C / C++ — ECE Master Cheat Sheet

C → Embedded C → Firmware → Hardware → C++ · Pointers · Memory · Bit manipulation · Registers · ISRs · 60+ traps, output puzzles and interview Q&A

BEHAVIOUR D deterministic I implementation-defined U unspecified UB undefined // ISO C11/C17 · ISO C++17 // CODE: keyword · type · literal · MACRO · comment

// §1 C FUNDAMENTALS — SKELETON, TYPES, LITERALS

1.1 Minimal program & the toolchain

```c
#include <stdint.h>       // fixed-width integer types
#include <stdio.h>        // printf: hosted targets only

#define LED_PIN 5U        // macro: no type, no storage, textual paste

int main(void) {          // (void) = takes NO arguments
    uint32_t x = 0U;      // ALWAYS initialise
    return 0;             // 0 = success
}
```

```
file.c --cpp--> file.i --cc1--> file.s --as--> file.o
 #include        C -> asm        asm -> obj       |
 #define                                          v
                          ld + linker script + libs
                                                  |
                                                  v
                  app.elf --objcopy--> app.bin / app.hex
```

Stage	Does	Typical flag
Preprocess	macros, `#include`, conditionals	`gcc -E`
Compile	C → assembly, optimisation	`gcc -S -O2`
Assemble	assembly → object	`gcc -c`
Link	resolve symbols, place sections	`-T link.ld -Wl,-Map=a.map`

Embedded build flags: `-Wall -Wextra -Werror` (treat warnings as errors), `-Os` (size), `-g3`, `-ffunction-sections -fdata-sections -Wl,--gc-sections` (drop unused code), `-fno-exceptions -fno-rtti` (C++ on MCU).

1.2 Data types — sizes are NOT fixed by the standard

Type	Typical size	Standard guarantee	Use
`char`	1 B	`sizeof` is **exactly 1**, ≥ 8 bits	byte / character
`short`	2 B	≥ 16 bits	small integer
`int`	4 B (2 B on AVR/MSP430)	≥ 16 bits, "natural" word	default integer
`long`	4 B Win / 8 B Linux-64	≥ 32 bits	wider integer
`long long`	8 B	≥ 64 bits	large integer
`float`	4 B	format not mandated (usually IEEE-754)	single precision
`double`	8 B	≥ precision of `float`	double precision
`void`	—	incomplete type, no objects	no value / generic ptr

Traps: plain `char` is a **third type**, distinct from `signed char` and `unsigned char`; its signedness is implementation-defined (signed on x86-GCC, **unsigned on ARM-GCC**). `sizeof(int)` is not portable — never assume 4. `CHAR_BIT` (bits per byte) is 8 on every mainstream target but is not fixed at 8 by the standard.

1.3 Fixed-width types — the embedded default

```c
#include <stdint.h>
uint8_t  u8;   int8_t  s8;   // exactly  8 bits, no padding
uint16_t u16;  int16_t s16;  // exactly 16 bits
uint32_t u32;  int32_t s32;  // 32 bits = Cortex-M reg width
uint64_t u64;  int64_t s64;  // exactly 64 bits
uint_fast8_t   f8;           // >= 8 bits, fastest on this CPU
uint_least16_t l16;          // smallest type with >= 16 bits
size_t     n;                // unsigned; result of sizeof
ptrdiff_t d;                 // signed; result of pointer - pointer
uintptr_t a;                 // unsigned int that can hold a void*
```

RULE: in firmware use `uintN_t` everywhere a width matters (registers, protocol frames, DMA buffers). `uintN_t` are **optional** in the standard but exist on every practical target; `uint_leastN_t` / `uint_fastN_t` are mandatory.

1.4 Literals, constants & qualified objects

```c
42     0x2A    052     0b101010  // dec, hex, OCTAL(!), binary
42U    42L     42UL    42ULL     // unsigned / long suffixes
3.14   3.14f   3.14e-2           // double, float, scientific
'A'    '\n'    '\0'    '\x41'    // character literals
"text"    "abc" "def"            // adjacent literals concatenate

// ---- three ways to name a constant ----
#define MAX  100       // no type, no scope, no debug symbol
const int kMax = 100;  // real object: typed, scoped, debuggable
enum { ARR_MAX = 100 };  // int constant, sizes an array in C
```

Trap — leading zero is OCTAL: `010` is **8**, not 10. Bit masks written as `0755` or dates as `09` silently change value (`09` is not even valid octal → compile error).

C vs C++: `'A'` has type `int` in C but `char` in C++, so `sizeof('A')` is typically **4 in C** and **1 in C++**. In C a `const int` is *not* a compile-time constant expression (cannot size an array in C89, cannot be a `case` label); in C++ it is.

1.5 printf / scanf format specifiers

Spec	Type	Spec	Type
`%d %i`	`int` (signed decimal)	`%u`	`unsigned int`
`%x %X`	unsigned hex	`%o`	unsigned octal
`%c`	character (passed as `int`)	`%s`	`char *`, NUL-terminated
`%f %e %g`	`double` (`float` promotes)	`%p`	`void *` — cast it
`%ld %lld`	`long` / `long long`	`%zu`	`size_t`
`%hhu %hu`	`unsigned char` / `short`	`%%`	literal %
`%08lX`	8 digits, zero-pad, upper hex	`%-10s`	left-justify in 10 cols

```c
#include <inttypes.h>
uint32_t r = 0xDEADBEEF;
printf("reg = 0x%08" PRIX32 "\n", r);  // width-correct, portable
printf("size = %zu\n", sizeof r);      // %zu for size_t, not %d
```

Trap: a mismatched specifier is **undefined behaviour**, not a wrong number — `printf("%d", 1.0)` or `%d` with a `long` can print garbage or crash. In variadic calls `float` promotes to `double` and small ints promote to `int`; `%f` covers both `float` and `double`. Many MCU libcs need `-u _printf_float` to print floats at all.

// §2 OPERATORS, PRECEDENCE & CONTROL FLOW

2.1 Master operator table

Group	Operators	One-line meaning
Arithmetic	`+ - * / %`	`%` is integer-only; `/` on two ints is integer division
Relational	`< > <= >=`	yield `int` 0 or 1 in C
Equality	`== !=`	value comparison — never use on `float`
Logical	`&& || !`	truth values; **short-circuit**, sequence point after left operand
Bitwise	`& | ^ ~`	per-bit AND / OR / XOR / NOT
Shift	`<< >>`	left / right shift by a bit count
Assign	`= += -= *= /= %= &= |= ^= <<= >>=`	compound forms evaluate the left operand once
Inc / dec	`++ --`	prefix returns new value, postfix returns old
Ternary	`c ? a : b`	only `c` plus one branch is evaluated
Sizeof	`sizeof`	compile-time size in bytes, type `size_t`
Pointer	`& *`	address-of / dereference
Member	`. ->`	object member / pointer-to-object member
Comma	`a, b`	evaluate `a`, discard it, result is `b`

The eight symbols interviewers test: `&` bitwise AND vs `&&` logical AND · `|` bitwise OR vs `||` logical OR · `~` bitwise NOT vs `!` logical NOT · `=` assignment vs `==` comparison.
`~1` is `0xFFFFFFFE` but `!1` is `0`. `2 & 4` is **0** but `2 && 4` is **1**.

2.2 Precedence — the levels that actually bite

Lvl	Operators (highest first)	Assoc
1	`() [] -> .` postfix `++ --`	L → R
2	unary `! ~ ++ -- + - * & sizeof (cast)`	R → L
3	`* / %`	L → R
4	`+ -`	L → R

Lvl	Operators (highest first)	Assoc
5	<< >>	L → R
6	< <= > >=	L → R
7	== !=	L → R
8–10	& then ^ then |	L → R
11–12	&& then ||	L → R
13	?:	R → L
14	= and all compound assignments	R → L
15	,	L → R

```
x & 1 == 0      // == BINDS TIGHTER: this is  x & (1 == 0)  =  0
(x & 1) == 0    // what you meant

1 << 2 + 3      // + BINDS TIGHTER: this is  1 << 5  =  32
(1 << 2) + 3    // what you meant

*p++            // == *(p++) : read *p, THEN advance p
(*p)++          // increment the pointed-to object
*++p            // advance first, then read
```

Golden rule: bitwise `& ^ |` sit **below** `==` and `!=`. Parenthesise every mixed bitwise/comparison expression — this single trap accounts for a large share of embedded OA mistakes.

2.3 `i++` vs `++i`, `=` vs `==`

Form	Value of the expression	Effect on `i`
`i++`	old value of `i`	incremented
`++i`	new value of `i`	incremented

```
int i = 5, j;
j = i++;      // j == 5, i == 6
i = 5;
j = ++i;      // j == 6, i == 6
```

Interview answer: for a plain `int` both compile to identical code as a loop step — the optimiser removes the unused temporary, so **"++i is faster" is false for scalars**. It matters for C++ iterators/heavy classes, where `it++` must copy the old object.

2.4 Control flow — syntax in one block

```
if (c) { ... } else if (d) { ... } else { ... }

switch (v) {                // v must be an integer type or enum
    case 1:  a(); break;    // NO break -> falls through to case 2
    case 2:
    case 3:  b(); break;    // shared body for 2 and 3
    default: c(); break;
}

for (int i = 0; i < n; i++) { ... }   // init ; test ; step
while (cond)      { ... }             // test  0..n times
do { ... } while (cond);              // body runs AT LEAST ONCE
for (;;) { ... }                      // infinite super-loop

break;     // leave innermost loop OR switch
continue;  // next iteration (in a for: the step still runs)
goto err;  // legitimate for single-exit error cleanup in C
```

Top control-flow traps

- `if (x = 5)` assigns then tests 5 → always true. Use `-Wall`, or write `if (5 == x)`.
- `if (cond);` — stray semicolon makes an empty body; the block always runs.
- Missing `break` → fall-through (sometimes intentional: mark it `/* FALLTHRU */`).
- `for (unsigned i = n-1; i >= 0; i--)` **never ends** — unsigned is always `>= 0`, it wraps to `UINT_MAX`.
- Off-by-one: `i <= n` with an `n`-element array writes one past the end (UB).
- `switch` on a `float` or `char *` is a compile error; `case` labels must be constant expressions.
- A variable declared inside a `case` without braces is a scope error in C.

// §3 FUNCTIONS & FUNCTION POINTERS

3.1 Declaration, definition, linkage

```
int  add(int a, int b);        // DECLARATION (prototype) - in the .h
int  add(int a, int b) {       // DEFINITION - in the .c
    return a + b;
}
static void helper(void);      // internal linkage: private to this .c
extern uint32_t g_ticks;       // declares an object DEFINED elsewhere
inline int sq(int x){return x*x;}   // a hint, not a guarantee
void no_args(void);            // C: (void) = no parameters
void old_style();              // C: unspecified args - AVOID
```

Keyword on a function	Meaning
`static`	internal linkage — invisible to the linker outside this translation unit; avoids name clashes and enables aggressive inlining
`extern`	external linkage (the default for functions) — defined in another TU
`inline`	request to inline; the compiler decides. In C99 needs an `extern` definition in exactly one TU; `static inline` in a header is the portable idiom

3.2 Pass-by-value — C has nothing else

```
void bad (int  x) { x = 99; }   // modifies a COPY, caller unaffected
void good(int *x) { *x = 99; }  // pass the ADDRESS instead
int v = 0;
bad(v);    // v == 0
good(&v);  // v == 99
```

RULE: C always copies the argument. "Passing an array" copies only the **pointer** to its first element (§5.3), which is why arrays look like by-reference and scalars do not.

3.3 Recursion & the stack

```
uint32_t fact(uint32_t n){ return (n < 2U) ? 1U : n * fact(n - 1U); }
```

Embedded view: every call costs a stack frame (locals + return address + saved registers). Deep or unbounded recursion overflows a 1–8 KB MCU stack and corrupts RAM **silently**. Safety-critical rules (MISRA C 17.2) **ban recursion**. Prefer iteration; if you must recurse, bound the depth.

3.4 Function pointers — dispatch tables, callbacks, vector tables

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

int (*fp)(int, int) = add;   // read inside-out: fp is a POINTER to
                             // a FUNCTION (int,int) returning int
int r = fp(2, 3);            // 5   -- (*fp)(2,3) is equally legal
fp = &sub;                   // the & is optional on both sides

typedef int (*op_t)(int, int);      // ALWAYS typedef it
static const op_t OPS[2] = { add, sub };
int q = OPS[1](9, 4);        // 5 : jump table, no if/else chain
```

```
// ---- callback: the driver calls UP into application code ----
typedef void (*rx_cb_t)(uint8_t byte);
static rx_cb_t s_on_rx;               // registered handler

void uart_set_rx_cb(rx_cb_t cb) { s_on_rx = cb; }

void USART1_IRQHandler(void) {        // ISR
    uint8_t b = (uint8_t)USART1->RDR;
    if (s_on_rx != NULL) {            // ALWAYS null-check a callback
        s_on_rx(b);
    }
}
```

Declaration traps: `int *f(void)` is a function returning `int *`; `int (*f)(void)` is a pointer to a function. `void (*vec[16])(void)` is an array of 16 function pointers — exactly how an ARM Cortex-M interrupt vector table is declared. A function pointer is **not** a `void *`: converting between object and function pointers is not guaranteed by the standard.

// §4 ARRAYS — LAYOUT, DECAY, PASSING

4.1 Declaration & initialisation

```
int   a[5];                 // 5 ints, INDETERMINATE values
int   b[5] = {1, 2, 3};     // b = {1,2,3,0,0} - the rest zeroed
int   c[]  = {1, 2, 3};     // size deduced = 3
int   d[5] = {0};           // whole array zeroed (idiom)
static int e[5];            // static storage -> zeroed at startup
uint8_t f[64] = {[63]=1};   // C99 designated initialiser

a[0] = 1;                   // indices run 0 .. N-1
size_t n = sizeof a / sizeof a[0];  // ONLY where a is an array
```

```
int a[5];   contiguous, no gaps, ascending addresses

 addr:  0x100   0x104   0x108   0x10C   0x110
        +-------+-------+-------+-------+-------+
  a =   | a[0]  | a[1]  | a[2]  | a[3]  | a[4]  |
        +-------+-------+-------+-------+-------+
        ^                                       ^
        a == &a[0]                      a+5 = one-past-the-end:
                                        legal to form, UB to deref
```

4.2 Array-to-pointer decay — the #1 C interview topic

RULE: in almost every expression an array **decays** to a pointer to its first element. The three exceptions: `sizeof a`, `&a`, and a string literal initialising a `char` array.

