Rust Programming Language: The Definitive Guide from Hello World to Expert-Level Topics

Rust is a modern systems programming language that combines C/C++-like performance with unparalleled memory and thread safety. It eliminates common bugs like null pointer dereferences, buffer overflows, and data races through compile-time checks, offers zero-cost abstractions, and enables fearless concurrency. This README is the ultimate guide to Rust, covering every topic from beginner to expert level, with exhaustive explanations, all subtypes, practical examples, and cross-topic connections.

Table of Contents

1. Introduction to Rust
2. Getting Started: Hello, World!
3. Rust Editions
4. Basic Concepts
    - Variables and Mutability
    - Data Types
    - Functions
    - Control Flow
5. Ownership and Borrowing
    - Ownership Rules
    - Borrowing and References
    - Slices
6. Structs and Enums
    - Structs
    - Enums and Pattern Matching
7. Modules and Crates
    - Modules
    - Crates
8. Collections
    - Vectors
    - Strings
    - HashMaps and Other Collections
9. Error Handling

    - Option and Result
    - Panic! and Unrecoverable Errors
10. Generics and Traits
    - Generics
    - Traits
    - Associated Types and Constants
    - Trait Objects
11. Lifetimes
    - Lifetime Annotations
    - Static and Elided Lifetimes
    - Variance in Lifetimes
12. Concurrency
    - Threads
    - Message Passing
    - Shared-State Concurrency
13. Advanced Topics
    - Unsafe Rust
    - Smart Pointers
    - Async/Await
    - Macros
    - Foreign Function Interface (FFI)
    - Closures and Iterators
        - Closures
        - Closures as Function Inputs
        - Iterators
    - WebAssembly
    - Inline Assembly
    - Dynamic vs. Static Dispatch
    - Attributes
    - Pin and Unpin

    - Type Aliases and Newtypes
    - Zero-Sized Types (ZSTs)
    - Dynamic Libraries and Plugins
    - Feature Gates and Nightly Rust
    - Higher-Ranked Trait Bounds (HRTBs)
    - Embedded Rust
    - Cross-Compilation
14. Testing in Rust
    - Unit Tests
    - Integration Tests
    - Doc Tests
    - Benchmarking
15. Metaprogramming and Build Tools
    - Build Scripts
    - Custom Derive and Procedural Macros
16. Memory Management and Allocators
17. Building a Sample Project
18. Resources

Introduction to Rust

Rust, initially developed by Mozilla and now maintained by the Rust Foundation, is a systems programming language that delivers high performance with robust safety guarantees. Its key features include:

- **Memory Safety**: Eliminates null pointer dereferences, buffer overflows, and data races via compile-time ownership and borrowing checks.
- **Zero-Cost Abstractions**: High-level features like pattern matching, iterators, and generics compile to efficient machine code, matching C/C++ performance.
- **Concurrency**: Ownership ensures thread safety, enabling fearless concurrent programming.
- **Ecosystem**: Cargo (package manager) and Crates.io provide a vibrant library ecosystem.
- **Use Cases**: Operating systems (Redox), web browsers (Firefox's Servo), WebAssembly, CLI tools, high-performance servers (Actix, Rocket), embedded systems, and blockchain.

Rust's design makes it ideal for performance-critical applications, safe concurrency, and modern systems programming, with growing adoption in industries like cloud infrastructure, IoT, and game development.

Getting Started: Hello, World!

Install Rust using `rustup` from rust-lang.org. Create a new project:

```
cargo new hello_world
cd hello_world
```

The basic "Hello, World!" program:

```
fn main() {
    println!("Hello, World!");
}
```

Run with:

```
cargo run
```

The `main` function is the entry point, and `println!` is a macro for formatted console output. Cargo manages dependencies, builds, tests, and documentation.

Rust Editions

Rust uses **editions** (2015, 2018, 2021, and upcoming editions) to introduce breaking changes while maintaining backward compatibility. Editions are specified in `Cargo.toml`:

```
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"
```

Key changes:

- **2015**: Initial release with core ownership model.
- **2018**: Introduced `async`/`await`, simplified `use` paths, `try` blocks, and raw identifiers (`r#struct`).
- **2021**: Enhanced `match` ergonomics, `const` generics, and array `into_iter`.

Use `cargo fix` to migrate code between editions. Editions ensure Rust evolves without breaking existing codebases.

