C for C++ Programmers

February 28, 2026

0. How to read this booklet

This is a short booklet to help a C++ programmer get up to speed with C. As a side benefit, you will probably find that you understand C++ better at the end of it.

Each chapter is meant to help you understand a topic, but you will still want to reference API descriptions for more specifics of the parameters and operating conditions. Hopefully, you'll have the context you need to understand API documentation.

Tips

Tips callout details that you need to pay special attention to. **Traps** warn you of common mistakes made. **Wut** calls out a detail that is counter-intuitive, so make sure you pay attention.

Try It

As the intro to the most amazing programming language book ever written starts out

> The only way to learn a new programming language is by writing programs in it.

You need to write some code. Make sure you try writing some programs from scratch. At the end of most sections is a starter program that you can type in and modify to play with. Don't use it as an excuse to avoid writing some of your own starter programs. It's the only way to master a language.

Exercises

Don't skip the exercises at the end of the chapters. You can get the answer key, but don't look at the answer key before you find the answer yourself. If you look at the answer key first, the concepts will not sink in.

1. Introduction

In the beginning, knowing C++ automatically meant you knew C. The original C++ compiler, `cfront`, literally translated C++ code into C before compiling it. But modern C++ has diverged dramatically from C. You have `std::string`, `std::vector`, smart pointers, RAII, templates, `iostream`, and exceptions — none of which exist in C. If someone hands you a C codebase today, your modern C++ instincts will not get you very far.

So why learn C? Because C is everywhere. Operating systems, embedded firmware, database engines, language runtimes — the foundational software that the world runs on is written in C. Even if you spend most of your career in C++, you will encounter C code, C libraries, and C APIs. Understanding C makes you a better programmer, period.

My go-to C textbook is *The C Programming Language (Second Edition)* by Brian Kernighan and Dennis Ritchie — often called "K&R." It is one of the most influential programming books ever written, and it opens like this:

> ### Getting Started
>
> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages: Print the words hello, world. This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.
>
> In C, the program to print "hello, world" is
>
> ```c
> #include <stdio.h>
> main()
> {
>     printf("hello, world\n");
> }
> ```
>
> Just how to run this program depends on the system you are using. As a specific example, on the UNIX operating system you must create the program in a file whose name ends in ".c", such as hello.c, then compile it with the command `cc hello.c`. If you haven't botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called a.out. If you run a.out by typing the command `a.out`, it will print `hello, world`. On other systems, the rules will be different; check with a local expert.

Differences Summary

Notice the differences from C++. There is no `#include <iostream>`, no `std::println`, no `std::cout`. In C, you use `printf` from `<stdio.h>` for output. The file ends in `.c`, not `.cpp`. You compile with `cc` (the C compiler) rather than `c++`.

Here is a quick summary of the biggest differences you will encounter:

C++	C
`std::string`	`char` arrays with `'\0'`
`std::vector`	raw arrays or `malloc`
`std::cout` / `std::println`	`printf`
`std::cin`	`scanf`
`new` / `delete`	`malloc` / `free`
Smart pointers	Raw pointers (only option)
Classes and objects	Structs and functions
References (`&`)	Pointers (`*`)
`bool` (built-in)	`_Bool` or `#include <stdbool.h>`
`// comments`	`/* comments */` (C89); `//` allowed since C99

Tip: C source files use the `.c` extension and are compiled with `cc`. If you accidentally compile a `.c` file with `c++`, it will be treated as C++ and may accept syntax that real C compilers reject. Always use `cc` when writing C.

Printing with `printf`

```
int printf(const char *format, ...);
```

In C++, you use `std::cout` or `std::println` for output. In C, you use `printf` from `<stdio.h>`. Unlike `std::println`, `printf` does not automatically add a newline at the end of the output. If you want each call to end on its own line, you must include `\n` in the format string yourself.

The first argument to `printf` is a **format string** containing literal text and **format specifiers** that start with `%`. Each specifier is replaced by the corresponding argument that follows:

```
int year = 1984;
printf("Year: %d\n", year);   // Year: 1984
```

Here are the format specifiers you will use most:

Specifier	Type	Example	Output
`%d`	`int` (decimal)	`printf("%d", 42)`	`42`
`%x`	`int` (hex, lowercase)	`printf("%x", 255)`	`ff`

Specifier	Type	Example	Output
`%X`	`int` (hex, uppercase)	`printf("%X", 255)`	`FF`
`%f`	`double`	`printf("%f", 3.14)`	`3.140000`
`%e`	`double` (scientific)	`printf("%e", 3.14)`	`3.140000e+00`
`%c`	`char`	`printf("%c", 'A')`	`A`
`%s`	`char *` (string)	`printf("%s", "hola")`	`hola`
`%p`	pointer	`printf("%p", ptr)`	`0x7ffd...`
`%zu`	`size_t`	`printf("%zu", sizeof(int))`	`4`

Trap: The first argument to `printf` must always be a string literal. Never pass a variable as the first argument. It may work, but it is a potential security vulnerability (format string attack). If you only want to print a string variable, don't do ~~`printf(str)`~~, do `printf("%s", str)`.

You can control the width and precision of output by placing numbers between the `%` and the specifier letter. A number before the specifier sets the minimum field width, and a `.` followed by a number sets the precision (decimal places for floats, max characters for strings):

```c
double score = 98.6;
printf("Score: %f\n", score);      // 98.600000  (default: 6 decimal places)
printf("Score: %.2f\n", score);    // 98.60      (2 decimal places)
printf("Score: %e\n", score);      // 9.860000e+01 (scientific notation)
```

Zero-filled output is useful for track numbers, timestamps, and hex addresses. Place a `0` before the width to pad with zeros instead of spaces:

```c
for (int i = 1; i <= 5; i++) {
    printf("Track %02d\n", i);
}
// Track 01
// Track 02
// Track 03
// Track 04
// Track 05

int color = 0xFF8800;
printf("Color: 0x%06X\n", color);   // Color: 0xFF8800

int score = 95;
printf("Score: %d%%\n", score);      // Score: 95%
```

Since `%` introduces a format specifier, you must write `%%` to print a literal percent sign.

Trap: `printf` does not check that your format specifiers match the types of your arguments. If you write `printf("%d", 3.14)`, the compiler may warn you, but it will not stop you. The result is garbage. Always match specifiers to types: `%d` for `int`, `%f` for `double`, `%s` for `char *`, and so on.

Key Points

- C uses `printf` from `<stdio.h>` for output — there is no `std::cout` or `std::println`.
- Format specifiers (`%d`, `%s`, `%f`, etc.) must match the types of the arguments passed to `printf`.
- C source files end in `.c` and are compiled with `cc`, not `c++`.
- C has no classes, no templates, no exceptions, no smart pointers, and no `std::string`.
- `printf` does not add a newline automatically — you must include `\n` yourself.

Exercises

1. **Think about it:** C uses format specifiers in `printf` while C++ uses `operator<<` or `std::format`. What advantage does the format string approach give you when writing output to a log file? What is a disadvantage?
2. **What does this print?**

   ```
   printf("%05d %x\n", 42, 255);
   ```

3. **Calculation:** How many bytes does the string literal `"hello"` occupy in memory?
4. **Where is the bug?**

   ```
   double pi = 3.14159;
   printf("Pi is %d\n", pi);
   ```

5. **Write a program** that prints a 5x5 multiplication table using `printf` with width formatting so the columns are aligned.

2. Pointers

If you have been writing modern C++, you may have rarely (or never) used raw pointers. Smart pointers like `std::unique_ptr` and `std::shared_ptr` manage memory for you. References let you pass objects without copying them. The standard library hides pointer details behind iterators and containers.

In C, none of that exists. Pointers are everywhere, and you must be comfortable with them. Every dynamic data structure, every function that needs to modify its arguments, every interaction with the operating system — all involve pointers.

What Is a Pointer?

A pointer is a variable that holds a memory address. That's it. Instead of holding a value like `42`, a pointer holds the *location* where `42` is stored.

Declaring Pointers

A pointer type is declared by placing a `*` after the base type. The type before the `*` tells you what kind of data lives at the address the pointer holds:

```
int *p;          // p is a pointer to an int
char *s;         // s is a pointer to a char
double *d;       // d is a pointer to a double
```

> **Trap:** The `*` belongs to the variable, not the type. This declaration creates one pointer and one regular int:
>
> ```
> int *p, q;    // p is a pointer to int; q is just an int
> ```
>
> To declare two pointers, you need two stars: `int *p, *q;`

The Address-Of Operator: `&`

The `&` operator returns the address of a variable. You have seen `&` in C++ for references — in C, it is strictly the address-of operator:

```
int score = 100;
int *p = &score;   // p now holds the address of score

printf("score = %d\n", score);    // 100
printf("address of score = %p\n", (void *)p);  // something like 0x7ffd5e8a3b2c
```

Dereferencing: `*`

The `*` operator on a pointer gives you the value at the address the pointer holds. This is called **dereferencing**:

```
int score = 100;
int *p = &score;
```

```c
printf("Value at p: %d\n", *p);   // 100

*p = 200;                          // modify score through the pointer
printf("score is now: %d\n", score);  // 200
```

Notice the dual use of `*`: in a declaration, it means "this is a pointer." In an expression, it means "follow the pointer to the value."

Pointers to Pointers

Since a pointer is just a variable, it has an address too. You can create a pointer to a pointer:

```c
int val = 42;
int *p = &val;       // p points to val
int **pp = &p;       // pp points to p

printf("val = %d\n", val);       // 42
printf("*p = %d\n", *p);         // 42
printf("**pp = %d\n", **pp);     // 42
```

You dereference `pp` twice: once to get `p`, and again to get `val`. Pointers to pointers show up frequently in C — for example, `main` can be declared as `int main(int argc, char **argv)`, where `argv` is a pointer to an array of string pointers.

