rust-memory-safety-examples examples/data_race_prevention.rs: Code Companion

Reference code for the Concurrent Code Introduction lecture. Sections correspond to the lecture document.

Section 1: From Memory Safety to Thread Safety

```
//! Data race prevention example

// This single use statement brings in all the demonstration functions
// The module system ensures clean separation of concerns
use rust_memory_safety_examples::data_race;
```

The use statement imports from a library crate, meaning these demonstrations live in src/lib.rs or a submodule. This separation allows the examples to be tested independently from the binary.

Section 2: The Main Function Structure

```
fn main() {
    println!("=== Data Race Prevention in Rust ===\n");

    // Each demonstration is clearly numbered and labeled
    println!("1. Safe Concurrent Access with Arc and Mutex");
    data_race::safe_concurrent_access();

    println!("\n2. Type System Prevents Races");
    data_race::type_system_prevents_races();

    println!("\n3. C vs Rust Comparison");
    data_race::compare_c_vs_rust();

    // Summary section reinforces key concepts
    println!("\n=== Key Takeaways ===");
    // ... takeaways follow
}
```

The function calls execute sequentially, ensuring each concurrent demonstration completes before the next begins. This prevents any cross-contamination between examples.

Section 3: Safe Concurrent Access with Arc and Mutex

```
// Conceptual structure (from the library module):
// Arc<Mutex<T>> combines two guarantees:

use std::sync::{Arc, Mutex};
use std::thread;

// Arc: Atomic Reference Counting - safe to clone across threads
// Mutex: Mutual Exclusion - only one thread accesses inner data at a time
let shared_data: Arc<Mutex-Vec<i32>>> = Arc::new(Mutex::new(Vec::new()));

// Clone Arc to share ownership with spawned thread
let data_for_thread = Arc::clone(&shared_data);

thread::spawn(move || {
    // .lock() returns a Result<MutexGuard<T>, PoisonError>
    // The guard provides exclusive access until dropped
let mut guard = data_for_thread.lock().unwrap();
    guard.push(42); // Safe: we hold the lock
}); // Guard drops here, releasing the lock
```

The MutexGuard returned by .lock() implements Deref and DerefMut, so you can use it like &mut T. The lock is automatically released when the guard goes out of scope.

Section 4: The Type System as Thread Safety Enforcer

```
// Send and Sync are auto-traits - the compiler derives them automatically

// This compiles: i32 is Send
thread::spawn(move || {
    let x: i32 = 42;
    println!("{}", x);
});

// This would NOT compile: Rc is not Send
// use std::rc::Rc;
// let rc = Rc::new(42);
// thread::spawn(move || {
    // println!("{}", rc); // ERROR: Rc<i32> cannot be sent between threads safely
// });

// The thread::spawn signature enforces Send:
// pub fn spawn<F, T>(f: F) -> JoinHandle<T>
// where
// F: FnOnce() -> T + Send + 'static, // <-- F must be Send
// T: Send + 'static, // <-- Return type must be Send</pre>
```

The compiler error for non-Send types is precise: "Rc cannot be sent between threads safely." This specificity helps developers understand exactly what needs to change.

Section 5: Connecting to Ownership Fundamentals

The 'static bound on thread::spawn requires that all captured data either be owned (move) or have 'static lifetime. This prevents dangling references across thread boundaries.

Section 6: C vs Rust Comparison

```
// In C (pseudocode showing the problem):
// int shared_counter = 0;
// pthread_mutex_t lock; // The lock exists, but...
//
// void* thread_func(void* arg) {
// shared_counter++; // Oops! Forgot to acquire lock
// return NULL; // Data race: undefined behavior
// }

// In Rust, the type system prevents this:
let counter = Arc::new(Mutex::new(0));

let counter_clone = Arc::clone(&counter);
thread::spawn(move || {
    // Cannot access the i32 without going through the Mutex
    // counter_clone += 1; // ERROR: no += on Arc<Mutex<i32>>

    // Must acquire lock to access the inner value
    let mut num = counter_clone.lock().unwrap();
    *num += 1; // Safe: lock is held
});
```

The Mutex<T> wrapper makes the protected data inaccessible without acquiring the lock. There's no way to "forget" to lock—the type system physically prevents unquarded access.

Section 7: The Key Takeaways Section

```
println!("\n=== Key Takeaways ===");
println!("\start Rust prevents data races at compile time");
println!("\start Send and Sync traits enforce thread safety");
println!("\start Arc and Mutex provide safe concurrent access");
println!("\start No undefined behavior in concurrent code");
```

Each takeaway maps to a concrete mechanism: compile-time prevention via the borrow checker, Send / Sync traits for type-level enforcement, Arc<Mutex<T>> for shared mutable state, and the absence of undefined behavior in safe Rust

Quick Reference

Туре		Thread Safety
Rc <t></t>		Not Send, not Sync
Arc <t></t>	Multi-threaded reference counting	Send + Sync if T: Send + Sync
Mutex <t></t>	Mutual exclusion wrapper	Sync if T: Send
RwLock <t></t>		Sync if T: Send + Sync

Trait	Meaning	Auto-derived?
Send	Safe to transfer to another thread	
Sync	Safe to share references across threads	

```
// Common pattern: shared mutable state across threads
let shared: Arc<Mutex<T>> = Arc::new(Mutex::new(value));

// Clone for each thread
let thread_copy = Arc::clone(&shared);

// Spawn and move ownership of the clone
thread::spawn(move || {
    let mut guard = thread_copy.lock().unwrap();
    // Use guard as &mut T
});
```