Chapter 4: Vec — Dynamic Arrays

> *"Vec is and always will be a (pointer, capacity, length) triplet. No more, no less."*
> — Vec documentation guarantees

Introduction

Box gives us a single heap allocation. But what if we need a collection that grows? Enter `Vec<T>`, Rust's dynamic array — the most commonly used collection type.

Vec builds on everything we've learned: Layout for calculating memory requirements, the Allocator traits for getting memory, and ownership semantics for safe resource management. But it adds crucial new complexity: **capacity management** and **growth strategies**.

4.1 The Vec Memory Model

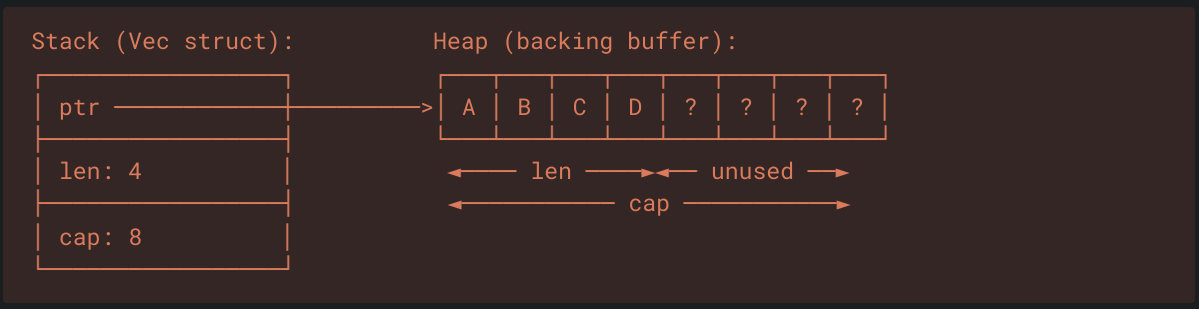
The Three-Field Structure

```
pub struct Vec<T, A: Allocator = Global> {
    buf: RawVec<T, A>,
    len: usize,
}
```

Vec delegates all allocation to `RawVec`, keeping only the element count itself. Effectively, Vec has three pieces of state: - **ptr**: Pointer to the heap allocation - **len**: Number of elements currently stored - **cap**: Total capacity (elements that *could* be stored)

Invariant: `len ≤ cap` always.

Memory Layout



On the stack: 24 bytes (three `usize` values on 64-bit)
On the heap: `cap * size_of::<T>()` bytes

Vec Guarantees

The documentation makes explicit promises:

1. **No small-buffer optimization** — Contents are always on the heap, never the stack
2. **Never auto-shrinks** — Capacity only grows unless explicitly shrunk
3. **Null-pointer optimized** — The pointer is never null (uses dangling for empty)
4. **No specific growth strategy guaranteed** — Only that push is amortized O(1)

Empty Vec: No Allocation

```
let v: Vec<i32> = Vec::new();
// ptr: dangling (not null, but not allocated)
// len: 0
// cap: 0
```

A brand new Vec doesn't allocate until you push the first element. The pointer is "dangling" — a valid, non-null pointer that doesn't point to allocated memory. This enables null-pointer optimization for `Option<Vec<T>>`.

4.2 RawVec: The Allocation Engine

Vec delegates all allocation logic to `RawVec`. This separation of concerns keeps Vec focused on element management while RawVec handles bytes.

The Inner/Outer Architecture

Modern Rust's RawVec uses a two-layer design to reduce code bloat:

```
pub(crate) struct RawVec<T, A: Allocator = Global> {
    inner: RawVecInner<A>,
    _marker: PhantomData<T>,
}

struct RawVecInner<A: Allocator = Global> {
    ptr: Unique<u8>,
    cap: Cap,
    alloc: A,
}
```

Why this split?

`RawVecInner` is only generic over the allocator, not the element type. Most operations (grow, shrink, deallocate) only need the element's Layout, not its actual type. This means: - `RawVecInner<Global>` is one type regardless of `T` - Less monomorphization = smaller binaries - Core allocation logic compiled once, not per-type

The Cap Type

```
type Cap = core::num::niche_types::UsizeNoHighBit;
```

Instead of raw `usize`, capacity uses a special type with a safety invariant: **cap must be in the range `0..=isize::MAX`**. This: - Matches the `isize::MAX` allocation limit from Layout - Enables the compiler to assume this invariant - The high bit is guaranteed zero (potential niche optimization)

Why Separate RawVec?