Visualizing Pointers in Memory

Consider this small program:

```c
int x = 1985;
int y = 80;
int *p = &x;
int **pp = &p;
```

Every variable lives at some address in memory. Here is what the layout looks like (using made-up but realistic addresses):

```
Variable    Address      Value
--------    ----------   ----------
x           0x1000       1985
y           0x1004       80
p           0x1008       0x1000  -------> x
pp          0x1010       0x1008  -------> p -------> x
```

The variable `x` lives at address `0x1000` and holds the value `1985`. The pointer `p` lives at address `0x1008` and holds the value `0x1000` — the address of `x`. The pointer-to-pointer `pp` lives at `0x1010` and holds `0x1008` — the address of `p`.

Following the chain: `*pp` gives you `p` (which is `0x1000`), and `**pp` gives you `x` (which is `1985`).

Notice that `p` and `pp` are just variables that hold numbers. Those numbers happen to be memory addresses. There is nothing magical about a pointer — it is just a variable whose value is an address.

> **Tip:** You can take the address of any variable with `&`, including the address of a pointer variable. The expression `&p` gives you the address where `p` itself is stored, not the address `p` points to.

NULL Pointers

A pointer that does not point to anything should be set to `NULL`:

```c
int *p = NULL;   // p points to nothing

if (p != NULL) {
    printf("Value: %d\n", *p);
} else {
    printf("Pointer is NULL\n");
}
```

Dereferencing a `NULL` pointer is undefined behavior and usually crashes your program with a segmentation fault. Always check before dereferencing a pointer you did not initialize yourself.

> **Tip:** In C, `NULL` is typically defined as `((void *)0)`. You may also see `0` used directly. C++11 introduced `nullptr` as a type-safe null pointer — C does not have `nullptr`, so use `NULL`.

Pointers and Arrays

A pointer might point to a single value in memory, or it might point to the first element of an array of values. There is nothing in the type system that tells you which — an `int *` looks the same either way:

```c
int score = 100;
int *p = &score;              // points to one int

int nums[] = {10, 20, 30};
int *q = nums;                // points to the first of three ints
```

Both `p` and `q` are `int *`. The compiler does not know whether there are more `int` values after the one being pointed to. It is up to you, the programmer, to keep track of how many elements a pointer refers to and to stay within bounds.

In C, arrays and pointers are intimately connected. When you use an array name in most expressions, it **decays** (that is the fancy technical term to mean it gets treated as) to a pointer to the first element:

```
int nums[] = {10, 20, 30, 40, 50};
int *p = nums;       // p points to nums[0]; no & needed

printf("%d\n", *p);       // 10 (same as nums[0])
printf("%d\n", *(p + 1)); // 20 (same as nums[1])
printf("%d\n", p[2]);     // 30 - yes, you can use [] on pointers!
```

Another way to see the decay in action is to notice that the expressions `nums` and `&nums` produce the same address.

```
int nums[] = {10, 20, 30, 40, 50};
printf("%p %p\n", nums, &nums);  // the same address is printed twice
```

Pointer arithmetic works in units of the pointed-to type. If `p` is an `int *` and `int` is 4 bytes, then `p + 1` advances the address by 4 bytes to the next `int`. You never have to think about byte sizes — the compiler handles it.

```
int nums[] = {10, 20, 30};
int *p = nums;

for (int i = 0; i < 3; i++) {
    printf("nums[%d] = %d\n", i, *(p + i));
}
```

> **Tip:** Array indexing `nums[i]` is actually syntactic sugar for `*(nums + i)`. This is why `2[nums]` technically works — it is `*(2 + nums)`, which is the same thing. Don't write code like that, but knowing this helps you understand how arrays and pointers relate.

Pointers and Structures

In C, you use `struct` to group related data — much like a class with only public data members in C++. Pointers to structures are extremely common; almost any non-trivial C program passes struct pointers around.

```
struct song {
    char title[40];
    int year;
};

struct song track = {"Karma Chameleon", 1983};
struct song *p = &track;
```

When a structure is stored in memory, all of its members will be stored in the same chunk of memory. For example, if you look in the memory where `track`

is stored, you will see 40 bytes reserved for the title and four bytes (on most systems) reserved for the year. The layout can include padding to make memory access more efficient. For example, if title was only 39 bytes there may still be a byte between the `title` and `year` to make `year`'s memory address 4-byte aligned.

You can access members of a structure directly using the `.` operator:

```
printf("Title: %s\n", track.title);
printf("Year: %d\n", track.year);
```

To access a field through a pointer, you must dereference the pointer first. But the `.` operator has higher precedence than `*`, so you need parentheses:

```
printf("Title: %s\n", (*p).title);    // parentheses required
printf("Year: %d\n", (*p).year);
```

Writing `(*p).field` everywhere is tedious. C provides the `->` operator as a convenient shorthand:

```
printf("Title: %s\n", p->title);      // same as (*p).title
printf("Year: %d\n", p->year);        // same as (*p).year
```

> **Tip:** The `->` operator is simply `(*p).field` written more clearly. You will see `->` far more often than `(*p).` in real C code. If you have a pointer to a struct, reach for `->`.

Pass by Value (and Pointers as a Workaround)

In C++, you can pass arguments by reference using `&`:

```
void increment(int &x) { x++; }   // C++ - modifies the original
```

C does not have references. **All function parameters in C are pass by value** — the function receives a copy of the argument, not the original. If you want a function to modify a variable in the caller, you pass a *pointer* to it:

```
void increment(int *x) {
    (*x)++;   // dereference the pointer, then increment
}

int main(void) {
    int score = 99;
    increment(&score);      // pass the ADDRESS of score
    printf("%d\n", score);  // 100
    return 0;
}
```

The function `increment` receives a *copy* of the pointer (the address), but since both the original and the copy point to the same memory, dereferencing either one reaches the same variable. This is how C simulates pass by reference.

Tip: Every time you see a function parameter with `*` in C, ask yourself: "Is this pointer here so the function can modify the caller's variable, or because it needs to access a block of memory (like an array)?" Often it is both.

Try It: Pointer Starter

This program demonstrates the core pointer operations:

```c
#include <stdio.h>

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

int main(void) {
    // Basic pointer usage
    int val = 1985;
    int *p = &val;
    printf("val = %d, *p = %d\n", val, *p);
    printf("Address of val: %p\n", (void *)&val);

    // Modify through pointer
    *p = 1989;
    printf("After *p = 1989: val = %d\n", val);

    // Pointer to pointer
    int **pp = &p;
    printf("**pp = %d\n", **pp);

    // Pass by value with pointers
    int x = 10, y = 20;
    printf("Before swap: x=%d, y=%d\n", x, y);
    swap(&x, &y);
    printf("After swap:  x=%d, y=%d\n", x, y);

    // Arrays and pointers
    char *words[] = {"Totally", "Radical", "Tubular"};
    for (int i = 0; i < 3; i++) {
        printf("words[%d] = %s\n", i, words[i]);
    }

    return 0;
}
```

Key Points

- A pointer holds a memory address. Use `&` to get an address and `*` to dereference it.
- All pointers are the same size on a given system, regardless of the type they point to.
- Arrays decay to pointers in most expressions. `a[i]` is equivalent to `*(a + i)`.
- Pointer arithmetic moves in units of the pointed-to type, not bytes.
- Use `->` to access struct fields through a pointer. It is shorthand for `(*p).field`.
- All function parameters in C are pass by value. Pass a pointer to modify the caller's variable.

Exercises

1. **Think about it:** In C++, you can pass by reference to modify a caller's variable. Why do you think C was designed with only pass by value? What does this simplify in the language?

2. **What does this print?**

```c
int a[] = {10, 20, 30, 40, 50};
int *p = a + 2;
printf("%d %d %d\n", *p, *(p - 1), p[1]);
```

3. **Calculation:** On a 64-bit system, what is `sizeof(int *)`, `sizeof(char *)`, and `sizeof(double *)`?

4. **Where is the bug?**

```c
int *get_value(void) {
    int result = 42;
    return &result;
}
```

5. **What does this print?**

```c
int x = 10;
int *p = &x;
int **pp = &p;
**pp = 20;
printf("%d\n", x);
```

6. **Where is the bug?**

```c
struct song {
    char title[40];
    int year;
};
```

```c
struct song *p = NULL;
printf("%s\n", p->title);
```

7. **Write a program** that declares an array of 5 integers, uses a pointer to iterate through the array, and prints each element along with its memory address.

3. Allocating Memory

Every variable in your program lives somewhere in memory, but not all memory is created equal. Understanding where variables live — and how long they last — is essential for writing correct C programs.

Global Variables

A **global variable** is declared outside of any function. It is created when the program starts and exists until the program exits:

```c
#include <stdio.h>

int high_score = 0;   // global - lives for the entire program

void update_score(int points) {
    if (points > high_score) {
        high_score = points;
    }
}

int main(void) {
    update_score(1000);
    update_score(500);
    printf("High score: %d\n", high_score);  // 1000
    return 0;
}
```

Global variables are visible to every function in the file. They are convenient but can make programs harder to reason about, since any function can change them.

> **Tip:** Use global variables sparingly. When every function can read and write the same variable, bugs become harder to track down. Prefer passing data through function parameters.

Local Variables

A **local variable** is declared inside a function (or block). It is created when the function is called and destroyed when the function returns:

```c
void greet(void) {
    char message[] = "Hola, amigo";  // local - exists only during greet()
    printf("%s\n", message);
}
// message is gone once greet() returns
```

Local variables live on the **stack** — a region of memory that grows and shrinks automatically as functions are called and return. You do not need to free stack

memory; it is reclaimed automatically.

