

In one line: **UB** = the standard places *no requirements* on the result — and the optimiser **assumes it never happens**, so it deletes checks and rewrites code around it; the bug shows up far away, often only at `-O2`. **Building** = per *translation unit* preprocess → compile → assemble to a `.o`, then **link** every `.o` + library into one image with real addresses.

The UB catalogue

UB	example · the defined alternative
signed overflow	<code>INT_MAX + 1</code> · unsigned wraps (defined)
shift $\geq$ width	<code>x << 32, 1 << 31, count < 0 · 1u << 31</code>
out of bounds	<code>a[5]</code> of <code>int a[5]</code> , reads too
null deref	<code>*p</code> with <code>p == NULL</code> · check <i>before</i> use
uninitialised read	<code>int x; return x;</code> · initialise it
use-after-free	<code>free(p); *p · p = NULL</code> after free
strict aliasing	<code>*(uint32_t *)&f · memcpy, union</code>
unsequenced	<code>i = i++; a[i] = i++;</code> · split them
data race	2 threads, 1 writes, no <code>sync · _Atomic, mutex</code>
literal write	<code>char *s = "hi"; s[0]='H';</code> · <code>char s[]</code>
div by zero	<code>x / 0, INT_MIN / -1</code> · (float: <code>inf/NaN</code>)
misaligned	<code>cast to <code>uint32_t *</code> of <code>buf+1 · memcpy</code></code>

Why the optimiser “breaks” it

```
int read_dev(struct dev *d) {
    int v = d->val; // deref => compiler infers d != NULL
    if (!d) return -1; // ...so this check is DELETED
    return v;
}
bool will_wrap(int x) { return x + 1 < x; } // folded: false
```

- Not “a garbage value”: a promise *you* made. Debug (`-O0`) works, `-O2` breaks → latent UB. Remove the UB; never “fix” it by lowering `-O`.
- Tests can’t prove absence: a path may look right today.

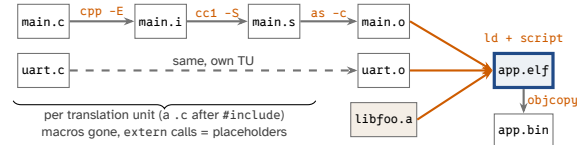
Catch it

- `-Wall -Wextra -Werror (+ -Wshadow -Wconversion)`.
- `-fsanitize=address`: OOB, use-after-free/return, double free, leaks. `-fsanitize=undefined`: overflow, shifts, null, misaligned. `-fsanitize=thread`: races (not with ASan). Uninitialised reads: `clang MSan / Valgrind`. Run on host unit tests.
- On the ESP32: heap poisoning, stack-smashing protection, FreeRTOS stack canaries (`menuconfig`); `cppcheck, clang --analyze`.

Static vs dynamic libraries

`.a` = archive of `.o`; only needed objects are copied in at link; order matters (`gcc main.o -lfoo`). `.so/.dylib` = loaded at run time, shared, updatable, needs PIC. MCU firmware is all static (each ESP-IDF component is a `.a`).

Picture — the pipeline



Linkage and the one-definition rule

- External** (default for functions/globals): one name across TUs. **Internal**: static at file scope = private to the `.c`. **None**: locals. (static in a function = storage, not linkage.)
- Header *declares* (extern `int count;`), exactly one `.c` *defines* (`int count = 0;`). `int count;` in a header → **multiple definition** (GCC ≥ 10 : `-fno-common`). Declared, never defined → **undefined reference**.
- Linker: resolves symbols, relocates to final addresses, places sections by the **linker script** (`.text/.data/.bss/IRAM` → flash/RAM), drops dead code (`-ffunction-sections -fdata-sections -WL,--gc-sections`).
- `__attribute__((weak))`: default, overridden by any strong definition. `-WL,-Map=app.map, size, nm, idf.py` size-components show who uses the flash.

Headers and macros

```
#ifndef UART_H // or: #pragma once */
#define UART_H
#define SQ(x) ((x) * (x)) // parens: arg AND body
#define SWAP(a,b) do{ int t_=(a); (a)=(b); (b)=t_; }while(0)
static inline int sq(int x) { return x * x; } // prefer
#endif
```

- `SQ(i++)` evaluates `i++` twice (UB); `do{}while(0)` makes `if (c) SWAP(x,y);` else parse; `#if F00` with `F00` undefined is silently `0`. `#x` stringifies, `a##b` pastes.

Interview traps

- ! Your Q22:** UB is not “may be `0` or `-1`” — no requirements; name 3 kinds.
- ! Your Q23:** the linker’s jobs: symbols, addresses, linker script, `--gc-sections` — not vague “optimisations”.
- ! `-O0` debug · `-Og` ESP-IDF default · `-Os` size · `-O2` speed.**

Remember

UB is a promise you made to the optimiser. `cpp` → `cc` → `as` per TU, then `ld` once. **Declare in the `.h`, define once in a `.c`.**

Likely questions

- Undefined reference vs multiple definition? — never defined vs defined twice.
- static on a global? — internal linkage: invisible to other `.c` files.
- Why `memcpy` for type punning? — aliasing-safe; compiles to one load.