vec_exploration.rs

```rust
//! Exploring Vec and RawVec Concepts
//!
//! This program demonstrates the allocation mechanics of Vec:
//! - Growth strategies and capacity changes
//! - Zero-sized type handling
//! - Drop behavior for collections
//! - Memory layout inspection

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

fn main() {
    println!("=== Rust Vec Exploration ===\n");

    // Section 1: Vec memory layout
    explore_vec_layout();

    // Section 2: Growth strategy observation
    explore_growth_strategy();

    // Section 3: Zero-sized types
    explore_zst_vec();

    // Section 4: Drop behavior
    explore_drop_behavior();

    // Section 5: Capacity vs Length
    explore_capacity_vs_length();

    // Section 6: Initial allocation sizes
    explore_min_non_zero_cap();
}

/// Demonstrates Vec's memory layout on stack and heap
fn explore_vec_layout() {
    println!("--- 1. Vec Memory Layout ---\n");

    // Vec is always 24 bytes on 64-bit (ptr + len + cap)
    println!("Vec<i32> stack size: {} bytes", mem::size_of::<Vec<i32>>());
    println!("Vec<u8> stack size: {} bytes", mem::size_of::<Vec<u8>>());
    println!("Vec<[u8; 1024]> stack size: {} bytes", mem::size_of::<Vec<[u8;
1024]>>());
    println!("→ Vec is always 24 bytes on stack regardless of T\n");

    // The heap allocation depends on capacity and element size
    let v: Vec<u64> = Vec::with_capacity(10);
    println!("Vec<u64> with capacity 10:");
    println!("  Stack: {} bytes", mem::size_of::<Vec<u64>>());
    println!("  Heap: {} bytes (cap × size_of::<u64>)", v.capacity() *
mem::size_of::<u64>());

    // Pointer inspection
    let v = vec![1u64, 2, 3, 4];
    println!("\nVec<u64> = [1, 2, 3, 4]:");
```

```
    println!("  Pointer to heap: {:p}", v.as_ptr());    println!("  Length: {}",
v.len());    println!("  Capacity: {}", v.capacity());    println!();}///
Demonstrates the doubling growth strategyfn explore_growth_strategy() {
println!("--- 2. Growth Strategy Observation ---\n");    let mut v: Vec<u64> =
Vec::new();    let mut last_cap = 0;    println!("Pushing elements and tracking
capacity changes:\n");    println!("{:>6} {:>10} {:>10} {:>15}", "Push#",
"Length", "Capacity", "Reallocated?");    println!("{}", "-".repeat(45));    for
i in 1..=33 {        v.push(i);        let reallocated = if v.capacity() !=
last_cap {            last_cap = v.capacity();            "YES"        } else {
""        };        // Only print on reallocation or first few        if
!reallocated.is_empty() || i <= 4 {            println!("{:>6} {:>10} {:>10}
{:>15}", i, v.len(), v.capacity(), reallocated);        }    }    println!("\n→
Capacity doubles: 4 → 8 → 16 → 32 → 64");    println!("→ Only ~5 reallocations
for 33 elements (amortized O(1) push)\n");}/// Zero-sized types get special
treatmentfn explore_zst_vec() {    println!("--- 3. Zero-Sized Type Vec ---\n");
// Unit type Vec    let mut v: Vec<()> = Vec::new();    println!
("Vec<()>::new():");    println!("  Length: {}", v.len());    println!("
Capacity: {} (usize::MAX for ZST)", v.capacity());    // Push a million units -
no allocation!    for _ in 0..1_000_000 {        v.push(());    }    println!
("\nAfter pushing 1,000,000 units:");    println!("  Length: {}", v.len());
println!("  Capacity: {} (still usize::MAX)", v.capacity());    println!("  Heap
memory used: 0 bytes");    // Empty struct is also ZST    struct Empty;    let
layout = Layout::new::<Empty>();    println!("\nLayout for Empty struct: size={},
align={}", layout.size(), layout.align());    let v_empty: Vec<Empty> = vec!
