! = %ld\n", factorial(7));   // 5040

    // Function pointers
    printf("sumar:\n");
    compute(sumar, 8, 3);     // result = 11
    printf("restar:\n");
    compute(restar, 8, 3);    // result = 5

    return 0;
}
```

Key Points

- C functions have no overloading, no default arguments, and no references. Every parameter is pass by value.
- Use `void` in empty parameter lists: `int foo(void)`. In C, `int foo()` means "unspecified parameters," not "no parameters."
- Use `const` on pointer parameters when the function does not modify the pointed-to data. It documents intent and catches bugs.
- For large structs, prefer `const struct type *param` over pass by value to avoid expensive copies.
- Function pointers let you store and pass functions as values — C's replacement for lambdas and `std::function`.
- Use `typedef` to make function pointer types readable.

Exercises

1. **Think about it:** C does not have function overloading. How does the C standard library handle providing similar functionality for different types? Look at `abs` (for `int`) and `fabs` (for `double`) as examples. What naming convention do you see?
2. **What does this print?**

```c
void mystery(int x) {
    x = x * 2;
    printf("inside: %d\n", x);
}

int main(void) {
    int val = 5;
    mystery(val);
    printf("outside: %d\n", val);
    return 0;
}
```

3. **Where is the bug?**

```c
int count_chars(const char *s) {
    int count;
    while (*s != '\0') {
        count++;
        s++;
    }
    return count;
}
```

4. **Calculation:** Given the struct below, approximately how many bytes are copied each time `process_data` is called with pass by value? Assume `int` is 4 bytes.

```c
struct data {
    int values[1000];
    int count;
};

void process_data(struct data d) { /* ... */ }
```

5. **What does this print?**

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

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

int main(void) {
    printf("%d\n", apply(mul, 6, 7));
    return 0;
}
```

6. **Where is the bug?**

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(void) {
    int x = 10, y = 20;
    swap(x, y);
    printf("x=%d y=%d\n", x, y);
    return 0;
}
```

7. **Write a program** that defines a function `void transform(int *arr, int n, int (*fn)(int))` which applies the function `fn` to each element of `arr`, modifying the array in place. Test it with a function that doubles each element and another that negates each element.