Same example in Rust

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

fn entry_point(data: Arc<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 = Arc::new(Mutex::new(String::from("Hello threads")));

    let mut threads = Vec::new();

    const NUM_THREADS: i32 = 15;

    for i in 0..NUM_THREADS {
        // Clone the Arc for this thread
        let data_clone = Arc::clone(&shared_data);

        // Spawn the thread and store its handle
        let handle = thread::spawn(move || {
            entry_point(data_clone, i);
        });

        threads.push(handle);
    }

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

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();
        }
    });
}
```

Key changes made in this version:

1. Removed `Arc` and now using direct references (`&Mutex<String>`)
2. Added `thread::scope` to ensure threads don't outlive the borrowed data
3. Changed the thread spawning to use scoped threads via `scope.spawn`
4. Simplified the function signature of `entry_point` to take a reference
5. No more need for explicit cloning since we're using references