Trap: Never return a pointer to a local variable. The memory is freed when the function returns, and the pointer becomes a **dangling pointer** — it points to memory that no longer belongs to you:

```
int *bad(void) {
    int x = 42;
    return &x;   // BUG: x is destroyed when bad() returns
}
```

Static Local Variables

A **static local variable** has the scope of a local variable but the lifetime of a global. It is declared inside a function with the `static` keyword, created once when the program starts, and retains its value between calls:

```
#include <stdio.h>

void count_calls(void) {
    static int count = 0;   // initialized once, persists between calls
    count++;
    printf("Called %d time(s)\n", count);
}

int main(void) {
    count_calls();   // Called 1 time(s)
    count_calls();   // Called 2 time(s)
    count_calls();   // Called 3 time(s)
    return 0;
}
```

Without `static`, `count` would be reset to 0 on every call. With `static`, it lives in the data segment (like a global) but is only accessible inside `count_calls`.

Dynamic Allocation: `malloc` and `free`

```
void *malloc(size_t size);
void free(void *ptr);
```

Sometimes you need memory that outlives the function that created it, or memory whose size you do not know at compile time. For this, C provides `malloc` and `free` from `<stdlib.h>`.

`malloc` allocates a block of memory on the **heap** and returns a pointer to it. The heap is a region of memory that persists until you explicitly release it:

```
#include <stdio.h>
#include <stdlib.h>
```

```c
int main(void) {
    int *nums = malloc(5 * sizeof(int));
    if (nums == NULL) {
        printf("Allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        nums[i] = (i + 1) * 10;
    }

    for (int i = 0; i < 5; i++) {
        printf("nums[%d] = %d\n", i, nums[i]);
    }

    free(nums);   // release the memory back to the system
    return 0;
}
```

`malloc` returns a `void *` — a generic pointer that can be assigned to any pointer type without a cast in C. It returns `NULL` if the allocation fails. Always check for `NULL` after calling `malloc`. Having said that, there is a school of thought held by some very good engineers that `NULL` checks just clutter the code. To handle `NULL` gracefully, there will be a check and logic every place where it could be `NULL`, so you end up with checks and logic that are rarely used and almost never tested. If you are out of memory, you'll probably need to shut down the program, so the reasoning goes that if you try to use a NULL pointer the CPU will do the check for you :). For safety-critical systems, the above argument does not hold. Although, often those systems forbid the use of dynamically allocated memory.

> **Tip:** There are no smart pointers in C. There is no RAII. There is no garbage collector. If you call `malloc`, you *must* call `free` when you are done. If you forget, you leak memory. If you call `free` twice on the same pointer, you get undefined behavior. If you use a pointer after freeing it, you get undefined behavior. Memory management in C is entirely your responsibility.

`calloc` is a variant that allocates memory and initializes it to zero:

```c
void *calloc(size_t count, size_t size);
```

```c
int *nums = calloc(5, sizeof(int));  // 5 ints, all initialized to 0
```

And `realloc` lets you resize a previously allocated block:

```c
void *realloc(void *ptr, size_t size);
```

```c
nums = realloc(nums, 10 * sizeof(int));  // grow to 10 ints
```

Working with Raw Memory: `memcpy` and `memset`

Two functions from `<string.h>` operate on raw bytes rather than strings. You will see them constantly in C code:

`memset` fills a block of memory with a byte value. It is commonly used to zero out a buffer:

```
void *memset(void *s, int c, size_t n);
```

```
int nums[10];
memset(nums, 0, sizeof(nums));   // set all bytes to 0
```

`memcpy` copies a block of bytes from one location to another. Unlike `strcpy`, it does not stop at a `'\0'` — you tell it exactly how many bytes to copy:

```
void *memcpy(void *dest, const void *src, size_t n);
```

```
int src[] = {10, 20, 30};
int dest[3];
memcpy(dest, src, sizeof(src));  // copy all 12 bytes (3 ints × 4 bytes)
```

Trap: `memcpy` requires that the source and destination do not overlap. If they might overlap (e.g., shifting elements within the same array), use `memmove` instead, which handles overlapping regions correctly.

```
void *memmove(void *dest, const void *src, size_t n);
```

Where Variables Live: A Summary

Kind	Where	Lifetime	Example
Global	Data segment	Entire program	`int count = 0;` (outside functions)
Local	Stack	Until function returns	`int x = 5;` (inside a function)
Static local	Data segment	Entire program	`static int n = 0;` (inside a function)
Dynamic	Heap	Until you call `free`	`int *p = malloc(...)`

Try It: Memory Lifetimes

```
#include <stdio.h>
#include <stdlib.h>

int total = 0;   // global - lives for the whole program
```

```c
void add_to_total(int n) {
    int local = n;       // local - gone when add_to_total returns
    total += local;
}

int main(void) {
    add_to_total(10);
    add_to_total(20);
    printf("Total: %d\n", total);    // 30

    // dynamic - lives until we free it
    int *data = malloc(3 * sizeof(int));
    if (data == NULL) return 1;

    data[0] = 1985;
    data[1] = 1986;
    data[2] = 1987;

    for (int i = 0; i < 3; i++) {
        printf("data[%d] = %d\n", i, data[i]);
    }

    free(data);
    return 0;
}
```

Key Points

- Global variables live for the entire program; local variables live only until the function returns.
- Static local variables have the scope of a local but the lifetime of a global.
- `malloc` allocates memory on the heap. You must call `free` when done.
- `calloc` allocates and zeroes memory. `realloc` resizes an allocation.
- `memcpy` copies bytes between non-overlapping regions. Use `memmove` for overlapping regions.
- `memset` fills a block of memory with a byte value.

Exercises

1. **Think about it:** Why would you choose `calloc` over `malloc` followed by `memset` to zero?
2. **What does this print?**

   ```c
   #include <stdio.h>
   ```

```c
void counter(void) {
    static int n = 0;
    n++;
    printf("%d ", n);
}

int main(void) {
    counter(); counter(); counter();
    return 0;
}
```

3. **Calculation:** On a system where `int` is 32 bits, how many bytes does `malloc(5 * sizeof(int))` allocate?

4. **Where is the bug?**

```c
int *p = malloc(10 * sizeof(int));
for (int i = 0; i < 10; i++) {
    p[i] = i;
}
free(p);
printf("%d\n", p[0]);
```

5. **Where is the bug?**

```c
int *a = malloc(5 * sizeof(int));
int *b = a;
free(a);
free(b);
```

6. **Write a program** that uses `malloc` to allocate an array of `n` integers (where `n` is provided by the user via `scanf`), fills the array with squares (0, 1, 4, 9, ...), prints them, and frees the memory.

4. Strings

In C++, you use `std::string` and barely think about what is happening under the hood. In C, there is no string type at all. A "string" in C is just an array of `char` that ends with a null character `'\0'`. Every string function in C depends on finding that null terminator to know where the string ends.

Declaring C Strings

There are several ways to create a string in C:

```c
char greeting[] = "Hola, mundo";        // compiler sizes it: 12 bytes (11 + '\0')
char band[20] = "Depeche Mode";         // 20-byte buffer, only 13 used
char empty[10] = "";                     // 10 bytes, first byte is '\0'
```

When you write `"Hola, mundo"`, the compiler automatically adds a `'\0'` at the end. The array `greeting` is 12 bytes: 11 visible characters plus the null terminator.

> **Trap:** Always remember the null terminator when sizing your buffers. The string `"hello"` needs 6 bytes, not 5. Off-by-one errors with null terminators are one of the most common bugs in C.

String Functions

C provides a library of string manipulation functions in `<string.h>`. These are the ones you will use most:

`strlen` — Get the Length

```c
size_t strlen(const char *s);
```

Returns the number of characters before the null terminator:

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char song[] = "Take On Me";
    printf("'%s' has %zu characters\n", song, strlen(song));  // 10
    return 0;
}
```

Note that `strlen` does not count the `'\0'`. The array holding `"Take On Me"` is 11 bytes, but `strlen` returns 10.

`strcpy` / `strncpy` — Copy a String

```c
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

`strcpy` copies the source string (including the null terminator) into the destination buffer. You must make sure the destination is large enough:

```
char dest[20];
strcpy(dest, "Tainted Love");  // copies 13 bytes (12 chars + '\0')
```

`strncpy` is the safer variant — it copies at most `n` characters:

```
char dest[10];
strncpy(dest, "Enjoy the Silence", 9);
dest[9] = '\0';  // strncpy does NOT guarantee null termination!
```

> **Wut:** `strncpy` does not null-terminate the destination if the source is longer than `n`. Always set the last byte yourself: `dest[sizeof(dest) - 1] = '\0';`

`strcmp` — Compare Strings

```
int strcmp(const char *s1, const char *s2);
```

In C++, you can compare strings with `==`. In C, you cannot — using `==` on two `char` arrays compares the *addresses*, not the contents. Use `strcmp` instead:

```
char a[] = "Rush";
char b[] = "Rush";

if (a == b) {
    // This compares ADDRESSES, not content - almost always false
}

if (strcmp(a, b) == 0) {
    // This compares the actual characters - correct!
    printf("Same band!\n");
}
```

`strcmp` returns 0 if the strings are equal, a negative value if the first is lexicographically less, and a positive value if it is greater.

> **Trap:** Yes, `strcmp` returns 0 for equal strings. It is a common source of confusion. Think of it as returning the "difference" between the strings — zero means no difference.

