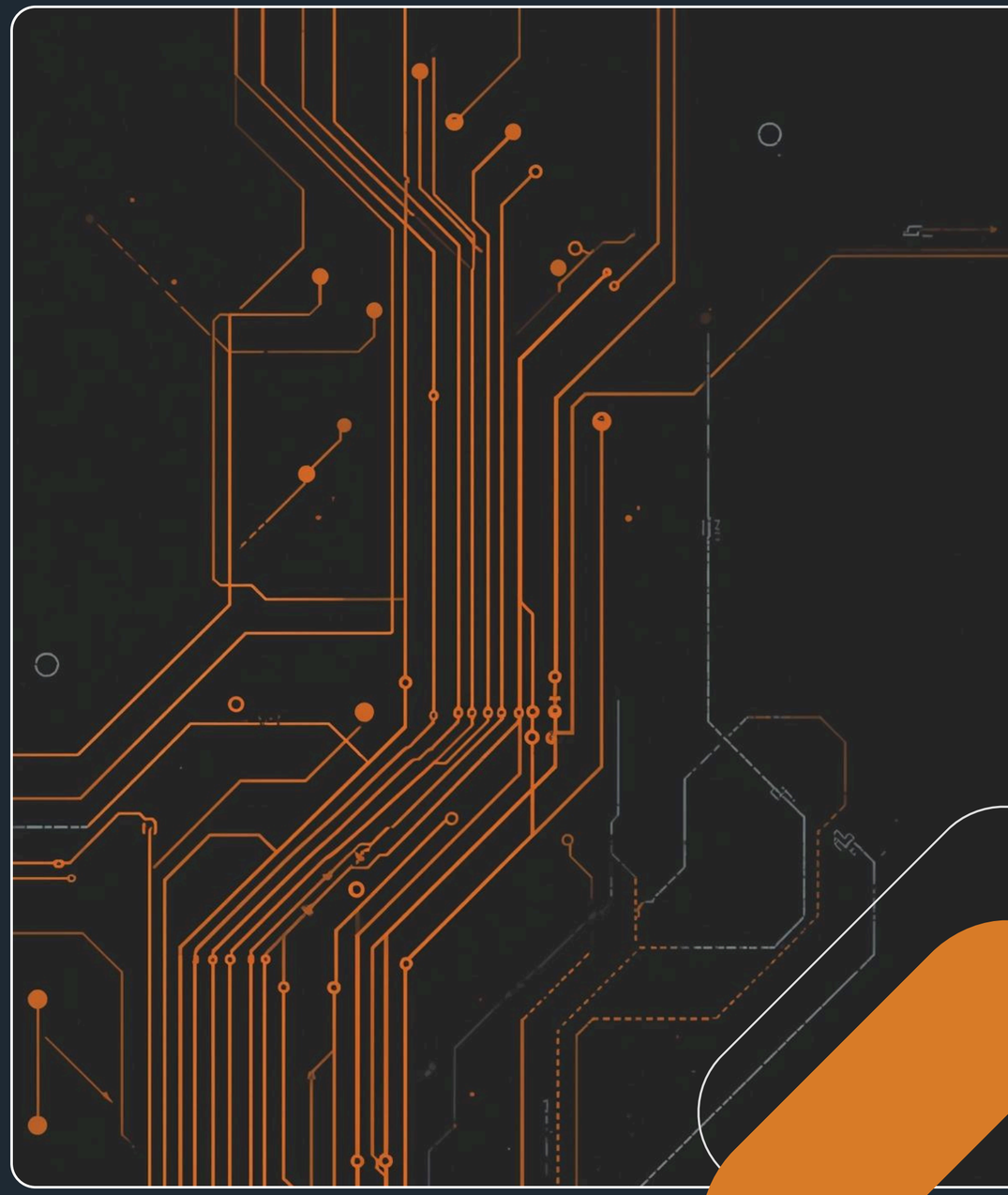
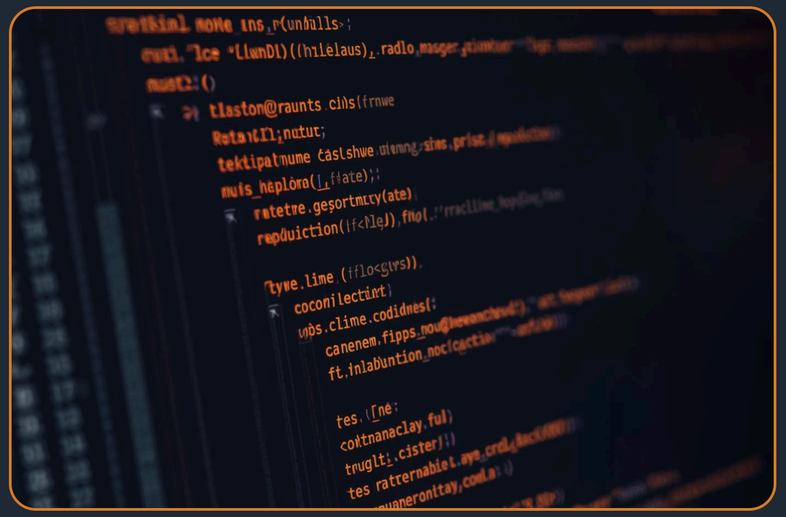
Rust vs C: Safer Systems Programming



