Advanced Rust Programming

Expert-Level Techniques and Internals

Rust Learning Series - Advanced Track

December 14, 2025

Prerequisites

This presentation assumes mastery of:

- All intermediate topics (traits, lifetimes, macros)
- Unsafe Rust and FFI
- Advanced async programming
- Performance optimization techniques
- Lock-free concurrency patterns

What we'll cover:

- Compiler internals and MIR
- Advanced proc macro techniques
- Custom allocators and memory management
- No-std and embedded systems
- WebAssembly deep dive
- Building language tooling

Rust Compilation Pipeline

Compilation Stages

1. Lexing and Parsing (AST)
2. Macro Expansion
3. HIR (High-level IR)
4. Type Checking and Inference
5. MIR (Mid-level IR)
6. Borrow Checking
7. Optimization
8. LLVM IR
9. Machine Code

```rust
// View MIR for a function
#[rustc_dump_mir(before = "all", after = "all")]
fn example(x: i32) -> i32 {
    x * 2 + 1
}
```

Working with MIR

```
// Simplified MIR representation
fn example(_1: i32) -> i32 {
    let mut _0: i32;
    let mut _2: i32;
    let mut _3: i32;

    bb0: {
        _2 = _1;
        _3 = const 2_i32;
        _2 = Mul(move _2, move _3);
        _3 = const 1_i32;
        _0 = Add(move _2, move _3);
        return;
    }
}
```

MIR is used for: Borrow checking, optimization, const evaluation

Compiler Plugins and Lints

```rust
#![feature(rustc_private)]
extern crate rustc_lint;
extern crate rustc_middle;

use rustc_lint::{LateContext, LateLintPass, LintContext};

declare_lint! {
    pub CUSTOM_LINT,
    Warn,
    "description of custom lint"
}

impl<'tcx> LateLintPass<'tcx> for CustomLint {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &
        Expr) {
        // Custom lint logic
    }
}
```

Complex Derive Macros

```rust
#[proc_macro_derive(Builder, attributes(builder))]
pub fn derive_builder(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);

    let name = &input.ident;
    let builder_name = format_ident!("{}Builder", name);

    let fields = match &input.data {
        Data::Struct(data) => &data.fields,
        _ => panic!("Builder only works on structs"),
    };

    // Generate builder methods
    let methods = fields.iter().map(|f| {
        let field_name = &f.ident;
        let field_type = &f.ty;
        quote! {
            pub fn #field_name(mut self, value: #field_type)
                -> Self
            {
                self.#field_name = Some(value);
```

Parsing Complex Syntax

```
use syn::{parse::Parse, Token};

struct MyMacroInput {
    name: Ident,
    _arrow: Token![=>],
    value: Expr,
}

impl Parse for MyMacroInput {
    fn parse(input: ParseStream) -> Result<Self> {
        Ok(MyMacroInput {
            name: input.parse()?,
            _arrow: input.parse()?,
            value: input.parse()?,
        })
    }
}

#[proc_macro]
pub fn my_macro(input: TokenStream) -> TokenStream {
    let parsed = parse_macro_input!(input as MyMacroInput);
```

Hygiene and Spans

```rust
use proc_macro::Span;
use quote::quote_spanned;

fn generate_with_span(span: Span) -> TokenStream {
    quote_spanned! {span=>
        compile_error!("Error at original location");
    }.into()
}

// Preserve spans for better error messages
let tokens = quote_spanned! {field.span()=>
    self.#field_name = value;
};
```

Hygiene

Macro-generated identifiers don't interfere with user code

The Allocator Trait

```rust
use std::alloc::{GlobalAlloc, Layout, System};

struct MyAllocator;

unsafe impl GlobalAlloc for MyAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        println!("Allocating {} bytes", layout.size());
        System.alloc(layout)
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        println!("Deallocating {} bytes", layout.size());
        System.dealloc(ptr, layout)
    }
}

#[global_allocator]
static ALLOCATOR: MyAllocator = MyAllocator;
```

Arena Allocators

```rust
struct Arena {
    buf: Vec<u8>,
    offset: usize,
}

impl Arena {
    fn alloc<T>(&mut self, value: T) -> &mut T {
        let layout = Layout::new::<T>();
        let offset = self.offset;
        self.offset += layout.size();

        unsafe {
            let ptr = self.buf.as_mut_ptr()
                .add(offset) as *mut T;
            ptr.write(value);
            &mut *ptr
        }
    }

    fn reset(&mut self) {
        self.offset = 0;
```