`strchr` / `strrchr` — Find a Character

```
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

`strchr` finds the first occurrence of a character. `strrchr` finds the last:

```
char lyric[] = "Don't You Forget About Me";
char *first_o = strchr(lyric, 'o');    // points to 'o' in "Don't"
char *last_o = strrchr(lyric, 'o');    // points to 'o' in "About"
```

```c
if (first_o != NULL) {
    printf("First 'o' at position %td\n", first_o - lyric);
}
```

Both functions return a pointer to the found character, or `NULL` if the character is not in the string.

`strstr` — Find a Substring

```c
char *strstr(const char *haystack, const char *needle);
```

Finds the first occurrence of a substring within a string:

```c
char title[] = "Blade Runner 1982";
char *found = strstr(title, "Runner");
if (found != NULL) {
    printf("Found: %s\n", found);  // "Runner 1982"
}
```

Like `strchr`, it returns a pointer to the start of the match, or `NULL` if not found.

`strcat` / `strncat` — Concatenate Strings

```c
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

`strcat` appends one string to the end of another:

```c
char message[50] = "I'll be ";
strcat(message, "back");
printf("%s\n", message);  // "I'll be back"
```

This works fine as long as the destination buffer is large enough. But `strcat` does not check bounds — if you run out of space, it writes past the end of the array.

> **Trap:** `strcat` is one of the most dangerous functions in C. It has no way to know how large the destination buffer is, so it blindly appends bytes. If the combined strings exceed the buffer size, you get a **buffer overflow** — one of the most common security vulnerabilities in the history of software. Use `strncat` instead.

`strncat` takes a maximum number of characters to append as the last argument. Make sure to leave space for the null terminator, so if there are three bytes left in a buffer, the maximum number of characters to append would be 2.

```c
char buf[20] = "Hello";
strncat(buf, ", World!", sizeof(buf) - strlen(buf) - 1);
```

`strdup` — Duplicate a String

```c
char *strdup(const char *s);
```

`strdup` allocates new memory on the heap, copies the string into it, and returns a pointer to the copy. You are responsible for freeing the memory when you are done:

```
char *copy = strdup("Video Killed the Radio Star");
printf("%s\n", copy);
free(copy);  // you must free memory allocated by strdup
```

> **Tip:** `strdup` calls `malloc` internally. Every call to `strdup` must eventually be paired with a call to `free`. If you forget, you have a memory leak. Note that `strdup` is a POSIX function, not part of the C standard until C23. It is available on virtually every system you will use, but compiling with `-std=c99 -pedantic` or `-std=c11 -pedantic` will produce a warning.

`strtok` — Tokenize a String

```
char *strtok(char *str, const char *delim);
```

`strtok` splits a string into tokens separated by any character in a delimiter set. In C++, you might use `std::istringstream` with `>>` or `std::string::find` — in C, `strtok` is the standard approach:

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char line[] = "Girls Just Want to Have Fun";

    char *tok = strtok(line, " ");
    while (tok != NULL) {
        printf("'%s'\n", tok);
        tok = strtok(NULL, " ");
    }
    return 0;
}
// Output:
// 'Girls'
// 'Just'
// 'Want'
// 'to'
// 'Have'
// 'Fun'
```

The first call passes the string to tokenize. Each subsequent call passes `NULL` to continue tokenizing the same string. `strtok` returns `NULL` when there are no more tokens.

There are two important gotchas. First, `strtok` **modifies the original string** by replacing delimiter characters with `'\0'`. If you need the original string

preserved, make a copy with `strdup` before tokenizing. Second, `strtok` stores its state in a hidden static variable, which means it is **not thread-safe** and you cannot tokenize two strings at the same time.

> **Tip:** Use `strtok_r` (POSIX) or `strtok_s` (C11 Annex K / Windows) instead of `strtok`. These reentrant versions take an extra `char **saveptr` parameter to store the tokenizer state, making them safe to use in multi-threaded programs and allowing nested tokenization:
>
> ```
> char *strtok_r(char *str, const char *delim, char **saveptr);
> ```
>
> ```
> char *saveptr;
> char *tok = strtok_r(line, " ", &saveptr);
> while (tok != NULL) {
>     tok = strtok_r(NULL, " ", &saveptr);
> }
> ```

The Dangers of `strcat`

Let's look at a concrete example of why `strcat` is dangerous:

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char buf[12] = "Buenas ";     // 7 chars + '\0', 4 bytes remaining

    // "noches" is 6 chars - doesn't fit in 4 remaining bytes!
    strcat(buf, "noches");        // BUFFER OVERFLOW - undefined behavior

    printf("%s\n", buf);          // might print garbage, might crash
    return 0;
}
```

The buffer is 12 bytes, `"Buenas "` uses 8 (including the `'\0'`), and `"noches"` needs 7 more bytes (including its `'\0'`). That is 14 bytes total in a 12-byte buffer. The extra bytes overwrite whatever happens to be next to the buffer in memory, which can corrupt other variables, crash the program, or — worst of all — create a security exploit.

A Preview: `sprintf` and `sscanf`

```
int sprintf(char *str, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

C has two powerful functions for building and parsing strings that we will cover in detail in the Standard I/O chapter: `sprintf` writes formatted output into a string buffer (like `printf` but to a string), and `sscanf` reads formatted input

from a string (like `scanf` but from a string). They are the C programmer's Swiss Army knife for string manipulation:

```
char result[50];
int year = 1985;
sprintf(result, "The year is %d. Que bueno!", year);
// result is now "The year is 1985. Que bueno!"
```

Try It: String Starter

This program exercises several string functions so you can see them in action:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
    // strlen
    char song[] = "Sweet Dreams";
    printf("Song: '%s' (%zu chars)\n", song, strlen(song));

    // strcpy
    char copy[20];
    strcpy(copy, song);
    printf("Copy: '%s'\n", copy);

    // strcmp
    printf("Compare '%s' to '%s': %d\n", song, copy, strcmp(song, copy));
    printf("Compare 'A' to 'B': %d\n", strcmp("A", "B"));

    // strcat
    char greeting[30] = "Buenos ";
    strcat(greeting, "dias");
    printf("Greeting: '%s'\n", greeting);

    // strchr and strstr
    char *ch = strchr(song, 'D');
    if (ch) printf("Found 'D' at position %td\n", ch - song);

    char *sub = strstr(song, "Dreams");
    if (sub) printf("Found substring: '%s'\n", sub);

    // strdup
    char *dup = strdup("Never Gonna Give You Up");
    printf("Duplicate: '%s'\n", dup);
    free(dup);
```

```
    return 0;
}
```

Key Points

- C strings are `char` arrays terminated by `'\0'`. There is no `std::string`.
- Always account for the null terminator when sizing buffers.
- Use `strcmp` to compare strings — the `==` operator compares addresses, not content.
- `strcpy` and `strcat` do not check buffer bounds. Prefer `strncpy` and `strncat`.
- `strdup` allocates memory with `malloc` — you must `free` the result.
- `strtok` splits strings into tokens but modifies the original string and is not thread-safe. Use `strtok_r` or `strtok_s` instead.
- `strlen` returns the number of characters *before* the `'\0'`, not the buffer size.

Exercises

1. **Think about it:** Why does `strcmp` return 0 for equal strings rather than 1? How does this relate to the function's actual purpose?
2. **What does this print?**

   ```
   char s[] = "Ghostbusters";
   printf("%zu %zu\n", strlen(s), sizeof(s));
   ```

3. **Calculation:** What is `sizeof(buf)` for `char buf[20] = "Hola";`?
4. **Where is the bug?**

   ```
   char buf[10] = "Livin'";
   strcat(buf, " on a Prayer");
   printf("%s\n", buf);
   ```

5. **Where is the bug?**

   ```
   char *a = "Hello";
   char *b = "Hello";
   if (a == b) {
       printf("Equal\n");
   } else {
       printf("Not equal\n");
   }
   ```

   (Hint: what does `==` actually compare here? Is the output guaranteed?)

6. **Write a program** that reads a string from the user, reverses it in place using pointer arithmetic, and prints the result.

5. Standard I/O

C's `<stdio.h>` library is your replacement for C++ `iostream`. It provides `printf` and `scanf` for formatted output and input, file operations with `fopen` and `fclose`, and binary I/O with `fread` and `fwrite`. Everything flows through the `FILE *` type — an opaque pointer to a structure that tracks the state of an I/O stream.

`scanf` for Input

```
int scanf(const char *format, ...);
```

You have already seen `printf` for output. For input, C uses `scanf`, which reads formatted data from standard input:

```
#include <stdio.h>

int main(void) {
    int year;
    printf("Enter a year: ");
    scanf("%d", &year);
    printf("You entered: %d\n", year);
    return 0;
}
```

Notice the `&` before `year`. Since C is pass by value, `scanf` needs the *address* of the variable so it can store the result there. Forgetting the `&` is a classic bug — the program compiles but crashes or produces garbage at runtime.

`scanf` uses similar format specifiers to `printf`, but not identical ones. Notably, `scanf` uses `%lf` for `double` while `printf` uses `%f`:

```
char name[50];
double gpa;
scanf("%s %lf", name, &gpa);
```

Note that `name` does not need `&` because an array name already points to the bytes we want to read into. But `gpa` does need a `&`, because `scanf` needs to know where `gpa` is stored to fill it in.

> **Trap:** `scanf("%s", ...)` reads a single word (stopping at whitespace). It also has no bounds checking — it will happily overflow your buffer. Use a width specifier like `%49s` to limit input to 49 characters (plus `'\0'`).

Scan Sets

`scanf` supports **scan set specifiers** with `%[...]`, which let you define exactly which characters to accept. The scan set reads characters as long as they are in the set, and stops at the first character that is not:

```c
char vowels[20];
scanf("%19[aeiouAEIOU]", vowels);  // reads only vowels, stops at first non-vowel
```

A caret `^` at the start of the set **negates** it — read everything *except* the listed characters. This gives you a way to read an entire line with `scanf`, since `%[^\n]` reads everything up to (but not including) the newline:

```c
char line[80];
scanf("%79[^\n]", line);   // reads a full line (up to 79 chars)
printf("You said: %s\n", line);
```

Always include a width limit to prevent buffer overflow, just like with `%s`.