CS333 - Final Presentation - Muneeb Azfar Nafees

Rust at a Glance

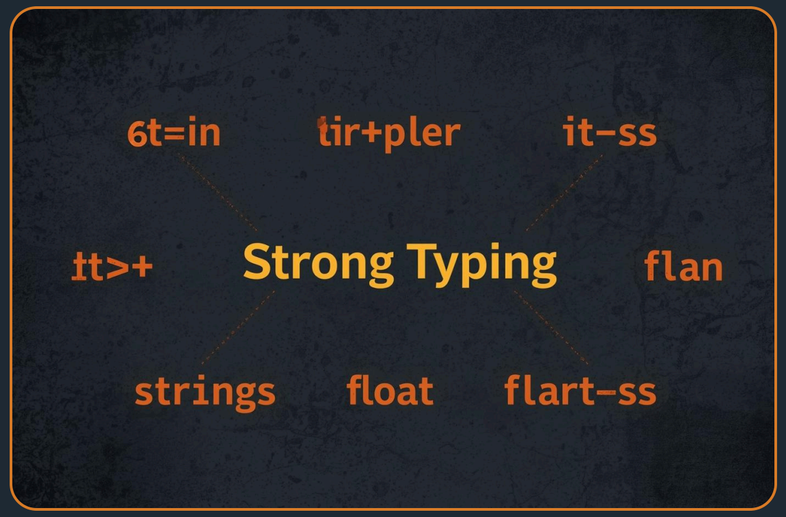


Systems

Rust focuses on system-level programming capabilities.

Typed

Rust’s strong typing helps catch errors at compile time.



Compiled

Rust’s compiled code ensures high performance and efficiency.

