Chapter 3: Box — Owned Heap Allocation

> *"Box provides the simplest form of heap allocation in Rust."*
> — Standard library documentation

Introduction

Having understood memory layouts (Chapter 1) and allocator traits (Chapter 2), we now see how they combine into something useful: `Box<T>`, Rust's simplest smart pointer.

Box is deceptively simple — it's just a pointer to heap memory. But that simplicity hides careful engineering around allocation, deallocation, and the ownership system.

3.1 What is Box?

At its core, Box is: 1. A pointer to heap-allocated memory 2. Ownership of that memory 3. Automatic deallocation when the Box goes out of scope

```
let boxed: Box<i32> = Box::new(42);
// - 4 bytes allocated on heap
// - Value 42 written there
// - boxed holds the pointer (8 bytes on stack)

// When boxed goes out of scope:
// - Memory is automatically freed
```

The Structure

```
pub struct Box<
    T: ?Sized,
    A: Allocator = Global,
>(Unique<T>, A);
```

Two fields: - **Field 0**: `Unique<T>` — A pointer wrapper that asserts exclusive ownership - **Field 1**: `A` — The allocator (defaults to `Global`, which is zero-sized)

For the common case of `Box<T>` with the global allocator, Box is exactly **one pointer** in size — 8 bytes on 64-bit systems.

Why Box Exists

1. Heap allocation for large data:

```
// This 1MB array would overflow the stack
let huge: Box<[u8; 1_000_000]> = Box::new([0u8; 1_000_000]);
// Only 8 bytes on stack, 1MB on heap
```

2. Recursive data structures:

```
enum List<T> {
    Cons(T, Box<List<T>>),  // Box breaks the infinite size
    Nil,
}
// Without Box: "recursive type has infinite size"
```

3. Trait objects:

```
let animal: Box<dyn Animal> = Box::new(Dog { name: "Rex" });
// Box holds a fat pointer: data ptr + vtable ptr
```

4. Transferring ownership efficiently:

```
fn transfer(data: Box<[u8; 1_000_000]>) { ... }
// Moving Box copies 8 bytes, not 1MB
```

3.2 Box Creation: The Allocation Path

The Simple Case: `Box::new`

```
pub fn new(x: T) -> Self {
    return box_new(x);
}

#[rustc_intrinsic]
pub fn box_new<T>(x: T) -> Box<T>;
```

`box_new` is a **compiler intrinsic** — the compiler generates specialized code for it. Why? To avoid unnecessary copies.

Without the intrinsic, the naive implementation would be:

```
fn new_naive(x: T) -> Box<T> {
    let mut uninit = Box::new_uninit();  // Allocate
    uninit.write(x);                     // Copy x to heap
    unsafe { uninit.assume_init() }
}
```

This copies `x` from wherever it starts → stack → heap. The intrinsic lets the compiler write `x` directly to heap memory.

The Real Allocation: `try_new_uninit_in`

```
pub fn try_new_uninit_in(alloc: A) -> Result<Box<MaybeUninit<T>, A>, AllocError>
where
    A: Allocator,
{
    let ptr = if T::IS_ZST {
        NonNull::dangling()
    } else {
        let layout = Layout::new::<MaybeUninit<T>>();
        alloc.allocate(layout)?.cast()
    };
    unsafe { Ok(Box::from_raw_in(ptr.as_ptr(), alloc)) }
}
```

Step by step:

1. **ZST check**: If `T` has zero size, don't allocate — return a "dangling" pointer
2. **Calculate layout**: Get size and alignment requirements
3. **Call allocator**: `alloc.allocate(layout)` — this eventually calls `malloc`
4. **Cast pointer**: From `NonNull<[u8]>` to `NonNull<MaybeUninit<T>>`
5. **Wrap in Box**: `Box::from_raw_in` creates the Box struct

Zero-Sized Types: No Allocation

```
let unit: Box<()> = Box::new(());
```

What happens: - `T::IS_ZST` is true for `()` - `NonNull::dangling()` returns a well-aligned, non-null pointer that points to nothing - No allocator call occurs - Box still has a valid (but meaningless) pointer

The dangling pointer is typically `0x1` for align=1 types — non-null, but obviously not from the heap.