```
int a[5];
a          // decays to  &a[0],  type int *
sizeof a   // 20 bytes   (no decay)      type of &a  is  int (*)[5]
a + 1      // address + 4 bytes  (one int)
&a + 1     // address + 20 bytes (one whole ARRAY of 5)
a[i] == *(a+i) == *(i+a) == i[a]   // subscript is commutative
```

IMPORTANT — array ≠ pointer. An array is *N contiguous objects*; a pointer is *one object holding an address*. `sizeof(array)` ≠ `sizeof(pointer)`. Declaring `extern int a[];` in one file when the definition is `int *a;` in another compiles and then corrupts memory at run time.

4.3 Passing an array to a function

```
void f(int a[5]) {              // BUG: a is an int*, not an array
    size_t n = sizeof a / sizeof a[0];  // n == 1 or 2, never 5
}
void g(int *a, size_t n) { }    // CORRECT: pass the length too

int a[5];
g(a, sizeof a / sizeof a[0]);  // count it where a IS an array
```

Trap: in a parameter list `int a[5]`, `int a[]` and `int *a` are the **same declaration** — the size is discarded by the compiler. Always pass the element count alongside the pointer. (C99 `int a[static 5]` documents "at least 5 elements" but still passes a pointer.)

4.4 Multidimensional arrays

```
int m[2][3] = {{1,2,3},{4,5,6}};  // ROW-MAJOR: 1 2 3 4 5 6
m[i][j] == *(*(m + i) + j);       // m+i steps a ROW (12 B)
sizeof m     // 24    sizeof m[0]  // 12    sizeof m[0][0]  // 4

void f(int m[][3],  size_t rows); // every dim EXCEPT the first
void f(int (*m)[3], size_t rows); // same: pointer to array-of-3
```

Cache / DMA note: iterate the **last** index in the inner loop (`for i { for j { m[i][j] } }`) to walk memory linearly. The reverse order strides across rows and destroys cache and burst-DMA efficiency.

4.5 Pointer-array declarations you must read correctly

Declaration	Reads as	`sizeof` (64-bit)
`int *a[5]`	array of 5 pointers to int	40
`int (*a)[5]`	pointer to an array of 5 int	8
`int **a`	pointer to pointer to int	8
`int *f(void)`	function returning `int *`	n/a
`int (*f)(void)`	pointer to function returning int	8

Right-left rule: start at the identifier, go right while you can, then left; `()` and `[]` bind tighter than `*`.

// §5 STRINGS & MEMORY FUNCTIONS

5.1 A C string is a convention, not a type

```
char s[6] = "hello";  // 'h','e','l','l','o','\0' -> 6 bytes
char t[]  = "hello";  // size 6, MODIFIABLE copy (.data / stack)
char *p   = "hello";  // points at a STRING LITERAL -> read-only
p[0] = 'H';           // UNDEFINED BEHAVIOUR (often a hard fault)
t[0] = 'H';           // fine

strlen(t)   // 5 - counts up to the NUL, at run time, O(n)
sizeof t    // 6 - includes the NUL, at compile time
sizeof p    // 8 - the POINTER, not the text
```

```
 t:  +---+---+---+---+---+----+
     | h | e | l | l | o | \0 |   <-- the NUL ENDS the string
     +---+---+---+---+---+----+
       0   1   2   3   4    5     lose it and str* runs off the end
```

5.2 <string.h> — what each one really does

Function	Purpose	Danger
`strlen(s)`	length before NUL	UB if s is not NUL-terminated
`strcpy(d,s)`	copy incl. NUL	**no bound** — classic overflow
`strncpy(d,s,n)`	copy at most n bytes	**may not NUL-terminate**; pads with NULs if short
`strcmp(a,b)`	`<0 / 0 / >0` ordering	0 means EQUAL — not "false"
`strcat / strncat`	append	O(n) rescan each call; n excludes the NUL
`snprintf(d,n,...)`	formatted, bounded	always NUL-terminates; returns the length it *wanted*
`memcpy(d,s,n)`	copy n bytes	**UB if the regions overlap**
`memmove(d,s,n)`	copy n bytes	overlap-safe (acts as if via a temp buffer)
`memset(d,c,n)`	fill n bytes with `(unsigned char)c`	cannot fill ints with 1s; fills BYTES
`memcmp(a,b,n)`	compare n raw bytes	compares **padding** too — never on structs

```
char dst[8];
strcpy (dst, "0123456789");    // OVERFLOW: 11 bytes into 8
strncpy(dst, "0123456789", sizeof dst);  // no NUL -> still broken
snprintf(dst, sizeof dst, "%s", src);    // BEST: bounded + NUL

int a[4];
memset(a, 0, sizeof a);   // OK: all-zero bytes = integer 0
memset(a, 1, sizeof a);   // byte 0x01 -> each int = 0x01010101
```

Buffer overflow = writing past the end of a buffer. It silently overwrites adjacent variables, saved registers or the return address (stack smashing). Rules: bound every copy, size with `sizeof dst`, prefer `snprintf` / `memcpy` with an explicit length, and never compute a length from attacker- or sensor-supplied data without clamping it.

memcpy vs memmove: `memcpy` may copy forwards, backwards or in wide blocks, so overlapping regions are UB even though it often "works". Shifting a ring-buffer window or scrolling a framebuffer overlaps → use `memmove`.

// §6 POINTERS — THE CORE OF EMBEDDED C

6.1 Address, dereference, the mental model

```
int  x = 10;      // an object
int *p = &x;      // p HOLDS the address of x   & = address-of
int  y = *p;      // y == 10                    * = dereference
*p = 20;          // writes THROUGH p -> x is now 20
```

```
       p                          x
  +------------+              +---------+
  | 0x2000A014 | ---------->  |   20    |   <- the object
  +------------+              +---------+
  at 0x2000A00C                int, 4 bytes

  &x = 0x2000A014  (address OF x)     p  = 0x2000A014  (same)
  &p = 0x2000A00C  (address OF p)     *p = 20      (the object)
  sizeof p = 4 or 8  (address width, NOT the size of x)
```

Written	Means
`int *p;`	p is a pointer to int (declaration — * is part of the type)
`*p`	the object p points at (expression — * is dereference)
`&x`	the address of x, type `int *`
`int *a, b;`	**Trap:** a is `int *`, b is a plain `int`. Declare one pointer per line.

6.2 Pointer arithmetic — scaled by the pointed-to type

```
int a[4] = {10,20,30,40};
int *p = a;              // p -> a[0]  (decay)
p + 1;                   // + 1 * sizeof(int) = +4 BYTES -> a[1]
p++;  *p;                // 20
p += 2;  *p;             // 40
p - a;                   // 3 : ptrdiff_t, ELEMENTS not bytes

char *cp = (char *)a;
cp + 1;                  // +1 byte: the scale follows the TYPE
```

Expression	Legal?	Result
`p + n, p - n, p++`	yes, inside the array (and one-past-the-end)	scaled address
`p1 - p2`	only if both point into the **same** array	`ptrdiff_t` element distance
`p1 == p2, p1 < p2`	equality always; ordering only within one array	`int` 0/1
`p1 + p2`	**no** — adding two pointers is meaningless	compile error
`*(a + 5)` on `int a[5]`	forming a+5 is legal, **dereferencing is UB**	—

6.3 The pointer taxonomy every interviewer asks for

Kind	What it is	Rule / fix
NULL pointer	guaranteed-invalid value, `((void*)0)`; compares unequal to any object	dereferencing is **UB** — test before every use
Wild pointer	never initialised — holds whatever was in that RAM	always initialise: `int *p = NULL;`
Dangling pointer	pointed-to object's lifetime has **ended** (freed, or a returned local)	`free(p); p = NULL;` — never return `&local`
void pointer	generic address, no type, no size	cannot dereference or do arithmetic; cast before use
Double pointer	`int **pp` — address of a pointer	lets a callee modify the caller's pointer
Function pointer	address of executable code	callbacks, jump tables, ISR vector tables
Dangling after realloc	old pointer invalid if the block moved	use only the returned pointer

```c
int *wild;              // WILD: garbage address, writing corrupts RAM
int *null = NULL;       // safe sentinel
int *dang;
{ int tmp = 5; dang = &tmp; }  // tmp dies at the closing brace
*dang;                         // DANGLING -> undefined behaviour

int *bad(void){ int lo = 5; return &lo; }  // DANGLING pointer
int *ok (void){ static int s = 5; return &s; }  // static: fine
```

MCU reality check: on a PC a NULL dereference traps instantly. On a bare-metal Cortex-M, address `0x00000000` is usually the **vector table in flash** — reading through NULL returns a plausible number and the bug shows up much later, somewhere else. Check pointers explicitly; do not rely on a fault.

6.4 `void *` and double pointers

```c
void *v = &x;        // object pointer -> void* needs no cast in C
int  *ip = v;        // C: implicit. C++: static_cast<int*>(v)
// *v;   v + 1;      // ILLEGAL in ISO C: the size is unknown

void store(void *dst, const void *src, size_t n);  // memcpy

// ---- double pointer: the callee changes the CALLER pointer ----
void alloc_buf(uint8_t **out, size_t n) { *out = malloc(n); }
uint8_t *buf = NULL;
alloc_buf(&buf, 64);  // without ** the caller still sees NULL

char *argv[] = { "cmd", "-v", NULL };  // decays to char **
```

6.5 `const` and pointers — memorise this table

```c
const int *p;          // pointer to const int  (= int const *p)
int *const p = &x;     // const pointer to int
const int *const p=&x; // const pointer to const int
```

Declaration	Change `*p` (data)?	Change `p` (pointer)?	Typical use
`int *p`	YES	YES	general working pointer
`const int *p`	**NO**	YES	read-only input parameter
`int *const p`	YES	**NO**	fixed hardware/buffer address
`const int *const p`	**NO**	**NO**	read-only ROM table

Read it right-to-left from the identifier: `int *const p` → "p is a const pointer to int". The rule of thumb: `const` **to the LEFT of** `*` **protects the data; to the RIGHT of** `*` **it protects the pointer.** `const` only blocks modification *through that access path* — casting it away and writing to a genuinely `const` object is UB.

6.6 Pointer to structure, `.` vs `->`

```c
typedef struct { uint32_t status; uint32_t data; } Dev;
Dev  d;
Dev *pd = &d;

d.status  = 1U;      //  .  on an OBJECT
pd->data  = 2U;      //  -> on a POINTER  ==  (*pd).data
(*pd).data = 2U;     // same; the () are REQUIRED (. binds first)
```

Trap: `*pd.data` parses as `*(pd.data)` and fails to compile. Passing a big struct by value copies every byte — pass `const Dev *` instead; on an MCU that is 4 bytes instead of dozens.

6.7 Pointer trap checklist

- Dereferencing `NULL`, a wild, or a dangling pointer — **UB**.
- Returning the address of a local (automatic) variable — **dangling**.
- `free(p)` twice, or using `p` after `free` — **UB**. Set `p = NULL` immediately.
- Arithmetic outside `[a, a+N]` — **UB** even if you never dereference.
- Casting a pointer to a wider-aligned type and dereferencing it — **alignment UB** (faults on Cortex-M0/M0+, which has no unaligned access at all).
- Reading an object through a pointer of an unrelated type — **strict-aliasing UB**; use `memcpy` or `char *`.
- Storing a pointer in an `int` — use `uintptr_t`.
- Comparing pointers into *different* objects with `<` — unspecified.

// §7 MEMORY LAYOUT, ALLOCATION & MEMORY BUGS

7.1 Program memory map

Region	Holds	Lifetime	On an MCU
`.text`	code, vector table	whole program	flash
`.rodata`	`const` data, string literals	whole program	flash
`.data`	initialised globals/statics	whole program	RAM, image in flash
`.bss`	zero/uninitialised globals/statics	whole program	RAM, zeroed by startup
heap	`malloc`-ed blocks	until `free`	RAM, often unused
stack	locals, args, return addresses	until the block exits	RAM, a few KB

```
HIGH addresses
  +--------------------+
  |  STACK   grows DOWN|  locals, args, return addresses
  |     |              |  automatic lifetime, LIFO
  |     v              |
  |                    |  <-- if they meet: STACK OVERFLOW
  |     ^              |
  |     |              |
  |  HEAP    grows UP  |  malloc/calloc/realloc, manual
  +--------------------+
  |  .bss              |  zero / uninit globals+statics    RAM
  +--------------------+  (no space in the image, zeroed at boot)
  |  .data             |  globals+statics with a NON-ZERO value
  +--------------------+  (image in flash, copied to RAM at boot)
  |  .rodata           |  const data, literals, tables   FLASH
  +--------------------+  (read-only: writing faults or is UB)
  |  .text             |  machine code + the vector table
  +--------------------+
LOW addresses
```

Startup code (before `main`): copy `.data` from flash to RAM → zero `.bss` → set the stack pointer and vector table → call C++ static constructors → `main()`. This is why a global `int g;` is guaranteed 0 but a local `int g;` is garbage.

7.2 Dynamic allocation

Call	Does	Initialises?	Returns
`malloc(n)`	allocate `n` bytes	**NO** — indeterminate	pointer or `NULL`
`calloc(k,n)`	allocate `k*n` bytes	**YES** — all bytes 0	pointer or `NULL`; detects `k*n` overflow
`realloc(p,n)`	resize a block	keeps old content, new tail uninitialised	new pointer (**may move**) or `NULL`
`free(p)`	release	—	nothing; `free(NULL)` is a safe no-op