Basic Concepts

Variables and Mutability

Rust variables are immutable by default to prevent unintended changes, enhancing safety. Use `mut` for mutability. **Shadowing** allows reusing variable names with different types or values. **Constants** (`const`) are computed at compile time, immutable, and scoped globally or locally. **Static** variables have a fixed memory location and `'static` lifetime, requiring `unsafe` for mutable access.

```
fn main() {
    let x = 5; // Immutable
    println!("x: {}", x);
    // x = 6; // Error: cannot assign to immutable variable

    let mut y = 10; // Mutable
    y = 15;
```

```
    println!("y: {}", y);

    let x = "shadowed"; // Shadowing with different type
    println!("x: {}", x);

    const MAX_POINTS: u32 = 100_000; // Constant with numeric separator
    println!("Max points: {}", MAX_POINTS);

    static GREETING: &str = "Hello, static!"; // Static, immutable
    println!("Static: {}", GREETING);

    static mut COUNTER: u32 = 0; // Mutable static requires unsafe
    unsafe {
        COUNTER += 1;
        println!("Counter: {}", COUNTER);
    }
}
```

Use Cases:

- Immutability: Prevents bugs in concurrent or complex code.
- Shadowing: Useful in functional-style transformations (e.g., parsing strings to numbers).
- Constants: For fixed values like configuration or mathematical constants.
- Statics: For global state, though use sparingly due to safety concerns.

Edge Cases:

- Shadowing vs. mutability: Shadowing creates a new variable, not modifying the old one.
- Static mutability: Concurrent access requires synchronization (e.g., `Mutex`).

Data Types

Rust is statically typed, requiring types to be known at compile time. It supports:

- **Scalar Types**:
    - **Integers**: Signed (`i8`, `i16`, `i32`, `i64`, `i128`, `isize`) and unsigned (`u8`, `u16`, `u32`, `u64`, `u128`, `usize`). `isize`/`usize` are architecture-dependent (e.g., 64-bit on x86_64). Supports literals like `1_000`, `0xFF`, `0b1010`.
    - **Floats**: `f32`, `f64`, adhering to IEEE-754, with operations like `sqrt`, `sin`.
    - **Booleans**: `bool` (`true`, `false`), used in control flow.
    - **Characters**: `char`, Unicode scalar values (4 bytes, e.g., `'A'`, `'😊'`).
- **Compound Types**:
    - **Tuples**: Fixed-size, heterogeneous collections, e.g., `(i32, f64, char)`. Access via `.0`, `.1`, or destructuring.
    - **Arrays**: Fixed-size, homogeneous collections, e.g., `[i32; 5]`. Stored on stack unless boxed.

```
fn main() {
    let i: i32 = -42;
    let u: u64 = 100;
    let f: f64 = 3.14;
    let b: bool = true;
    let c: char = '😊';

    let tuple: (i32, f64, char) = (500, 6.4, 'Z');
    let (x, y, z) = tuple; // Destructuring
    let array: [i32; 3] = [1, 2, 3];

    println!("i: {}, u: {}, f: {}, b: {}, c: {}", i, u, f, b, c);
    println!("Tuple: ({}, {}, {})", x, y, z);
    println!("Array: {:?}", array);

    // Array bounds safety
    // let out_of_bounds = array[10]; // Compile-time error or panic
    println!("Safe access: {:?}", array.get(1)); // Returns Option<&i32>
}
```

Edge Cases:

- Integer overflow: Panics in debug mode, wraps in development environments unless using `checked_add`, `wrapping_add`, or `saturating_add`.
- Float precision: Beware of inaccuracies (e.g., `0.1 + 0 != 0.3`). Use `approx` crate for comparisons.
- Unicode chars: `char` is not a byte; use `String` or `&str` for text processing.

Cross-Connections:

- Ownership: Arrays and tuples follow ownership rules; `String` requires borrowing.
- Generics: Arrays with `const N` leverage const generics.