Per-Collection Allocators

```rust
#![feature(allocator_api)]

use std::alloc::Allocator;

// Custom allocator for specific use case
struct BumpAllocator { /* ... */ }

unsafe impl Allocator for BumpAllocator {
    fn allocate(&self, layout: Layout)
        -> Result<NonNull<[u8]>, AllocError>
    {
        // Bump allocation logic
    }

    unsafe fn deallocate(&self, ptr: NonNull<u8>, layout:
        Layout) {
        // No-op for bump allocator
    }
}

// Use with collections
```

No-Std Fundamentals

```rust
#![no_std]
#![no_main]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}

#[no_mangle]
pub extern "C" fn _start() -> ! {
    // Entry point for bare metal
    loop {}
}

// Use core instead of std
use core::ptr;
use core::mem;
```

No-std environment: No heap, no OS, minimal runtime

Embedded HAL (Hardware Abstraction Layer)

```rust
use embedded_hal::digital::v2::OutputPin;

struct Led<P: OutputPin> {
    pin: P,
}

impl<P: OutputPin> Led<P> {
    fn on(&mut self) -> Result<(), P::Error> {
        self.pin.set_high()
    }

    fn off(&mut self) -> Result<(), P::Error> {
        self.pin.set_low()
    }
}

// Works with any GPIO implementation
let mut led = Led { pin: gpio_pin };
led.on().unwrap();
```

Volatile Memory Access

```rust
use core::ptr::{read_volatile, write_volatile};

// Memory-mapped IO
const GPIO_BASE: usize = 0x4000_0000;

#[repr(C)]
struct GpioRegisters {
    data: u32,
    direction: u32,
    interrupt: u32,
}

fn set_gpio(bit: u8) {
    unsafe {
        let gpio = GPIO_BASE as *mut GpioRegisters;
        let mut data = read_volatile(&(*gpio).data);
        data |= 1 << bit;
        write_volatile(&mut (*gpio).data, data);
    }
}
```

Interrupt Handling

```rust
use cortex_m_rt::interrupt;

#[interrupt]
fn TIM2() {
    // Timer 2 interrupt handler
    static mut COUNT: u32 = 0;

    unsafe {
        *COUNT += 1;
        // Clear interrupt flag
        (*TIM2::ptr()).sr.modify(|_, w| w.uif().clear_bit())
            ;
    }
}

// Configure interrupt
unsafe {
    cortex_m::peripheral::NVIC::unmask(Interrupt::TIM2);
}
```

Wasm Bindgen

```rust
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn fibonacci(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fibonacci(n - 1) + fibonacci(n - 2),
    }
}

#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    log(&format!("Hello, {}!", name));
}
```

JavaScript Interop

```rust
use wasm_bindgen::JsCast;
use web_sys::{Document, Element, HtmlElement};

#[wasm_bindgen(start)]
pub fn main() -> Result<(), JsValue> {
    let window = web_sys::window().unwrap();
    let document = window.document().unwrap();

    let body = document.body().unwrap();
    let div = document.create_element("div")?;
    div.set_inner_html("Hello from Rust!");

    body.append_child(&div)?;
    Ok(())
}
```

web-sys: Auto-generated bindings for Web APIs

Optimizing Wasm Size

```
# Cargo.toml
[profile.release]
opt-level = "z"             # Optimize for size
lto = true                  # Link-time optimization
codegen-units = 1           # Better optimization
panic = "abort"             # Smaller binary
strip = true                # Strip symbols

# Additional tools
# wasm-opt for further optimization
# wasm-snip to remove unused code
```

Size Reduction Techniques

- Avoid formatting macros in release
- Use `wee_alloc` instead of default allocator
- Tree-shake with `wasm-gc`

Building a Simple Executor

```rust
use std::future::Future;
use std::task::{Context, Poll, RawWaker, RawWakerVTable,
    Waker};

struct SimpleExecutor {
    tasks: Vec<Pin<Box<dyn Future<Output = ()>>>>,
}

impl SimpleExecutor {
    fn run(&mut self) {
        while !self.tasks.is_empty() {
            self.tasks.retain_mut(|task| {
                let waker = create_waker();
                let mut cx = Context::from_waker(&waker);

                match task.as_mut().poll(&mut cx) {
                    Poll::Ready(()) => false,  // Remove
                    Poll::Pending => true,     // Keep
                }
            });
        }
```

Waker Implementation

```
fn create_waker() -> Waker {
    unsafe fn clone(ptr: *const ()) -> RawWaker {
        RawWaker::new(ptr, &VTABLE)
    }

    unsafe fn wake(_: *const ()) {
        // Wake the task
    }

    unsafe fn wake_by_ref(_: *const ()) {
        // Wake without consuming
    }

    unsafe fn drop(_: *const ()) {}

    static VTABLE: RawWakerVTable = RawWakerVTable::new(
        clone, wake, wake_by_ref, drop
    );

    let raw = RawWaker::new(std::ptr::null(), &VTABLE);
    unsafe { Waker::from_raw(raw) }
}
```

