Intermediate Rust Programming

Advanced Concepts and Patterns

Rust Learning Series

December 14, 2025

Prerequisites

This presentation assumes you understand:

- Ownership, borrowing, and lifetimes (basics)
- Structs, enums, and pattern matching
- Traits and generics (fundamentals)
- Error handling with Result and Option
- Basic concurrency concepts

What we'll cover:

- Advanced trait patterns and design
- Lifetime complexity and variance
- Macro programming (declarative & procedural)
- Advanced async patterns
- Unsafe Rust and FFI
- Performance optimization

Associated Types

Why Associated Types?

Associated types allow traits to define placeholder types that implementers specify.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

impl Iterator for Counter {
    type Item = u32;
    fn next(&mut self) -> Option<u32> { /* ... */ }
}
```

vs. Generic Parameters:

- Associated types: one implementation per type
- Generics: multiple implementations possible

Trait Objects and Dynamic Dispatch

Object Safety

A trait is object-safe if:

- All methods return types are sized
- Methods don't use `Self` in return position
- No generic methods

```rust
// Object-safe trait
trait Draw {
    fn draw(&self);
}

// Usage with trait objects
let objects: Vec<Box<dyn Draw>> = vec![
    Box::new(Circle { radius: 5 }),
    Box::new(Rectangle { width: 10, height: 20 }),
];
```

Advanced Trait Bounds

```rust
// Higher-Ranked Trait Bounds (HRTB)
fn apply<F>(f: F) -> i32
where
    F: for<'a> Fn(&'a i32) -> &'a i32
{
    let x = 42;
    *f(&x)
}

// Trait bound with associated type constraints
fn process<T>(items: T)
where
    T: Iterator<Item = String> + Clone
{
    for item in items.clone() {
        println!("{}", item);
    }
}
```

Blanket Implementations

```rust
// Implement trait for all types that satisfy
    constraints
impl<T: Display> ToString for T {
    fn to_string(&self) -> String {
        format!("{}", self)
    }
}

// Negative trait bounds (unstable)
trait Special {}
impl<T> Special for T where T: !Copy {}
```

Use Case

Blanket implementations provide functionality to many types at once, reducing code duplication.

Lifetime Variance

Variance Types

- **Covariant**: `&'a T` - longer lifetimes can substitute shorter
- **Contravariant**: `fn(&'a T)` - shorter lifetimes can substitute longer
- **Invariant**: `&'a mut T` - exact match required

```rust
// Covariance example
fn covariant<'a>(x: &'a str) -> &'a str {
    let y: &'static str = "hello";
    x  // 'a can be 'static (longer lifetime)
}

// Invariance: mutable references
fn invariant<'a>(x: &'a mut i32) {
    // Cannot substitute different lifetimes
}
```

Multiple Lifetime Parameters

```rust
struct Context<'s, 'c> {
    session: &'s Session,
    config: &'c Config,
}

impl<'s, 'c> Context<'s, 'c> {
    fn new(s: &'s Session, c: &'c Config) -> Self {
        Context { session: s, config: c }
    }
}

// Lifetime bounds
fn process<'a, 'b: 'a>(
    x: &'a str,
    y: &'b str
) -> &'a str
where
    'b: 'a  // 'b outlives 'a
{
```

Lifetime Elision Advanced Cases

```rust
// Case 1: No elision - multiple inputs
fn longest(x: &str, y: &str) -> &str {  // Error!
    if x.len() > y.len() { x } else { y }
}

// Case 2: Explicit lifetimes required
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

// Case 3: Struct lifetimes
struct Parser<'a> {
    buffer: &'a str,
}

impl<'a> Parser<'a> {
    fn parse(&self) -> &'a str {  // Elided!
        self.buffer
    }
```

Declarative Macros (macro_rules!)

```
macro_rules! hashmap {
    ($($key:expr => $val:expr),* $(,)?) => {
        {
            let mut map = HashMap::new();
            $(
                map.insert($key, $val);
            )*
            map
        }
    };
}

// Usage
let map = hashmap! {
    "one" => 1,
    "two" => 2,
};
```

Procedural Macros - Derive

```
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(Builder)]
pub fn derive_builder(input: TokenStream) ->
    TokenStream {
    let ast = syn::parse(input).unwrap();
    impl_builder(&ast)
}

fn impl_builder(ast: &syn::DeriveInput) -> TokenStream
    {
    let name = &ast.ident;
    let gen = quote! {
        impl #name {
            pub fn builder() -> Builder {
                Builder::default()
            }
```

Attribute and Function-like Macros

```rust
// Attribute macro
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream)
    -> TokenStream
{
    // Parse attr as route path
    // Transform function into route handler
}

// Function-like macro
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
    // Parse SQL at compile time
    // Generate type-safe query code
}