Functions

Functions are declared with `fn`, with optional return types specified via `->`. The last expression in a block is implicitly returned without a semicolon. Rust supports:

- **Free Functions**: Standalone functions.
- **Associated Functions**: Static methods (no `self`self).
- **Methods**: Take `self`, `self`self, or `&mut self`.
- **Diverging Functions**: Return `!` (never return, e.g., (e.g., `panic!`).

```
struct Point {
    width: u32,
    height: u32,
}

impl Rectangle {
    // Associated function
    fn square(size: u32) -> Rectangle {
```

```
        Rectangle { width: size, height: size }
    }

    // Method
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn add(a: i32, b: i32) -> i32 {
    a + b // Implicit return
}

fn diverge() -> ! {
    panic!("This function never returns");
}

fn main() {
    let sum = add(5, 3);
    let rect = Rectangle::square(5);
    println!("Sum: {}", sum);
    println!("Area: {}", rect.area());
    // diverge(); // Would panic
}
```

Advanced Features:

- **Function Pointers**: `fn(i32,) -> i32` for passing functions as arguments.
- **Overloading**: Not supported; use traits or different names.
- **Closures**: Functions can accept closures (covered later).

Use Cases:

- Implicit returns: Omitting `return` for non-final expressions requires semicolons.
- Diverging functions: Used in loop control or flow match or arms to ensure type consistency.

Edge Cases:

- Name conflicts: Resolved by module paths or renaming.
- FFI**: Functions Functions for C interop require `extern "C"`.

Cross-Connections:

- Traits: Methods enable trait implementations.
- Ownership: Ownership: `self` affects borrowing rules.
- Async: Functions can be `async fn` for asynchronous programming.

Control Flow

Rust provides:

- **If**: Requires boolean conditions, can return values.

- **Loop**: Infinite loop, supports `break` with values or `continue`.
- **While**: Condition-based looping.
- **For**: Iterates over iterators or ranges (e.g., `1..5`, `1..=5`).
- **Match**: Pattern matching (covered with enums).

```
fn main() {
    let number = 7;
    let result = if number > 5 {
        "greater than"
    } else if number == 5 {
        "equals"
    } else {
        "less than"
    };
    println!("Number is {}", result);
}
    let mut counter = 0;
    let loop_result = loop {
        counter += 1;
        if counter == 5 {
            break counter * 2;
        }
    };
    println!("Loop result: {}", loop_result);

    while counter > 0 {
        println!("Counter: {}", counter);
        counter -= 1;
    }

    for i in 1..=3 {
        println!("i: {}", i);
    }

    // Labeled loops
    'outer: for i in 1..3 {
        for j in 1..3 {
            if i * j > 2 {
                break 'outer;
            }
            println!("i: {}, j: {}", i, j);
        }
    }
}
```

Advanced Features:

- **Labeled Loops**: Control outer loops with `'label`.
- **If as Expressions**:
- **Range Syntax**: `a..b` (exclusive), `a..=b` (inclusive), supports custom ranges via `RangeBounds`.
  Edge Cases:

- Empty ranges: `1..1` or `1..=0` produce no iterations.
- Match exhaustion: All match arms must return compatible types or diverge.

Cross-Connections:

- Iterators: `for` loops leverage `IntoIterator`.
- Enums: `match` is central to enum handling.

Ownership and Borrowing

Rust's ownership model is its defining feature, ensuring memory safety without a garbage collector. It enforces strict rules that govern how values are managed in memory.

Ownership Rules:

- Each value has a single owner.
- Ownership can be moved to another variable.
- When the owner goes out of scope, the value is dropped (via `Drop` trait).
- Types implementing `Copy` (e.g., integers, booleans, `char`) are copied, not moved.

```
fn main() {
    let s1 = String::from("hello"); // s1 owns the String
    let s2 = s1; // Ownership moves to s2
    // println!("{}", s1); // Error: s1 is invalid
    println!("s2: {}", s2);

    let x = 5; // i32 implements Copy
    let y = x; // Copied
    println!("x: {}, y: {}", x, y);

    // Custom Drop
    struct Resource;
    impl Drop for Resource {
        fn drop(&mut self) {
            println!("Resource dropped");
        }
    }
    let r = Resource;
} // r dropped here
```

Edge Cases:

- **Partial Moves**: Moving a struct field invalidates the struct unless handled.
- **Drop Order**: Follows scope exit, last-in-first-out (LIFO).
- **Copy Trait**: Custom types can't implement `Copy` if they manage resources.

Cross-Connections:

- Borrowing: Ownership enables safe borrowing.
- **Smart Pointers**: `Box`, `Rc` manage ownership differently.

- **Concurrency**: Ownership prevents data races.