[Empty, Empty, Empty];    println!("Vec<Empty> with 3 elements: cap = {}",
v_empty.capacity());    println!();}/// Drop behavior for collectionsfn
explore_drop_behavior() {    println!("--- 4. Collection Drop Behavior ---\n");
struct Noisy(u32);    impl Drop for Noisy {        fn drop(&mut self) {
println!("    Dropping Noisy({})", self.0);        }    }    println!("Creating
Vec<Noisy> with 3 elements...");    {        let v = vec![Noisy(1), Noisy(2),
Noisy(3)];        println!("  Vec created with len={}, cap={}", v.len(),
v.capacity());        println!("  Dropping Vec...");    }    println!("  →
Elements dropped in order, then memory freed\n");    // Pop doesn't drop - the
value is moved out    println!("Pop behavior:");    {        let mut v = vec!
[Noisy(10), Noisy(20)];        println!("  Created Vec with Noisy(10),
Noisy(20)");        let popped = v.pop();        println!("  After pop: len={},
popped value exists", v.len());        println!("  Dropping popped value...");
drop(popped);        println!("  Dropping remaining Vec...");    }    println!
();}/// Capacity vs Length distinctionfn explore_capacity_vs_length() {
println!("--- 5. Capacity vs Length ---\n");    // with_capacity allocates but
doesn't initialize    let mut v: Vec<i32> = Vec::with_capacity(10);    println!
("Vec::with_capacity(10):");    println!("  Length: {} (no elements yet)",
v.len());    println!("  Capacity: {} (space allocated)", v.capacity());    //
Push some elements    v.extend([1, 2, 3]);    println!("\nAfter pushing 3
elements:");    println!("  Length: {}", v.len());    println!("  Capacity: {}
(unchanged)", v.capacity());    // Pop doesn't reduce capacity    v.pop();
v.pop();    println!("\nAfter popping 2 elements:");    println!("  Length: {}",
v.len());    println!("  Capacity: {} (still 10!)", v.capacity());    //
shrink_to_fit releases unused capacity    v.shrink_to_fit();    println!("\nAfter
shrink_to_fit:");    println!("  Length: {}", v.len());    println!("  Capacity:
{} (shrunk to fit)", v.capacity());    // Clear doesn't deallocate
v.extend([1, 2, 3, 4, 5]);    let cap_before = v.capacity();    v.clear();
println!("\nAfter clear:");    println!("  Length: {} (all elements gone)",
v.len());    println!("  Capacity: {} (memory still allocated)", v.capacity());
```

```
assert_eq!(v.capacity(), cap_before);    println!();}/// Demonstrates
min_non_zero_cap behaviorfn explore_min_non_zero_cap() {    println!("--- 6.
Initial Allocation Sizes ---\n");    // The stdlib uses min_non_zero_cap to avoid
tiny allocations:    // - 8 for size 1 (bytes)    // - 4 for size <= 1024    // -
1 for larger elements    println!("First push allocation (min_non_zero_cap
behavior):\n");    // u8: starts at capacity 8    let mut v_u8: Vec<u8> =
Vec::new();    v_u8.push(1);    println!("Vec<u8>: first push → capacity {}",
v_u8.capacity());    println!("  (min 8 because heap allocators round up small
requests)");    // u64: starts at capacity 4    let mut v_u64: Vec<u64> =
Vec::new();    v_u64.push(1);    println!("\nVec<u64>: first push → capacity {}",
v_u64.capacity());    println!("  (min 4 for moderate-sized elements <= 1KB)");
// Large element: starts at capacity 1    let mut v_large: Vec<[u8; 4096]> =
Vec::new();    v_large.push([0; 4096]);    println!("\nVec<[u8; 4096]>: first
push → capacity {}", v_large.capacity());    println!("  (min 1 for large
elements to avoid wasting space)");    println!("\n→ This optimization reduces
wasted memory for different use cases\n");}// Bonus: A minimal Vec implementation
for educational purposes#[allow(dead_code)]mod simple_vec {    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");            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) {            let new_cap = if self.cap == 0
{ 4 } else { self.cap * 2 };            let new_layout = Layout::array::<T>
(new_cap).unwrap();            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 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); }            }        }    }}
```