Gorgo C for C++ Programmers

June 10, 2026

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 no templates, its `constexpr` only arrived in C23 (in a limited form), and `inline` (added in C99) is little more than a hint. 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); */
```

```c
/* 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:

```c
#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__`:

```c
#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 — `##__VA_ARGS__` itself was never standardized. C23 solves the same problem portably with `__VA_OPT__` (e.g. `fmt "\n" __VA_OPT__(,) __VA_ARGS__`) and also allows passing zero variadic arguments.

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.

```c
#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.

Try It: Macro Starter

```c
#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);          \
    (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");
```

```c
#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?**

```
#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.