Rust Enhances C with Safety Features

Balancing Control and Safety

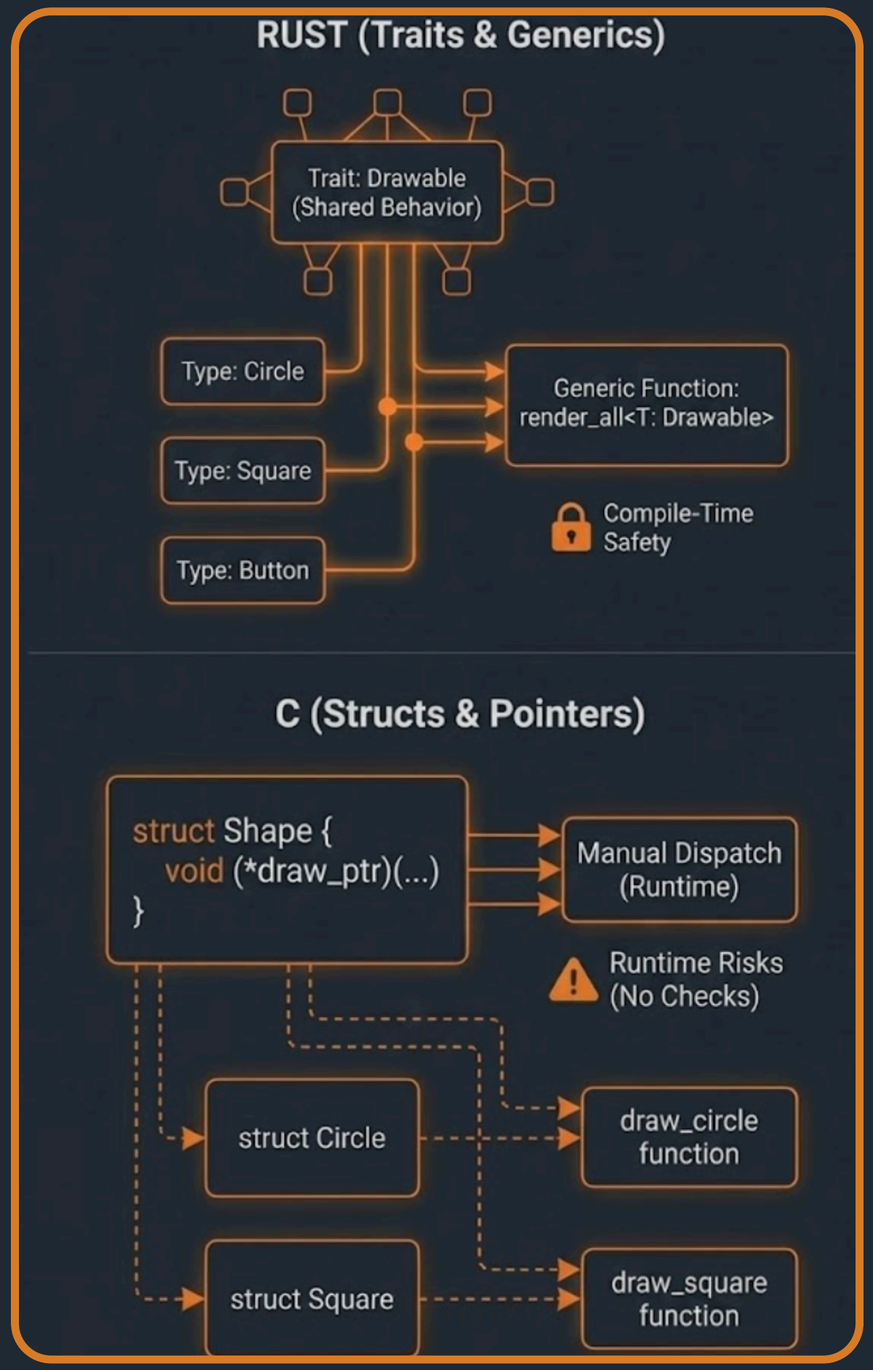
- Rust keeps C-level performance and low-level control.
- It adds safety with traits, ownership, and typed error handling.
- **Today:** polymorphism, memory, error handling, and concurrency (bonus feature!)

Polymorphism in Rust

Traits and Generics for Flexibility

- **Traits** define shared behavior across types.
- **Generics** let one function work for many types.
- Together, they give flexible, reusable, type-safe code.

```rust
trait Shape {
    fn area(&self) -> f64;
}

struct Circle {
    radius: f64,
}

struct Rectangle {
    width: f64,
    height: f64,
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        3.14 * self.radius * self.radius
    }
}

impl Shape for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}

fn total_area<T: Shape>(shapes: &[T]) -> f64 {
    shapes.iter().map(|s| s.area()).sum()
}
```



Polymorphism: Rust vs C

Traits vs function pointers

Rust: Traits and Generics

- Traits define shared behavior for many types.
- Generics provide a single function with many type-safe versions.
- Compiler checks trait use at compile time.

