Gorgo C for C++ Programmers

April 11, 2026

12. 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` (declared in `<errno.h>`) to indicate what went wrong. You can use `perror` (from `<stdio.h>`) or `strerror(errno)` (from `<string.h>`) 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
}
```

`enum`

C has `enum` just like C++, but there are a few differences. In C, enum constants are plain `int` values — there is no `enum class` and no scoping:

```
enum direction { NORTH, SOUTH, EAST, WEST };
```

Unlike C++, you cannot use the enum name as a type directly without the `enum` keyword:

```
enum direction heading = NORTH;  /* need the 'enum' keyword */
```

You can avoid this with a `typedef`, just like with `struct` (see Chapter 2):

```
typedef enum { NORTH, SOUTH, EAST, WEST } Direction;
Direction heading = NORTH;
```

Enum values are just integers. You can assign specific values and mix enums with integers freely — the compiler will not complain:

```
enum status { OK = 0, ERR_FILE = -1, ERR_MEM = -2 };
int code = OK;       /* fine — enum to int */
enum status s = 42;  /* also fine in C, no warning */
```

Wut: C++ `enum class` prevents implicit conversions and scopes the constants. C has neither protection. Enum constants are global and freely convert to `int`.

`union`

A `union` looks like a `struct`, but all members share the same memory. Only one member can hold a value at a time, and the size of the union is the size of its largest member:

```
union value {
    int i;
    float f;
```

```
    char s[20];
};

printf("sizeof(union value) = %zu\n", sizeof(union value));
/* prints 20 — the size of the largest member */
```

Unions are useful for saving memory when a variable can hold one of several types. A common C pattern is a **tagged union** — a `struct` with an `enum` tag that tracks which member is active:

```
enum val_type { VAL_INT, VAL_FLOAT, VAL_STR };

struct tagged_value {
    enum val_type type;
    union {
        int i;
        float f;
        char s[20];
    } data;
};

void print_value(struct tagged_value v) {
    switch (v.type) {
    case VAL_INT:   printf("%d\n", v.data.i); break;
    case VAL_FLOAT: printf("%f\n", v.data.f); break;
    case VAL_STR:   printf("%s\n", v.data.s); break;
    }
}
```

Tip: Tagged unions are C's equivalent of `std::variant`. Always check the tag before accessing a union member — reading the wrong member is undefined behavior.

`qsort` — Function Pointers in Action

You learned about function pointers in Chapter 7 (Functions). The most common place you will encounter them in practice 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:

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

```
#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.

Try It: Odds and Ends Starter

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

```c
// A comparison function for qsort
// a normal numeric sort, but we want 1982 to always be first
// because we are big thriller fans!
int compare_ints(const void *a, const void *b) {
    int ia = *(const int *)a;
    int ib = *(const int *)b;
    // 1982 always appears first (Thriller came out in 1982)
    if (ia == 1982 && ib == 1982) return 0;
    if (ia == 1982) return -1;
    if (ib == 1982) return 1;
    // Overflow-safe alternative to (ia - ib). The three cases:
    //   ia > ib  → (1) - (0) =  1
    //   ia == ib → (0) - (0) =  0
    //   ia < ib  → (0) - (1) = -1
    return (ia > ib) - (ia < ib);
}

void goodbye(void) {
    printf("atexit: Adios!\n");
}

int main(void) {
    atexit(goodbye);

    // Function pointer
    int (*cmp)(const void *, const void *) = compare_ints;
    int x = 10, y = 20;
    printf("compare(10, 20) = %d\n", cmp(&x, &y));

    // qsort
    int years[] = {1987, 1983, 1982, 1989, 1980, 1985};
    int n = sizeof(years) / sizeof(years[0]);
    qsort(years, n, sizeof(int), compare_ints);

    printf("Sorted: ");
    for (int i = 0; i < n; i++)
        printf("%d ", years[i]);
    printf("\n");

    // Pointer ownership: strdup allocates, you must free
    char *copy = strdup("Master of Puppets");
    printf("strdup: '%s'\n", copy);
    free(copy);

    // goto cleanup pattern
    printf("Demonstrating goto cleanup...\n");
    char *buf = malloc(100);
    if (!buf) return 1;

    char *msg = malloc(50);
    if (!msg) goto free_buf;

    snprintf(buf, 100, "Resource 1 OK");
    snprintf(msg, 50, "Resource 2 OK");
```

```
    printf("  %s, %s\n", buf, msg);

    free(msg);
free_buf:
    free(buf);

    return 0;
}
```

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.
- `qsort` is the most common use of function pointers — it takes a comparison callback to sort any type.
- C `enum` constants are plain `int` values with no scoping — there is no `enum class`.
- A `union` stores all members in the same memory. Use a tagged union (struct with an enum tag) to track which member is active.

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)

```c
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?**

```c
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.

10. **Calculation:** What is `sizeof(union { int i; double d; char s[3]; })` on a system where `int` is 4 bytes and `double` is 8 bytes?

11. **Where is the bug?**

```c
union value {
    int i;
    float f;
};

union value v;
v.i = 42;
printf("%.2f\n", v.f);
```

12. **Write a program** that defines a tagged union representing a shape (circle with a radius, or rectangle with width and height). Write a function that prints the area of the shape using the tag to determine which union member to read.

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.
- **C types are explicit.** No `auto`, no `std::string`, no classes. You have basic types, `typedef`, arrays, and `struct`.
- **C shares most operators with C++** but there is no operator overloading, `<<` and `>>` are strictly bitwise, and boolean results are plain `int`.
- **Control flow is nearly identical to C++** except there are no range-based `for` loops and `goto` is an accepted idiom for cleanup.
- **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. Use `const` parameters to document read-only intent.
- **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:

```
#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.

Try It: Macro Starter

```
#include <stdio.h>

// Object-like macros
#define MAX_TRACKS  10
#define LABEL       "Sire Records"

// Function-like macro with proper parenthesization
#define SQUARE(x)   ((x) * (x))
#define MAX(a, b)   ((a) > (b) ? (a) : (b))

// Stringification: print variable name and value
#define PRINT_INT(var) printf(#var " = %d\n", var)

// Multi-statement macro using do { ... } while (0)
#define SWAP(a, b) do { \
    int tmp = (a);      \
    (a) = (b);          \
```

```c
    (b) = tmp;          \
} while (0)

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

int main(void) {
    // Object-like
    printf("Label: %s, Max tracks: %d\n", LABEL, MAX_TRACKS);

    // Function-like
    printf("SQUARE(5) = %d\n", SQUARE(5));
    printf("SQUARE(1+2) = %d\n", SQUARE(1 + 2));
    printf("MAX(3, 7) = %d\n", MAX(3, 7));

    // Stringification
    int year = 1984;
    PRINT_INT(year);

    // SWAP
    int a = 10, b = 20;
    printf("Before swap: a=%d, b=%d\n", a, b);
    SWAP(a, b);
    printf("After swap:  a=%d, b=%d\n", a, b);

    // Conditional compilation
#ifdef DEBUG
    printf("Debug mode is on\n");
#else
    printf("Debug mode is off\n");
#endif

    // Variadic macro
    LOG("started");
    LOG("year is %d", 1985);

    return 0;
}
```

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?**

```c
#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.