Same example in Rust (with borrows)

```rust
use std::sync::Mutex;
use std::thread;

fn entry_point(data: &Mutex<String>, thread_id: i32) {
    let mut guard = data.lock().unwrap();
    guard.push_str("🔥");
    println!("Thread {}: {}", thread_id, *guard);
}

pub fn main() {
   let shared_data = Mutex::new(String::from("Hello threads"));

    const NUM_THREADS: i32 = 15;

    // Use scope to ensure threads don't outlive our data
    thread::scope(|scope| {
        let mut threads = Vec::new();

        for i in 0..NUM_THREADS {
            let local_data = &shared_data;
            let handle = scope.spawn(move || {
                entry_point(local_data, i);
            });

            threads.push(handle);
        }

        for handle in threads {
            handle.join().unwrap();
        }
    });
}
```

This version has several advantages:

- More efficient (no atomic reference counting)
- Cleaner code (no clone operations)
- Compile-time guarantees about data lifetime
- Still maintains thread safety through the `Mutex`

Without a way to properly express lifetimes (in terms of borrows/relocations/drops) we don't get the same level of safety and performance