Reactor Pattern

```rust
use mio::{Events, Interest, Poll, Token};

struct Reactor {
    poll: Poll,
    events: Events,
    handlers: HashMap<Token, Box<dyn FnMut()>>,
}

impl Reactor {
    fn register<S: Source>(&mut self, source: &mut S,
                           handler: impl FnMut() + 'static)
    {
        let token = Token(self.handlers.len());
        self.poll.registry()
            .register(source, token, Interest::READABLE)
            .unwrap();
        self.handlers.insert(token, Box::new(handler));
    }

    fn run(&mut self) {
        loop {
```

ABA Problem and Solutions

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

// Simple (unsafe) stack - has ABA problem
struct Node<T> {
    data: T,
    next: *mut Node<T>,
}

struct Stack<T> {
    head: AtomicPtr<Node<T>>,
}

impl<T> Stack<T> {
    fn push(&self, data: T) {
        let node = Box::into_raw(Box::new(Node {
            data,
            next: std::ptr::null_mut(),
        }));

        loop {
            let head = self.head.load(Ordering::Acquire);
```

Tagged Pointers for ABA Prevention

```rust
// Pack counter with pointer
struct Tagged<T> {
    ptr: usize,  // Bottom bits: counter, top bits: pointer
}

impl<T> Tagged<T> {
    fn new(ptr: *mut T, tag: usize) -> Self {
        let addr = ptr as usize;
        Tagged { ptr: addr | (tag & 0xFFFF) }
    }

    fn get_ptr(&self) -> *mut T {
        (self.ptr & !0xFFFF) as *mut T
    }

    fn get_tag(&self) -> usize {
        self.ptr & 0xFFFF
    }
}

// Use AtomicUsize for tagged pointer
```

Epoch-Based Reclamation

```
use crossbeam_epoch::{self as epoch, Atomic, Owned};

struct Node<T> {
    data: T,
    next: Atomic<Node<T>>,
}

struct Stack<T> {
    head: Atomic<Node<T>>,
}

impl<T> Stack<T> {
    fn push(&self, data: T) {
        let node = Owned::new(Node {
            data,
            next: Atomic::null(),
        });

        let guard = epoch::pin();
        loop {
            let head = self.head.load(Ordering::Acquire, &
```

Using rust-analyzer APIs

```rust
use ra_ap_syntax::{ast, AstNode, SyntaxKind};
use ra_ap_ide::{Analysis, AnalysisHost, FileId};

fn analyze_code(source: &str) -> Vec<String> {
    let parse = ast::SourceFile::parse(source);
    let root = parse.tree();

    let mut functions = Vec::new();

    for node in root.syntax().descendants() {
        if let Some(func) = ast::Fn::cast(node) {
            if let Some(name) = func.name() {
                functions.push(name.to_string());
            }
        }
    }

    functions
}
```

Custom Cargo Subcommands

```rust
// cargo-mycmd/src/main.rs
use clap::Parser;

#[derive(Parser)]
#[command(name = "cargo")]
#[command(bin_name = "cargo")]
enum Cargo {
    Mycmd(Args),
}

#[derive(Parser)]
struct Args {
    #[arg(long)]
    verbose: bool,
}

fn main() {
    let Cargo::Mycmd(args) = Cargo::parse();

    // Access cargo metadata
    let metadata = cargo_metadata::MetadataCommand::new()
```

LSP Server Implementation

```rust
use tower_lsp::{LspService, Server};
use tower_lsp::lsp_types::*;

struct Backend;

#[tower_lsp::async_trait]
impl LanguageServer for Backend {
    async fn initialize(&self, _: InitializeParams)
        -> Result<InitializeResult>
    {
        Ok(InitializeResult {
            capabilities: ServerCapabilities {
                text_document_sync: Some(
                    TextDocumentSyncCapability::Kind(
                        TextDocumentSyncKind::FULL
                    )
                ),
                completion_provider: Some(CompletionOptions
                    ::default()),
                ..Default::default()
            }
```

Benchmarking with Criterion

```rust
use criterion::{black_box, criterion_group, criterion_main,
    Criterion};

fn fibonacci_benchmark(c: &mut Criterion) {
    c.bench_function("fib 20", |b| {
        b.iter(|| fibonacci(black_box(20)))
    });

    c.bench_function("fib_iterative 20", |b| {
        b.iter(|| fib_iterative(black_box(20)))
    });
}

criterion_group!(benches, fibonacci_benchmark);
criterion_main!(benches);
```

Criterion Features

- Statistical analysis
- HTML reports with plots

CPU Profiling