Borrowing and References

Borrowing provides temporary access to data:

- **Immutable References (`&T`)**: Multiple allowed, read-only.
- **Mutable References (`&mut T`)**: Only one at a time, read-write.
- **Rules**:
    - Cannot have mutable and immutable borrows simultaneously.
    - References must not outlive their referents.

```
fn main() {
    let mut s = String::from("hello");
    let len = calculate_length(&s); // Immutable borrow
    println!("Length: {}", len);

    change(&mut s); // Mutable borrow
    println!("Modified: {}", s);

    // let r1 = &s;
    // let r2 = &mut s; // Error: cannot borrow as mutable
}

fn calculate_length(s: &String) -> usize {
    s.len()
}

fn change(s: &mut String) {
    s.push_str(", world!");
}
```

Edge Cases:

- **Borrow Checker**: Enforces rules at compile time, preventing data results races.
- **Non-Lexical Lifetimes (NLL)**: Allows more flexible borrowing since Rust 2018.
- **Reborrowing**: `&mut T` can be reborrowed as `&T` or another `&mut T` in nested scopes.

Cross-Connections:

- Lifetimes: Borrowing relies on lifetime rules.
- Slices: References to slices are a form of borrowing.
- Concurrency: Borrowing ensures thread safety.

Slices

Slices reference a contiguous sequence portion of a collection:

- **String Slices (`&str`)**: View into a `String` or string literals.

- **Array Slices (`&[T]`)**: View into arrays or vectors.
- **Custom Slices**: Implement `SliceIndex` for custom types.

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[..i];
        }
    }
    &s[..]
}

fn main() {
    let s = String::from("hello world");
    let word = first_word(&s);
    println!("First word: {}", word);

    let arr = [1, 2, 3, 4, 5];
    let slice = &arr[1..3];
    println!("Slice: {:?}", slice);

    // Slice safety
    // let invalid_slice = &arr[10..]; // Panic at runtime
    println!("Safe slice: {:?}", arr.get(1..3)); // Returns Option<&[i32]>
}
```

Edge Cases:

- **UTF-8 Boundaries**: Slicing `&str` must align with character boundaries to avoid panics.
- **Fat Pointers**: Slices are fat pointers (pointer + length), impacting memory layout.
- **Zero-Length Slices**: Valid but require careful handling in loops.
  Cross-Connections:
- Borrowing: Slices are borrowed views.
- **Iterators**: Slices support iteration via `iter()` or `into_iter()`.
- **Collections**: Slices support are used with `Vec` and `String`.

Structs and Enums

```
### Structs
Rust supports three struct types:
- **Named-Field Structs**: Fields with names, ideal for structured data.
- **Tuple Structs**: Unnamed fields, accessed by index, for lightweight types.
- **Unit Structs**: No fields, used as markers or placeholders.

```rust
struct Point {
    x: i32,
    y: i32,
}
```
```

```
struct Color(i32, i32, i32); // Tuple struct
struct Unit; // Unit struct

impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x: x, y: y }
    }

    fn distance(&self, other: &Point) -> f64 {
        let dx = (self.x - other.x as) as f64;
        let dy = (self.y - other.y) as f64;
        (dx * dx * + dy * dy).sqrt()
    }

}
}

fn main() {
    let x = 1;
    let y = 2;
    let point = Point { x, y }; // Field shorthand
    let other = Point::new(3, 4);
    println!("Distance: {}", point.distance(&other));

    let color = Color(255, 128, 0);
    println!("Color: ({}, {}, {})", color.0, color.1, color.2);