// Usage
#[route("/users/:id")]
fn get_user(id: u32) -> User { /* ... */ }
```

Async Traits

```rust
// Using async-trait crate
use async_trait::async_trait;

#[async_trait]
trait AsyncDatabase {
    async fn query(&self, sql: &str)
        -> Result<Vec<Row>, Error>;
}

#[async_trait]
impl AsyncDatabase for PostgresDB {
    async fn query(&self, sql: &str)
        -> Result<Vec<Row>, Error>
    {
        self.conn.query(sql).await
    }
}
```

Note

Pinning and Futures

```rust
use std::pin::Pin;
use std::future::Future;

// Manual Future implementation
struct MyFuture {
    state: State,
}

impl Future for MyFuture {
    type Output = i32;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output> {
        // Poll logic
        Poll::Ready(42)
    }
}
```

Async Cancellation and Select

```rust
use tokio::select;
use tokio::time::{sleep, Duration};

async fn process_with_timeout() -> Result<Data, Error>
    {
    select! {
        result = fetch_data() => {
            result
        }
        _ = sleep(Duration::from_secs(5)) => {
            Err(Error::Timeout)
        }
    }
}

// Graceful cancellation
use tokio::sync::CancellationToken;

async fn worker(token: CancellationToken) {
```

Unsafe Superpowers

What `unsafe` Allows

- Dereference raw pointers
- Call unsafe functions
- Access/modify mutable statics
- Implement unsafe traits
- Access union fields

```
// Raw pointers
let mut num = 5;
let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1: {}", *r1);
    *r2 = 10;
}
```

Safe Abstractions over Unsafe Code

```rust
pub struct Vec<T> {
    ptr: *mut T,
    len: usize,
    cap: usize,
}

impl<T> Vec<T> {
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap {
            self.grow();
        }
        unsafe {
            std::ptr::write(
                self.ptr.add(self.len),
                elem
            );
        }
        self.len += 1;
    }
```

Foreign Function Interface (FFI)

```rust
// Calling C from Rust
extern "C" {
    fn abs(input: i32) -> i32;
}

unsafe {
    println!("Absolute value: {}", abs(-3));
}

// Exposing Rust to C
#[no_mangle]
pub extern "C" fn rust_function(x: i32) -> i32 {
    x * 2
}

// Use repr(C) for C-compatible structs
#[repr(C)]
pub struct CPoint {
    x: i32,
```

Zero-Cost Abstractions

```rust
// Iterator chains compile to efficient loops
let sum: i32 = (1..100)
    .filter(|x| x % 2 == 0)
    .map(|x| x * x)
    .sum();

// Compiles to roughly:
let mut sum = 0;
for x in 1..100 {
    if x % 2 == 0 {
        sum += x * x;
    }
}
```

Inline Optimization

Use `#[inline]` and `#[inline(always)]` for performance-critical functions.

Memory Layout and Alignment

```rust
use std::mem::{size_of, align_of};

// Control layout with repr
#[repr(C)]
struct CLayout { x: u8, y: u32 }

#[repr(packed)]
struct Packed { x: u8, y: u32 }

#[repr(align(16))]
struct Aligned { data: [u8; 16] }

println!("CLayout: {} bytes", size_of::<CLayout>());
println!("Packed: {} bytes", size_of::<Packed>());
println!("Align: {}", align_of::<Aligned>());
```

Trade-offs: Packing reduces size but may reduce performance on some architectures.

SIMD and Platform-Specific Code

```rust
#[cfg(target_arch = "x86_64")]
use std::arch::x86_64::*;

#[target_feature(enable = "avx2")]
unsafe fn process_avx2(data: &[f32]) -> f32 {
    // SIMD operations
    // Process 8 floats at once
    let mut sum = _mm256_setzero_ps();
    for chunk in data.chunks_exact(8) {
        let v = _mm256_loadu_ps(chunk.as_ptr());
        sum = _mm256_add_ps(sum, v);
    }
    // Extract and sum all lanes
    // ...
}
```

Note: Use portable_simd for stable Rust or platform-specific intrinsics.

Error Type Design

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum AppError {
    #[error("Database error: {0}")]
    Database(#[from] sqlx::Error),

    #[error("Invalid input: {field}")]
    Validation { field: String },

    #[error("Not found: {0}")]
    NotFound(String),

    #[error(transparent)]
    Other(#[from] anyhow::Error),
}
```

Best Practices:

- Use `thiserror` for library errors

Error Context and Chaining

```rust
use anyhow::{Context, Result};

fn process_file(path: &Path) -> Result<Data> {
    let content = fs::read_to_string(path)
        .with_context(|| {
            format!("Failed to read file: {}", path.
                display())
        })?;

    parse_content(&content)
        .context("Failed to parse file content")?;

    Ok(data)
}

// Error chain provides full context:
// Error: Failed to parse file content
// Caused by: Invalid JSON at line 42
// Caused by: unexpected character '}'
```