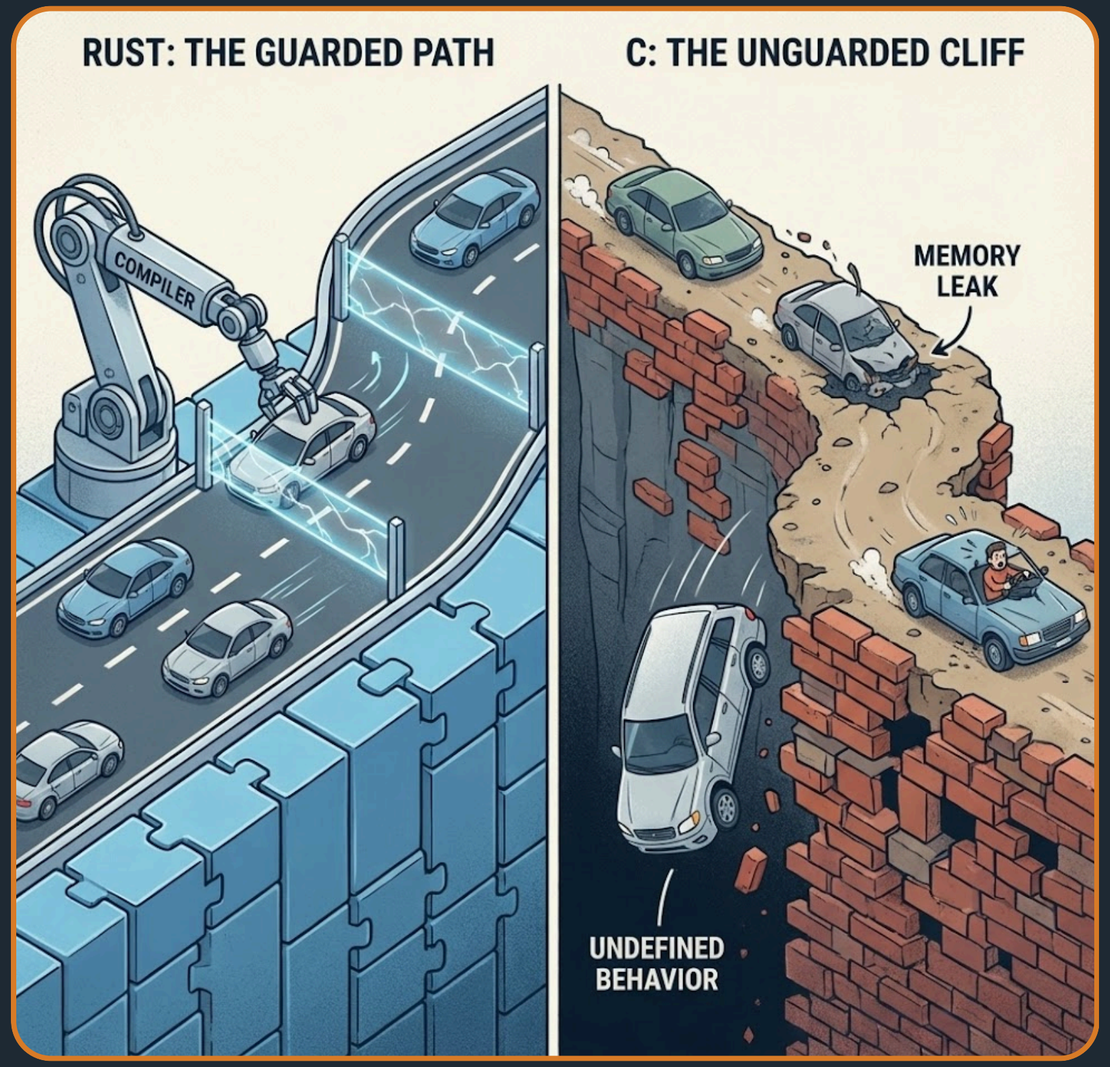
C: Structs and Function Pointers

- Struct + function-pointer patterns emulate polymorphism.
- No built-in interfaces or generics.
- More boilerplate and easier to misuse pointers.

Ownership and Borrowing in Rust

Memory safety without a garbage collector

- Each value in Rust has exactly **one owner**.
- Ownership can move; borrowing (& / &mut) gives **temporary access**.
- The compiler enforces these rules, preventing common memory bugs.



Memory Management: Rust vs C

Comparing ownership and pointers

Rust: Safe Ownership and Lifetimes

- Strict ownership rules prevent dangling pointers.
- Values are dropped automatically when they go out of scope.
- Compile-time checks stop many memory bugs before running.