```c
uint8_t *b = malloc(n * sizeof *b);  // sizeof *b survives a retype
if (b == NULL) { /* HANDLE IT */ }   // ALWAYS check: it can fail
...
free(b);
b = NULL;                            // kill the dangling pointer

uint8_t *t = realloc(b, n2);         // NEVER b = realloc(b, n2);
if (t == NULL) {                     // on failure b is still valid,
    free(b); return -1;              // so that assignment would leak
}
b = t;
```

Traps: `malloc` does **not** zero memory (`calloc` does). `malloc(0)` may return `NULL` or a unique non-dereferenceable pointer — both are conforming. The returned pointer is suitably aligned for any fundamental type; a cache- or DMA-aligned buffer needs `aligned_alloc`. Every `malloc` needs exactly one `free`, on every path including error paths.

7.3 The six memory bugs — name, cause, symptom

Bug	Cause	Symptom
Memory leak	allocated, never freed (or the last pointer was overwritten)	slow growth → allocation failure after hours/days
Dangling pointer	object lifetime ended, pointer still held	random corruption, works in debug, fails in release
Use-after-free	reading/writing through a freed pointer	corrupts the allocator's own metadata; a classic exploit
Double free	`free(p)` called twice on one block	heap corruption / abort — prevented by `p = NULL`
Buffer overflow	writing past the end of an array	neighbouring variables or the return address overwritten
Stack overflow	deep recursion or a huge local array	silently overwrites `.bss`/heap on an MCU with no MPU

```c
void leak(void){ char *p = malloc(64); }   // block outlives p
void uaf (void){ char *p = malloc(64); free(p); p[0]=1; }  // UAF
void dbl (void){ char *p = malloc(64); free(p); free(p); } // 2x
void ovf (void){ char b[8]; strcpy(b, "0123456789"); }  // overflow
void so  (void){ uint8_t big[16*1024]; }   // blows a 4 KB stack
```

Why firmware avoids `malloc`: (1) **fragmentation** — after mixed alloc/free a 1 KB request can fail with 4 KB free; (2) **non-deterministic** allocation time breaks hard real-time deadlines; (3) no MMU, so a heap/stack collision is silent corruption, not a segfault; (4) out-of-memory has nowhere to go in a device with no user. **Standard practice:** static buffers, fixed-size pools sized at compile time, or allocate once at init and never free. MISRA C 21.3 bans the `malloc` family outright.

Tools to name in an interview: Valgrind / AddressSanitizer (`-fsanitize=address,undefined`) on host builds, stack-usage analysis (`-fstack-usage`), a stack painting pattern (`0xDEADBEEF`) plus a high-water-mark check, an MPU guard page, and the linker `.map` file for section sizes.

// §8 STORAGE CLASSES, SCOPE & LINKAGE

Keyword	Lifetime	Scope / linkage	Key point
`auto`	block (automatic)	block, no linkage	the default for locals; the keyword is never written in C. **In C++11 it means type deduction instead.**
`static` (local)	**whole program**	block, no linkage	retains its value between calls; zero-initialised once, before `main`
`static` (file)	whole program	file, **internal linkage**	private to this .c file — the C way to say "module-private"
`extern`	whole program	external linkage	a *declaration*, not a definition: the object lives in another TU
`register`	block	block, no linkage	optimisation hint (ignored today). **You may not take its address** — that is a constraint violation. Removed in C++17.

```
int counter(void){ static int n = 0; return ++n; }  // 1,2,3,...
static uint32_t s_ticks;   // private global, zeroed at startup

// ---- header / source pair for a shared global ----
// driver.h :  extern volatile uint32_t g_tick;  // DECLARATION
// driver.c :  volatile uint32_t g_tick;         // DEFINITION x1
```

Traps: a `static` local is **not** re-initialised on later calls and is **not** thread- or ISR-safe. Putting a definition (no `extern`) in a header and including it twice gives a duplicate-symbol link error. Scope is compile-time visibility; linkage is what the **linker** can see; lifetime is how long the storage exists — three independent questions.

// §9 CONST · VOLATILE · RESTRICT

Qualifier	Promise to the compiler	Typical use
`const`	"I will not modify the object **through this access path**"	read-only parameters, ROM tables
`volatile`	"this object may change outside normal program flow — do not optimise its accesses away or reorder them"	hardware registers, ISR-shared flags, DMA buffers
`restrict` (C99)	"for the lifetime of this pointer, the object it points to is accessed **only** through it"	`memcpy`-style APIs, DSP inner loops

9.1 `volatile` — the ECE keyword

RULE: every read and write of a `volatile` object must actually be performed, exactly as written, in program order relative to other volatile accesses. The compiler may not cache it in a register, may not delete a "redundant" access, and may not invent new ones.

```
volatile uint32_t status;          // shared with an ISR
volatile uint32_t *const REG =     // fixed address, changing CONTENTS
        (volatile uint32_t *)0x40021000U;

// ---- without volatile the load can be hoisted out of the loop ----
while (flag == 0U) { }      // NOT volatile -> may become "while (1);"
while (REG_SR & BUSY) { }   // must re-read the register each pass
```

Declaration	Meaning
`volatile uint32_t *p`	pointer to volatile data — the **register** is volatile
`uint32_t *volatile p`	volatile pointer to normal data — the **pointer** may change (rare)
`volatile uint32_t *const p`	**the MMIO idiom:** fixed address, volatile contents
`const volatile uint32_t *p`	read-only status register that hardware updates by itself

CRITICAL TRAP — `volatile` does NOT mean:

- **atomic** — `count++` on a volatile is still load / modify / store; an interrupt between the load and the store loses the update.
- **thread-safe / race-free** — it is not a lock and provides no mutual exclusion.
- **a CPU memory barrier** — it constrains the *compiler*, not a write buffer, cache or an out-of-order core. Cross-core or cache-coherency ordering needs `DMB` / `DSB` or C11 atomics.

Use `volatile` for visibility; use `_Atomic` / `atomic_*`, a critical section (disable interrupts), or a lock for atomicity.

```
volatile uint32_t count;
count++;             // NOT atomic: LDR / ADD / STR -- an ISR fits
                     // between the load and the store

uint32_t pm = __get_PRIMASK();  __disable_irq();  // critical section
count++;                                          // now indivisible
__set_PRIMASK(pm);                    // RESTORE, never just enable

_Atomic uint32_t c2;  c2++;  // C11: truly atomic where supported
volatile sig_atomic_t f;     // the only type a signal handler may set
```

9.2 `const` and `restrict` in practice

```
static const uint8_t SINE[256] = { ... };  // .rodata: FLASH, 0 RAM
void print(const char *s);                 // says "I only read s"

// restrict = the caller promises these do NOT overlap
void *memcpy(void *restrict d, const void *restrict s, size_t n);
void *memmove(void *d, const void *s, size_t n);   // may overlap
```

Traps: `const` is a compiler-checked contract, not memory protection — casting it away and writing to an object that is genuinely `const` is **UB** (on an MCU it may target flash and simply do nothing). Violating a `restrict` promise by passing overlapping pointers is **UB** and produces wrong results only at high optimisation levels.

// §10 STRUCT · UNION · ENUM · TYPEDEF

```
struct Device {            // struct: members get DISTINCT storage
    uint32_t status;
    uint32_t data;
};
struct Device d1;          // C needs the "struct" keyword
typedef struct Device Dev; // alias -> now just "Dev d2;"

typedef union {            // union: all members SHARE the storage
    uint32_t word;         // sizeof = largest member (+ padding)
    uint8_t  byte[4];
} Word;

typedef enum { IDLE = 0, RUN, ERROR } State;   // 0, 1, 2
```

Construct	Storage	Use
`struct`	members occupy **distinct** storage (+ padding)	group related data: a device, a packet, a sample
`union`	members **overlap**; only one is meaningful at a time	byte/word views, RAM saving, tagged variants
`enum`	named integral constants (an `int`-compatible type in C)	FSM states, error codes, register field values
`typedef`	no storage — a **type alias**, not a new type	readability: `uint32_t`, callbacks, handles

```
Word w;  w.word = 0x11223344U;
w.byte[0];        // 0x44 little-endian, 0x11 big-endian
```

Union type punning: in **C**, reading a member other than the last one written is permitted and reinterprets the bytes (the value may be a trap representation). In **C++** it is formally **UB** — use `memcpy`, or `std::bit_cast` in C++20. Either way the result depends on endianness and padding, so it is not portable across targets.

Traps: `typedef` creates an alias, so it gives you **no type safety** — a `typedef`-ed `int` still converts freely. An `enum`'s underlying type is implementation-defined in C and may be signed or unsigned; values are not range-checked, so `State s = (State)99;` compiles. Comparing two structs with `==` is a compile error in C (compare member by member).

// §11 PADDING & ALIGNMENT

```
struct A { char a; int b; char c; };  // 32-bit target: sizeof 12
struct B { int b; char a; char c; };  //                sizeof  8
```

```
struct A  offset: 0    1  2  3    4  5  6  7    8    9 10 11
                +----+---------+------------+----+---------+
                | a  | PADDING |     b      | c  | PADDING |
                +----+---------+------------+----+---------+
                  1B     3B          4B        1B     3B    = 12 B
    b must start on a 4-byte boundary ---^        ^--- TRAILING
    padding, so that A arr[2] keeps every b aligned

struct B  +------------+----+----+---------+
          |     b      | a  | c  | PADDING |  = 8 B  (-33%)
          +------------+----+----+---------+
```

Rules: every member sits at an offset that is a multiple of its own alignment; the struct's alignment is the **largest** member alignment; the total size is rounded up to that alignment so arrays stay aligned. Therefore `sizeof(struct)` may be **>** the sum of member sizes, and the exact layout is **implementation-defined**.
Fix: declare members in **descending size order** to minimise padding.

```
#include <stddef.h>
offsetof(struct A, b);        // member offset - never hand-compute
_Alignof(uint32_t);           // 4  (C11; alignof in C++11)
_Alignas(32) uint8_t dma[64]; // force cache-line / DMA alignment

// GCC: sizeof == 5, no padding at all
struct __attribute__((packed)) Pkt { uint8_t id; uint32_t val; };
#pragma pack(push, 1)   /* MSVC form ... */   #pragma pack(pop)
```

Packed-struct traps: packing removes padding but creates **unaligned members**. Taking a pointer to one and dereferencing it is UB: it faults on Cortex-M0/M0+ (ARMv6-M has no unaligned access), and even on M3/M4 — which do allow unaligned `LDR` / `STR` — it still faults for `LDM` / `LDRD` or when `UNALIGN_TRP` is set, and costs extra cycles. Never `memcmp` two structs — the padding bytes are indeterminate, so equal structs can compare unequal. Never map a packed struct directly onto a network/CAN frame without also fixing **endianness**; the portable path is explicit byte packing/unpacking.

Bit-fields: `struct { uint32_t en:1; uint32_t mode:3; };` looks perfect for registers but the **bit order, storage unit and straddling rules are implementation-defined**. Use explicit shifts and masks (§14) for hardware; keep bit-fields for internal flags only.

// §12 INTEGER CONVERSIONS — WHERE SILENT BUGS LIVE

12.1 The two rules that drive everything

1. Integer promotion. Any type narrower than `int` (`char`, `short`, `_Bool`, `uint8_t`, `uint16_t`, bit-fields) is promoted to `int` before almost any arithmetic — or to `unsigned int` if `int` cannot hold all its values.
2. Usual arithmetic conversions. The two operands are then brought to a common type: higher rank wins; at equal rank the **unsigned** type wins. A signed operand is converted to unsigned unless the signed type can represent every value of the unsigned one.