    let unit = Unit;
}
```

Advanced Features:

- **Field shorthand**: Shorthand Simplifies struct initialization when variable names match fields.
- **Update Syntax**: `Point { x: 1, ..other }` copies fields from another instance.
- **Zero-sized Structs**: Optimize memory (covered in zero-sized ZSTs).

Edge Cases:

- **Partial Moves**: Moving a field invalidates the struct.
- **Alignment**: Structs fields are padded for alignment; use `#[repr(packed)]` for control.
- **Privacy**: Fields are private by default unless `pub`.

Cross-Connections:

- Ownership: Structs own their fields.
- Traits: Structs implement traits for behavior.
- Generics: Structs can be generic.

Enums and Pattern Matching

Enums define types with multiple variants, used with `match` or `if let`. Rust's built-in `Option` and `Result` are built-in enums.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn process_message(msg: Message) {
    match msg {
        Message::Quit => println!("Quit"),
        Message::Move { x, y } if x > 0 => println!("Positive move: ({}, {})", x, y),
        Message::Move { x, y } => println!("Move: ({}, {}", x, y),
        Message::Write(text) => println!("Text: {}", text),
        Message::ChangeColor(r, g, b) => println!("Color: ({}, {}, {})", r, g, b),
    }
}

fn main() {
    let messages = [
        Message::Write(String::from("Hello")),
        Message::Move(Move { x: 10, y: 20 }),
        Message::ChangeColor(255, 0, 0),
        Message::Quit,
    ];

    for msg in messages {
        process_message(msg);
    }

    // If let
    let msg = Message::Write(String::from("Test"));
    if let Message::Write(text) = msg {
        println!("Text (if let): {}", text);
    }

    // Match destructuring
    let complex = Message::Move { x: -5, y: 10 };
    match complex {
        Message::Move { x, y: y @ 10..=20 } => println!("Move with y in range: x={},
y={}", x, y),
        _ => println!("Other"),
    }
}
```

Advanced Pattern Matching:

- **Guards**: Add conditions to patterns.
- **Destructuring**: Extract fields or tuple elements.
- **Wildcard (_)**: Catch-all pattern.

- **Ranges**: Match ranges like `1..=10`.
- **Nested Patterns**: Match nested enums or structs.

Edge Cases:

- **Exhaustiveness**: `match` must cover all variants or use `_`.
- **Ownership**: Patterns can move or borrow values.
- **Unreachable Code**: Compiler warns about unreachable patterns.

Cross-Connections:

- Error Handling: `Option` and `Result` rely on enums.
- **Iterators**: Patterns work with iterator methods like `next`.
- Generics: Enums with can be generic.

Modules and Crates

Modules

Rust uses modules to organize code and control visibility with `pub`. Use `mod` for defining to define modules and `use` for importing to import paths. Modules can be inline or in separate files.

##rust

```
use front_of {
    pub mod hosting {
        pub fn add_to_waitlist() {
            println!("Added to waitlist!");
        }
        fn private_function() {
            println!("Private");
        }
    }

    mod serving {
        fn take_order() {} // Private
    }
}

fn main() {
    use front_of_house::hosting;
    hosting::add_to_waitlist();
    // hosting::private_function(); // Error: private
}
```

Path Syntax:

- **Absolute**: `crate::module::item`.
- **Relative**: `self::item`, `super::item`.

- **Re-exports**: `pub use` to expose items publicly.

File Structure:

- `mod hosting;` loads `hosting.rs;` or `hosting/mod.rs`.
- Nested modules mirror directory structure.

Edge Cases:

- **Privacy**: Non-`pub items items are private to the module.
- **Module Cycles**: Avoid cyclic dependencies.
- **Orphan Rules**: Trait implementations must be in the trait's or type's crate.

Cross-Connections:

- Crates: Modules organize code within crates.
- Traits: Modules group related traits implementations.
- **Macros**: Modules control macro visibility.

Crates

A crate is a binary (executable) or library. The `Cargo.toml` file defines metadata and dependencies.

```
toml
[package]
name = "my-project"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.10"
serde = { version = "1.0", features = ["derive"] }
tokio = { version = "1.4", features = ["full"] }
clap = { version = "4.2", features = ["derive"] }
criterion = "0.4"
libloading = "0.7"
rayon = "1.5"
wasm-bindgen = "0.2"
async-trait = "0.2"

[dev-dependencies]
criterion = "0.4"

[[bench]]
name = "my_bench"
harness = false
```

Use external crates for added functionality:

```
use rand::Rng;

fn main() {
    let random = rand::thread_rng().gen_range(1..=100);
```

```
    println!("Random: {}", random);
}
```

Workspace Crates: Manage multiple crates in one project.

```
toml
[workspace]
members = ["crate1", "crate2"]
```

Crate Types:

- `bin`: Executable binary.
- `lib`: Library (e.g., `rlib`, `dylib`).
- `proc-macro`: For procedural macros.

Edge Cases:

- **Dependency Conflicts**: Use `cargo tree` to resolve conflicts.
- **Feature Flags**: Enable optional dependencies with `[features]`.
- **SemVer**: Ensure compatible versions to avoid breakage.

Cross-Connections:

- Modules: Crates contain modules.
- **FFI**: Crates can link to C libraries.
- **WebAssembly**: Crates target WebAssembly with specific configurations.