You already saw a scan set used with `sscanf` earlier in the strings chapter preview. Here it is again to parse a structured string:

```c
char buf[] = "Track 03: 99 Luftballons";
int track;
char title[50];
sscanf(buf, "Track %d: %49[^\n]", &track, title);
// track is 3, title is "99 Luftballons"
```

> **Tip:** `%[^\n]` is the `scanf` way to read a line, but `fgets` is generally safer and simpler for line-oriented input. Use scan sets when you need to parse structured input where only certain characters are valid.

The %m Modifier (POSIX)

With `%s` and `%[...]`, you must always provide a buffer that is large enough. The POSIX `%m` modifier (called the **assignment-allocation** modifier) tells `scanf` to `malloc` the buffer for you. Instead of passing a `char[]`, you pass a `char **` and `scanf` allocates exactly enough memory:

```c
char *line = NULL;
scanf("%m[^\n]", &line);       // scanf mallocs the buffer
printf("You said: %s\n", line);
free(line);                     // you must free it
```

Notice `&line` — `scanf` needs a pointer to your `char *` so it can fill it in with the address of the newly allocated buffer. This eliminates buffer overflow risk entirely, since the buffer is always the right size.

`%ms` works the same way for single words:

```c
char *word = NULL;
scanf("%ms", &word);   // reads one word, malloc'd to fit
free(word);
```

Tip: `%m` is a POSIX extension (available on Linux, macOS, and most Unix systems) and is not part of the C standard. It will not work with MSVC on Windows. When portability is not a concern, `%m` is an excellent way to avoid buffer sizing headaches.

stdin, stdout, and stderr

When your C program starts, three streams (of type `FILE *`) are already open:

Stream	Purpose	C++ equivalent
`stdin`	Standard input (keyboard)	`std::cin`
`stdout`	Standard output (screen)	`std::cout`
`stderr`	Standard error (screen)	`std::cerr`

`printf(...)` is actually shorthand for `fprintf(stdout, ...)`. You can write to `stderr` for error messages:

```
fprintf(stderr, "Error: file not found\n");
```

Error messages sent to `stderr` are not affected by output redirection (`./program > output.txt` only redirects `stdout`), so error messages still appear on the screen. `./program 2> err.txt` will redirect errors to `err.txt` and `stdout` will appear on the screen.

fprintf and fscanf

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

`fprintf` and `fscanf` are the file versions of `printf` and `scanf`. They take a `FILE *` as the first argument:

```
fprintf(stdout, "Hello\n");           // same as printf("Hello\n")
fprintf(stderr, "Something broke\n"); // write to stderr
```

More usefully, you can use them with files you have opened yourself. Here is `fscanf` reading from a file:

```
FILE *f = fopen("scores.txt", "r");
if (f != NULL) {
    char name[50];
    int score;
    while (fscanf(f, "%49s %d", name, &score) == 2) {
        printf("%s scored %d\n", name, score);
    }
    fclose(f);
}
```

`fscanf` returns the number of items successfully read, so checking the return value tells you whether the read succeeded.

Opening and Closing Files

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
```

To read or write a file, you open it with `fopen` and close it with `fclose`:

```
#include <stdio.h>

int main(void) {
    FILE *f = fopen("log.txt", "w");
    if (f == NULL) {
        fprintf(stderr, "Cannot open file\n");
        return 1;
    }

    fprintf(f, "Under Pressure\n");
    fprintf(f, "Year: %d\n", 1981);

    fclose(f);
    return 0;
}
```

The second argument to `fopen` is the **mode string**:

Mode	Meaning
`"r"`	Read (file must exist)
`"w"`	Write (creates or truncates)
`"a"`	Append (creates or appends)
`"r+"`	Read and write (file must exist)
`"w+"`	Read and write (creates or truncates)
`"a+"`	Read and append

To open a file in **binary mode**, add `b` to the mode string: `"rb"`, `"wb"`, `"ab"`, etc. On Unix systems, binary and text modes behave identically. On Windows, text mode translates `\r\n` to `\n` on input and vice versa on output — binary mode does not.

`sprintf` and `sscanf`

`sprintf` writes formatted output into a string buffer instead of a stream. `sscanf` reads formatted input from a string:

```c
char buf[100];
sprintf(buf, "Track %02d: %s", 3, "99 Luftballons");
// buf is now "Track 03: 99 Luftballons"

int track;
char title[50];
sscanf(buf, "Track %d: %49[^\n]", &track, title);
// track is 3, title is "99 Luftballons"
```

> **Trap:** `sprintf` has the same buffer overflow risk as `strcpy` — it does not check the size of the destination buffer. Use `snprintf` for safety:
>
> ```c
> int snprintf(char *str, size_t size, const char *format, ...);
> ```
>
> ```c
> snprintf(buf, sizeof(buf), "Track %02d: %s", 3, "99 Luftballons");
> ```
>
> `snprintf` guarantees it will not write more than `sizeof(buf)` bytes, including the null terminator.

`asprintf` (POSIX) goes one step further — it `malloc`s a buffer that is exactly the right size, so you never have to guess:

```c
int asprintf(char **strp, const char *format, ...);
```

```c
char *msg;
asprintf(&msg, "Track %02d: %s", 3, "Mis Ojos Lloran Por Ti");
printf("%s\n", msg);   // "Track 03: Mis Ojos Lloran Por Ti"
free(msg);              // you must free it
```

Like `%m` in `scanf`, you pass a pointer to a `char *` and `asprintf` fills it in with the address of the newly allocated string. It returns the number of characters written, or -1 on failure.

> **Tip:** `asprintf` is a POSIX/GNU extension, not part of the C standard. It is available on Linux, macOS, and most Unix systems but not MSVC. When it is available, it is the safest and most convenient way to build formatted strings — no buffer sizing, no truncation, no overflow.

Binary I/O: `fread` and `fwrite`

```c
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
```

For reading and writing raw binary data (not text), use `fread` and `fwrite`:

```c
#include <stdio.h>

int main(void) {
    int nums[] = {10, 20, 30, 40, 50};

    // Write binary data
```

```c
    FILE *f = fopen("data.bin", "wb");
    fwrite(nums, sizeof(int), 5, f);
    fclose(f);

    // Read it back
    int result[5];
    f = fopen("data.bin", "rb");
    fread(result, sizeof(int), 5, f);
    fclose(f);

    for (int i = 0; i < 5; i++) {
        printf("%d ", result[i]);   // 10 20 30 40 50
    }
    printf("\n");
    return 0;
}
```

Both functions take four arguments: a pointer to the data, the size of each element, the number of elements, and the file stream. `fwrite` returns the number of elements successfully written; `fread` returns the number of elements successfully read.

Buffering and `fflush`

`stdio` does not write directly to the output device on every call. Instead, it accumulates data in an internal buffer and writes it in larger chunks for efficiency. There are three buffering modes:

- **Full buffering:** Output is written when the buffer is full (default for files).
- **Line buffering:** Output is written when a `\n` is encountered (default for `stdout` when connected to a terminal).
- **Unbuffered:** Output is written immediately (default for `stderr`).

This means that `printf("Working...")` (no newline) might not appear on screen immediately when `stdout` goes to a terminal, and will almost certainly not appear immediately when redirected to a file. Use `fflush` to force the buffer to be written:

```c
int fflush(FILE *stream);
```

```c
printf("Working...");
fflush(stdout);   // force output to appear now
// ... long computation ...
printf(" done!\n");
```

Trap: When `stdout` is connected to a terminal, output is **line buffered** — a `\n` triggers a flush. When `stdout` is redirected to a file or pipe, it is **fully buffered** — output may not appear until the buffer fills up or the program exits. If you need output to appear immediately (e.g., progress indicators), call `fflush(stdout)` after printing. `stderr` is always unbuffered, which is why error messages appear immediately.

Key Points

- `printf` writes to `stdout`; `fprintf` writes to any `FILE *`.
- `scanf` needs the address (`&`) of each variable — arrays are the exception since they decay to pointers.
- `fopen` returns `NULL` on failure — always check before using the file pointer.
- Add `"b"` to the mode string for binary files. This matters on Windows.
- `fread` and `fwrite` transfer raw bytes — no format conversion.
- `stdout` is line buffered at a terminal and fully buffered when redirected. Use `fflush` when you need output immediately.

Exercises

1. **Think about it:** Why does `scanf` need the `&` operator for scalar variables but not for arrays?
2. **What does this print?**

   ```c
   char buf[50];
   sprintf(buf, "%s: %d", "Score", 100);
   printf("%zu\n", strlen(buf));
   ```

3. **Calculation:** If `buf` is declared as `char buf[20]` and you call `snprintf(buf, sizeof(buf), "Year: %d", 1984)`, how many characters (excluding the null terminator) are written to `buf`?
4. **Where is the bug?**

   ```c
   int x;
   scanf("%d", x);
   ```

5. **Where is the bug?**

   ```c
   FILE *f = fopen("noexist.txt", "r");
   fprintf(f, "Hello\n");
   fclose(f);
   ```

6. **Think about it:** You run `./program > output.txt` and your program contains both `printf` and `fprintf(stderr, ...)` calls. Which messages appear in `output.txt` and which appear on the screen? Why?
7. **Write a program** that opens a text file, writes five lines to it (your choice of content), closes it, reopens it for reading, reads and prints each line

using `fgets`, then closes it again.

6. Low-Level I/O

The `<stdio.h>` functions you learned in the previous chapter are built on top of a lower-level I/O interface provided by the operating system. These system calls — `read`, `write`, `open`, and `close` — work directly with **file descriptors** rather than `FILE *` pointers. You will encounter them in systems programming, and understanding them helps you see what `stdio` is actually doing under the hood.

File Descriptors

A file descriptor is a small non-negative integer that the operating system uses to identify an open file (or pipe, socket, device, etc.). When your program starts, three file descriptors are already open:

File Descriptor	POSIX Name	Purpose
0	`STDIN_FILENO`	Standard input
1	`STDOUT_FILENO`	Standard output
2	`STDERR_FILENO`	Standard error

These constants are defined in `<unistd.h>`. They correspond to `stdin`, `stdout`, and `stderr` from `<stdio.h>`, but at a lower level.

`read` and `write`

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

The `read` and `write` system calls transfer raw bytes between a file descriptor and a buffer:

```
#include <unistd.h>

// write(fd, buffer, count) - returns bytes written
write(1, "Blue Monday\n", 12);   // write 12 bytes to stdout

// read(fd, buffer, count) - returns bytes read
char buf[100];
ssize_t n = read(0, buf, sizeof(buf));  // read from stdin
write(1, buf, n);                        // echo it back
```

`read` returns the number of bytes actually read (which may be less than requested), 0 at end of file, or -1 on error. `write` returns the number of bytes actually written, or -1 on error.

Unlike `printf` and `scanf`, these functions perform no formatting — they transfer raw bytes. There are no format specifiers, no newline handling, no buffering.

open and close

```
int open(const char *path, int flags, ... /* mode_t mode */);
int close(int fd);
```

To open a file at the system call level, use `open` from `<fcntl.h>`:

```
#include <fcntl.h>
#include <unistd.h>

int fd = open("data.txt", O_RDONLY);
if (fd == -1) {
    write(2, "Cannot open file\n", 17);
    return 1;
}

char buf[256];
ssize_t n = read(fd, buf, sizeof(buf));
write(1, buf, n);

close(fd);
```

The second argument to `open` is a set of flags combined with bitwise OR:

Flag	Purpose
`O_RDONLY`	Open for reading only
`O_WRONLY`	Open for writing only
`O_RDWR`	Open for reading and writing
`O_CREAT`	Create the file if it does not exist
`O_TRUNC`	Truncate the file to zero length
`O_APPEND`	Append to the file

To **create a new file** (or truncate an existing one), combine flags:

```
int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

The third argument (`0644`) is the **file permissions** — only used with `O_CREAT`. The value `0644` means the owner can read and write, and everyone else can only read.

There is also `creat`, which is equivalent to `open` with `O_WRONLY | O_CREAT | O_TRUNC`:

```
int creat(const char *path, mode_t mode);
```

```
int fd = creat("output.txt", 0644);
// same as: open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644)
```

Wut: Yes, it is `creat` with no `e`. Ken Thompson was once asked what he would do differently if he were redesigning Unix. His answer: "I'd spell creat with an e."

Seeking: `lseek`

```
off_t lseek(int fd, off_t offset, int whence);
```

`lseek` repositions the file offset for an open file descriptor:

```
#include <unistd.h>

lseek(fd, 0, SEEK_SET);      // go to beginning
lseek(fd, 0, SEEK_END);      // go to end
lseek(fd, 100, SEEK_SET);    // go to byte 100

off_t pos = lseek(fd, 0, SEEK_CUR);  // get current position (no move)
```

The three `SEEK_` constants control where the offset is relative to:

Constant	Meaning
`SEEK_SET`	Relative to the beginning of the file
`SEEK_CUR`	Relative to the current position
`SEEK_END`	Relative to the end of the file

`lseek` returns the new offset from the beginning of the file, or -1 on error.

Tip: In `<stdio.h>`, the equivalent functions are `fseek` and `ftell`. The low-level `lseek` combines both: calling `lseek(fd, 0, SEEK_CUR)` returns the current position without moving, just like `ftell`.

`pread` and `pwrite`

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

`pread` and `pwrite` are like `read` and `write` but take an explicit offset instead of using (or modifying) the file's current position:

```
// Read 100 bytes starting at offset 200, without changing the file position
ssize_t n = pread(fd, buf, 100, 200);

// Write 50 bytes at offset 0, without changing the file position
pwrite(fd, data, 50, 0);
```

These are useful in multi-threaded programs where multiple threads share a file descriptor — since they do not modify the file position, there is no race condition.

Key Points

- File descriptors are small integers: 0 is stdin, 1 is stdout, 2 is stderr.
- `read` and `write` transfer raw bytes — no formatting, no buffering.
- `open` returns a file descriptor; `fopen` returns a `FILE *`. They are different levels of abstraction.
- Use `O_CREAT` with `open` to create files. Always provide a permissions argument when using `O_CREAT`.
- `lseek` repositions the read/write offset. Use `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- `pread` and `pwrite` read/write at a specific offset without changing the file position.

Exercises

1. **Think about it:** Why would you use low-level `read`/`write` instead of `fprintf`/`fscanf`? When would `stdio` be the better choice?
2. **What does this print?**

   ```
   write(1, "ABC", 3);
   write(1, "DEF\n", 4);
   ```

3. **Calculation:** If `read(fd, buf, 1024)` returns 512, what does that tell you? Does it mean there was an error?
4. **Where is the bug?**

   ```
   int fd = open("newfile.txt", O_WRONLY | O_CREAT);
   write(fd, "Hello\n", 6);
   close(fd);
   ```

5. **Think about it:** Explain the difference between `lseek(fd, 0, SEEK_END)` and `lseek(fd, -1, SEEK_END)`. What does each return?
6. **Write a program** that uses low-level I/O (`open`, `read`, `write`, `close`) to copy the contents of one file to another. The source and destination filenames should be taken from `argv`.

7. Odds and Ends

This chapter covers a few remaining topics that do not fit neatly into the previous chapters but are important for writing real C programs and for working with C code from C++.

`exit` vs `return`

```
void exit(int status);
```

You already know that `return` in `main` ends the program. The `exit` function from `<stdlib.h>` does the same thing, but it can be called from *any* function — not just `main`:

```
#include <stdio.h>
#include <stdlib.h>

void check_file(const char *path) {
    FILE *f = fopen(path, "r");
    if (f == NULL) {
        fprintf(stderr, "Fatal: cannot open %s\n", path);
        exit(1);   // end the program immediately
    }
    // ... work with the file ...
    fclose(f);
}
```

`exit` is useful when an error deep inside a call chain is unrecoverable and there is no reasonable way to propagate the error back through multiple layers of callers. In C++, you would throw an exception; in C, `exit` is sometimes the pragmatic choice.

`exit` also flushes all open `stdio` streams and calls any functions registered with `atexit` before terminating the program.

```
int atexit(void (*func)(void));
```

> **Tip:** Use `exit` sparingly. It is a blunt instrument — it ends the entire program immediately, skipping any cleanup code in calling functions. If you can reasonably propagate an error code back to `main` and let `main` return, prefer that approach. Reserve `exit` for truly fatal errors.

`extern "C"` — Calling C from C++

If you are writing C++ code that needs to call functions from a C library, you need `extern "C"`. The reason: C++ **mangles** function names to support overloading (so `void foo(int)` and `void foo(double)` have different symbol

names), but C does not. Without `extern "C"`, the C++ linker looks for the mangled name and cannot find the C function.

```
// In your C++ code:
extern "C" {
    #include "my_c_library.h"   // treat these declarations as C
}
```

Or for a single function:

```
extern "C" void c_function(int x);
```

Many C headers protect themselves with this pattern so they work from both C and C++:

```
#ifdef __cplusplus
extern "C" {
#endif

void some_function(int x);
int another_function(const char *s);

#ifdef __cplusplus
}
#endif
```

The `__cplusplus` macro is only defined when compiling with a C++ compiler, so the `extern "C"` wrapper only appears in C++ compilation.

> **Tip:** The reason C++ mangles names is to support **function overloading** — having multiple functions with the same name but different parameter types. C has no function overloading. Every function name must be unique. If you need two functions that do the same thing for different types, you give them different names — for example, `abs` for `int`, `fabs` for `double`, and `labs` for `long`. The upside is that when you see a function call in C, you know exactly which function will be invoked — there is only one version.

Pointer Ownership

In C++, smart pointers make ownership clear: a `std::unique_ptr` owns the memory, and when it goes out of scope, the memory is freed. In C, there are no smart pointers. When a function returns a pointer, you must ask: **who owns this memory?**

There are three common patterns:

1. The caller owns it (you must free). The function allocates memory and hands ownership to you:

```c
char *copy = strdup("Everybody Wants to Rule the World");
// You own this memory. You must free it.
free(copy);
```

2. The library owns it (do not free). The function returns a pointer to memory it manages internally:

```c
struct hostent *h = gethostbyname("example.com");
// The library owns this. Do NOT free it.
```

3. You own it (you passed it in). You allocated the memory and passed a pointer to the function. The function used it but did not take ownership:

```c
char buf[100];
fgets(buf, sizeof(buf), stdin);
// You still own buf. Nothing to free (it's on the stack).
```

> **Trap:** Always read the documentation of a C function that returns a pointer. Look for words like "the caller must free the returned pointer" or "the returned pointer points to a static buffer." If the documentation does not say, look at the source code. Getting ownership wrong leads to either memory leaks (never freeing) or double-free bugs (freeing what you do not own).

Error Handling Without Exceptions

In C++, you can `throw` an exception and let a `catch` block handle it several call levels up. C has no exceptions. Error handling is done through return codes, and cleanup is your responsibility.

The simplest pattern is to check return values and bail out:

```c
void perror(const char *s);
```

```c
FILE *f = fopen("La Isla Bonita.txt", "r");
if (!f) {
    perror("fopen");
    return -1;
}
```

But what happens when a function acquires multiple resources? You need to release them in the correct order when something goes wrong. The idiomatic C pattern uses `goto` to jump to cleanup labels:

```c
int process(const char *path) {
    int status = -1;

    FILE *f = fopen(path, "r");
    if (!f) return -1;

    char *buf = malloc(1024);
```

```
    if (!buf) goto close_file;

    char *line = malloc(256);
    if (!line) goto free_buf;

    /* do work with f, buf, and line ... */
    status = 0;

    free(line);
free_buf:
    free(buf);
close_file:
    fclose(f);
    return status;
}
```

Each resource acquired gets a corresponding cleanup label below it. If any allocation fails, control jumps to the label that releases everything acquired so far, in reverse order. This pattern is used extensively in real C code including the Linux kernel.