3.3 Box Destruction: The Drop Implementation

```
unsafe impl #[may_dangle] T: ?Sized, A: Allocator> Drop for Box<T, A> {
    fn drop(&mut self) {
        // The T in the Box is dropped by the compiler before this runs

        let ptr = self.0;

        unsafe {
            let layout = Layout::for_value_raw(ptr.as_ptr());
            if layout.size() != 0 {
                self.1.deallocate(From::from(ptr.cast()), layout);
            }
        }
    }
}
```

Drop Order

Critical detail: **T's destructor runs before Box's destructor**.

```
struct Noisy(i32);
impl Drop for Noisy {
    fn drop(&mut self) { println!("Dropping {}", self.0); }
}

{
    let b = Box::new(Noisy(42));
}  // Prints "Dropping 42", then Box::drop runs
```

The compiler inserts `drop_in_place` for the contents, *then* calls `Box::drop` which only handles memory.

The `#[may_dangle]` Attribute

This tells the compiler that even though `T` might contain dangling references after its destructor runs, that's okay — Box won't access `T`'s data, only deallocate the memory.

Zero-Size Check

```
if layout.size() != 0 {
    self.1.deallocate(...);
}
```

For ZSTs, we never allocated, so we must not deallocate. The dangling pointer never touched the allocator.

3.4 Raw Pointer Conversions

Box provides escape hatches for interop with raw pointers:

`into_raw`: Box → Raw Pointer

```
pub fn into_raw(b: Self) -> *mut T {
    let mut b = mem::ManuallyDrop::new(b);
    &raw mut **b
}
```

This: 1. Wraps Box in `ManuallyDrop` to prevent Drop from running 2. Returns the raw pointer

After calling `into_raw`: - You own the raw pointer - You're responsible for eventually freeing the memory - The Box no longer exists

`from_raw`: Raw Pointer → Box

```
pub unsafe fn from_raw(raw: *mut T) -> Self {
    unsafe { Self::from_raw_in(raw, Global) }
}
```

Safety requirements: - Pointer must have come from `Box::into_raw` (or equivalent) - Pointer must not have been freed - Must be called at most once per allocation

The Round-Trip Pattern

```rust
let original = Box::new(String::from("hello"));
let ptr = Box::into_raw(original);

// ... do things with raw pointer ...

let recovered = unsafe { Box::from_raw(ptr) };
// Memory will be freed when recovered is dropped
```

This is essential for FFI — passing heap memory to C code and getting it back.

3.5 Memory Layout and ABI

Box is ABI-Compatible with C Pointers

```rust
// Rust
#[no_mangle]
pub extern "C" fn create_foo() -> Box<Foo> {
    Box::new(Foo::new())
}

// C
struct Foo* create_foo(void);
```

For sized types, `Box<T>` has the exact same representation as `T*` in C. This enables zero-cost FFI.

Box with Custom Allocators

```rust
Box<T, A: Allocator>
```

When `A` is not `Global`: - The allocator instance is stored in the Box - Box size increases by `size_of::<A>()` - The allocator must be used for deallocation

For `Global` (a zero-sized type), no overhead is added.

3.6 The Deref Magic

Box implements `Deref` and `DerefMut`:

```rust
impl<T: ?Sized, A: Allocator> Deref for Box<T, A> {
    type Target = T;

    fn deref(&self) -> &T {
        &**self
    }
}
```

This enables:

```rust
let b: Box<String> = Box::new(String::from("hello"));
println!("{}", b.len());  // Calls String::len through Deref
b.push_str(" world");     // Calls String::push_str through DerefMut
```

The `**self` syntax: - First `*` dereferences the `&self` to get `Box<T>` - Second `*` is the built-in dereference for `Box<T>` to get `T`

3.7 Box and Pinning

```rust
impl<T: ?Sized, A: Allocator> Unpin for Box<T, A> {}
```

Box is **always Unpin**, even when T is not. This is a deliberate design choice:

- Box owns its contents and can always move them
- The address of the Box can change (Box is on the stack)
- But `Pin<Box<T>>` pins the *heap* contents, not the Box itself

```rust
let pinned: Pin<Box<MyFuture>> = Box::pin(my_future);
// The Box can move, but the MyFuture on the heap cannot
```

3.8 Practical Examples

Recursive Data Structures

```
enum BinaryTree<T> {
    Leaf(T),
    Node {
        left: Box<BinaryTree<T>>,
        right: Box<BinaryTree<T>>,
    },
}

let tree = BinaryTree::Node {
    left: Box::new(BinaryTree::Leaf(1)),
    right: Box::new(BinaryTree::Node {
        left: Box::new(BinaryTree::Leaf(2)),
        right: Box::new(BinaryTree::Leaf(3)),
    }),
};
```

Trait Objects

```
trait Animal {
    fn speak(&self);
}

struct Dog;
impl Animal for Dog {
    fn speak(&self) { println!("Woof!"); }
}

let animal: Box<dyn Animal> = Box::new(Dog);
animal.speak();
```

`Box<dyn Animal>` is a **fat pointer**: 16 bytes containing: - Pointer to the data (8 bytes) - Pointer to the vtable (8 bytes)

FFI with C

```
#[no_mangle]
pub extern "C" fn create_buffer(size: usize) -> *mut u8 {
    let buffer: Box<[u8]> = vec![0u8; size].into_boxed_slice();
    Box::into_raw(buffer) as *mut u8
}

#[no_mangle]
pub unsafe extern "C" fn free_buffer(ptr: *mut u8, size: usize) {
    let slice = std::slice::from_raw_parts_mut(ptr, size);
    let _ = Box::from_raw(slice as *mut [u8]);
}
```

3.9 Exploration Program

```
use std::alloc::Layout;
use std::mem;

fn main() {
    // Box is pointer-sized
    println!("Box<i32> size: {} bytes", mem::size_of::<Box<i32>>());
    println!("Box<[u8; 1000]> size: {} bytes", mem::size_of::<Box<[u8; 1000]>>
());

    // ZST Box has a dangling pointer
    let unit_box: Box<()> = Box::new(());
    let ptr = Box::into_raw(unit_box);
    println!("Box<()> pointer: {:p}", ptr);  // Usually 0x1
    let _ = unsafe { Box::from_raw(ptr) };

    // Drop order demonstration
    struct Loud(i32);
    impl Drop for Loud {
        fn drop(&mut self) { println!("Dropping Loud({})", self.0); }
    }

    {
        let outer = Box::new(Loud(1));
        let inner = Box::new(Loud(2));
        println!("Both alive");
    }
    println!("Both dropped (LIFO order)");
}
```

3.10 Key Takeaways

1. **Box is just a pointer** — 8 bytes for the common case
2. **Allocation happens through the Allocator trait** — Eventually calls malloc
3. **ZSTs don't allocate** — They get dangling pointers
4. **Drop is two-phase** — Content destructor first, then memory deallocation
5. **Box is ABI-compatible with C pointers** — Enables zero-cost FFI
6. **Deref makes Box transparent** — You can call T's methods directly

Source Files

File	Purpose
`library/alloc/src/boxed.rs`	Box type and all implementations
`library/alloc/src/boxed/convert.rs`	From/Into implementations
`library/alloc/src/boxed/thin.rs`	ThinBox for DSTs

Exercises

1. What's the size of `Box<Box<i32>>`? Why?
2. Why can't you call `Box::from_raw` twice on the same pointer?
3. Implement a simple smart pointer that's like Box but logs all allocations/deallocations.
4. What happens if you call `mem::forget` on a Box? Is this a memory leak?
5. Why does `Box<[T]>` (boxed slice) need to be a fat pointer, unlike `Box<[T; N]>` (boxed array)?

Next Chapter

Chapter 4: Vec — Dynamic Arrays →

We'll see how `Vec<T>` builds on these primitives to provide a growable array, introducing the complexity of capacity management and reallocation strategies.