Collections

Rust's standard `std::collections` module provides various collection types, all heap-allocated unless specified otherwise.

Vectors

`Vec<T>` is a growable, heap-allocated array that supports dynamic resizing, iteration, and safe indexing.

```
fn main() {
    let mut v = vec![1, 2, 3];
    v.push(4);
    v.pop();
    println!("Vector: {:?}", v);

    // Accessing elements
    let second = v[1]; // Panics if index out of bounds
    let second_safe = v.get(1); // Returns Option<&T>
    println!("Second: {}, second_safe: {:?}", second, second_safe);

    // Iterating over elements
    for i in &mut v {
        *i += 10;
    }
    println!("Modified: {:?}", v);
```

```
    // Capacity management
    v.reserve(10); // Pre-allocate space
    println!("Capacity: {}", v.capacity());

    // Slicing
    let slice = &v[1..];
    println!("Slice: {:?}", slice);
}
```

Edge Cases:

- **Reallocation**: Pushing may trigger reallocation, invalidating references unless `reserve` is used.
- **Ownership**: Moving elements out requires care to avoid invalidating the vector.
- **Drain**: `drain` removes elements while allowing iteration.

Cross-Connections:

- Slices: `Vec<T>` dereferences to `&[T]`.
- Iterators: `Vec<T>` supports `iter`, `iter_mut`, `into_iter`.
- Concurrency: Use `Arc<Vec<T>>` for shared vectors.

Strings

Rust has two main string types:

- **`String`: Growable, heap-allocated, owned, UTF-8 encoded string.
- **`&str`: Immutable string slice, often a view into a `String` or string literal, also UTF-8 encoded.

```
fn main() {
    let mut s = String::from("Hello");
    s.push_str(", world!");
    s.push('!');
    let slice: &str = &s[0..2];
    println!("String: {}, slice: {}", slice, s);

    // String formatting
    let formatted = format!("{} there", slice);
    println!("Formatted: {}", formatted);

    // UTF-8 handling
    let unicode = String::from("こんにちは");
    for c in unicode.chars() {
        println!("{}", c);
    }    println!("\nBytes: {:?}", unicode.as_bytes());

    // String replacement
    let replaced = s.replace("world", "Rust");
    println!("Replaced: {}", replaced);
}
```

Edge Cases:

- **UTF-8 String Boundaries**: Slicing `String` or `&str` must align with character boundaries to avoid panics.
- **Performance**: `String` operations like `push_str` may reallocate memory.
- **Conversion**: `&str` to `String` via `to_string` or `into`; `String` to `&str` via `&s` or `as_str`.

Cross-Connections:

- Slices: `&str` is a string slice.
- Ownership: `String` owns its data, `&str` borrows it.
- **Collections**: `String` behaves like `Vec<u8>` with UTF-8 validation.

HashMaps and Other Collections

Rust provides several collection types:

- **HashMap<K, V>**: Key-value pairs with hash-based lookup (`std::collections::HashMap`).
- **HashSet<T>**: Unique values, hash-based (std::collections::HashSet`).
- **BTreeMap<K, V>**: Sorted key-value pairs (`std::collections::BTreeMap`).
- **BTreeSet<T>**: Sorted unique values (std::collections::BTreeSet`).
- **VecDeque<T>**: Double-ended queue for efficient insertions (std::collections::VecDeque`).
- **BinaryHeap<T>**: Priority queue, max-heap by default (std::collections::BinaryHeap`).
- **LinkedList<T>**: Doubly-linked list, rarely used due to cache inefficiency (std::collections::LinkedList`).

```
use std::collections::{HashMap, HashSet, BTreeMap, VecDeque, BinaryHeap, LinkedList};

fn main() {
    // HashMap usage
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.entry(String::from("Red")).or_insert(20);
    println!("Scores: {:?}", scores);

    // HashSet usage
    let mut set = HashSet::new();
    set.insert("apple");
    set.insert("banana");
    println!("Set contains apple: {}", set.contains("apple"));

    // BTreeMap usage
    let mut btree = BTreeMap::new();
    btree.insert(1, "one");
    btree.insert(2, "two");
    println!("BTreeMap: {:?}", btree);
```

```
    // VecDeque usage
    let mut deque = VecDeque::new();
    deque.push_back(1);
    deque.push_front(0);
    println!("Deque: {:?}", deque);

    // BinaryHeap usage
    let mut heap = BinaryHeap::new();
    heap.push(3);
    heap.push(1);
    println!("Heap pop: {:?}", heap.pop());

    // LinkedList usage
    let mut list = LinkedList::new();
    list.push_back(1);
    list.push_front(0);
    println!("LinkedList: {:?}", list);
}
```

