Chapter 2: The Allocator Traits

> *"It's undefined behavior if global allocators unwind."*
> — Safety requirements for GlobalAlloc

Introduction

With `Layout` defining *what* memory we need, we now examine *how* to get it. Rust provides two allocator traits:

- **`GlobalAlloc`** — Stable, simple, the current production workhorse
- **`Allocator`** — Unstable, more sophisticated, the future

Both define contracts that any memory allocator must satisfy. We'll trace the path from these abstract traits down to actual `libc::malloc` calls.

2.1 GlobalAlloc: The Stable Interface

```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);

    // Provided defaults (can be overridden for efficiency)
    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8;
    unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) ->
*mut u8;
}
```

Why `unsafe trait`?

The trait itself is unsafe to implement because the compiler cannot verify the guarantees. Implementors must manually ensure:

1. **No unwinding** — Panicking from an allocator is undefined behavior
2. **Correct layouts** — Returned memory must match the requested size and alignment

3. **Valid pointers** — Returned pointers must be usable for reads/writes

The Four Methods

`alloc` — The core allocation method - Takes a `Layout`, returns a raw pointer - Returns null on failure (never panics!) - Memory is uninitialized — contains garbage - **Zero-sized layouts are undefined behavior**

`dealloc` — Frees memory - Must receive a pointer from a previous `alloc` on the same allocator - Must receive the *same layout* used to allocate - Using the wrong layout is undefined behavior

`alloc_zeroed` — Allocation with zero-initialization - Default implementation: `alloc` then `memset` to zero - Can be overridden to use `calloc` for efficiency

`realloc` — Resize an allocation - May return the same pointer (grown in place) or a new one - Copies `min(old_size, new_size)` bytes to the new location - Old pointer becomes invalid after successful realloc

---

2.2 The Safety Contracts

These contracts are **not checked by the compiler**. Violating them causes undefined behavior — crashes, corruption, security vulnerabilities.

Contract for `alloc`

```
unsafe fn alloc(&self, layout: Layout) -> *mut u8;
```

Preconditions (caller must ensure): - `layout.size() > 0` — Zero-sized allocations are UB

Postconditions (implementor must ensure): - Returns null OR a valid pointer that is: - Properly aligned to `layout.align()` - Valid for reads and writes of `layout.size()` bytes - Not overlapping with any other live allocation

Contract for `dealloc`

```
unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
```

Preconditions (caller must ensure): - `ptr` was returned by a previous `alloc` or `realloc` on this allocator - `layout` is the same layout used to allocate - The memory has not already been

deallocated

Postconditions: - The memory is released back to the allocator - The pointer becomes invalid — any use is UB

Contract for `realloc`

```
unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) -> *mut
u8;
```

Preconditions: - Same as `dealloc` for `ptr` and `layout` - `new_size > 0` - `new_size` rounded up to `layout.align()` must not overflow `isize`

Postconditions: - If returns null: original allocation is unchanged, `ptr` still valid - If returns non-null: - Original `ptr` is now invalid - Returned pointer has the new size - First `min(old_size, new_size)` bytes are preserved

---

2.3 The Default Implementations

GlobalAlloc provides default implementations for `alloc_zeroed` and `realloc`. Understanding these reveals optimization opportunities.

Default `alloc_zeroed`

```
unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 {
    let size = layout.size();
    let ptr = unsafe { self.alloc(layout) };
    if !ptr.is_null() {
        unsafe { ptr::write_bytes(ptr, 0, size) };
    }
    ptr
}
```

This is inefficient — it allocates uninitialized memory then writes zeros. A smart implementation uses `calloc`, which: 1. May get pre-zeroed pages from the OS 2. Uses copy-on-write to share zero pages 3. Avoids touching memory until actually needed

Default `realloc`

```
unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) -> *mut
u8 {
    let new_layout = unsafe { Layout::from_size_align_unchecked(new_size,
layout.align()) };
    let new_ptr = unsafe { self.alloc(new_layout) };
    if !new_ptr.is_null() {
        unsafe {
            ptr::copy_nonoverlapping(ptr, new_ptr, cmp::min(layout.size(),
new_size));
            self.dealloc(ptr, layout);
        }
    }
    new_ptr
}
```

This is the naive **allocate-copy-free** strategy: 1. Allocate new block 2. Copy old data 3. Free old block

Real allocators (like glibc) try to **grow in place** first — if there's free space after the current block, just extend it. This is much faster for growing `Vec`s.