```
# Using perf on Linux
$ cargo build --release
$ perf record --call-graph=dwarf ./target/release/myapp
$ perf report

# Using flamegraph
$ cargo install flamegraph
$ cargo flamegraph

# Using samply (modern alternative)
$ cargo install samply
$ samply record ./target/release/myapp
```

Profile-guided optimization (PGO):

```
# 1. Build with instrumentation
RUSTFLAGS="-Cprofile-generate=/tmp/pgo" cargo build --
    release

# 2. Run typical workloads
./target/release/myapp < typical_input.txt
```

Memory Profiling

```rust
// Using dhat for heap profiling
#[global_allocator]
static ALLOC: dhat::Alloc = dhat::Alloc;

fn main() {
    let _profiler = dhat::Profiler::new_heap();

    // Your code here
    let v: Vec<u64> = (0..1_000_000).collect();

    // Profiler drops, generating report
}

// Using valgrind massif
$ valgrind --tool=massif ./target/release/myapp
$ ms_print massif.out.12345
```

Rust Compiler Development

```
# Clone and build rustc
$ git clone https://github.com/rust-lang/rust.git
$ cd rust
$ ./x.py build

# Run tests
$ ./x.py test

# Build specific component
$ ./x.py build library/std

# Build documentation
$ ./x.py doc
```

Key Areas

- Compiler: Type checking, borrow checking, MIR
- Standard library: Core, alloc, std
- Cargo: Build system and package manager

RFC Process

1. **Idea**: Discuss on internals.rust-lang.org
2. **RFC**: Submit RFC (Request for Comments)
3. **Discussion**: Community reviews and suggests changes
4. **FCP**: Final Comment Period (10 days)
5. **Merge**: RFC accepted, implementation begins
6. **Implementation**: Write code, tests, docs
7. **Stabilization**: Feature gate removal after testing

Example RFCs:

- RFC 2229: Capture disjoint fields in closures
- RFC 2585: FC-solve
- RFC 3086: metaevar expr

Writing Compiler Tests

```
// tests/ui/my-feature.rs
fn main() {
    let x: i32 = "hello"; //~ ERROR mismatched types
}

// Run test
$ ./x.py test tests/ui/my-feature.rs

// Update expected output
$ ./x.py test tests/ui --bless
```

Test Types

- `ui`: Compiler error/warning tests
- `compile-fail`: Tests that should fail compilation
- `run-pass`: Tests that should compile and run
- `incremental`: Incremental compilation tests

Key Takeaways

1. **Compiler**: Understand MIR and compilation pipeline
2. **Proc Macros**: Master complex code generation
3. **Allocators**: Implement custom memory management
4. **No-std**: Build for embedded and bare metal
5. **Wasm**: Deploy Rust to the web
6. **Async**: Build custom executors and runtimes
7. **Lock-Free**: Implement advanced concurrent data structures
8. **Tooling**: Create language tools and IDE support
9. **Profiling**: Optimize for maximum performance
10. **Contributing**: Give back to the Rust ecosystem

Mastery Projects

Challenge Yourself

- Build a custom async runtime
- Implement a garbage collector
- Create a programming language in Rust
- Write a kernel module or OS
- Build a database engine
- Implement a JIT compiler
- Create embedded firmware
- Contribute to rustc or cargo

Expert Territory

You're now equipped to tackle the most challenging Rust projects!

Advanced Resources

Deep Dives

- Rust Compiler Development Guide
- Embedded Rust Book
- Rust and WebAssembly Book
- Lock-Free Programming Papers
- "Writing an OS in Rust" blog series
- Rustc Dev Guide

Communities

- #rustc on Rust Zulip
- Embedded WG
- Compiler Team meetings
- Working Groups (async, wasm, etc.)

The Journey Continues

From Beginner to Expert

You've completed the full Rust learning path:

Beginner → Fundamentals
Intermediate → Advanced Patterns
Advanced → Expert Techniques

Keep Exploring, Keep Building!

Resources

Official Documentation

- Rustc Dev Guide: `rustc-dev-guide.rust-lang.org`
- Forge: `forge.rust-lang.org`
- Embedded Book: `docs.rust-embedded.org`
- Wasm Book: `rustwasm.github.io`

Community

- Zulip: `rust-lang.zulipchat.com`
- Internals Forum: `internals.rust-lang.org`
- GitHub: `github.com/rust-lang`
- This Week in Rust