box_exploration.rs

```rust
//! Exploring Box and Allocation Concepts
//!
//! This program demonstrates the allocation mechanics we've studied:
//! - Box creation and deallocation
//! - Zero-sized types (ZSTs)
//! - Layout calculations
//! - Custom allocators (using the unstable allocator_api)
//! - Memory layout inspection

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

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

    // Chapter 1: Basic Box mechanics
    explore_basic_box();

    // Chapter 2: Layout calculations
    explore_layouts();

    // Chapter 3: Zero-sized types
    explore_zst();

    // Chapter 4: Box memory layout
    explore_box_memory_layout();

    // Chapter 5: Drop behavior
    explore_drop_behavior();

    // Chapter 6: Raw pointer round-trip
    explore_raw_pointers();
}

/// Demonstrates basic Box creation and the heap allocation it triggers
fn explore_basic_box() {
    println!("--- 1. Basic Box Mechanics ---\n");

    // Simple heap allocation
    let boxed_int: Box<i32> = Box::new(42);
    println!("Boxed i32 value: {}", *boxed_int);
    println!("Box<i32> size: {} bytes", mem::size_of::<Box<i32>>());
    println!("i32 size: {} bytes", mem::size_of::<i32>());
    println!("→ Box is just a pointer ({} bytes on this platform)\n",
mem::size_of::<*const ()>());

    // Larger type to see the benefit of heap allocation
    let large_array: Box<[u8; 1024]> = Box::new([0u8; 1024]);
    println!("Box<[u8; 1024]> size on stack: {} bytes", mem::size_of::<Box<[u8;
1024]>>());
    println!("Actual [u8; 1024] size: {} bytes", mem::size_of::<[u8; 1024]>());
    println!("→ Only the pointer lives on stack, data is on heap\n");

    // The value is dropped when boxed_int and large_array go out of scope
```

```
    // Drop order: large_array first (LIFO), then boxed_int    drop(large_array);
drop(boxed_int);}/// Demonstrates Layout calculations - the foundation of
allocationfn explore_layouts() {    println!("--- 2. Layout Calculations ---\n");
// Layout for primitive types    let i32_layout = Layout::new::<i32>();
println!("Layout for i32:  size={}, align={}", i32_layout.size(),
i32_layout.align());    let u64_layout = Layout::new::<u64>();    println!
("Layout for u64:  size={}, align={}", u64_layout.size(), u64_layout.align());
// Layout for a struct - demonstrates padding    #[repr(C)]  // Use C layout for
predictable ordering    struct PaddedStruct {        a: u8,    // 1 byte        //
7 bytes padding here to align b        b: u64,   // 8 bytes, needs 8-byte
alignment        c: u8,    // 1 byte        // 7 bytes padding at end to make
struct size multiple of alignment    }    let padded_layout = Layout::new::
<PaddedStruct>();    println!("\nLayout for PaddedStruct (u8, u64, u8):");
println!("  size={}, align={}", padded_layout.size(), padded_layout.align());
println!("  → Expected: 1 + 7(pad) + 8 + 1 + 7(pad) = 24 bytes");    // Rust's
default repr can reorder fields!    struct RustStruct {        a: u8,        b:
u64,        c: u8,    }    let rust_layout = Layout::new::<RustStruct>();
println!("\nLayout for RustStruct (same fields, Rust repr):");    println!("
size={}, align={}", rust_layout.size(), rust_layout.align());    println!("  →
Rust may reorder to: u64, u8, u8 = 8 + 1 + 1 + 6(pad) = 16 bytes");    // Array
layout    let array_layout = Layout::array::<u32>(10).unwrap();    println!
("\nLayout for [u32; 10]: size={}, align={}",              array_layout.size(),
array_layout.align());    // The isize::MAX constraint    println!("\nMax
allocation size for align=8: {} bytes",              isize::MAX as usize - 7);
println!("That's approximately {} GB",              (isize::MAX as usize - 7) /
(1024 * 1024 * 1024));    println!();}/// Zero-sized types don't actually
allocate!fn explore_zst() {    println!("--- 3. Zero-Sized Types (ZSTs) ---\n");
// Unit type is zero-sized    let unit_layout = Layout::new::<()>();    println!
("Layout for (): size={}, align={}", unit_layout.size(), unit_layout.align());
// Empty struct is zero-sized    struct Empty;    let empty_layout =
Layout::new::<Empty>();    println!("Layout for Empty struct: size={}, align={}",
             empty_layout.size(), empty_layout.align());    // Box<()> is still
pointer-sized, but no heap allocation occurs!    let boxed_unit: Box<()> =
Box::new(());    println!("\nBox<()> stack size: {} bytes", mem::size_of::
<Box<()>>());    println!("→ Box still needs a pointer, but it's 'dangling' (no
heap alloc)");    // The pointer is non-null but doesn't point to allocated
memory    let ptr = Box::into_raw(boxed_unit);    println!("Box<()> raw pointer:
{:p}", ptr);    println!("→ This is a well-aligned dangling pointer, not from the
allocator");    // We must reconstruct the Box to avoid leaking (even though
there's nothing to leak)    let _ = unsafe { Box::from_raw(ptr) };    // Array of
ZSTs    let zst_array_layout = Layout::array::<()>(1_000_000).unwrap();
println!("\nLayout for [(); 1_000_000]: size={}, align={}",
zst_array_layout.size(), zst_array_layout.align());    println!("→ A million
units take zero bytes!\n");}/// Inspect the actual memory layout of Boxfn
explore_box_memory_layout() {    println!("--- 4. Box Memory Layout ---\n");
let boxed: Box<u64> = Box::new(0xDEADBEEF_CAFEBABE);    // Get the raw pointer
without consuming the Box    let ptr: *const u64 = &*boxed;    println!("Value:
0x{:016X}", *boxed);    println!("Pointer to heap data: {:p}", ptr);    println!
("Address of Box on stack: {:p}", &boxed);    // The Box itself (on stack)
contains just the pointer    // We can see this by examining the raw bytes    let
box_bytes: &[u8] = unsafe {        std::slice::from_raw_parts(            &boxed
as *const Box<u64> as *const u8,            mem::size_of::<Box<u64>>()        )
};    println!("\nBox bytes on stack (the pointer itself):");    print!("  ");
for byte in box_bytes {        print!("{:02X} ", byte);    }    println!("\n→
```

```
This is the heap address in little-endian\n");}/// Demonstrates Drop behavior
with a custom typefn explore_drop_behavior() {    println!("--- 5. Drop Behavior
---\n");    struct Noisy {        id: u32,    }    impl Drop for Noisy {
fn drop(&mut self) {            println!("  Dropping Noisy({})", self.id);
}    }    println!("Creating Box<Noisy>...");    {        let boxed =
Box::new(Noisy { id: 1 });        println!("  Noisy({}) is alive", boxed.id);
println!("Leaving scope...");    }    println!("After scope ends.\n");    //
Nested boxes - drop order is important    println!("Creating nested boxes...");
{        let outer = Box::new(Noisy { id: 10 });        let inner =
Box::new(Noisy { id: 20 });        println!("  Both alive: outer={}, inner={}",
outer.id, inner.id);        println!("Leaving scope (inner dropped first -
LIFO)...");    }    println!("After scope ends.\n");    // Box containing another
Box    println!("Creating Box<Box<Noisy>>...");    {        let nested:
Box<Box<Noisy>> = Box::new(Box::new(Noisy { id: 100 }));        println!("
Nested Noisy is alive: {}", nested.id);        println!("Leaving scope (drops
cascade from outer to inner)...");    }    println!("After scope ends.\n");}///
Demonstrates raw pointer conversion (into_raw / from_raw)fn
explore_raw_pointers() {    println!("--- 6. Raw Pointer Round-Trip ---\n");
// Convert Box to raw pointer    let boxed = Box::new(String::from("Hello,
heap!"));    println!("Original Box contains: \"{}\"", boxed);    let raw_ptr:
*mut String = Box::into_raw(boxed);    println!("Converted to raw pointer: {:p}",
raw_ptr);    println!("→ Box is consumed, we now own the raw pointer");
println!("→ No destructor ran yet - memory is still allocated");    // We can use
the raw pointer (carefully!)    unsafe {        println!("Dereferencing raw
pointer: \"{}\"", *raw_ptr);        // Modify through raw pointer
(*raw_ptr).push_str(" Modified!");        println!("After modification: \"{}\"",
*raw_ptr);    }    // Convert back to Box - this is how we ensure cleanup    let
recovered_box = unsafe { Box::from_raw(raw_ptr) };    println!("Recovered Box
contains: \"{}\"", recovered_box);    println!("→ Box will deallocate when
dropped");    // Manual deallocation example (educational - don't do this
normally!)    println!("\nManual allocation/deallocation:");    unsafe {
use std::alloc::{alloc, dealloc};        let layout = Layout::new::<u64>();
println!("  Allocating {} bytes with align {}", layout.size(), layout.align());
let ptr = alloc(layout) as *mut u64;        if ptr.is_null() {            panic!
("Allocation failed!");        }        println!("  Allocated at: {:p}", ptr);
// Write to the memory        ptr.write(0x1234_5678_9ABC_DEF0);        println!("
Wrote value: 0x{:016X}", ptr.read());        // Deallocate - must use same
layout!        dealloc(ptr as *mut u8, layout);        println!("  Deallocated
successfully");    }    println!();}// Bonus: Recursive data structure that
requires Box#[allow(dead_code)]enum List<T> {    Cons(T, Box<List<T>>),    Nil,}#
[allow(dead_code)]impl<T> List<T> {    fn new() -> Self {        List::Nil    }
fn prepend(self, elem: T) -> Self {        List::Cons(elem, Box::new(self))    }
fn len(&self) -> usize {        match self {            List::Nil => 0,
List::Cons(_, tail) => 1 + tail.len(),        }    }}
```