2.4 The Unix Implementation

Now let's see how these traits connect to actual OS memory:

```rust
// From library/std/src/sys/alloc/unix.rs

unsafe impl GlobalAlloc for System {
    #[inline]
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        if layout.align() <= MIN_ALIGN && layout.align() <= layout.size() {
            unsafe { libc::malloc(layout.size()) as *mut u8 }
        } else {
            unsafe { aligned_malloc(&layout) }
        }
    }

    #[inline]
    unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
        unsafe { libc::free(ptr as *mut libc::c_void) }
    }

    #[inline]
    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 {
        if layout.align() <= MIN_ALIGN && layout.align() <= layout.size() {
            unsafe { libc::calloc(layout.size(), 1) as *mut u8 }
        } else {
            let ptr = unsafe { self.alloc(layout) };
            if !ptr.is_null() {
                unsafe { ptr::write_bytes(ptr, 0, layout.size()) };
            }
            ptr
        }
    }

    #[inline]
    unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) ->
*mut u8 {
        if layout.align() <= MIN_ALIGN && layout.align() <= new_size {
            unsafe { libc::realloc(ptr as *mut libc::c_void, new_size) as *mut u8
}
        } else {
            unsafe { realloc_fallback(self, ptr, layout, new_size) }
        }
    }
}
```

The Two-Path Strategy

Every method has the same pattern: 1. **Simple path**: If alignment requirements are modest, use standard libc functions 2. **Aligned path**: If alignment is strict, use special aligned allocation

The check `layout.align() <= MIN_ALIGN && layout.align() <= layout.size()` determines which path:

- `MIN_ALIGN` is the guaranteed alignment from `malloc` (usually 8 or 16 bytes)
- The second condition (`align <= size`) works around jemalloc quirks where small allocations might not respect alignment

Aligned Allocation

```
unsafe fn aligned_malloc(layout: &Layout) -> *mut u8 {
    let mut out = ptr::null_mut();
    let align = layout.align().max(size_of::<usize>());
    let ret = unsafe { libc::posix_memalign(&mut out, align, layout.size()) };
    if ret != 0 { ptr::null_mut() } else { out as *mut u8 }
}
```

`posix_memalign` is the POSIX standard for aligned allocation: - First argument: output pointer (modified by the function) - Second argument: alignment (must be ≥ `sizeof(void*)` and power of 2) - Third argument: size - Returns 0 on success, error code on failure

Platform-Specific Bug Workarounds

```
#[cfg(target_vendor = "apple")]
{
    if layout.align() > (1 << 31) {
        return ptr::null_mut();
    }
}
```

Older macOS/iOS versions have a bug where `posix_memalign` with huge alignments (>2GB) returns a pointer that isn't actually aligned. Rather than silently corrupt memory, Rust returns null immediately.

2.5 The Allocator Trait (Unstable)

The newer `Allocator` trait fixes several limitations of `GlobalAlloc`:

```rust
pub unsafe trait Allocator {
    fn allocate(&self, layout: Layout) -> Result<NonNull<[u8]>, AllocError>;
    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout: Layout);

    // Provided defaults
    fn allocate_zeroed(&self, layout: Layout) -> Result<NonNull<[u8]>,
AllocError>;
    unsafe fn grow(...) -> Result<NonNull<[u8]>, AllocError>;
    unsafe fn grow_zeroed(...) -> Result<NonNull<[u8]>, AllocError>;
    unsafe fn shrink(...) -> Result<NonNull<[u8]>, AllocError>;
}
```

Key Differences from GlobalAlloc

Aspect	GlobalAlloc	Allocator
Error handling	Null pointer	`Result<_, AllocError>`
Pointer type	`*mut u8`	`NonNull<[u8]>`
Zero-sized alloc	Undefined behavior	Allowed
Resize operations	Single `realloc`	Separate `grow` / `shrink`
Alignment changes	Not supported	Supported in grow/shrink

The Fat Pointer Return

`NonNull<[u8]>` is a **fat pointer** — it contains both the address and the length. This lets the allocator tell you the *actual* size allocated:

```rust
// You ask for 30 bytes, allocator gives you 32 due to internal binning
let result = allocator.allocate(Layout::from_size_align(30, 1)?)?;
let actual_size = result.len();  // Might be 32!
```

Collections like `Vec` can use this extra space for free.

