Chapter 1: Memory Layout and Alignment

> *"Your performance intuition is useless. Run perf."*
> — Comment in Rust's layout.rs

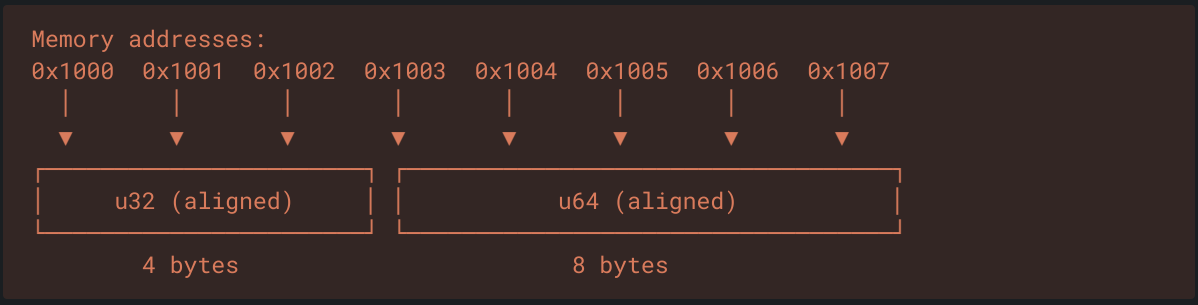
Introduction

Before Rust can allocate memory, it needs to answer two fundamental questions: 1. **How many bytes do I need?** 2. **What address boundaries must the memory respect?**

The `Layout` type encapsulates these requirements. It's the foundation upon which all of Rust's allocation machinery is built.

1.1 What is Memory Alignment?

At the hardware level, CPUs access memory most efficiently when data sits at addresses that are multiples of the data's size.



A 32-bit integer (4 bytes) wants addresses like `0x1000`, `0x1004`, `0x1008` — addresses divisible by 4. A 64-bit value wants addresses divisible by 8.

Why Does Alignment Matter?

Hardware efficiency: On many architectures, misaligned access requires multiple memory bus cycles instead of one. On x86, it's slower. On older ARM, it causes a hardware fault and crashes your program.

Atomics: Atomic operations often require alignment to work correctly. A misaligned atomic might not be atomic at all.

SIMD: Vector instructions typically require 16-byte or 32-byte alignment.

1.2 The Layout Type

Layout lives in `core::alloc` and has exactly two fields:

```
pub struct Layout {
    size: usize,      // How many bytes
    align: Alignment, // What address boundaries (power of 2)
}
```

That's it. Everything else is methods for constructing, validating, and manipulating these two numbers.

The Core Invariant

This is critical and worth memorizing:

> **"The size, when rounded up to the nearest multiple of align, does not overflow isize."**

Why `isize` and not `usize`? Because pointer arithmetic in Rust uses signed offsets. The `ptr.offset(i)` method takes an `isize`. If your allocation exceeds `isize::MAX`, pointer arithmetic could overflow, causing undefined behavior.

The maximum size for a given alignment is calculated as:

```
const fn max_size_for_align(align: Alignment) -> usize {
    // max_size = (isize::MAX + 1) - align
    unsafe { unchecked_sub(isize::MAX as usize + 1, align.as_usize()) }
}
```

On a 64-bit system with 8-byte alignment, the maximum allocation is approximately **8 exabytes** (8,589,934,591 GB). You won't hit this limit.

1.3 Creating Layouts

Safe Constructor

```
pub const fn from_size_align(size: usize, align: usize) -> Result<Self,
LayoutError>
```

This validates three things: 1. `align` must be non-zero 2. `align` must be a power of two 3. `size` must not exceed `max_size_for_align`

Type-Based Constructors

```
// For any sized type - computed at compile time
let layout = Layout::new::<i32>();  // size=4, align=4

// For dynamically-sized types (slices, trait objects)
let layout = Layout::for_value(&some_slice);

// For arrays
let layout = Layout::array::<u32>(100)?;  // 100 u32s
```

The Unsafe Escape Hatch

```
pub const unsafe fn from_size_align_unchecked(size: usize, align: usize) -> Self
```

Skips validation. In debug builds, it still asserts. In release builds, it trusts you completely. Use only when you can prove the invariants hold.

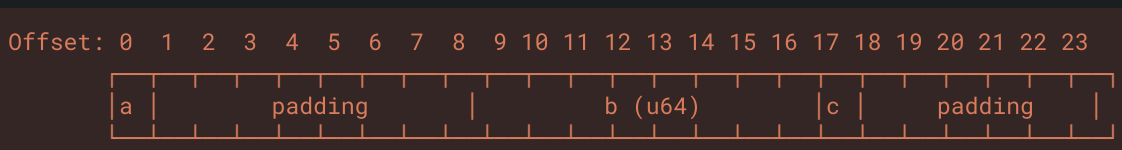
1.4 Struct Layout and Padding

When Rust lays out a struct, it must ensure each field is properly aligned. This often requires **padding** — unused bytes inserted between fields.