```
unsigned int u =  1U;
int          s = -2;
if (s < u) { ... }   // FALSE: s becomes (unsigned)(-2) = 4294967294
                     // -2 < 1 is true in maths, false in C

if ((size_t)-1 > 0) ...     // TRUE: (size_t)-1 is SIZE_MAX
for (size_t i = n-1; i >= 0; i--)  // INFINITE: never < 0
if (strlen(s) - 1 >= 0) ...        // ALWAYS true, even if empty
```

RULE: never mix signed and unsigned in one comparison. Enable `-Wsign-compare` / `-Wconversion`, cast explicitly, or keep loop counters and sizes in the same signedness. This is the single most common OA "why is the output wrong" question.

12.2 Promotion in bitwise code

```
uint8_t a = 0xFFU;
~a              // NOT 0x00: a promotes to int 0xFF -> ~ = 0xFFFFFF00
(uint8_t)~a     // 0x00 - cast back explicitly
uint8_t b = 0x80U;
b << 1          // int 0x100 (256), NOT 0: the shift is int-wide
uint16_t c = 0xFFFFU;
c * c           // int overflow -> UB (0xFFFE0001 will not fit)
(uint32_t)c * c  // correct: widen FIRST
```

Conversion	What happens	Example
Zero extension	unsigned → wider: fill with 0s	`uint8_t 0xFF` → `0x000000FF`
Sign extension	signed → wider: copy the sign bit	`int8_t 0xFF` (−1) → `0xFFFFFFFF`
Truncation	wider → narrower: keep the low bits	`0x1234` → `uint8_t 0x34`
To signed, out of range	**implementation-defined** (two's-complement wrap in practice)	`(int8_t)200` → typically −56
To unsigned	well defined: reduce modulo 2^N	`(uint8_t)-1` → 255
Float → int	truncates toward zero; UB if out of range	`(int)-2.7` → −2

```
char ch = -1;          // plain char: signedness is IMPL-DEFINED
if (ch == 0xFF) { }    // false if char is signed, true if unsigned
int c = getchar();     // MUST be int: EOF is -1, not a char value
```

12.3 Overflow & shifts

Operation	Behaviour
Signed overflow (`INT_MAX + 1`)	UB — the compiler may assume it never happens (and delete your check)
Unsigned overflow	D wraps modulo 2^N — well defined and portable
`x << n`, `n >=` width, or negative `n`	UB
`1 << 31` with 32-bit `int`	UB (does not fit in `int`) — write `1U << 31`
`-1 >> 1` (right shift of a negative)	I implementation-defined (arithmetic shift in practice)
Division by zero	UB — including `%`
`-7 / 2` and `-7 % 2`	D since C99: truncates toward zero → `-3` and `-1`

Overflow check done right: never write `if (a + b < a)` for signed types — the addition already invoked UB. Test **before**: `if (a > INT_MAX - b)`, or do the arithmetic in `unsigned` /a wider type, or use `__builtin_add_overflow`.

// §13 UNDEFINED · UNSPECIFIED · IMPLEMENTATION-DEFINED

Class	Meaning	Can you predict the output?
UB **Undefined**	the standard imposes **no requirements at all**	**No.** Anything may happen, including "it worked yesterday"
U **Unspecified**	one of several valid behaviours; no documentation required	No — but the program stays well-formed
I **Implementation-defined**	the implementation chooses **and documents** it	Yes, per compiler/target — read the manual

Class	Classic examples
UB	signed overflow · out-of-bounds access · dereferencing NULL/dangling · use-after-free · double free · modifying a string literal · `i = i++` (in C) · shift ≥ width · division by zero · strict-aliasing violation · falling off the end of a value-returning function · misaligned access · data race
U	order of evaluation of function arguments · order of evaluation of `a() + b()` · comparing pointers into different objects · the value of padding bytes
I	`sizeof(int)` · signedness of plain `char` · right shift of a negative value · struct padding and layout · endianness · `enum` underlying type · out-of-range conversion to a signed type

Why UB is worse than "a wrong answer": the optimiser is allowed to assume UB never occurs, so it may delete the code path you wrote to guard against it. Never answer an interview question about UB with a fixed number — say **"it is undefined behaviour"** and then explain what a typical compiler happens to emit. Build with `-fsanitize=undefined` to catch it at run time.

// §14 PREPROCESSOR & MACROS

```
#include <stdint.h>        // system header: search the system paths
#include "board.h"         // project header: this directory first

#define LED      5U                       // object-like
#define BIT(n)   (1UL << (n))             // PARENTHESISE EVERYTHING
#define MIN(a,b) ((a) < (b) ? (a) : (b)) // beware double evaluation
#define ARRAY_LEN(a) (sizeof(a) / sizeof((a)[0]))

#define LOG(f, ...) printf("[%s:%d] " f, __FILE__, __LINE__, \
                           __VA_ARGS__)

#ifndef BOARD_H            // include guard  (or  #pragma once )
#define BOARD_H
...
#endif

#if defined(DEBUG) && (DEBUG > 1)
#  define DBG(x) (x)
#else
#  define DBG(x) ((void)0)      // nothing, and no unused warning
#endif

#define STR(x)  #x             // stringify:  STR(abc)   -> "abc"
#define CAT(a,b) a##b          // token paste: CAT(x, 1) -> x1
```

Macro traps

- `#define SQ(x) x*x` → `SQ(1+2)` expands to `1+2*1+2` = **5**. Parenthesise: `((x)*(x))`.
- `MAX(i++, j)` evaluates `i++` **twice** — macros paste text, they do not evaluate arguments.
- A multi-statement macro needs `do { ... } while (0)` so it works after a one-line `if`.
- Macros ignore scope and types, and are invisible to the debugger. Prefer `static inline` functions, `const` objects and `enum` constants — **except** where you need the textual power (register maps, `__LINE__`, conditional compilation).

Macro	textual substitution before compilation; no type check, no scope, no address, no debug symbol; can act on tokens (#, ##)
`inline`	a real function: typed, scoped, arguments evaluated **once**, debuggable, and the compiler still usually removes the call

// §15 BIT MANIPULATION — MEMORISE COLD

15.1 The five single-bit operations

Operation	Idiom	Why it works
SET bit n	`x |= (1U << n);`	OR with 1 forces 1, OR with 0 keeps
CLEAR bit n	`x &= ~(1U << n);`	AND with 0 forces 0, AND with 1 keeps
TOGGLE bit n	`x ^= (1U << n);`	XOR with 1 flips, XOR with 0 keeps
READ bit n → 0/1	`(x >> n) & 1U`	shift the bit down, mask the rest
TEST bit n	`if (x & (1U << n))`	non-zero, but **not** necessarily 1

```c
uint32_t x = 0U;
x |=  (1U << 3);            // set    bit 3   -> 0x00000008
x &= ~(1U << 3);            // clear  bit 3   -> 0x00000000
x ^=  (1U << 3);            // toggle bit 3
uint32_t b = (x >> 3) & 1U; // read   bit 3   -> 0 or 1
if (x & (1U << 3)) { }      // test   bit 3

#define BIT(n)         (1UL << (n))
#define SET_BIT(r,n)   ((r) |=  BIT(n))
#define CLR_BIT(r,n)   ((r) &= ~BIT(n))
#define TGL_BIT(r,n)   ((r) ^=  BIT(n))
#define GET_BIT(r,n)   (((r) >> (n)) & 1UL)
```

Always write `1U`, **never** `1`. `1 << 31` shifts into the sign bit of a signed `int` — **UB**. `~(1U << n)` on a `uint8_t` target still computes in `int` width (§12.2), so mask the result back: `x = (uint8_t)(x & ~(1U << n));`. Shifting by ≥ the type width (`1U << 32`) is UB — it does **not** give 0.

15.2 Masks and multi-bit fields

```c
mask = (1U << n) - 1U;        // n low bits: n=4 -> 0x0F  (n < 32!)
x &= ~MASK;                   // clear every bit in MASK
x |=  MASK;                   // set   every bit in MASK
if ((x & MASK) == MASK) { }   // ALL bits of MASK are set
if ((x & MASK) != 0U)   { }   // ANY bit of MASK is set

// ---- a 3-bit field at bit 4 :  bits [6:4] ----
#define MODE_SHIFT   4U
#define MODE_MASK    (0x7U << MODE_SHIFT)     // 0x00000070

// EXTRACT
field = (reg >> MODE_SHIFT) & 0x7U;

// INSERT (read-modify-write: CLEAR the field, then OR the value)
reg = (reg & ~MODE_MASK) | ((value << MODE_SHIFT) & MODE_MASK);
```

Never write `reg |= (value << SHIFT);` **alone** — OR can only set bits, so an old `0b111` can never become `0b010`. Clear the field with `& ~MASK` first. The trailing `& MODE_MASK` defends against an out-of-range `value` spilling into neighbouring fields.

Shortcut (unsigned only)	Equivalent
`x >> n`	$x / 2^n$ (unsigned; on signed negatives it is implementation-defined)
`x & ((1U << n) - 1U)`	$x \% 2^n$ — the ring-buffer wrap trick
`x << n`	$x * 2^n$ (watch overflow)

15.3 Bit tricks worth memorising

Trick	Does	Note
`x & (x - 1)`	clears the **lowest set bit**	0 → detects a power of two
`x & -x`	isolates the **lowest set bit**	use an **unsigned** x: `-x` on `INT_MIN` is UB
`x && !(x & (x - 1))`	power-of-two test	the `x &&` rejects 0
`x | (x - 1)`	sets all bits below the lowest set bit	mask building
`(x ^ y)` then popcount	Hamming distance	ECC / error metrics

```c
// ---- COUNT SET BITS (population count) ----
uint32_t popcount(uint32_t x) {
    uint32_t c = 0U;
    while (x) { x &= (x - 1U); c++; }  // 1 pass per SET bit
    return c;
}
// hardware/compiler intrinsic when available: __builtin_popcount(x)
```

```c
// ---- PARITY (1 if an odd number of bits is set) ----
uint32_t parity(uint32_t x) {
    x ^= x >> 16; x ^= x >> 8; x ^= x >> 4; x ^= x >> 2; x ^= x >> 1;
    return x & 1U;                    // XOR-fold: log2(32) = 5 steps
}
```

```c
// ---- REVERSE BITS ----
uint32_t reverse(uint32_t x) {
    uint32_t r = 0U;
    for (uint32_t i = 0U; i < 32U; i++) {
        r = (r << 1) | (x & 1U);  x >>= 1;
    }
    return r;              // ARM does this in 1 cycle: the RBIT insn
}
```

```c
// ---- SWAP BYTES (endian conversion) ----
uint16_t swap16(uint16_t v){ return (uint16_t)((v >> 8) | (v << 8)); }
uint32_t swap32(uint32_t v){
    return ((v >> 24) & 0x000000FFU) | ((v >> 8) & 0x0000FF00U)
         | ((v << 8)  & 0x00FF0000U) | ((v << 24) & 0xFF000000U);
}
// ---- SWAP NIBBLES of a byte ----
uint8_t swap_nib(uint8_t b){ return (uint8_t)((b << 4) | (b >> 4)); }
```

```c
// ---- ROTATE (not a C operator - build it) ----
uint32_t rotl(uint32_t v, uint32_t n) {
    n &= 31U;                         // mask BOTH shift counts
    return (v << n) | (v >> ((32U - n) & 31U));
}
```

Rotate trap: the naive `(v << n) | (v >> (32 - n))` is **UB when** `n == 0` (a shift by 32). Mask both counts as above. **XOR swap** (`a^=b; b^=a; a^=b;`) is a party trick: it fails when both operands are the same object, it is slower than a temporary, and the one-line form `a^=b^=a^=b` is UB.

15.4 Counting in a fixed number of steps

```c
static const uint8_t NIB[16] = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
uint32_t popcount_lut(uint32_t x){   // table in flash, no branches
    uint32_t c = 0U;
    for (uint32_t i = 0U; i < 8U; i++) {
        c += NIB[x & 0xFU];  x >>= 4;      // 8 lookups, constant time
    }
    return c;
}
```

Interview framing: Kernighan's loop is O(number of set bits); the lookup table is O(1) with constant time (better for a **constant-time** requirement, e.g. crypto); `__builtin_popcount` maps to a single instruction where the ISA has one. Say which trade-off you are making — that is the answer they want.

// §16 MEMORY-MAPPED I/O & REGISTER PROGRAMMING

16.1 The MMIO idiom, built in four steps

```c
#define REG (*(volatile uint32_t *)0x40000000U)
```

```
   0x40000000U           a plain integer: the datasheet address
        |
        v  (volatile uint32_t *)  cast to "pointer to a 32-bit
        |                          volatile object"
        v  volatile               every access must really happen,
        |                          in order, never held in a reg
        v  * ( ... )              dereference -> REG behaves like
        |                          an ordinary variable
   +-------------------------------------------------------+
   |  REG |= (1U << 3);  ==  read the register, set bit 3,  |
   |                        write it back (read-modify-write)|
   +-------------------------------------------------------+
```

```c
// ---- how vendor headers do it: a struct overlay on the block ----
typedef struct {
    volatile uint32_t MODER;   // offset 0x00
    volatile uint32_t OTYPER;  // offset 0x04
    volatile uint32_t IDR;     // 0x10 read-only -> const volatile
    volatile uint32_t ODR;     // 0x14 output
} GPIO_TypeDef;

#define GPIOA  ((GPIO_TypeDef *)0x40020000U)

GPIOA->ODR |=  (1U << 5);        // drive PA5 high
GPIOA->ODR &= ~(1U << 5);        // drive PA5 low
```

Why the struct works: members are laid out in order at increasing offsets, so one base address plus the declaration order reproduces the whole register map. It only holds because every field is exactly 32 bits with natural alignment — so **no padding** is inserted. Insert a `uint32_t RESERVED[3];` for gaps in the map, never a packed struct.

16.2 Register operation table

Operation	Pattern	Caution
Set bit	`REG |= MASK;`	read-modify-write — not atomic
Clear bit	`REG &= ~MASK;`	read-modify-write
Toggle bit	`REG ^= MASK;`	read-modify-write
Test bit	`if (REG & MASK)`	a read may have side effects
Extract field	`(REG >> SHIFT) & MASK`	—
Insert field	`REG = (REG & ~FMASK) | ((v << SHIFT) & FMASK);`	clear before OR
Clear a W1C flag	`REG = MASK;` (**plain write**)	**never** `|=` — see below
Write-only register	`REG = value;`	reading may return garbage

Read-modify-write hazards — the questions that separate candidates

- **Write-1-to-clear (W1C) status registers:** `SR |= UART_TC;` reads the register, sees *every* pending flag as 1, and writes them all back — **clearing interrupts you never handled**. Correct form: `SR = UART_TC;`.
- **Read-to-clear registers:** reading clears the flag, so a second read (e.g. inside a `printf` of the same expression) loses the event. Read once into a local.

...and three more that cost silicon time

- **Reserved bits:** the datasheet dictates whether they must be preserved or written as 0. A blind `REG = value;` can clobber them.
- **ISR vs main loop:** an interrupt between the read and the write of a RMW loses one update. Use a critical section, or the hardware's atomic set/clear registers (STM32 `BSRR`, many parts have SET/CLR/TOG aliases).
- **Access width** must match the datasheet — a 32-bit-only peripheral may fault or corrupt on a byte write.

```
GPIOA->BSRR = (1U << 5);          // atomic SET of PA5, no RMW
GPIOA->BSRR = (1U << (5 + 16));  // atomic CLEAR: use this in ISRs
```

// §17 INTERRUPTS & ISRS

```
   main loop running ...
        |  hardware event (edge, timer, byte received)
        v
  [1] CPU finishes the current instruction   [4] ISR returns
  [2] pushes context (PC, status, regs)      [5] context restored
  [3] jumps via the VECTOR TABLE to the ISR  [6] main loop resumes
```

Term	One line
Interrupt	a hardware signal that makes the CPU suspend normal flow and service an event
ISR	the function that runs on that event: `void TIM2_IRQHandler(void)` — no arguments, no return value, never called by you
Vector table	an array of function pointers, one per exception/IRQ, at a fixed address (offset 0 on Cortex-M, after the initial stack pointer)
Latency	time from the event to the first ISR instruction (pipeline + context save + higher-priority ISRs + interrupt-disabled windows)
Priority / nesting	a higher-priority IRQ can pre-empt a running ISR; equal priority waits
Re-entrancy	a function safe to enter again before the first call finishes: no static/global state, no shared hardware

ISR rules

- **Keep it short** — do the minimum, defer the work to the main loop or an RTOS task (top half / bottom half).
- **Never block**: no busy-wait on another peripheral, no delay loops, no mutex that a task holds.
- **No `malloc`, no `printf`** — not re-entrant, unbounded run time.
- **Clear the interrupt flag**, or the ISR re-enters forever the moment it returns.
- Every variable shared with the main loop is `volatile` — and `volatile` alone is not enough for read-modify-write or multi-byte data.
- Beware floating point / DSP registers if the context save does not include them.

```
volatile uint32_t g_ticks;      // written by ISR, read by main
volatile uint8_t  g_flag;       // event notification

void SysTick_Handler(void) {    // ISR: short, no blocking
    g_ticks++;                  // one 32-bit store on a 32-bit core
    g_flag = 1U;
}

int main(void) {
    for (;;) {
        if (g_flag) {           // deferred processing
            g_flag = 0U;
            do_work();
        }
    }
}

// ---- a multi-word shared value needs a critical section ----
uint64_t read_us(void) {
    uint32_t p = __get_PRIMASK();  __disable_irq();
    uint64_t t = g_micros64;    // two 32-bit loads: do not split
    __set_PRIMASK(p);           // restore the PREVIOUS state
    return t;
}
```

Race condition: two contexts access the same data and at least one writes, with no ordering guarantee — the result depends on timing. Classic form: `count++` in both the main loop and an ISR. Fixes: disable interrupts around the sequence (shortest possible window), use an atomic instruction (LDREX/STREX, `_Atomic`), make the ISR the only writer, or use a lock-free single-producer/single-consumer queue (§19).

17.1 Polling vs interrupt vs DMA

	Polling	Interrupt	DMA
CPU cost	burns cycles while waiting	only when the event happens	near zero during transfer
Latency	up to one poll period	low and bounded	lowest for bulk data
Complexity	trivial	concurrency bugs, `volatile`	buffers, alignment, cache coherency
Power	worst — cannot sleep	good — CPU sleeps until IRQ	best for streams
Best for	startup, very fast/rare events, simple loops	asynchronous events, byte-level UART	ADC blocks, SPI/I2S streams, framebuffers

DMA + cache/compiler: a DMA buffer is written by hardware behind the compiler's back — declare it `volatile` (or re-read after the completion flag), align it, and on a cached core invalidate before reading and clean before transmitting.

// §18 ENDIANNESS

```
uint32_t x = 0x12345678U;      // stored at address A
```

```
                A     A+1   A+2   A+3
LITTLE endian  78    56    34    12    LSB at the LOWEST address
BIG    endian  12    34    56    78    MSB at the LOWEST address
                                       (= "network byte order")

little: x86, ARM (usual), RISC-V
big   : many network protocols, some DSPs, SPARC
```

Endianness is BYTE order in memory, not bit order. Inside a byte, bit 7 is always the MSB. The value `x` is `0x12345678` on every machine; only the byte-by-byte layout differs — so it is visible only when you access the object through a different-sized type, or send it over a wire.

```
uint32_t v = 0x12345678U;
uint8_t *p = (uint8_t *)&v;
p[0];                          // 0x78 little-endian, 0x12 big-endian

// ---- runtime detection (a char pointer view is legal in C) ----
int is_little(void){ uint16_t t = 1U; return *(uint8_t*)&t == 1U; }

// ---- portable: never memcpy a struct onto the wire ----
void put_be32(uint8_t *b, uint32_t v){   // big-endian pack, ANY host
    b[0] = (uint8_t)(v >> 24);  b[1] = (uint8_t)(v >> 16);
    b[2] = (uint8_t)(v >>  8);  b[3] = (uint8_t) v;
}
uint32_t get_be32(const uint8_t *b){     // big-endian unpack
    return ((uint32_t)b[0] << 24) | ((uint32_t)b[1] << 16)
         | ((uint32_t)b[2] <<  8) |  (uint32_t)b[3];
}
```

Where it bites: casting a `uint8_t` receive buffer to a `uint32_t *` (endianness **and** alignment UB), `memcpy`-ing a struct into a CAN/Ethernet frame, sharing a binary file or EEPROM image between a big- and little-endian part, and multi-byte register access on a peripheral whose bus differs from the core. Use explicit shift-based pack/unpack, or `htons/htonl` on hosted systems.

// §19 RING BUFFER (CIRCULAR FIFO)

```
SIZE = 8 (power of two).  Producer writes HEAD, consumer reads TAIL.

  idx   0    1    2    3    4    5    6    7
      +----+----+----+----+----+----+----+----+
      | D  | E  |    |    |    | A  | B  | C  |  <-- wraps round
      +----+----+----+----+----+----+----+----+
             ^                   ^
           HEAD                TAIL      count = HEAD - TAIL
        next write          next read          (unsigned wrap)
                                      EMPTY: HEAD == TAIL
                                      FULL : count == SIZE
```

```
#define RB_SZ 64U                    // MUST be a power of two
typedef struct {
    uint8_t  buf[RB_SZ];
    volatile uint32_t head;          // written ONLY by the producer
    volatile uint32_t tail;          // written ONLY by the consumer
} rb_t;

// unsigned subtraction wraps correctly, so no modulo is needed
static inline uint32_t rb_count(const rb_t *r)
                                     { return r->head - r->tail; }
static inline bool rb_empty(const rb_t *r)
                                     { return r->head == r->tail; }
static inline bool rb_full(const rb_t *r)
                                     { return rb_count(r) >= RB_SZ; }

bool rb_put(rb_t *r, uint8_t v) {    // ENQUEUE - producer only
    if (rb_full(r)) { return false; }        // overrun: drop + flag
    r->buf[r->head & (RB_SZ - 1U)] = v;      // &(SZ-1) == %SZ
    r->head++;                       // publish AFTER the data is stored
    return true;
}
bool rb_get(rb_t *r, uint8_t *out) {  // DEQUEUE - consumer only
    if (rb_empty(r)) { return false; }
    *out = r->buf[r->tail & (RB_SZ - 1U)];
    r->tail++;
    return true;
}
```

Why this is lock-free for one producer + one consumer: each index has exactly **one writer**, both are single 32-bit aligned words (so loads/stores are indivisible on a 32-bit core), and the data store is ordered before the index update. No disabling of interrupts is needed — that is why every UART driver looks like this. With **multiple** producers or consumers you must add a critical section or CAS.

Design choice	Trade-off
Free-running counters + `& (SZ-1)`	uses all SIZE slots; needs a power-of-two size; relies on unsigned wraparound

Design choice	Trade-off
Sacrifice one slot (`full = (head+1)%N == tail`)	any size, no count variable, capacity is N-1
Explicit `count` field	simplest to read, but `count` has **two** writers → needs atomicity

Traps: `%` with a non-power-of-two size costs a division on MCUs with no divider. Overrun handling must be explicit — decide whether to drop the newest byte, drop the oldest, or set an error flag. Never `memcpy` across the wrap point in one call: split it into two copies.

// §20 FINITE STATE MACHINES

```
   CURRENT STATE  +  INPUT/EVENT   ->   NEXT STATE  +  ACTION (output)

        +--------+  EV_START   +-------+  EV_FAULT   +---------+
        |  IDLE  |------------>|  RUN  |------------>|  ERROR  |
        +--------+             +-------+             +---------+
             ^   EV_STOP          |                       |
             +--------------------+   EV_RESET            |
             +-------------------------------------------+
```

```c
typedef enum { IDLE, RUN, ERROR } State;
typedef enum { EV_START, EV_STOP, EV_FAULT, EV_RESET } Event;

static State s_state = IDLE;        // ONE owner of the state variable

void fsm_step(Event e) {
    switch (s_state) {
    case IDLE:
        if (e == EV_START) { motor_on();  s_state = RUN;   }
        break;
    case RUN:
        if      (e == EV_FAULT) { motor_off(); s_state = ERROR; }
        else if (e == EV_STOP)  { motor_off(); s_state = IDLE;  }
        break;
    case ERROR:
        if (e == EV_RESET) { s_state = IDLE; }
        break;
    default:
        s_state = ERROR;            // defensive: unreachable states
        break;
    }
}
```

Style	How	When
`switch` on an `enum`	the code above	default choice: readable, debuggable, no RAM cost
Transition table	`static const State NEXT[N_STATE][N_EVENT];`	many states; table lives in flash, O(1) dispatch
Function-pointer state	`void (*state)(Event);`	large per-state behaviour, hierarchical FSMs

ECE crossover: a **Moore** machine's outputs depend only on the current state (glitch-free, registered outputs); a **Mealy** machine's outputs depend on state **and** input (fewer states, but outputs can glitch). The same distinction you draw in RTL applies to firmware FSMs. Always have an explicit reset/default state, and drive the FSM from one place — an event queue (§19) fed by ISRs.

// §21 PERIPHERALS & FIRMWARE PRACTICE

Bus	Shape	The C-side concern
UART	asynchronous, 2 wires, point-to-point	byte stream → ring buffer per direction; overrun flag; baud mismatch
SPI	synchronous, full duplex, master/slave + CS	every transfer is a simultaneous TX **and** RX — you must read the RX byte you did not want; mode (CPOL/CPHA)
I2C	2 wires, addressed, multi-drop	address + register-pointer transactions, ACK/NACK checking, repeated START, clock stretching, bus-hang recovery
CAN	differential, multi-master, arbitrated	frames with message IDs, mailboxes/filters, priority by ID, error counters

Driver skeleton that works for all four: `init()` configures clocks/pins/registers → transfers run by **polling** (simplest), **interrupt** (per byte/frame, feeds a ring buffer) or **DMA** (block, one interrupt at the end) → a status/error path clears flags and reports up. Keep the ISR to "move one byte, set a flag"; put protocol logic in the main loop or a task.

Firmware practice	Why
Fixed-width types + `volatile` for all MMIO	correctness across compilers and optimisation levels
No dynamic allocation after init	determinism, no fragmentation (§7.3)
Timer/tick instead of `for`-loop delays	a busy delay changes with clock speed and optimisation
Watchdog kicked from one place only	kicking it inside an ISR hides a hung main loop
Debounce inputs (time or state based)	a mechanical switch generates dozens of edges → spurious IRQs
Check the `.map` file and stack high-water mark	silent RAM exhaustion is the classic field failure
Compile the logic for the host and unit-test it	separating hardware access from algorithm makes both testable

// §22 C++ FUNDAMENTALS

22.1 Namespaces, references, classes

```cpp
namespace drv { void init(); }  // avoids cross-module name clashes
drv::init();
using namespace std;            // handy, but NEVER in a header

int  x = 10;
int &r = x;             // REFERENCE: an alias for x
r = 20;                 // x is now 20 - no dereference syntax
int *p = &x;  *p = 30;  // the pointer equivalent

void byref(int &v)       { v = 99; }  // caller sees the change
void byptr(int *v)       { *v = 99; } // the C style
void ro   (const int &v) { }          // read-only, and no copy
```

```cpp
class Motor {           // class:  members PRIVATE by default
public:                 // struct: members PUBLIC (only difference)
    Motor() : pin_(0), on_(false) {}       // default constructor
    explicit Motor(uint8_t pin)            // explicit: blocks an
        : pin_(pin), on_(false) {}         //   implicit conversion
    ~Motor() { stop(); }                   // destructor: cleanup

    void start()       { on_ = true; }     // member function
    bool is_on() const { return on_; }     // const: cannot modify
    static uint8_t count();                // belongs to the CLASS

private:
    uint8_t pin_;       // DECLARATION ORDER == INITIALISATION ORDER
    bool    on_;
    static uint8_t s_count;   // ONE copy shared by all objects
};
uint8_t Motor::s_count = 0;   // a static member needs a definition

Motor m(5);               // an object; the constructor runs here
Motor *pm = new Motor(6); // on the heap; delete pm runs the dtor
```

Member	Runs when	Signature
Default ctor	`Motor m;`	`Motor()`
Parameterised ctor	`Motor m(5);`	`Motor(uint8_t)`
Copy ctor	copy-init, pass/return by value	`Motor(const Motor&)`
Move ctor	init from a temporary / `std::move`	`Motor(Motor&&)`
Copy / move assignment	`a = b;` / `a = std::move(b);`	`operator=(const Motor&)` / `(Motor&&)`
Destructor	scope exit, `delete`, container teardown	`~Motor()` — one per class, no args

Traps: members are initialised in **declaration order**, not in the order you write the initialiser list (`-Wreorder`). Objects are destroyed in **reverse** order of construction. A one-argument constructor without `explicit` creates surprise implicit conversions. Calling a **virtual function from a constructor or destructor** dispatches to the *current* class's version — the derived object does not exist yet, or no longer does.

22.2 Overloading & `this`

```cpp
void log(int);  void log(double);  void log(const char *);
// OVERLOAD: one name, DIFFERENT parameter lists
// int log(int);  ERROR: the return type alone cannot overload

class C {
    int v_;
public:
    C& set(int v) { this->v_ = v;    // this = ptr to this object
                    return *this; }  // returning *this -> chaining
};

extern "C" void isr_handler(void);   // no C++ NAME MANGLING, so
                                     // C and asm can link to it
```

// §23 OOP — FOUR PILLARS & POLYMORPHISM

Concept	Meaning	Mechanism in C++
Encapsulation	bundle data with the code that operates on it, hide the internals	`class`, `private`, accessors
Abstraction	expose only the essential interface	abstract base class, header/impl split
Inheritance	derive a new class from an existing one ("is-a")	`class D : public B`
Polymorphism	one interface, many behaviours	**compile time:** overloading, templates · **run time:** `virtual`

```cpp
class Sensor {              // ABSTRACT: cannot be instantiated
public:
    virtual ~Sensor() = default;  // VIRTUAL DTOR - mandatory here
    virtual int  read() = 0;      // PURE virtual: a derived class
                                  //   MUST implement it
    virtual void calibrate() { }  // virtual, with a default body
    const char *name() const { return "adc"; }  // non-virtual =
};                                              // statically bound

class Adc : public Sensor {
public:
    int  read() override { return sample(); } // compiler-checked
    void calibrate() final { }                // no further override
};

Sensor *s = new Adc();
s->read();      // RUNTIME dispatch via the vtable -> Adc::read
delete s;       // needs a virtual ~Sensor(), otherwise UB
```

```
   Adc object            vtable for Adc (FLASH, one per CLASS)
  +-----------+         +----------------------+
  |  vptr     |-------->| ~Adc()               |
  +-----------+         | Adc::read()          |  a virtual call =
  |  members  |         | Sensor::calibrate()  |  load vptr, load
  +-----------+         +----------------------+  slot, branch
   the vptr costs 4/8 bytes PER OBJECT; the table is shared
```

Term	One line
`virtual` function	may be overridden; the call is resolved at **run time** from the object's actual type
Pure virtual (`= 0`)	declares an interface with no body; makes the class abstract
Abstract class	has ≥ 1 pure virtual; cannot be instantiated, only used via pointer/reference
Virtual destructor	required whenever you `delete` a derived object through a base pointer — otherwise only the base dtor runs: **UB and a leak**
Overloading vs overriding	overloading = same name, **different parameters**, resolved at compile time. Overriding = same signature in a derived class, resolved at run time
Object slicing	copying a `Derived` into a `Base` **by value** discards the derived part and the vptr — pass by reference/pointer

Rule of 0 / 3 / 5: if a class needs a custom destructor, copy constructor or copy assignment, it almost certainly needs **all three** (Rule of 3), plus the move constructor and move assignment in modern C++ (Rule of 5). Best of all is the **Rule of 0**: hold resources in `std::unique_ptr` /containers and write none of them.

// §24 POINTER VS REFERENCE

Pointer `T *p`	Reference `T &r`
can be `nullptr`	must bind to a valid object at initialisation; normally cannot be null
can be reseated to another object	**cannot be reseated** — assignment writes through to the referent
may be uninitialised	must be initialised when declared
explicit dereference: `*p`, `p->m`	implicit: use it like the object itself
pointer arithmetic allowed	no arithmetic
can point to a pointer (`T **`)	no references to references, no arrays of references
`sizeof p` = address width	`sizeof r` = `sizeof(T)` (the referent)
Use for: optional/"may be absent", C APIs, arrays, ownership transfer	Use for: parameters that must exist, operator overloads, `const T&` to avoid copies

Trap: both compile to the same machine code (a reference is normally implemented as a pointer) — the difference is what the **compiler enforces**, not cost. Returning a reference to a local is a dangling reference, exactly like returning `&local`. In C there are no references at all: `&` is only address-of and bitwise AND.

// §25 C++ MEMORY: NEW, RAII, SMART POINTERS

```cpp
int   *p = new int(5);    // allocate + CONSTRUCT -> delete p;
int   *a = new int[10];   // allocate + construct N -> delete[] a;
Motor *m = new Motor(3);  // the constructor runs
delete p;  delete[] a;  delete m;   // dtor first, then free
delete nullptr;           // safe no-op
```

Traps: mixing forms is **UB** — `new[]` must be paired with `delete[]`, `malloc` with `free`, and never `free` a `new`-ed object. `new` **throws** `std::bad_alloc` on failure (it does not return null unless you write `new (std::nothrow)`), so checking for null after plain `new` is pointless. Deleting twice, or deleting a stack object, is UB.

25.1 RAII — the central C++ idea

RULE: **R**esource **A**cquisition **I**s **I**nitialisation — a resource (memory, file, lock, interrupt state, GPIO) is acquired in a **constructor** and released in the **destructor**. Because destructors run automatically at scope exit — on every path, including an early `return` or a thrown exception — the resource cannot leak.

```cpp
{
    std::lock_guard<std::mutex> lk(mtx);   // ctor locks
    if (bad) { return; }     // the dtor unlocks HERE, on every path
}                            // and here on the normal path

// ---- the same idea bare-metal: a scoped critical section ----
class IrqLock {
    uint32_t prev_;
public:
    IrqLock()  : prev_(__get_PRIMASK()) { __disable_irq(); }
    ~IrqLock() { __set_PRIMASK(prev_); }  // restore, never enable
};
void update(void) { IrqLock lk; g_shared++; }  // early-return safe
```

25.2 Smart pointers

Type	Ownership	Cost	Use
`std::unique_ptr<T>`	**exclusive** — move-only, cannot be copied	same size as a raw pointer; **zero** runtime overhead	the default owning pointer
`std::shared_ptr<T>`	**shared** — a reference count; the object dies when the count hits 0	2 pointers + a control block; the count is **atomic**	genuinely shared lifetime only
`std::weak_ptr<T>`	**non-owning** observer of a `shared_ptr`	no count contribution; `lock()` to use it	**breaks reference cycles**, caches

```cpp
auto u = std::make_unique<Motor>(5);  // no explicit new needed
auto v = std::move(u);        // ownership MOVED; u is now null
auto s = std::make_shared<Motor>(6);  // 1 alloc: object + block
std::weak_ptr<Motor> w = s;   // observes, does not keep it alive
if (auto sp = w.lock()) {     // safe promotion, null if gone
    sp->start();
}
```

Traps: two `shared_ptr`s that point at each other **never** free — break the cycle with `weak_ptr`. Never build two independent `shared_ptr`s from the same raw pointer (two control blocks → double free). `unique_ptr` cannot be copied, only moved. In firmware, `shared_ptr`'s atomic refcount and heap use are usually unacceptable — prefer `unique_ptr` or static storage.

// §26 MODERN C++ (C++11 → C++17) ESSENTIALS

Feature	One line	Example
`auto`	deduce the type from the initialiser (never "any type")	`auto i = v.begin();`
`nullptr`	typed null pointer constant (`std::nullptr_t`) — fixes `NULL`/`0` overload ambiguity	`T *p = nullptr;`
`constexpr`	computable at **compile time** — folds tables and maths into flash, zero run-time cost	`constexpr int N = 4*8;`
`enum class`	scoped, strongly typed enum — no implicit int conversion, no name leaking	`enum class St : uint8_t { Idle };`
Range-based `for`	iterate any container/array with a `begin/end`	`for (auto &x : buf) x = 0;`
Lambda	an anonymous function object; `[ ]` lists what it captures	`auto f = [&](int x){ return x+n; };`
Rvalue ref `T&&`	binds to temporaries — the hook that enables moving	`void f(std::string &&s);`
Move semantics	**steal** the internals of a dying object instead of copying them	`b = std::move(a);`
`std::array`	fixed-size array with a size, no decay, no heap	`std::array<int,4> a{};`
`std::vector`	dynamic contiguous array (heap)	`v.push_back(1);`
`std::string`	owning, resizable text (heap, with a small-string buffer)	`s += "abc";`
`std::optional`	a value that may be absent — replaces sentinel returns	`if (o) use(*o);`

```cpp
std::vector<int> a = {1,2,3};
std::vector<int> b = a;              // COPY: allocates + copies
std::vector<int> c = std::move(a);   // MOVE: steals the buffer, O(1)
// a is now valid but UNSPECIFIED: only assign to it or destroy it

int n = 5;
auto add = [n](int x){ return x + n; };  // capture BY VALUE
auto acc = [&n](int x){ n += x; };       // BY REFERENCE: lifetime!
```

Traps: `std::move` does not move anything — it is a cast to `T&&` that *permits* a move. A lambda that captures by reference and outlives the captured object dangles (never `[&]` in a callback stored for later). `auto` strips references and `const` by default (`auto&` / `const auto&` keep them).

// §27 STL CONTAINERS — COMPLEXITY & EMBEDDED FIT

Container	Main use	Key complexity	Memory
`array`	fixed-size sequence	index O(1)	stack/static, no heap
`vector`	dynamic array (default choice)	index O(1); `push_back` amortised O(1); insert/erase middle O(n)	heap, contiguous, grows geometrically
`deque`	push/pop at both ends	index O(1), ends O(1)	heap, chunked
`queue`	FIFO adapter	push/pop O(1)	wraps `deque`
`stack`	LIFO adapter	push/pop/top O(1)	wraps `deque`
`map` / `set`	ordered key-value / unique keys	find, insert, erase O(log n)	heap, red-black tree, per-node allocation
`unordered_map` / `_set`	hash lookup	**average** O(1), worst O(n)	heap, buckets; rehashing is a latency spike
`list`	splice-heavy sequences	insert/erase O(1) *at a known position*	heap per node; cache-hostile

Embedded reality: every container except `std::array` allocates. `vector` reallocation is an unbounded latency spike and **invalidates every iterator and pointer** into it — call `reserve()` up front. Node-based containers (`map`, `list`) fragment the heap. For firmware prefer `std::array`, a fixed-capacity ring buffer (§19), or a static pool allocator; if you must use `vector`, `reserve()` once at init.

C++ on an MCU — what to keep and what to drop. Keep: classes, RAII, `constexpr`, templates, `enum class`, `std::array`, references, `unique_ptr` — these are zero-cost or near-zero-cost abstractions. Drop (build with `-fno-exceptions -fno-rtti`): exceptions and RTTI (code size + unbounded unwinding), `iostream` (tens of KB), `std::function` (may heap-allocate — use a plain function pointer or a template parameter), and heavy dynamic containers. Watch the **static initialisation order fiasco**: the construction order of globals across translation units is unspecified.

// §28 THE COMPARISON TABLES INTERVIEWERS ASK FOR

28.1 Pointer vs Array

Aspect	Array `int a[5]`	Pointer `int *p`
What it is	5 contiguous `int` objects	one object holding an address
`sizeof`	20 (whole array)	4 or 8 (the address)
Assignable	**no** — `a = ...` is an error	yes — `p = ...`
Storage	the elements themselves	an address; the data lives elsewhere
In an expression	decays to `&a[0]` (except `sizeof`, `&a`)	already a pointer
As a parameter	silently becomes a pointer	a pointer

28.2 Stack vs Heap

Aspect	Stack	Heap
Allocated by	the compiler, on entering a block	you, via `malloc`/`new`
Freed by	automatically at scope exit	you, via `free`/`delete` (or a smart pointer)
Speed	one register adjust — fastest	allocator bookkeeping, may block
Size	small and fixed (KB on an MCU)	large, limited by free RAM
Fragmentation	impossible (LIFO)	yes — the reason firmware avoids it
Lifetime	the enclosing block	until explicitly released
Failure mode	stack overflow (silent on an MCU)	`NULL`/`bad_alloc`, leaks, fragmentation

28.3 `malloc` vs `calloc` vs `new`

Aspect	`malloc`	`calloc`	`new`
Arguments	total bytes	count, element size	a type
Initialises	no	**yes, all bytes 0**	runs the **constructor**
Returns	`void *`, `NULL` on failure	`void *`, `NULL` on failure	typed pointer; **throws** `bad_alloc`
Type safety	none (cast needed in C++)	none	full — no cast
Overflow check	none	checks `count * size`	n/a
Released by	`free`	`free`	`delete` (runs the destructor)
Overridable	no	no	yes — `operator new`, placement new

28.4 `struct` vs `union` · `struct` vs `class`

Aspect	`struct`	`union`
Storage	members at distinct offsets	all members share offset 0
`sizeof`	≥ sum of members (padding)	= largest member (+ padding)
Valid members	all at once	**one at a time** (the last written)
Use	records, register maps, packets	byte/word views, RAM reuse, tagged variants

`struct` vs `class` in C++: the **only** differences are the default member access (`struct` public, `class` private) and the default inheritance access. Both can have constructors, destructors, virtuals and templates. Convention: `struct` for plain data, `class` for invariants.

28.5 `static` vs `extern` · `const` vs `volatile`

Aspect	`static`	`extern`
Meaning	internal linkage / permanent storage	"defined in another translation unit"
Visibility	this .c file only	global across the program
Defines storage?	yes	**no** — it is a declaration
On a local variable	value survives between calls	not applicable

Aspect	`const`	`volatile`
Says	"this code will not modify it"	"something else may modify it"
Enforced by	the compiler (diagnostic)	the compiler (access generation)
Optimisation	enables it (value may be cached/folded)	**disables** it (every access is real)
Together?	`const volatile` is valid and common — a read-only hardware status register	

28.6 `volatile` vs atomic

Aspect	`volatile`	`_Atomic` / `std::atomic`
Guarantees	the access is really performed, not reordered w.r.t. other volatile accesses	**indivisible** read-modify-write + a defined memory order
Atomic?	**no**	yes
Thread/ISR safe?	no	yes (for the operations it defines)
Memory barrier?	compiler only	hardware fences as requested
Right tool for	MMIO registers, DMA buffers	counters and flags shared between contexts

Say this in the interview: "`volatile` solves *visibility*, atomics solve *indivisibility*. A hardware register needs `volatile`; a shared counter needs an atomic or a critical section. A variable shared with an ISR often needs **both**."

28.7 Macro vs `inline` · `memcpy` vs `memmove`

Aspect	Macro	`inline` function
Handled by	preprocessor (text)	compiler (code)
Type checking	none	full
Arguments	may be evaluated **many times**	evaluated exactly once
Debugger	invisible	full symbol information
Scope	ignores it	respects it
Still needed for	`#include` guards, `__LINE__`, conditional compilation, token pasting	everything else

Aspect	`memcpy`	`memmove`
Overlapping regions	**UB**	safe — behaves as if copied via a temporary
Speed	faster (may copy in any order/width)	slightly slower (direction check)
Signature	both pointers are `restrict`	no `restrict`
Use when	you know the buffers are disjoint	shifting within one buffer, scrolling, sliding windows

28.8 Shallow vs Deep Copy

Aspect	Shallow copy	Deep copy
Copies	the members, including the **pointer value**	the members **and** the pointed-to data
Result	two objects share one buffer	two independent objects

Aspect	Shallow copy	Deep copy
Danger	**double free** and use-after-free when both are destroyed; a write through one is visible in the other	cost of the allocation and copy
In C	`struct` assignment / `memcpy` is always shallow	write it by hand
In C++	the implicit copy constructor is shallow (member-wise)	write a copy ctor + copy assignment (Rule of 3/5), or hold a `unique_ptr`/container and get it free

28.9 C vs C++ — the ECE view

Topic	C	C++
Paradigm	procedural, very explicit	multi-paradigm: procedural, OOP, generic
OOP / templates / RAII	no (emulated with structs + function pointers)	yes
Type system	weaker: implicit `void*` conversions	stricter: casts required, overloading, `enum class`
Hardware access	excellent	equally excellent (same MMIO idioms)
Binary/runtime	minimal, fully predictable	near-identical *if* exceptions, RTTI and heap use are avoided
Firmware usage	very high — drivers, RTOS, boot code, MISRA-constrained work	high and growing — HALs, device abstraction, C++ on Cortex-M/automotive
Compiling C as C++	not automatic: `void*` needs casts, `sizeof('a')` differs, extra keywords (`class`, `new`, `template`) become reserved; use `extern "C"` for linkage	

Answer for "why is C used in embedded?" — a thin, predictable mapping to machine code; no hidden allocation, no hidden dispatch, no runtime; direct memory and register access; a tiny standard library that is easy to port; every silicon vendor ships a C compiler and every RTOS/driver stack is written in it.

// §29 OUTPUT-BASED DA TRAPS

Every card is tagged: D deterministic · I implementation-defined · U unspecified · UB undefined. **For a UB question the correct interview answer is "undefined behaviour", never a number.**

T1. Two `sizeof`s, two answers

D

```
void f(int a[5]) { printf("%zu ", sizeof a); }
int main(void) { int a[5]; f(a); printf("%zu", sizeof a); }
```

A **Pointer size (4 or 8), then 20.** A parameter written `int a[5]` *is* an `int *` — the size is discarded. Inside `main`, `a` is still an array. Pass the length explicitly.

T2. Pointer arithmetic scale

D

```
int a[5] = {10,20,30,40,50};
int *p = a;
printf("%d %d %d", *(p+2), *(a+1), p[3]);
```

A **30 20 40.** `p+2` advances `2 * sizeof(int)` = 8 bytes. `a[i]`, `*(a+i)` and even `i[a]` are all the same expression.

T3. `&a` vs `a`

D

```
int a[5];
printf("%td %td", (char*)(a+1)-(char*)a,
                  (char*)(&a+1)-(char*)a);
```

A **4 and 20.** `a` decays to `int *` so `a+1` skips one `int`; `&a` has type `int (*)[5]`, so `&a+1` skips the **whole array**.

T4. String size vs length

D

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

A **6 and 5.** `sizeof` counts the NUL terminator, `strlen` stops at it. For `char *p = "hello";`, `sizeof p` would be the pointer size instead.

T5. Writing to a string literal

UB

```
char *p = "hello";
p[0] = 'H';
```

A **Undefined behaviour.** String literals are not modifiable; they usually sit in read-only memory (`.rodata`/flash), so this typically faults — but the standard promises nothing. `char p[] = "hello";` makes a writable copy.

T6. Static local

D

```
void f(void){ static int s = 0; int a = 0; s++; a++;
              printf("%d%d ", s, a); }
int main(void){ f(); f(); f(); }
```

A **11 21 31.** The `static` is initialised **once** before `main` and keeps its value; the automatic `a` is created fresh on every call.

T7. The signed/unsigned classic

D

```
unsigned int u = 1U;
int s = -2;
printf("%s", (s < u) ? "less" : "not less");
```

A **"not less".** The usual arithmetic conversions turn `-2` into `4294967294U`. Deterministic, and the most common wrong answer in embedded OAs.

T8. Promotion of a small unsigned type

D

```
uint8_t a = 0xFFU;
printf("%X %X", ~a, (uint8_t)~a);
```

A `FFFFFF00` **and** `0`. `a` is promoted to `int` before `~`, so the result is 32 bits wide. Cast back whenever you need the narrow value.

T9. Precedence of `&` vs `==`

D

```
int x = 5;
if (x & 1 == 0) puts("even"); else puts("odd");
```

A **"odd".** Parsed as `x & (1 == 0)` = `5 & 0` = 0 → false. The test is *meant* to be `(x & 1) == 0`, which is also false here — but for `x = 4` the buggy version still prints "odd". Parenthesise.

T10. Two side effects, one variable

UB

```
int i = 5;
printf("%d %d", i++, i++);
int j = i++ + ++i;
```

A **Undefined behaviour in C** — two unsequenced modifications of `i`. Do not quote a number. (In C++17 the argument evaluations are indeterminately sequenced, so the *order* is merely unspecified; `i = i++` also became defined there. Never write either.)

T11. Struct size

I

```
struct A { char a; int b; char c; };
printf("%zu", sizeof(struct A));
```

A **Typically 12** on a 32-bit-aligned target (1 + 3 pad + 4 + 1 + 3 pad), but the layout is **implementation-defined**. Reordering to `int, char, char` gives 8. Say "sum of members plus padding, so it depends on alignment".

T12. Shifting into the sign bit

UB

```
int  a = 1 << 31;
uint32_t b = 1U << 31;
uint32_t c = 1U << 32;
```

A `a`**: UB** (does not fit in a 32-bit signed `int`). `b`**: fine** — `0x80000000`. `c`**: UB** — shifting by ≥ the width is undefined; it does *not* yield 0. Always shift an unsigned value by less than its width.

T13. Reading uninitialised heap

UB

```
int *p = malloc(4 * sizeof *p);
printf("%d", p[0]);
```

A **Undefined** — `malloc` does not initialise. The value is indeterminate and reading it is UB. Use `calloc`, or write before you read. (Also: `p` may be `NULL`.)

T14. Returning a local

UB

```
int *f(void){ int x = 42; return &x; }
printf("%d", *f());
```

A **Dangling pointer → UB.** `x` dies when `f` returns; the stack slot is reused by the very next call. It often "prints 42" in a debug build and fails in release — the worst kind of bug. Return by value, or use `static`/caller-provided storage.

T15. Integer division with negatives

D

```
printf("%d %d %d", 5/2, -5/2, -5%2);
```

A **2, −2, −1.** Since C99, division truncates **toward zero** and the remainder takes the sign of the dividend. (C89 left it implementation-defined.)

T16. Float equality

I

```
if (0.1 + 0.2 == 0.3) puts("eq"); else puts("ne");
```

A **"ne"** on any IEEE-754 implementation — neither 0.1 nor 0.2 is exactly representable in binary. Compare with a tolerance: `fabs(a-b) <= eps`. (The floating-point format itself is implementation-defined.)

T17. C++ construction and destruction order

D

```
struct T { char c; T(char c):c(c){ putchar(c); }
                   ~T(){ putchar((char)toupper(c)); } };
int main(){ T a('a'); { T b('b'); } T c('c'); }
```

A `abBcCA`. `b` is destroyed at its inner closing brace; the rest are destroyed at the end of `main` in **reverse** order of construction.

T18. `*p++` vs `(*p)++`

D

```
int a[3] = {10, 20, 30};
int *p = a;
int x = *p++;      // A
int y = (*p)++;    // B
printf("%d %d %d %d", x, y, a[0], a[1]);
```

A **10 20 10 21.** `*p++` reads `a[0]` then advances `p`; `(*p)++` returns `a[1]`'s old value 20 and increments the *object* to 21. Postfix binds tighter than `*`.

T19. Fall-through

D

```
int x = 2;
switch (x) {
    case 1: printf("A");
    case 2: printf("B");
    case 3: printf("C"); break;
    default: printf("D");
}
```

A **BC.** Execution enters at `case 2` and falls through `case 3` until the `break`. Missing `break` is a bug unless you mark it deliberately.

T20. Union byte view

I

```
union { uint32_t w; uint8_t b[4]; } u;
u.w = 0x12345678U;
printf("%02X", u.b[0]);
```

A `78` **on a little-endian machine,** `12` **on big-endian** — the answer is **implementation-defined**, so name the endianness in your answer. (Reading a member other than the last written is allowed in C; in C++ it is formally UB.)

T21. Macro text substitution

D

```
#define SQ(x) x*x
printf("%d ", SQ(1+2));

#define MAX(a,b) ((a)>(b)?(a):(b))
int i = 5, j = 3;
printf("%d", MAX(i++, j));
```

A **5, then 6 — and `i` ends at 7.** `SQ(1+2)` pastes to `1+2*1+2` = 5. `MAX` expands `i++` **twice**: the comparison uses 5 (`i`→6), then the winning branch evaluates `i++` again, yielding 6 (`i`→7). Never pass a side effect to a macro.

T22. Shallow copy of a struct with a pointer

UB

```
typedef struct { char *buf; } S;
S a; a.buf = malloc(16);
S b = a;                    // member-wise = SHALLOW
free(a.buf);  free(b.buf);
```

A **Double free → UB.** Struct assignment copies the *pointer*, so both objects own one buffer. Fix with a deep copy, single ownership, or (C++) a copy constructor / `unique_ptr`.

T23. Counting loop

D

```
uint32_t x = 0xF0U, c = 0U;
while (x) { x &= (x - 1U); c++; }
printf("%u", c);
```

A **4.** `x & (x-1)` clears the lowest set bit each pass, so the loop runs once per set bit — `0xF0` has four. If the answer were 8 the question would be about a shift loop instead.

T24. Missing virtual destructor

UB

```
struct B { ~B(); };            // NOT virtual
struct D : B { int *buf; ~D(); };
B *p = new D();  delete p;
```

A **Undefined behaviour** — deleting a derived object through a base pointer whose destructor is not `virtual`. In practice `~D()` never runs and `buf` leaks. Rule: **any class used as a polymorphic base needs a virtual destructor.**

// §30 TOP 30 ECE INTERVIEW QUESTIONS — ANSWER IN 10 SECONDS

1. Why is C used in embedded?	Thin, predictable mapping to machine code; no hidden allocation, dispatch or runtime; direct memory/register access; tiny portable library; every vendor ships a C compiler and every RTOS/driver stack is written in it.
2. Why are pointers important?	They are how C names hardware addresses (MMIO), passes data without copying, builds dynamic structures, and implements callbacks and vector tables.
3. What is `volatile`?	A qualifier telling the compiler the object may change outside normal program flow, so every access must actually be performed, in order, and never cached in a register or optimised away.
4. Is `volatile` atomic?	**No.** It gives visibility, not indivisibility. `v++` is still load-modify-store. Use an atomic type or a critical section for atomicity.
5. Set / clear / toggle a bit?	`x |= (1U<<n);` · `x &= ~(1U<<n);` · `x ^= (1U<<n);` · read with `(x>>n)&1U`. Always `1U`, never `1`.
6. What is memory-mapped I/O?	Peripheral registers appear as addresses in the CPU's normal address space, so ordinary loads and stores drive hardware: `#define REG (*(volatile uint32_t*)0x40000000U)`.
7. What is an ISR?	The function the CPU jumps to via the vector table when an interrupt fires. No arguments, no return value, never called by your code; it must be short and must clear the interrupt flag.
8. Polling vs interrupt?	Polling burns CPU and cannot sleep, but is simple and has no concurrency issues. Interrupts give low bounded latency and let the CPU idle, at the cost of shared-state and `volatile` discipline. Bulk streams → DMA.
9. What is a race condition?	Two contexts access the same data with at least one writing, and the outcome depends on timing. Fix with a critical section, an atomic operation, or a single-writer design.
10. Stack vs heap?	Stack: compiler-managed, LIFO, fast, small, freed automatically, cannot fragment. Heap: manually managed, large, slow, fragments, leaks.
11. Why avoid `malloc` in firmware?	Fragmentation can fail an allocation while free RAM exists; allocation time is non-deterministic; with no MMU a heap/stack collision silently corrupts RAM; and there is no sensible recovery from out-of-memory in a headless device.

12. What is endianness?	The order of **bytes** of a multi-byte value in memory. Little-endian puts the least significant byte at the lowest address; big-endian the reverse. It is not bit order.
13. What is structure padding?	Unused bytes inserted so each member sits at an address that is a multiple of its alignment, plus trailing bytes so arrays stay aligned — which is why `sizeof(struct)` can exceed the sum of members.
14. Why fixed-width integers?	`int` may be 16 or 32 bits, so register maps, protocol frames and stored data must use `uintN_t` to be portable and to match hardware exactly.
15. What is a function pointer?	A variable holding the address of executable code: `int (*fp)(int,int);`. It powers jump tables, callbacks, plug-in drivers and the interrupt vector table.
16. What is a callback?	A function you register with a lower layer so it can call *up* into your code when an event happens — the standard way a driver notifies an application without depending on it.
17. What is a ring buffer?	A fixed-size FIFO with head and tail indices that wrap around. With one producer and one consumer it is lock-free — the standard UART/DMA queue.
18. What is read-modify-write?	Read a register, change some bits, write it back. It is not atomic (an ISR can interrupt it) and it is wrong on write-1-to-clear registers, where it clears unrelated flags.
19. What is an FSM?	A machine with a finite set of states where the current state plus an input determines the next state and the action — implemented as an `enum` plus a `switch`, or a transition table.
20. `memcpy` vs `memmove`?	`memcpy` is UB if the regions overlap (it may copy in any order); `memmove` handles overlap correctly and is marginally slower.
21. Pointer vs reference?	A pointer can be null, reseated and used in arithmetic, and needs explicit dereference. A reference must be bound at initialisation, can never be reseated, and is used like the object itself.
22. What is a virtual function?	A member function whose call is resolved at run time from the object's actual type, via a vptr into the class's vtable — runtime polymorphism.
23. Why a virtual destructor?	Deleting a derived object through a base pointer with a non-virtual destructor is UB: only the base destructor runs, so derived resources leak. Any polymorphic base needs one.
24. What is RAII?	Tie a resource's lifetime to an object: acquire in the constructor, release in the destructor. Scope exit — including early return and exceptions — cannot leak.
25. `nullptr` vs `NULL`?	`NULL` is an integer null-pointer constant, so `f(NULL)` can select an `int` overload. `nullptr` has its own type `std::nullptr_t`, converts only to pointers, and is type-safe.
26. `const` vs `volatile`?	`const` = "I will not modify it through this path" (enables optimisation). `volatile` = "something else may modify it" (disables it). `const volatile` = a read-only status register.
27. `malloc` vs `new`?	`malloc` returns untyped raw bytes and never runs a constructor; `new` is typed, constructs the object, throws on failure, and is overridable. Pair with `free` / `delete` respectively — never mix.
28. `struct` vs `class`?	In C++ only the defaults differ: `struct` members and inheritance are public, `class` private. In C a `struct` is data only — no member functions, no access control.
29. Why do signed/unsigned comparisons bug out?	The usual arithmetic conversions convert the signed operand to unsigned, so a negative value becomes huge: `-2 < 1U` is **false**, and `for (unsigned i = n-1; i >= 0; i--)` never terminates.
30. What causes undefined behaviour?	Out-of-bounds access, dereferencing null/dangling pointers, use-after-free and double free, signed overflow, shifting by ≥ the type width, modifying a string literal, misaligned access, strict-aliasing violations, and data races.

// §31 CLASSIC "IMPLEMENT IT" OA TASKS

Write these from memory. In an OA, state your **edge cases** (NULL input, empty string, overlap, overflow) *before* you code — that is most of the mark.

31.1 The string / memory primitives

```
size_t my_strlen(const char *s) {
    const char *p = s;
    while (*p != '\0') { p++; }      // walk to the terminator
    return (size_t)(p - s);          // ptrdiff -> length
}

char *my_strcpy(char *d, const char *s) {
    char *r = d;                     // keep the start to return
    while ((*d++ = *s++) != '\0') { }  // copies NUL, then stops
    return r;                        // NOTE: no bound - see snprintf
}

int my_strcmp(const char *a, const char *b) {
    while (*a != '\0' && *a == *b) { a++; b++; }
    return (int)(unsigned char)*a - (int)(unsigned char)*b;
}                                    // compare as UNSIGNED char
```

```
void *my_memcpy(void *d, const void *s, size_t n) {
    uint8_t *dp = d;  const uint8_t *sp = s;
    while (n-- > 0U) { *dp++ = *sp++; }   // forward only
    return d;                             // UB if they overlap
}

void *my_memmove(void *d, const void *s, size_t n) {
    uint8_t *dp = d;  const uint8_t *sp = s;
    if (dp < sp) {                        // copy FORWARD
        while (n-- > 0U) { *dp++ = *sp++; }
    } else {                              // overlap -> go BACKWARD
        dp += n;  sp += n;
        while (n-- > 0U) { *--dp = *--sp; }
    }
    return d;
}
```

The mark they look for: `my_memmove` chooses the copy direction from the pointer order — that single `if` is the whole difference from `memcpy`. In `my_strcmp` the operands must be converted to `unsigned char`, or bytes ≥ 0x80 compare wrongly where `char` is signed.

31.2 Reverse, parse, classify

```
void reverse_str(char *s) {          // in place, two indices
    size_t i = 0U, j = my_strlen(s);
    while (j > i) {
        j--;
        char t = s[i];  s[i] = s[j];  s[j] = t;
        i++;
    }
}

int my_atoi(const char *s) {         // no overflow handling:
    int sign = 1, v = 0;             //   SAY SO in the interview
    if (*s == '-') { sign = -1; s++; }
    else if (*s == '+') { s++; }
    while (*s >= '0' && *s <= '9') {
        v = v * 10 + (*s - '0');     // signed overflow = UB
        s++;
    }
    return sign * v;
}

bool is_pow2(uint32_t x) {
    return (x != 0U) && ((x & (x - 1U)) == 0U);
}
```

31.3 Pointer-heavy classics

```
typedef struct Node { int v; struct Node *next; } Node;

Node *reverse_list(Node *head) {     // iterative, O(n), O(1) space
    Node *prev = NULL;
    while (head != NULL) {
        Node *next = head->next;     // save BEFORE overwriting
        head->next = prev;           // flip the link
        prev = head;                 // advance both
        head = next;
    }
    return prev;                     // the new head
}

int bsearch_idx(const int *a, size_t n, int key) {
    size_t lo = 0U, hi = n;          // half-open [lo, hi)
    while (lo < hi) {
        size_t mid = lo + (hi - lo) / 2U;   // NOT (lo+hi)/2: overflow
        if      (a[mid] == key) { return (int)mid; }
        else if (a[mid] <  key) { lo = mid + 1U; }
        else                    { hi = mid; }
    }
    return -1;                       // not found
}
```

Two details that separate answers: compute the midpoint as `lo + (hi-lo)/2` — `(lo+hi)/2` can overflow; and in `reverse_list` you must save `head->next` *before* you overwrite it, or the rest of the list is lost. Both are asked about explicitly.

31.4 Acronyms you are expected to recognise

Term	Meaning	Term	Meaning
MMIO	memory-mapped I/O	**RMW**	read-modify-write
ISR	interrupt service routine	**W1C**	write-1-to-clear
IRQ	interrupt request	**NVIC**	nested vectored interrupt controller (Cortex-M)
DMA	direct memory access	**MPU**	memory protection unit
MMU	memory management unit (virtual memory)	**BSS**	the zero-initialised data section

Term	Meaning	Term	Meaning
UB	undefined behaviour	**TU**	translation unit (one .c after preprocessing)
RAII	resource acquisition is initialisation	**SPSC**	single producer, single consumer
FSM	finite state machine	**POD**	plain old data (no ctor/vtable)
MISRA	safety-critical C coding standard	**HAL**	hardware abstraction layer
LSB / MSB	least / most significant bit or byte	**ABI**	application binary interface (calling convention)

Last-minute drill: cover the right-hand column of §33 and rebuild each pattern from memory — set/clear/toggle/test a bit, extract and insert a field, declare an MMIO register, and write the two `const`-pointer forms. If you can write those six from a blank page, you can answer most of the C portion of an embedded OA.

// §32 50 THINGS TO MEMORISE

1. An array is N contiguous objects; a pointer is one object holding an address. **They are not the same thing.**
2. `sizeof(array)` is the whole array; `sizeof(pointer)` is the address width.
3. In a parameter list `int a[5]`, `int a[]` and `int *a` are identical — the size is discarded.
4. An array decays to `&a[0]` everywhere except `sizeof a`, `&a`, and string-literal initialisation.
5. `a[i]` is `*(a+i)`, so `i[a]` is legal C.
6. `a+1` advances one element; `&a+1` advances the **whole array**.
7. Pointer arithmetic is scaled by `sizeof(*p)`; going outside `[a, a+N]` is UB.
8. `sizeof(char)` is exactly 1 by definition; `sizeof(int)` is **not** fixed by the standard.
9. Plain `char` is a third type distinct from `signed` / `unsigned char`; its signedness is implementation-defined.
10. A C string is terminated by `'\0'`, and that byte costs storage: `sizeof("hello")` is 6, `strlen` is 5.
11. Modifying a string literal is **undefined behaviour**.
12. `strncpy` may leave the destination without a NUL terminator.
13. `memcpy` on overlapping regions is UB — use `memmove`.
14. `memset` fills **bytes**: `memset(arr,1,n)` does not make ints equal to 1.
15. Never `memcmp` two structs — padding bytes are indeterminate.
16. Dereferencing a NULL, wild or dangling pointer is UB.
17. A dangling pointer refers to an object whose **lifetime has ended**; `free(p); p = NULL;` prevents use-after-free and double free.
18. `free(NULL)` is a safe no-op; freeing twice is UB.
19. Never write `p = realloc(p, n);` — on failure you lose the original block.
20. Returning the address of a local variable gives a dangling pointer.
21. `malloc` does not initialise memory; `calloc` sets every byte to zero.
22. Always check the allocation result; `malloc(0)` may legitimately return `NULL`.
23. Locals hold indeterminate values; globals and statics are zeroed before `main`.
24. A `static` local is initialised once and keeps its value between calls.
25. `static` at file scope means internal linkage; `extern` declares but never defines storage.
26. You may not take the address of a `register` variable.
27. On an MCU with no MMU, a stack overflow is silent corruption — not a crash.
28. Startup code copies `.data` from flash to RAM and zeroes `.bss` before `main` runs.
29. **Signed** overflow is UB; **unsigned** arithmetic wraps modulo 2^N.
30. Shifting by ≥ the width of the type, or by a negative count, is UB.
31. `1 << 31` on a 32-bit `int` is UB — write `1U << 31`.
32. Right-shifting a negative value is implementation-defined.
33. Types narrower than `int` are promoted to `int` before arithmetic: `~(uint8_t)0xFF` is `0xFFFFFF00`.
34. In a mixed comparison of equal rank the signed operand converts to **unsigned**: `-2 < 1U` is false.
35. An unsigned loop counter is never negative, so `i >= 0` loops forever.
36. `i = i++` and `printf("%d%d", i++, i++)` are UB in C.
37. The evaluation order of function arguments is **unspecified**.
38. `&` is bitwise AND, `&&` is logical AND; likewise `|` / `||`, `~` / `!`, and `=` / `==`.
39. Bitwise `& ^ |` bind **looser** than `==` and `!=` — always parenthesise.
40. `+` binds tighter than `<<`, so `1 << 2 + 3` is 32.
41. `*p++` means `*(p++)`.
42. Parenthesise every macro parameter and the whole body; a macro argument may be evaluated more than once.
43. `sizeof(struct)` may exceed the sum of its members because of padding; declare members largest-first.
44. Bit-field layout and bit order are implementation-defined — use explicit shifts and masks for hardware registers.
45. `volatile` is not atomic, not thread-safe and not a CPU memory barrier.
46. On a write-1-to-clear register, `REG |= MASK;` clears flags you never handled — write `REG = MASK;`.
47. Endianness is the order of **bytes** in memory, not the order of bits.
48. `new[]` pairs with `delete[]`; mixing `new` / `free` or `new[]` / `delete` is UB.
49. Any class used as a polymorphic base needs a `virtual` destructor; C++ members are initialised in **declaration order** and destroyed in reverse.
50. `std::move` only casts — it permits a move; and a `vector` reallocation invalidates every iterator and pointer into it.

// §33 FINAL RAPID REVISION SHEET

C → DATA TYPES → OPERATORS → CONTROL FLOW → FUNCTIONS → ARRAYS → STRINGS →
POINTERS → const/volatile → MEMORY (stack · heap · .data · .bss · .text) → STORAGE CLASSES →
STRUCT / UNION / ENUM → PADDING & ALIGNMENT → INTEGER CONVERSIONS → UNDEFINED BEHAVIOUR →
BIT MANIPULATION → MMIO → REGISTERS (RMW · W1C) → ISR → RACE / CRITICAL SECTION →
RING BUFFER → FSM → ENDIAN → UART/SPI/I2C/CAN →
C++ → CLASSES → OOP (encapsulation · abstraction · inheritance · polymorphism) → VIRTUAL →
REFERENCES → RAII → SMART POINTERS → MODERN C++ → STL → INTERVIEW TRAPS

The patterns to have in muscle memory

```
x |= (1U << n);                 // SET bit n
x &= ~(1U << n);                // CLEAR bit n
x ^= (1U << n);                 // TOGGLE bit n
(x >> n) & 1U;                  // READ bit n  -> 0 or 1
if (x & (1U << n)) { }          // TEST bit n

mask  = (1U << n) - 1U;                          // n low bits
field = (reg >> SHIFT) & MASK;                   // EXTRACT a field
reg = (reg & ~FMASK) | ((v << SHIFT) & FMASK);   // INSERT: clear 1st

#define REG (*(volatile uint32_t *)0x40000000U)      // MMIO register
volatile uint32_t *const p = (volatile uint32_t *)ADDR;

int  x = 10;
int *p = &x;      *p = 20;   // pointer: address in, value out
const int *cp;               // the DATA is const
int *const pc = &x;          // the POINTER is const

size_t n = sizeof a / sizeof a[0];  // only where a is an ARRAY
buf[head & (SIZE - 1U)] = v;        // ring wrap, SIZE = 2^k
if (p != NULL) { }                  // check every pointer
free(p); p = NULL;                  // kill the dangling pointer
```

The ten rules that carry the most marks

1. Say "**undefined behaviour**" — never invent an output for UB.
2. `volatile` = visibility. Atomics / critical sections = indivisibility.
3. Arrays decay; `sizeof` inside a function is a pointer size.
4. Never mix signed and unsigned in a comparison.
5. Parenthesise bitwise expressions and every macro argument.
6. Use `1U` in shifts, and fixed-width types for hardware.
7. Clear a bit-field before OR-ing a new value in; never RMW a W1C register.
8. Every `malloc` has exactly one `free` — and firmware prefers static buffers.
9. Keep ISRs short, clear the flag, defer the work, mark shared state `volatile`.
10. In C++: RAII, `unique_ptr`, virtual destructors, and pass big objects by `const&`.