Zero-Sized Allocations

Unlike GlobalAlloc, Allocator handles ZSTs gracefully:

```
// This is fine with Allocator
let layout = Layout::new::<()>();  // size=0
let ptr = allocator.allocate(layout)?;  // Returns a valid NonNull
```

The implementation must catch this case and return a dangling-but-valid pointer without calling the underlying allocator.

2.6 The Global Allocator

By default, Rust uses the system allocator. You can change it:

```
use std::alloc::{GlobalAlloc, Layout, System};

#[global_allocator]
static ALLOCATOR: System = System;
```

Or use a custom allocator:

```
struct MyAllocator;

unsafe impl GlobalAlloc for MyAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // Your implementation
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        // Your implementation
    }
}

#[global_allocator]
static ALLOCATOR: MyAllocator = MyAllocator;
```

Example: A Simple Bump Allocator

From the GlobalAlloc documentation:

```
const ARENA_SIZE: usize = 128 * 1024;

#[repr(C, align(4096))]
struct SimpleAllocator {
    arena: UnsafeCell<[u8; ARENA_SIZE]>,
    remaining: AtomicUsize,
}

unsafe impl GlobalAlloc for SimpleAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        let size = layout.size();
        let align = layout.align();
        let align_mask = !(align - 1);

        if align > 4096 {
            return null_mut();
        }

        let mut allocated = 0;
        self.remaining
            .fetch_update(Relaxed, Relaxed, |mut remaining| {
                if size > remaining { return None; }
                remaining -= size;
                remaining &= align_mask;  // Round down for alignment
                allocated = remaining;
                Some(remaining)
            })
            .ok()?;

        unsafe { self.arena.get().cast::<u8>().add(allocated) }
    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        // Bump allocators don't free individual allocations
    }
}
```

This "bump allocator" is extremely fast (just an atomic decrement) but never reuses memory. It's used in compilers, game frames, and other short-lived contexts.

2.7 The Optimizer's Freedom

A critical note from the docs:

> You must not rely on allocations actually happening, even if there are explicit heap allocations in the source.

The optimizer can: - **Eliminate allocations** — `drop(Box::new(42))` might not allocate at all - **Move to stack** — Small heap allocations might become stack allocations - **Merge allocations** — Multiple small allocations might become one

This means debugging allocators that count allocations can be misleading. The count depends on optimization level!

2.8 Key Takeaways

1. **GlobalAlloc is stable and simple** — Four methods, raw pointers, null for errors
2. **Allocator is the future** — Result types, fat pointers, ZST support
3. **Safety contracts are critical** — Violating them is undefined behavior
4. **Default implementations are naive** — Override for efficiency
5. **Platform quirks exist** — Real allocators handle bugs in underlying systems
6. **The optimizer can eliminate allocations** — Don't rely on allocation counts

---

Source Files

File	Purpose
`library/core/src/alloc/global.rs`	GlobalAlloc trait definition
`library/core/src/alloc/mod.rs`	Allocator trait and AllocError
`library/std/src/sys/alloc/unix.rs`	Unix implementation
`library/std/src/sys/alloc/windows.rs`	Windows implementation

Exercises

1. Why must allocators never panic? What would happen if `alloc` panicked inside `Box::new`?

2. The default `realloc` always allocates new memory. When would `libc::realloc` be able to grow in place?
3. Write a simple allocator that tracks total bytes allocated (hint: wrap the System allocator).
4. Why does `Allocator::allocate` return `NonNull<[u8]>` instead of just `NonNull<u8>`?

Next Chapter

Chapter 3: Box — Owned Heap Allocation →

We'll see how `Box<T>` uses these allocator traits to provide safe, owned heap memory with automatic cleanup.