C-Compatible Layout (`#[repr(C)]`)

```rust
#[repr(C)]
struct PaddedStruct {
    a: u8,   // offset 0, size 1
    // 7 bytes padding (to align b to 8)
    b: u64,  // offset 8, size 8
    c: u8,   // offset 16, size 1
    // 7 bytes padding (to make total size multiple of 8)
}
// Total: 24 bytes, align 8
```

Memory layout:



Rust's Default Layout

Without `#[repr(C)]`, Rust can **reorder fields** to minimize padding:

```rust
struct RustStruct {
    a: u8,
    b: u64,
    c: u8,
}
// Rust reorders to: b, a, c
// Layout: 8 + 1 + 1 + 6(padding) = 16 bytes
```

This is why you can't reliably take pointers to struct fields and do arithmetic between them unless you use `#[repr(C)]`.

1.5 Combining Layouts

The `extend` Method

Places one layout after another with proper padding:

```rust
let a = Layout::new::<u8>();   // size=1, align=1
let b = Layout::new::<u32>();  // size=4, align=4

let (combined, offset_of_b) = a.extend(b)?;
// combined: size=8, align=4
// offset_of_b: 4 (after 3 bytes of padding)
```

This is how you manually compute struct layouts:

```rust
pub fn repr_c(fields: &[Layout]) -> Result<(Layout, Vec<usize>), LayoutError> {
    let mut offsets = Vec::new();
    let mut layout = Layout::from_size_align(0, 1)?;

    for &field in fields {
        let (new_layout, offset) = layout.extend(field)?;
        layout = new_layout;
        offsets.push(offset);
    }

    // Don't forget trailing padding!
    Ok((layout.pad_to_align(), offsets))
}
```

The `pad_to_align` Method

Adds trailing padding so the total size is a multiple of alignment. This is essential for arrays — without it, elements beyond the first would be misaligned.

```rust
let layout = Layout::from_size_align(5, 4)?;  // size=5, align=4
let padded = layout.pad_to_align();           // size=8, align=4
```

The `repeat` Method

Creates a layout for an array of `n` elements:

```rust
let element = Layout::new::<u32>();
let (array_layout, stride) = element.repeat(10)?;
// array_layout: size=40, align=4
// stride: 4 (distance between elements)
```

1.6 The Bit Manipulation Tricks

Layout code uses fast bitwise operations because alignment is always a power of two.

Rounding Up to Alignment

```
// Round size up to next multiple of align
fn round_up(size: usize, align: usize) -> usize {
    (size + align - 1) & !(align - 1)
}
```

How it works: - `align - 1` gives a mask of the low bits (e.g., align=8 → mask=0b111) - Adding `align - 1` ensures we round up, not down
- `!(align - 1)` inverts the mask - AND-ing clears the low bits, rounding down to alignment

Example with size=5, align=4:

```
size + align - 1 = 5 + 3 = 8 = 0b1000
!(align - 1)     = !3    = ...11111100
8 & ...11111100  = 8
```

Calculating Padding

```
fn padding_needed(size: usize, align: usize) -> usize {
    let rounded = (size + align - 1) & !(align - 1);
    rounded - size
}
```

1.7 Zero-Sized Types

Rust has types with size 0:

```
let unit_layout = Layout::new::<()>();      // size=0, align=1
let empty_layout = Layout::new::<[u8; 0]>(); // size=0, align=1

struct Empty;
let empty_struct = Layout::new::<Empty>();  // size=0, align=1
```

Arrays of ZSTs also have size 0:

```rust
let million_units = Layout::array::<()>(1_000_000)?;
// size=0, align=1
// A million units take zero bytes!
```

This has important implications for allocation — you can't actually allocate zero bytes from most allocators, so Rust handles ZSTs specially (we'll see this in Chapter 3).

1.8 Key Takeaways

1. **Layout = size + alignment**, nothing more
2. **Alignment must be a power of two** — required by hardware and OS allocators
3. **Size limit is isize::MAX** — to ensure pointer arithmetic is safe
4. **Padding is invisible but real** — it affects struct sizes and memory usage
5. **Rust can reorder struct fields** — use `#[repr(C)]` for predictable layout
6. **Zero-sized types have size 0** — they're handled specially throughout the allocation system

---

Source Files

File	Purpose
`library/core/src/alloc/layout.rs`	The Layout type and all its methods
`library/core/src/alloc/mod.rs`	Module organization and re-exports

Exercises

1. Calculate the layout of this struct by hand, then verify with `Layout::new`:

   ```rust
   #[repr(C)]
   struct Mixed { a: u8, b: u32, c: u8, d: u16 }
   ```

2. Why can't alignment be zero? What would happen if you tried?

3. The `extend` method doesn't add trailing padding. Why is `pad_to_align` a separate step?
4. What's the maximum number of `u8` values you could store in a single allocation?

Next Chapter

Chapter 2: The Allocator Traits →

We'll see how Layout gets used by the allocation interfaces — `GlobalAlloc` and `Allocator` — and trace the path from Rust code down to `libc::malloc`.