Rust Allocation from First Principles

A deep dive into how Rust manages memory, from raw bytes to high-level collections.

Overview

This guide takes you through Rust's allocation system by reading the actual standard library source code. Rather than learning abstractions first, we start at the bottom — how memory is requested from the operating system — and build up to the smart pointers and collections you use every day.

Prerequisites

- Basic Rust syntax (variables, functions, structs, enums)
- Comfort with the idea of pointers (even if not with raw pointer manipulation)
- Curiosity about how things work under the hood

Chapters

Chapter 1: Memory Layout and Alignment

Before allocating memory, Rust must know *what* to allocate. The `Layout` type captures two pieces of information: how many bytes, and what address boundaries the memory must respect.

Key concepts: - Memory alignment and why CPUs care - The Layout type and its invariants - Struct padding and field ordering - The isize::MAX size limit

Chapter 2: The Allocator Traits

With Layout defining requirements, we need interfaces for actually getting memory. Rust provides two traits: the stable `GlobalAlloc` and the future-facing `Allocator`.

Key concepts: - GlobalAlloc: simple, stable, production-ready - Allocator: sophisticated, handles ZSTs, returns fat pointers - Safety contracts and undefined behavior - The Unix implementation calling libc

Chapter 3: Box — Owned Heap Allocation

`Box<T>` is Rust's simplest smart pointer: a single heap allocation with ownership. It's the foundation for understanding how Rust combines allocation with the ownership system.

Key concepts: - Box structure: just a pointer (plus allocator) - The allocation path: from Box::new to malloc - Drop behavior: content destructors then deallocation - Zero-sized types and dangling pointers - Raw pointer escape hatches for FFI

Chapter 4: Vec — Dynamic Arrays

`Vec<T>` adds dynamic sizing: the ability to grow and shrink at runtime. This introduces capacity management, growth strategies, and reallocation.

Key concepts: - The (ptr, len, cap) triplet - RawVec: the allocation engine - Amortized O(1) push via doubling - Reallocation: in-place vs allocate-copy-free - ZST handling with infinite capacity

Source Code References

All content is based on the Rust standard library source:

Topic	Source File
Layout	`library/core/src/alloc/layout.rs`
GlobalAlloc	`library/core/src/alloc/global.rs`
Allocator	`library/core/src/alloc/mod.rs`
Unix allocator	`library/std/src/sys/alloc/unix.rs`
Box	`library/alloc/src/boxed.rs`
Vec	`library/alloc/src/vec/mod.rs`
RawVec	`library/alloc/src/raw_vec.rs`

Companion Code

The `box_exploration.rs` file contains runnable examples demonstrating: - Box memory layout inspection - Layout calculations for various types - Zero-sized type behavior - Drop order visualization - Raw pointer round-trips - Manual allocation/deallocation

Learning Path

```
Chapter 1: Layout
      │
      ▼
Chapter 2: Allocator Traits
      │
      ├──────────────────┐
      ▼                  ▼
Chapter 3: Box    Chapter 4: Vec
      │                  │
      └──────┬───────────┘
             ▼
    Further exploration:
    String, HashMap, etc.
```

Key Insights

1. **Allocation is two questions**: How many bytes? What alignment?
2. **Traits abstract the allocator**: Your code doesn't care if it's malloc, jemalloc, or a custom arena.
3. **Zero-sized types are special**: They never allocate but maintain valid (dangling) pointers.
4. **Drop is two-phase**: Destructors first, then memory deallocation.
5. **Growth strategies matter**: Doubling gives O(1) amortized push; naive growth gives O(n²).
6. **Safety contracts are manual**: The compiler can't verify allocator correctness — you must uphold invariants.

Next Steps

After completing these chapters:

- **Read the Rustonomicon**: Deeper unsafe Rust coverage

- **Implement your own allocator**: Apply what you've learned
- **Study other collections**: HashMap, BTreeMap, VecDeque
- **Explore String**: It's Vec with UTF-8 validation

"Your performance intuition is useless. Run perf."
— Rust standard library source code