1. **Reusability**: Other types can use RawVec (`VecDeque`, `String`)
2. **Complexity isolation**: Allocation edge cases live in one place
3. **Unsafe encapsulation**: RawVec contains the unsafe code
4. **Monomorphization reduction**: Inner type is allocator-generic only

4.3 Growth Strategies: The Amortized Analysis

The Problem with Naive Growth

If Vec grew by exactly one element each time:

```
let mut v = Vec::new();
for i in 0..1000 {
    v.push(i);  // Reallocates every single time!
}
```

Each push would: 1. Allocate new buffer (size n+1) 2. Copy n elements 3. Free old buffer

Total work: 1 + 2 + 3 + ... + 999 = **O(n²)** for n pushes!

The Doubling Strategy

The core growth logic in `grow_amortized`:

```
fn grow_amortized(&mut self, len: usize, additional: usize, elem_layout: Layout)
    -> Result<(), TryReserveError>
{
    // Calculate required capacity
    let required_cap = len.checked_add(additional).ok_or(CapacityOverflow)?;

    // Exponential growth: at least double, but satisfy requirement
    let cap = cmp::max(self.cap.as_inner() * 2, required_cap);

    // Ensure we don't allocate tiny buffers
    let cap = cmp::max(min_non_zero_cap(elem_layout.size()), cap);

    // ... allocate with new capacity
}
```

The min_non_zero_cap Optimization

Tiny allocations are inefficient. The standard library avoids them:

```
const fn min_non_zero_cap(size: usize) -> usize {
    if size == 1 {
        8   // Heap allocators round up to at least 8 bytes anyway
    } else if size <= 1024 {
        4   // Moderate-sized elements: start with 4
    } else {
        1   // Large elements: even 1 is significant
    }
}
```

This means: - `Vec<u8>` starts at capacity 8 (not 1) - `Vec<u64>` starts at capacity 4 - `Vec<[u8; 4096]>` starts at capacity 1

Growth Timeline

```
Push 1: cap 0 → 8 (for u8) or 4 (for u64)
Push 2-8: no realloc
Push 9: cap 8 → 16 (doubled)
Push 17: cap 16 → 32
Push 33: cap 32 → 64
...
```

For 1000 elements in a `Vec<u64>`: ~8 reallocations total.

The Math (Optional)

For n elements with doubling: - Number of reallocations: O(log n) - Total elements copied: 4 + 8 + 16 + ... + n/2 ≈ n

So n pushes cost O(n) work total, meaning **O(1) amortized per push**.

4.4 The Growth Machinery

finish_grow: The Core Reallocation Function

All growth paths eventually call `finish_grow`:

```rust
fn finish_grow<A: Allocator>(
    new_layout: Layout,
    current_memory: Option<(NonNull<u8>, Layout)>,
    alloc: &mut A,
) -> Result<NonNull<[u8]>, TryReserveError> {
    let memory = if let Some((ptr, old_layout)) = current_memory {
        // Have existing allocation: try to grow it
        unsafe { alloc.grow(ptr, old_layout, new_layout) }
    } else {
        // No existing allocation: fresh allocate
        alloc.allocate(new_layout)
    };
    memory.map_err(|_| AllocError { layout: new_layout, non_exhaustive: () }
.into())
}
```

This is marked `#[cold]` — the compiler assumes it's rarely called, optimizing the hot path (no growth needed).

The Two Growth Modes

Amortized Growth (`grow_amortized`): - Doubles capacity or grows to required, whichever is larger - Used by `push`, `reserve`, `extend` - Provides O(1) amortized operations

Exact Growth (`grow_exact`): - Grows to exactly the required capacity - Used by `reserve_exact` - No extra space beyond what's needed

---

4.5 Push and Pop

Push: The Heart of Vec

```
pub fn push(&mut self, value: T) {
    // 1. Check if we need to grow
    if self.len == self.buf.capacity() {
        self.buf.grow_one();
    }

    // 2. Write the value
    unsafe {
        let end = self.as_mut_ptr().add(self.len);
        ptr::write(end, value);
    }

    // 3. Increment length
    self.len += 1;
}
```

Note the specialized `grow_one()` method — it's optimized for the common case of adding one element, avoiding some of the overhead of the general `reserve` calculation.