C: Manual Memory Management Risks

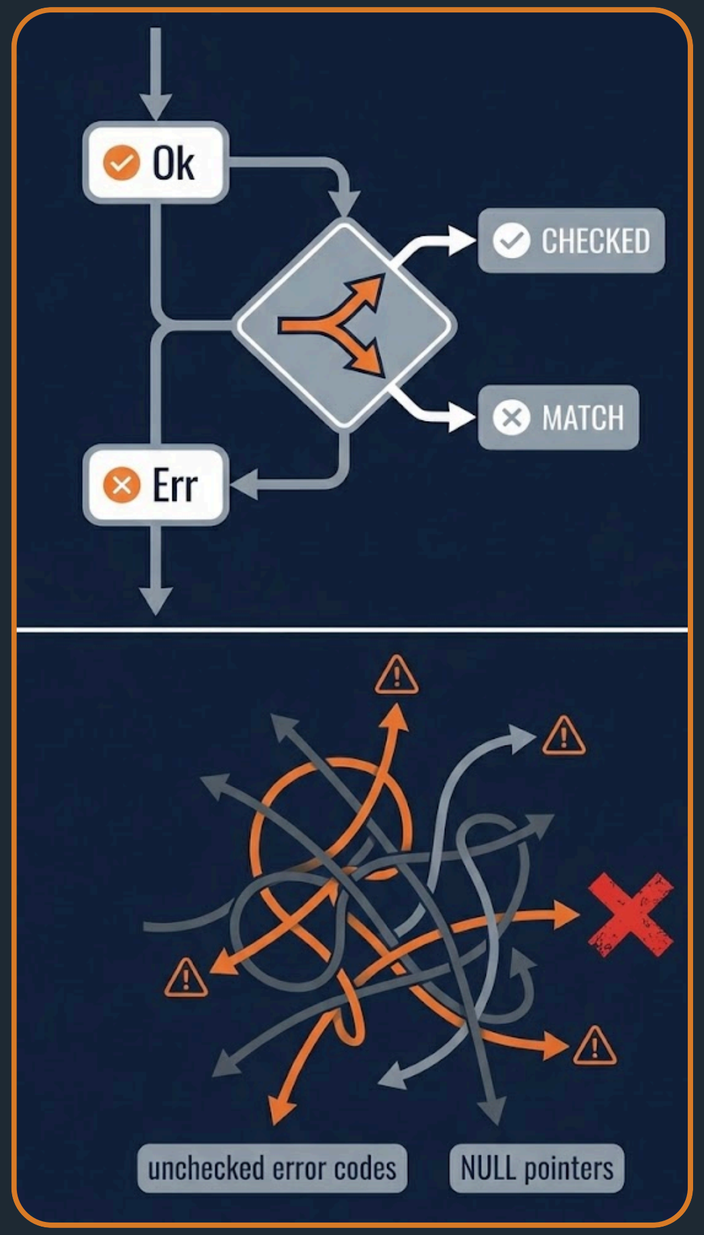
- A programmer must manage malloc/free by hand.
- Raw pointers can outlive the data they point to.
- Easy to create leaks, use-after-free, and undefined behavior.

Result and Option in Rust

Typed error handling

- **Result<T, E>** encodes success (Ok) or failure (Err).
- **Option<T>** encodes Some value or None instead of null.
- The compiler forces you to handle errors instead of ignoring them.

```rust
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err("cannot divide by zero".into())
    } else {
        Ok(a / b)
    }
}

fn main() {
    match divide(10.0, 0.0) {
        Ok(value) => println!("Result: {value}"),
        Err(err)  => println!("Error: {err}"),
    }
}
```



Error Handling: Rust vs C

Comparing approaches to error management

Rust: Types enforce handling

- Errors are values: Result<T, E> and Option<T>.
- Compiler ensures you handle success and failure.
- Pattern matching makes handling every case explicit.

C: Errors are easy to ignore

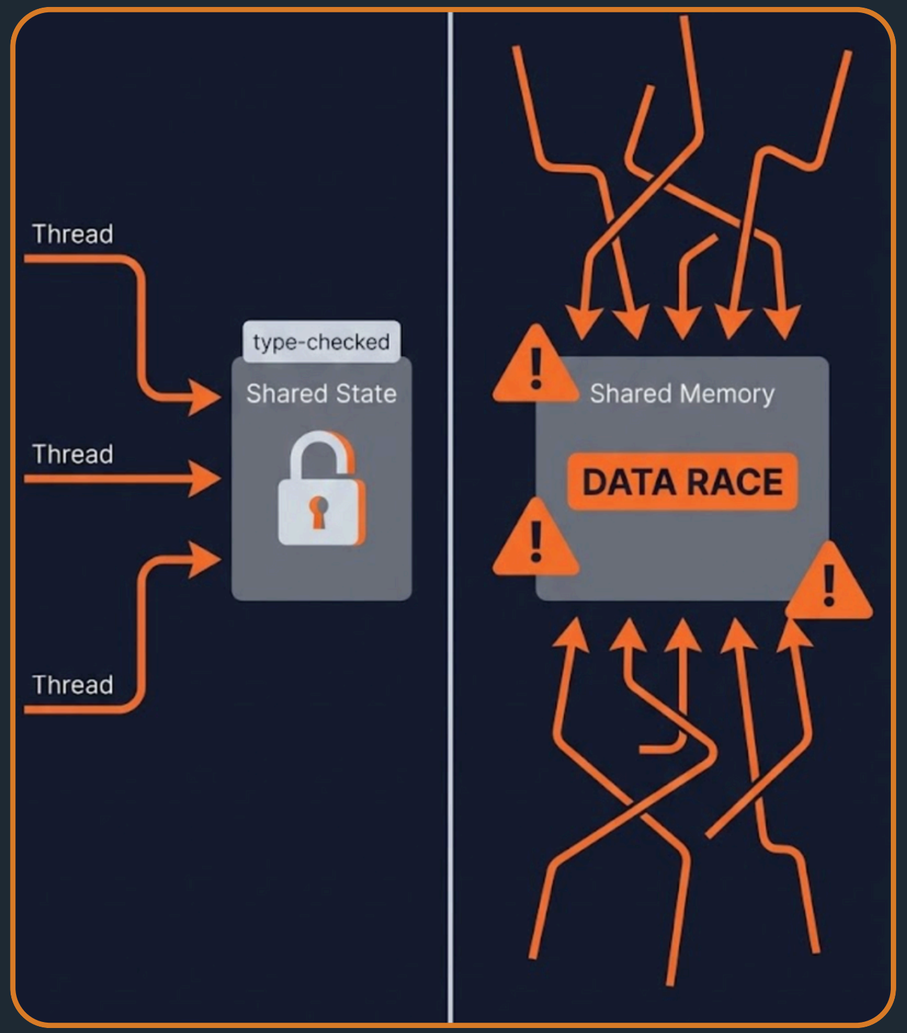
- Functions often return error codes or NULL.
- A programmer must remember to check every result.
- Unchecked errors can crash programs or corrupt data.