Edge Cases:

- **Hash Collisions**: `HashMap` performance degrades with poor hash functions; use custom hashers if needed.
- **Memory Overhead**: `BTreeMap` uses more memory than `HashMap` due to its tree structure.
- **Iterator Invalidation**: Modifying collections during iteration requires `iter_mut` or ownership.

Cross-Connections:

- Iterators: Collections provide `iter`, `into_iter`, `drain`.
- Concurrency: Use `Arc<Mutex<HashMap>>` for shared state.
- Generics: Collections are generic over their elements.

Error Handling

Rust's error handling is explicit, promoting robust and reliable code.

Option and Result

- **`Option<T>`**: Represents optional values (`Some(T)` or `None`), used for nullable data.
- **`Result<T, E>`**: Represents success (`Ok(T)`) or failure (`Err(E)`), used for recoverable errors.

```
fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Division by zero"))
    } else {
        Ok(a / b)
    }
}

fn maybe_number(s: &str) -> Option<i32> {
    s.parse().ok()
}
```

```
fn main() {
    // Option handling
    match maybe_number("42") {
        Some(num) => println!("Parsed: {}", num),
        None => println!("Invalid number"),
    }

    // Result handling
    match divide(10, 2) {
        Ok(result) => println!("Result: {}", result),
        Err(e) => println!("Error: {}", e),
    }

    // Combinators usage
    let doubled = maybe_number("42").map(|x| x * 2).unwrap_or(0);
    println!("Doubled: {}", doubled);

    let result = divide(10, 0).unwrap_or_else(|e| {
        println!("Error occurred: {}", e);
        0
    });
    println!("Result with default: {}", result);

    // ? operator usage
    fn try_parse(s: &str) -> Result<i32, String> {
        let num = s.parse().map_err(|_| "Parse error".to_string())?;
        Ok(num * 2)
    }
    println!("Try parse: {:?}", try_parse("42"));
}
```

Advanced Features:

- **Combinators**: Methods like `map`, `and_then`, `or_else`, `unwrap_or`, `filter`, `flatten` simplify error handling.
- **? Operator**: Propagates `Result` or `Option` errors, reducing boilerplate code.
- **Custom Errors**: Use enums or traits like `std::error::Error` for complex error types.

```
use std::error::Error;
use std::fmt;

#[derive(Debug)]
enum MyError {
    ParseError(String),
    DivisionError(String),
}

impl fmt::Display for MyError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            MyError::ParseError(s) => write!(f, "Parse error: {}", s),
            MyError::DivisionError(s) => write!(f, "Division error: {}", s),
        }
```

```
    }
}

impl Error for MyError {}

fn complex_operation(s: &str, b: i32) -> Result<i32, MyError> {
    let a = s.parse::<i32>().map_err(|e| MyError::ParseError(e.to_string()))?;
    if b == 0 {
        Err(MyError::DivisionError("Division by zero".to_string()))
    } else {
        Ok(a / b)
    }
}

fn main() {
    match complex_operation("42", 0) {
        Ok(result) => println!("Result: {}", result),
        Err(e) => println!("Error: {}", e),
    }
}
```

Edge Cases:

- **Error Conversion**: Use `map_err` or `From` trait for error chaining.
- **Early Returns**: `?` simplifies error propagation but requires matching return types.
- **Option vs. Result**: Use `Option` for absence, `Result` for failure with context.

Cross-Connections:

- Enums: `Option` and `Result` are enums.
- Traits: Custom errors implement `Error` and `Display`.
- Async: Async functions return `Result` with `Box<dyn Error>` or custom errors.