Pop: Simple and Efficient

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.as_ptr().add(self.len)))
        }
    }
}
```

Pop never shrinks the buffer! Capacity stays the same. This is intentional: - Shrinking would require reallocation - You might push again soon - Use `shrink_to_fit()` if you really want to reclaim memory

---

4.6 Reallocation: How grow() Works

When Vec needs more space, the allocator has options:

The Allocator::grow Method

```
unsafe fn grow(
    &self,
    ptr: NonNull<u8>,
    old_layout: Layout,
    new_layout: Layout,
) -> Result<NonNull<[u8]>, AllocError>
```

The allocator can:

1. **Extend in place** (optimal) — If free space exists after the allocation
2. **Allocate-copy-free** (common) — New location, copy data, free old
3. **Fail** — Return `AllocError`

What Happens on Failure

For infallible methods (`push`, `reserve`):

```
fn handle_error(e: TryReserveError) -> ! {
    match e.kind() {
        CapacityOverflow => capacity_overflow(), // panic!
        AllocError { layout, .. } => handle_alloc_error(layout), // abort
    }
}
```

For fallible methods (`try_reserve`): - Return `Err(TryReserveError)` — caller decides what to do

4.7 Drop: Cleaning Up Collections

```
unsafe impl<#[may_dangle] T, A: Allocator> Drop for Vec<T, A> {
    fn drop(&mut self) {
        unsafe {
            // Drop all elements
            ptr::drop_in_place(ptr::slice_from_raw_parts_mut(
                self.as_mut_ptr(),
                self.len
            ))
        }
        // RawVec::drop handles deallocation
    }
}
```

Two phases: 1. **Drop each element**: Calls destructor on `v[0]`, `v[1]`, ..., `v[len-1]` 2. **Deallocate buffer**: RawVec's Drop frees the memory

RawVec's Drop

```
unsafe impl<#[may_dangle] T, A: Allocator> Drop for RawVec<T, A> {
    fn drop(&mut self) {
        unsafe { self.inner.deallocate(T::LAYOUT) }
    }
}
```

Note: RawVec **does not** drop contents — it only frees the allocation. This separation of concerns is crucial.

Panic Safety

If an element's destructor panics: - Remaining elements might not be dropped (potential leak) - But memory safety is preserved — no double-free or use-after-free - This is a known limitation,

documented in the source

4.8 Zero-Sized Types in Vec

Vec handles ZSTs specially:

```
let mut v: Vec<()> = Vec::new();
for _ in 0..1_000_000 {
    v.push(());  // No allocation ever happens!
}
println!("len: {}, cap: {}", v.len(), v.capacity());
// len: 1000000, cap: 18446744073709551615 (usize::MAX)
```

How It Works

```
const fn capacity(&self, elem_size: usize) -> usize {
    if elem_size == 0 { usize::MAX } else { self.cap.as_inner() }
}
```

For ZSTs: - `ptr` is dangling (never allocated) - `cap` returns `usize::MAX` (effectively infinite) - `len` tracks count normally - No memory is used except the Vec struct itself (24 bytes)

Why usize::MAX?

This clever design means: - `len < cap` is always true for ZSTs (can always "push") - Overflow checking in len catches real issues - No special-casing in most Vec methods

---

4.9 Common Vec Operations

Reserve: Ensure Capacity

```
let mut v = Vec::new();
v.reserve(100);  // Guarantee space for 100 more elements
// v.capacity() >= 100 + v.len()
```

Internally:

```rust
pub fn reserve(&mut self, additional: usize) {
    self.buf.reserve(self.len, additional);
}
```

Shrink: Reclaim Memory

```rust
let mut v = vec![1, 2, 3, 4, 5];
v.pop();
v.pop();
v.shrink_to_fit();  // Reduce capacity to match length
// v.capacity() == 3
```

Note the check in `shrink_to_fit`:

```rust
pub fn shrink_to_fit(&mut self) {
    if self.capacity() > self.len {  // Avoid panic in RawVec
        self.buf.shrink_to_fit(self.len);
    }
}
```

With Capacity: Avoid Reallocations

```rust
// If you know the size upfront
let mut v = Vec::with_capacity(1000);
for i in 0..1000 {
    v.push(i);  // No reallocations!
}
```

Clear: Remove All Elements

```rust
let mut v = vec![1, 2, 3];
v.clear();
// len: 0, but capacity unchanged
```

4.10 Optimizer Hints

The source uses hints to help LLVM:

```rust
fn try_reserve(&mut self, len: usize, additional: usize, ...) -> Result<(),
TryReserveError> {
    if self.needs_to_grow(len, additional, elem_layout) {
        self.grow_amortized(len, additional, elem_layout)?;
    }
    unsafe {
        // Tell optimizer that growth succeeded
        hint::assert_unchecked(!self.needs_to_grow(len, additional,
elem_layout));
    }
    Ok(())
}
```

`hint::assert_unchecked` allows the optimizer to assume the condition is true, enabling better code generation for subsequent operations.

4.11 Key Takeaways

1. **Vec is a (ptr, len, cap) triplet** — 24 bytes on stack
2. **RawVec uses inner/outer split** — Reduces monomorphization
3. **Cap has a safety invariant** — Always ≤ `isize::MAX`
4. **Doubling growth gives O(1) amortized push** — Essential for performance
5. **min_non_zero_cap avoids tiny allocations** — 8 for bytes, 4 for moderate, 1 for large
6. **Pop doesn't shrink** — Use `shrink_to_fit` explicitly
7. **ZSTs are special-cased** — Infinite capacity, no allocation
8. **Drop is two-phase** — Elements first, then memory

Source Files

File	Purpose
`library/alloc/src/vec/mod.rs`	Main Vec implementation
`library/alloc/src/raw_vec/mod.rs`	RawVec and RawVecInner

`library/alloc/src/vec/into_iter.rs`	IntoIterator implementation
`library/alloc/src/vec/drain.rs`	Drain iterator

Exercises

1. Why does RawVec use `RawVecInner` instead of putting everything in one struct?
2. What's the minimum number of allocations for pushing 1000 `u64` values starting from an empty Vec?
3. Implement a `SimpleVec<T>` with just `new`, `push`, `pop`, and `Drop`.
4. Why is `Vec<()>` capacity `usize::MAX` instead of tracking a real count?
5. What happens if you `mem::forget` a Vec? Is this a memory leak? What about the elements?

Hands-On Project: SimpleVec

Build a minimal Vec to solidify understanding:

```rust
use std::alloc::{alloc, dealloc, realloc, Layout};
use std::ptr::{self, NonNull};
use std::mem;

pub struct SimpleVec<T> {
    ptr: NonNull<T>,
    len: usize,
    cap: usize,
}

impl<T> SimpleVec<T> {
    pub fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "ZST not supported in SimpleVec");
        SimpleVec {
            ptr: NonNull::dangling(),
            len: 0,
            cap: 0,
        }
    }

    pub fn push(&mut self, value: T) {
        if self.len == self.cap {
            self.grow();
        }
        unsafe {
            ptr::write(self.ptr.as_ptr().add(self.len), value);
        }
        self.len += 1;
    }

    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            self.len -= 1;
            unsafe { Some(ptr::read(self.ptr.as_ptr().add(self.len))) }
        }
    }

    fn grow(&mut self) {
        // Start with 4, then double (like std for moderate types)
        let new_cap = if self.cap == 0 { 4 } else { self.cap * 2 };
        let new_layout = Layout::array::<T>(new_cap).unwrap();

        assert!(new_layout.size() <= isize::MAX as usize, "Allocation too
large");

        let new_ptr = if self.cap == 0 {
            unsafe { alloc(new_layout) }
        } else {
            let old_layout = Layout::array::<T>(self.cap).unwrap();
            unsafe { realloc(self.ptr.as_ptr() as *mut u8, old_layout,
new_layout.size()) }
        };
```

```
        self.ptr = NonNull::new(new_ptr as *mut T).expect("allocation failed");
self.cap = new_cap;    }    pub fn len(&self) -> usize { self.len }    pub fn
capacity(&self) -> usize { self.cap }}impl<T> Drop for SimpleVec<T> {    fn
drop(&mut self) {        if self.cap > 0 {            // Drop all elements first
for i in 0..self.len {                unsafe {
ptr::drop_in_place(self.ptr.as_ptr().add(i)); }            }            // Then
deallocate            let layout = Layout::array::<T>(self.cap).unwrap();
unsafe { dealloc(self.ptr.as_ptr() as *mut u8, layout); }        }    }}
```

What's Next?

With Vec mastered, you understand Rust's core collection machinery. Suggested further reading:

- **String**: Essentially `Vec<u8>` with UTF-8 guarantees
- **HashMap**: Uses RawTable, a more complex allocation strategy
- **VecDeque**: Ring buffer built on RawVec
- **BTreeMap**: Node-based allocation

You now have the foundation to understand any Rust collection at the allocation level!