> **Wut:** C++ programmers are taught "never use `goto`." In C, `goto` for cleanup is an accepted and widely used idiom. It is the closest thing C has to RAII — a structured way to ensure resources are always released.

The other common strategy is to return an error code (often `-1` or `NULL`) and let the caller decide what to do. Many C library functions set the global variable `errno` to indicate what went wrong, and you can use `perror` or `strerror(errno)` to get a human-readable message:

```
char *strerror(int errnum);
```

```
FILE *f = fopen("No Existe.txt", "r");
if (!f) {
    perror("fopen");    // prints: fopen: No such file or directory
}
```

Function Pointers and `qsort`

In C++, you pass behavior to functions using lambdas, `std::function`, or template parameters. C has none of those — instead, you use **function pointers**. A function pointer holds the address of a function, and you can call it just like a regular function.

Here is a simple example:

```
#include <stdio.h>
```

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

int main(void) {
    int (*op)(int, int);   // declare a function pointer

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

    op = mul;
    printf("%d\n", op(3, 4));   // 12
    return 0;
}
```

The declaration `int (*op)(int, int)` reads: `op` is a pointer to a function that takes two `int` parameters and returns an `int`. The parentheses around `*op` are required — without them, `int *op(int, int)` would declare a function that returns an `int *`.

> **Tip:** Function pointer declarations are notoriously hard to read. A `typedef` makes them much clearer:
>
> ```c
> typedef int (*binop_fn)(int, int);
> binop_fn op = add;
> printf("%d\n", op(3, 4));
> ```

The most common place you will encounter function pointers is the standard library function `qsort` from `<stdlib.h>`. In C++, you would use `std::sort` with a lambda or comparator. In C, `qsort` takes a comparison function pointer:

```c
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int compare_ints(const void *a, const void *b) {
    int ia = *(const int *)a;
    int ib = *(const int *)b;
    return (ia > ib) - (ia < ib);
}

int main(void) {
    int years[] = {1989, 1982, 1985, 1980, 1987};
    int n = sizeof(years) / sizeof(years[0]);

    qsort(years, n, sizeof(int), compare_ints);
```

```
    for (int i = 0; i < n; i++) {
        printf("%d ", years[i]);
    }
    printf("\n");
    // Output: 1980 1982 1985 1987 1989
    return 0;
}
```

`qsort` takes four arguments: the array, the number of elements, the size of each element, and a pointer to a comparison function. The comparison function receives `const void *` pointers — you must cast them to the correct type inside the function. It returns a negative value if the first argument is less than the second, zero if equal, and a positive value if greater.

You can sort anything with `qsort` by writing different comparison functions. Here is one that sorts strings:

```
int compare_strings(const void *a, const void *b) {
    const char *sa = *(const char **)a;
    const char *sb = *(const char **)b;
    return strcmp(sa, sb);
}

int main(void) {
    const char *songs[] = {
        "Maniac", "Footloose", "Flashdance", "Fame"
    };
    int n = sizeof(songs) / sizeof(songs[0]);

    qsort(songs, n, sizeof(char *), compare_strings);

    for (int i = 0; i < n; i++) {
        printf("%s\n", songs[i]);
    }
    // Output:
    // Fame
    // Flashdance
    // Footloose
    // Maniac
    return 0;
}
```

Notice the double cast in `compare_strings`: `qsort` passes a pointer *to* each array element, and each element is already a `char *`, so you receive a `char **` disguised as `const void *`.

Wut: A common mistake is to write `return a - b` in integer comparison functions. This can overflow when `a` and `b` have very different signs (e.g., `INT_MAX - (-1)` overflows). The pattern `(a > b) - (a < b)` is safe and returns -1, 0, or 1.

Key Points

- `exit` terminates the program from any function. Use it for unrecoverable errors.
- `exit` flushes `stdio` streams and calls `atexit` handlers before terminating.
- `extern "C"` tells the C++ compiler not to mangle function names, so it can link to C libraries.
- C headers often use `#ifdef __cplusplus` to wrap declarations in `extern "C"` automatically.
- When you receive a pointer from a function, always determine who owns the memory: you, the function, or a library.
- C has no exceptions. Use return codes for errors and `goto` cleanup for releasing resources in the correct order.
- Function pointers let you pass behavior to functions. `qsort` is the most common example — it takes a comparison function pointer to sort any type.

Exercises

1. **Think about it:** In C++ you would use exceptions for error handling. In C there are no exceptions. What strategies can you use to handle errors in deeply nested function calls? When is `exit` appropriate and when is it not?

2. **What happens here?**

```c
#include <stdlib.h>
#include <stdio.h>

void cleanup(void) {
    printf("Adios!\n");
}

int main(void) {
    atexit(cleanup);
    printf("Starting...\n");
    exit(0);
}
```

3. **Where is the bug?**

```c
char *get_greeting(void) {
    char buf[50];
```

```
    sprintf(buf, "Hola, mundo");
    return buf;
}
```

4. **Think about it:** You call a function `char *get_name(int id)` from a library. How would you determine whether you need to `free` the returned pointer?

5. **Where is the bug?** (Hint: ownership)

```
char *name = strdup("Walking on Sunshine");
char *alias = name;
free(name);
printf("%s\n", alias);
```

6. **Calculation:** Given `int nums[] = {5, 10, 15, 20};`, what is the value of `sizeof(nums) / sizeof(nums[0])`?

7. **What does this print?**

```
int compare_desc(const void *a, const void *b) {
    int ia = *(const int *)a;
    int ib = *(const int *)b;
    return (ib > ia) - (ib < ia);
}

int main(void) {
    int vals[] = {3, 1, 4, 1, 5};
    qsort(vals, 5, sizeof(int), compare_desc);
    printf("%d %d %d %d %d\n", vals[0], vals[1], vals[2], vals[3], vals[4]);
    return 0;
}
```

8. **Write a program** that uses `qsort` to sort an array of strings in reverse alphabetical order. Write a custom comparison function that calls `strcmp` with the arguments swapped.

9. **Write a program** in C++ that uses `extern "C"` to call the C function `strlen` from `<string.h>`, passes it a string, and prints the result. Compile it with `c++` to verify it works.

Conclusion

You have covered a lot of ground — from `printf` format specifiers to file descriptors to pointer ownership. Here are the key takeaways:

- **C and C++ are different languages.** Modern C++ has evolved far from C. Knowing one does not mean you automatically know the other.
- **`printf` and `scanf` replace `iostream`.** Format specifiers must match argument types. `scanf` needs `&` for scalar variables.
- **Pointers hold memory addresses.** Use `&` to get an address, `*` to follow one, and `->` to access struct fields through a pointer. Arrays decay to pointers, and pointer arithmetic moves in units of the pointed-to type.
- **All function arguments are pass by value.** To modify a caller's variable, pass a pointer to it.
- **Know where your memory lives.** Global variables last the whole program, local variables live on the stack, and dynamic memory from `malloc` lives on the heap until you `free` it.
- **Strings in C are `char` arrays** terminated by `'\0'`. You must manage buffer sizes manually and use functions like `strlen`, `strcpy`, `strcmp`, and `strcat` instead of `std::string` methods.
- **`stdio` provides buffered I/O** through `FILE *` pointers. Low-level I/O uses file descriptors and system calls like `read`, `write`, and `open`.
- **C has no exceptions.** Use return codes for errors and `goto` cleanup to release resources in reverse order.
- **Function pointers replace lambdas.** `qsort` is the classic example — pass a comparison function to sort any type.
- **`exit` terminates from anywhere.** Use it for fatal errors. `extern "C"` bridges C and C++. Always know who owns a pointer.

Es un mundo nuevo, but you have the C++ foundation to build on. The syntax will feel familiar even when the idioms are different. Write small programs, compile them with `cc`, and get comfortable with the compiler's warnings — they are your best amigo.

Buena suerte — you have got this.

Content outline and editorial support from Ben. Words by Claude, the Opus.

Appendix A: Macros

In C++, you have `constexpr` for compile-time constants, templates for generic code, and `inline` functions to avoid call overhead. C has none of those. Instead, C leans heavily on the **preprocessor** — the `#define` macro system that rewrites your source code *before* the compiler sees it.

Macros are pure textual substitution. The preprocessor does not know about types, scope, or expressions — it just replaces text. This makes macros powerful and flexible, but also a source of subtle bugs if you are not careful.

Object-Like Macros

The simplest macros define named constants:

```
#define MAX_BUF   1024
#define PI        3.14159265
#define GREETING  "Hola, amigo"
```

Everywhere the preprocessor sees `MAX_BUF`, it replaces it with `1024`. No semicolons — a common mistake is writing `#define MAX_BUF 1024;`, which would paste `1024;` everywhere, breaking expressions like `malloc(MAX_BUF * sizeof(int))`.

> **Trap:** Do not put a semicolon at the end of a `#define`. The semicolon becomes part of the replacement text and will cause surprising errors.

Conditional Compilation

Macros also control which code the compiler sees:

```
#define DEBUG

#ifdef DEBUG
    printf("x = %d\n", x);
#endif
```

`#ifdef` checks whether a macro is defined (regardless of its value). Its complement `#ifndef` checks that a macro is *not* defined. You can also use `#if`, `#elif`, and `#else` for more complex conditions:

```
#if VERBOSE_LEVEL >= 2
    printf("Detailed trace...\n");
#elif VERBOSE_LEVEL == 1
    printf("Basic trace...\n");
#else
    /* no tracing */
#endif
```

Include Guards

The most common use of `#ifndef` is protecting header files from being included more than once:

```
/* myheader.h */
#ifndef MYHEADER_H
#define MYHEADER_H

struct point {
    int x, y;
};

void draw_point(struct point p);

#endif /* MYHEADER_H */
```

The first time `myheader.h` is included, `MYHEADER_H` is not defined, so the contents are processed and `MYHEADER_H` gets defined. Any subsequent include finds `MYHEADER_H` already defined and skips the entire file.

> **Tip:** Many compilers support `#pragma once` as a non-standard alternative to include guards. It is simpler to write but not portable to all compilers. When in doubt, use the `#ifndef` guard — it works everywhere.

Function-Like Macros

Macros can take parameters, making them look like functions:

```
#define SQUARE(x)   ((x) * (x))
#define MAX(a, b)   ((a) > (b) ? (a) : (b))
#define ABS(x)      ((x) < 0 ? -(x) : (x))
```

But they are *not* functions — they are text substitution with parameter placeholders. This distinction matters.

The Parenthesization Rules

Always parenthesize every parameter use *and* the entire macro body:

```
/* Wrong: */
#define SQUARE(x) x * x

/* SQUARE(1 + 2) expands to: 1 + 2 * 1 + 2 = 5 (not 9!) */

/* Right: */
#define SQUARE(x) ((x) * (x))

/* SQUARE(1 + 2) expands to: ((1 + 2) * (1 + 2)) = 9 */
```

Without parentheses, operator precedence in the surrounding expression can silently rearrange the computation.

The Double-Evaluation Trap

Since macros substitute text, each parameter reference evaluates the argument again:

```
#define SQUARE(x) ((x) * (x))

int i = 3;
int result = SQUARE(i++);
/* Expands to: ((i++) * (i++)) - i is incremented TWICE */
/* Undefined behavior: two unsequenced modifications of i */
```

A real function evaluates its argument once. A macro evaluates it once per appearance in the replacement text. This is the most important difference between macros and functions.

> **Trap:** Never pass expressions with side effects (like `i++`, `f()`, or assignment) to function-like macros. The expression will be evaluated multiple times, producing unexpected results or undefined behavior.

Multi-Statement Macros: `do { ... } while (0)`

If a macro needs to execute multiple statements, wrap them in `do { ... } while (0)`:

```
#define SWAP(a, b) do { \
    int tmp = (a);      \
    (a) = (b);          \
    (b) = tmp;          \
} while (0)
```

Why not just use braces? Consider:

```
if (x > y)
    SWAP(x, y);
else
    printf("Already sorted\n");
```

If `SWAP` expanded to a bare `{ ... }`, the semicolon after `SWAP(x, y)` would terminate the `if` statement, and the `else` would become a syntax error. The `do { ... } while (0)` idiom creates a single statement that works correctly with semicolons and control flow.

Tip: The `do { ... } while (0)` pattern is everywhere in C codebases. It looks odd at first, but it is the standard way to make multi-statement macros behave like ordinary statements.

Stringification and Token Pasting

The preprocessor has two special operators for macro arguments.

Stringification: `#`

The `#` operator turns a macro argument into a string literal:

```
#define PRINT_VAR(x) printf(#x " = %d\n", x)

int score = 42;
PRINT_VAR(score);
/* Expands to: printf("score" " = %d\n", score); */
/* Adjacent string literals are concatenated: "score = %d\n" */
/* Output: score = 42 */
```

This is useful for debug macros where you want to print both the variable name and its value.

Token Pasting: `##`

The `##` operator joins two tokens into one:

```
#define DECLARE_PAIR(type) \
    type type##_first;     \
    type type##_second;

DECLARE_PAIR(int)
/* Expands to:
   int int_first;
   int int_second;
*/
```

Token pasting is commonly used to generate families of related variables or functions from a single macro.

Variadic Macros

Macros can accept a variable number of arguments using `...` and `__VA_ARGS__`:

```
#define LOG(fmt, ...) fprintf(stderr, "[LOG] " fmt "\n", __VA_ARGS__)

LOG("score is %d", 42);
/* Expands to: fprintf(stderr, "[LOG] " "score is %d" "\n", 42); */
```

This is commonly used to wrap `printf`-style functions with extra decoration like timestamps or log levels.

> **Tip:** When `__VA_ARGS__` is empty, the trailing comma before it can cause a compilation error. GNU C provides `##__VA_ARGS__` which swallows the comma when the argument list is empty:
>
> ```
> #define LOG(fmt, ...) fprintf(stderr, "[LOG] " fmt "\n", ##__VA_ARGS__)
> LOG("started");  /* No extra args - comma is removed */
> ```
>
> This is a GCC/Clang extension. C23 standardizes this behavior with `__VA_OPT__`.

Multi-Level Expansion

Macros can expand to other macros, and the preprocessor **rescans** the result to expand again. But the `#` and `##` operators are special — they operate on the raw argument text *before* any expansion happens.

```
#define MAX_BUF 1024
#define STRINGIFY(x)  #x
#define XSTRINGIFY(x) STRINGIFY(x)

printf("%s\n", STRINGIFY(MAX_BUF));
/* # operates before expansion: prints "MAX_BUF" */

printf("%s\n", XSTRINGIFY(MAX_BUF));
/* First pass: XSTRINGIFY(MAX_BUF) → STRINGIFY(1024) */
/* Rescan:     STRINGIFY(1024) → "1024" */
/* Prints "1024" */
```

`STRINGIFY(MAX_BUF)` gives `"MAX_BUF"` because `#` stringifies its argument before expansion. `XSTRINGIFY(MAX_BUF)` first expands `MAX_BUF` to `1024` (since the outer macro does not use `#` directly), then passes `1024` to `STRINGIFY`, producing `"1024"`.

This two-level indirect pattern is used whenever you need the *expanded* value of a macro as a string.

> **Tip:** Whenever you need a macro's expanded value as a string, use the two-level indirect pattern. It comes up often when embedding version numbers or configuration values in strings.

X-Macros

X-macros are a technique for defining a list of items once and expanding it in multiple ways. The idea: define the list as a macro that calls an unspecified "action" macro on each item, then define that action differently for each use.

Here is a concrete example that generates both an enum and a string table from a single list of log levels:

```c
#include <stdio.h>

/* Define the list once */
#define LOG_LEVELS(X) \
    X(LOG_DEBUG)      \
    X(LOG_INFO)       \
    X(LOG_WARN)       \
    X(LOG_ERROR)      \
    X(LOG_FATAL)

/* Generate the enum */
#define AS_ENUM(name) name,
enum log_level { LOG_LEVELS(AS_ENUM) LOG_COUNT };

/* Generate the string table */
#define AS_STRING(name) #name,
const char *log_level_names[] = { LOG_LEVELS(AS_STRING) };

int main(void) {
    for (int i = 0; i < LOG_COUNT; i++) {
        printf("%d = %s\n", i, log_level_names[i]);
    }
    return 0;
}
```

Output:

```
0 = LOG_DEBUG
1 = LOG_INFO
2 = LOG_WARN
3 = LOG_ERROR
4 = LOG_FATAL
```

Add a new log level? Add one line to `LOG_LEVELS` and the enum and string table stay in sync automatically. Without X-macros, you would need to update both the enum and the string array separately — and hope you never forget one.

> **Tip:** X-macros are one of the preprocessor's most powerful patterns. You will see them in real codebases for error codes, command tables, and state machines. The key advantage: a single source of truth for a list of items.

Key Points

- Macros are **textual substitution** performed before compilation. They are not functions and do not respect scope or type rules.
- Object-like macros define constants and feature flags. Never end a `#define` with a semicolon.
- Function-like macros must have every parameter use and the entire body parenthesized to avoid precedence bugs.
- Macro arguments are evaluated each time they appear — do not pass expressions with side effects.
- Use `do { ... } while (0)` for multi-statement macros so they work correctly with `if`/`else` and semicolons.
- The `#` operator stringifies a macro argument; `##` pastes tokens together.
- `#` and `##` prevent argument expansion. Use the two-level indirect pattern (e.g., `XSTRINGIFY`/`STRINGIFY`) when you need the expanded value.
- `__VA_ARGS__` enables variadic macros for wrapping `printf`-style functions.
- X-macros define a list once and expand it multiple ways, keeping enums and string tables in sync.

Exercises

1. **Think about it:** C++ uses `constexpr` and `inline` functions to replace many uses of macros. What specific problems do macros have that these C++ features solve? Why does C still rely on macros despite these problems?

2. **What does this produce?**

```c
#define DOUBLE(x) ((x) + (x))

int i = 5;
printf("%d\n", DOUBLE(i++));
```

3. **Calculation:** Given the macro `#define BUFSIZE 256`, how many bytes does `char buf[BUFSIZE + 1]` allocate? Why is the `+ 1` a common pattern?

4. **Where is the bug?**

```c
#define MUL(a, b)  a * b

int result = MUL(2 + 3, 4 + 5);
printf("%d\n", result);
```

5. **What does this produce?**

```c
#define STRINGIFY(x)  #x
#define XSTRINGIFY(x) STRINGIFY(x)
#define VERSION 3
```

```
printf("[%s] [%s]\n", STRINGIFY(VERSION), XSTRINGIFY(VERSION));
```

6. **Where is the bug?**

```
#define LOG_IF(cond, msg) \
    if (cond) \
        printf("[WARN] %s\n", msg);

if (x > 100)
    LOG_IF(x > 200, "very high");
else
    printf("normal\n");
```

7. **Write a program** that defines an X-macro list of at least four colors, then uses it to generate both an `enum` and a function that returns the string name for a given enum value. Print each color's enum value and name.