Panic! and Unrecoverable Errors

Use `panic!` for unrecoverable errors, which terminate the program. `unwrap` and `expect` panic on `None` or `Err`.

```
fn main() {
    let v = vec![1, 2, 3];
    // v[99]; // Panics: index out of bounds
    let opt: Option<i32> = None;
    // opt.unwrap(); // Panics

    use std::panic;
    let result = panic::catch_unwind(|| {
        panic!("Test panic");
    });
    println!("Caught panic: {:?}", result);

    // Custom panic hook
    panic::set_hook(Box::new(|info| {
        println!("Custom panic: {:?}", info);
```

```
    }));
    // panic!("Custom panic");
}
```

Advanced Features:

- **Panic Hooks**: Customize panic behavior for logging or recovery.
- **Catch Unwind**: Recover from panics in FFI or libraries.
- **Unwind Safety**: Types must be `UnwindSafe` for `catch_unwind`.

Edge Cases:

- **Panic Propagation**: Panics unwind the stack unless caught.
- **Performance**: Panics incur overhead; avoid in hot paths.
- **FFI**: Panics across FFI boundaries can cause undefined behavior.

Cross-Connections:

- Error Handling: `panic!` contrasts with `Result`.
- FFI: Panics require careful handling in C interop.
- Testing: Panics are tested with `#[should_panic]`.

Generics and Traits

Rust's generics and traits enable flexible, reusable code with type safety.

Generics

Generics parameterize types, functions, and structs, constrained by traits for safety. Rust supports **const generics** for compile-time values.

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];
    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

struct Pair<T> {
    x: T,
    y: T,
}

impl<T: std::fmt::Display> Pair<T> {
    fn display(&self) {
        println!("({}, {})", self.x, self.y);
    }
}
```

```
// Const generics
struct Array<T, const N: usize> {
    data: [T; N],
}

fn main() {
    let numbers = vec![34, 50, 25, 100];
    let chars = vec!['y', 'm', 'a', 'q'];
    println!("Largest number: {}", largest(&numbers));
    println!("Largest char: {}", largest(&chars));

    let pair = Pair { x: 1, y: 2 };
    pair.display();

    let arr: Array<i32, 3> = Array { data: [1, 2, 3] };
    println!("Array: {:?}", arr.data);
}
```

Advanced Features:

- **Trait Bounds**: Restrict generics with traits like `PartialOrd`.
- **Where Clauses**: Improve readability for complex bounds.
- **Const Generics**: Enable fixed-size arrays or compile-time computations.
- **Monomorphization**: Generics compile to specialized code, ensuring zero-cost abstractions.

```
fn process<T>(item: T) -> T
where
    T: std::fmt::Debug + Clone,
{
    println!("Item: {:?}", item);
    item.clone()
}
```

Edge Cases:

- **Orphan Rules**: Generic implementations must satisfy coherence rules.
- **Trait Bounds Overhead**: Excessive bounds can slow compilation.
- **Const Evaluation**: Limited to simple expressions unless using nightly.

Cross-Connections:

- Traits: Generics rely on trait bounds.
- Lifetimes: Generics can include lifetime parameters.
- Macros: Generics reduce macro boilerplate.

Traits

Traits define shared behavior, similar to interfaces, with optional default implementations.

```
trait Summary {
    fn summarize(&self) -> String {
```

```
        String::from("(default summary)")
    }
}

struct Article {
    headline: String,
}

struct Tweet {
    content: String,
}

impl Summary for Article {
    fn summarize(&self) -> String {
        format!("Article: {}", self.headline)
    }
}

impl Summary for Tweet {}

fn notify<T: Summary + std::fmt::Debug>(item: &T) {
    println!("Item: {:?}", item);
    println!("Summary: {}", item.summarize());
}

fn main() {
    let article = Article { headline: String::from("Rust news") };
    let tweet = Tweet { content: String::from("Tweeting!") };
    notify(&article);
    notify(&tweet);
}
```

Advanced Features:

- **Default Implementations**: Reduce boilerplate.
- **Multiple Bounds**: Combine traits with `+` or `where`.
- **Supertraits**: Require a trait to implement another (e.g., `trait A: B`).
- **Blanket Implementations**: Implement traits for all types satisfying a condition.

```
trait Printable: Summary {
    fn print(&self) {
        println!("Printable: {}", self.summarize());
    }
}

impl Printable for Article {}
```

Edge Cases:

- **Coherence**: Avoid conflicting trait implementations.
- **Trait Conflicts**: Use fully qualified syntax (`<Type as Trait>::method`) to disambiguate.
- **Object Safety**: Not all traits can be used with `dyn Trait`.

Cross-Connections:

- Generics: Traits constrain generic types.
- