Understanding #[repr(C)] in Rust

This is a concise, corrected version of the original material with edits applied directly. Scope: predictable layout via C ABI, padding/alignment realities, safe parsing patterns, and when to use/avoid #[repr(C)]. Assumptions: 64-bit, little-endian target (e.g., Apple Silicon/macOS) unless stated.

Revision Summary (Applied Edits)

• Corrected claim about default layout producing 16-byte struct — actual size is typically 24 unless fields are manually reordered.

• Clarified that #[repr(C)] is stable for the target's C ABI, not universally across platforms.

• Added explicit alignment and endianness caveats for unsafe byte casts; provided safer parsing pattern.

• Strengthened warnings for #[repr(packed)] with unaligned access; showed read_unaligned usage.

• Tightened guidance on enums (fieldless vs data-carrying).

• Added a practical test/checklist section: size_of/align_of and (optional) field offset checks.

1) What #[repr(C)] DOES

• Uses the C ABI's field order and alignment for structs, exactly as written in source.

• Produces a predictable, C-compatible layout on the CURRENT TARGET (not a universal layout).

• Retains padding required by the C ABI—#[repr(C)] does not remove padding.

2) What #[repr(C)] Does NOT Do

• It does NOT automatically compress structs or remove padding.

• It does NOT guarantee the same layout on all platforms—only compatibility with the present target's C ABI.

• It does NOT make pointer casts safe: alignment and endianness still matter.

3) When to Use vs Avoid

Use: FFI with C, binary/network/file formats that specify a fixed layout, carefully-designed register maps.

Avoid: Pure Rust internals where layout freedom aids optimization, or when space/perf is critical without C interop.

4) Related reprs

• #[repr(transparent)]: one-field wrapper has the same layout as the field; ideal for FFI newtypes.

• #[repr(packed)]: removes padding; USE WITH CAUTION—requires unaligned access patterns.

• #[repr(align(N))]: increases alignment (e.g., SIMD/cache-line).

5) Corrected Example — Sizes, Alignment & Padding

The original text implied the compiler would reorder fields under #[repr(Rust)] to yield a smaller size.

In practice, repr(Rust) layout is unspecified; do not rely on any order. On common 64-bit targets:

```
// 64-bit target (typical results)
use std::mem::{size_of, align_of};

struct S1 { a: u8, b: u64, c: u8 }          // likely size 24, align 8
#[repr(C)] struct S2 { a: u8, b: u64, c: u8 } // size 24, align 8 (C layout)
struct S3 { b: u64, c: u8, a: u8 }          // size 16, align 8 (manual reordering)

fn main() {
   println!("S1 size={} align={}", size_of::<S1>(), align_of::<S1>());
   println!("S2 size={} align={}", size_of::<S2>(), align_of::<S2>());
   println!("S3 size={} align={}", size_of::<S3>(), align_of::<S3>());
}
```

6) Safer Parsing From Bytes (Avoid Whole-Struct Transmutes)

Prefer parsing integral fields with from_be/le/ne_bytes and constructing the struct explicitly. This avoids alignment/endianness pitfalls and keeps invariants clear.

```
#[repr(C)]
struct Header { magic: u32, length: u32, checksum: u32 }

fn parse_header_be(pkt: &[u8]) -> Option<Header> {
   if pkt.len() < 12 { return None; }
   let magic    = u32::from_be_bytes(pkt[0..4].try_into().ok()?);
   let length   = u32::from_be_bytes(pkt[4..8].try_into().ok()?);
   let checksum = u32::from_be_bytes(pkt[8..12].try_into().ok()?);
   Some(Header { magic, length, checksum })
}
```

If alignment and endianness are fully controlled and guaranteed, you can use unsafe casts, but document those guarantees and add tests.

7) #[repr(packed)] — Unaligned Access Caveat

Accessing packed fields via references can cause undefined behavior (unaligned loads). Use read_unaligned/write_unaligned or copy to an aligned local.

```
#[repr(packed)]
struct Packed { a: u8, b: u64, c: u8 }

fn read_b(p: &Packed) -> u64 {
   unsafe { core::ptr::read_unaligned(&p.b as *const u64) }
}
```

8) Enums & #[repr(C)]

• Fieldless enums with #[repr(C)] map to a C-like integer representation (e.g., repr(u32)).

• Data-carrying enums are not simple C structs/unions; do not assume C compatibility without a precise ABI strategy.

9) Testing & Checklist

• Validate with size_of::<T>(), align_of::<T>().

• Where critical, verify offsets (e.g., memoffset::offset_of!).

• Confirm endianness and alignment preconditions in FFI/binary parsing code.

• Only use #[repr(C)] when layout compatibility actually matters; otherwise prefer default layout for flexibility.