Concurrency in Rust

Threads without data races

- Rust tags types with **Send/Sync** to control what can cross threads.
- Shared state uses **Mutex**, **Arc**, or **message-passing** channels.
- The compiler blocks data races at compile time, not at runtime.

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

fn main() {
    let counter = Arc::new(Mutex::new(0));

    let mut handles = vec![];

    for _ in 0..4 {
        let counter = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            for _ in 0..1000 {
                let mut n = counter.lock().unwrap();
                *n += 1;
            }
        }));
    }

    for h in handles {
        h.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```



Concurrency: Rust vs C

Type-checked vs manual thread safety

Rust: Type-checked concurrency

- Ownership + borrowing rules apply across threads.
- Send/Sync traits restrict which values can be shared.
- Libraries (Mutex, Arc, channels) are safe by default.

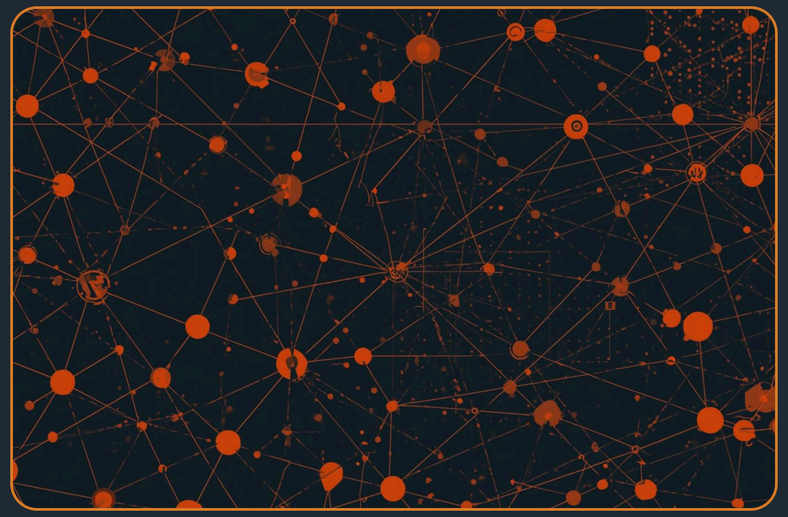
C: Manual concurrency

- Uses pthreads or OS threads with shared memory.
- A programmer must manage locks and shared state by hand.
- Data races and subtle timing bugs are easy to introduce.

When to Choose Rust Over C

Safety

Rust provides memory safety and data-race freedom by default.



Performance

Rust maintains high efficiency without sacrificing safety.

Scalability

Rust’s strong types and modules help large codebases stay reliable.

Code Examples

Scan the QR to explore the Rust vs C repo

That’s it from me. If you’d like to look at any of these examples in more detail, you can scan this QR code to open the GitHub repo. I’ve included separate files for polymorphism, memory, concurrency, and error handling!