Lock-Free Data Structures

```rust
use std::sync::atomic::{AtomicUsize, Ordering};

struct LockFreeCounter {
    count: AtomicUsize,
}

impl LockFreeCounter {
    fn increment(&self) {
        self.count.fetch_add(1, Ordering::SeqCst);
    }

    fn get(&self) -> usize {
        self.count.load(Ordering::SeqCst)
    }
}
```

Memory Orderings:

- Relaxed: No synchronization
- Acquire/Release: Synchronize memory operations

Channels and Message Passing Patterns

```rust
use tokio::sync::{mpsc, oneshot};

// Multi-producer, single-consumer
let (tx, mut rx) = mpsc::channel(100);

// Worker pool pattern
for _ in 0..4 {
    let tx = tx.clone();
    tokio::spawn(async move {
        let result = do_work().await;
        tx.send(result).await.unwrap();
    });
}

// Request-response pattern
let (resp_tx, resp_rx) = oneshot::channel();
tx.send(Request { data, resp_tx }).await?;
let response = resp_rx.await?;
```

Actor Pattern

```rust
use tokio::sync::mpsc;

struct Actor {
    receiver: mpsc::Receiver<Message>,
    state: State,
}

impl Actor {
    async fn run(mut self) {
        while let Some(msg) = self.receiver.recv().
            await {
            self.handle_message(msg).await;
        }
    }

    async fn handle_message(&mut self, msg: Message) {
        match msg {
            Message::DoWork(data) => { /* ... */ }
            Message::GetState(tx) => {
```

Phantom Types and State Machines

```rust
use std::marker::PhantomData;

struct Locked;
struct Unlocked;

struct Door<State> {
    _state: PhantomData<State>,
}

impl Door<Locked> {
    fn unlock(self) -> Door<Unlocked> {
        Door { _state: PhantomData }
    }
}

impl Door<Unlocked> {
    fn open(&self) { println!("Door opened!"); }
    fn lock(self) -> Door<Locked> {
        Door { _state: PhantomData }
```

Const Generics

```rust
// Fixed-size arrays without macros
struct Matrix<T, const ROWS: usize, const COLS: usize>
    {
    data: [[T; COLS]; ROWS],
}

impl<T, const R: usize, const C: usize> Matrix<T, R, C
    > {
    fn new(data: [[T; C]; R]) -> Self {
        Matrix { data }
    }
}

// Type-safe matrix multiplication
fn multiply<T, const M: usize, const N: usize, const P
    : usize>(
    a: &Matrix<T, M, N>,
    b: &Matrix<T, N, P>,
) -> Matrix<T, M, P> {
```

GATs - Generic Associated Types

```rust
// Unstable/stabilizing feature
trait LendingIterator {
    type Item<'a> where Self: 'a;

    fn next<'a>(&'a mut self) -> Option<Self::Item<'a
        >>;
}

// Allows lending references with lifetimes tied to
    self
struct WindowsMut<'data, T> {
    data: &'data mut [T],
    size: usize,
}

impl<'data, T> LendingIterator for WindowsMut<'data, T
    > {
    type Item<'a> = &'a mut [T] where Self: 'a;
```

Key Takeaways

1. **Traits**: Use associated types, object safety, and blanket impls effectively
2. **Lifetimes**: Understand variance and complex lifetime relationships
3. **Macros**: Leverage declarative and procedural macros for code generation
4. **Async**: Master pinning, futures, and async patterns
5. **Unsafe**: Write safe abstractions over unsafe code
6. **Performance**: Use zero-cost abstractions and control memory layout
7. **Concurrency**: Apply lock-free structures and message passing patterns
8. **Type System**: Exploit phantom types, const generics, and GATs

Advanced Resources

Books and Guides

- The Rustonomicon (unsafe Rust)
- Rust Performance Book
- Rust Async Book
- Programming Rust (O'Reilly)

Practice

- Contribute to open source Rust projects
- Build systems programming projects
- Implement data structures from scratch
- Explore embedded Rust and WebAssembly

Next Steps

1. **Specialize**: Choose a domain (systems, web, embedded, etc.)
2. **Deep Dive**: Study source code of major Rust projects
3. **Performance**: Learn profiling and optimization techniques
4. **Ecosystem**: Explore crates ecosystem and contribute
5. **Community**: Join Rust working groups and discussions

Keep Building and Learning!

Resources

Official Resources

- The Rustonomicon: `doc.rust-lang.org/nomicon/`
- Async Book: `rust-lang.github.io/async-book/`
- Reference: `doc.rust-lang.org/reference/`

Community

- Rust Users Forum
- r/rust on Reddit
- Rust Discord
- This Week in Rust newsletter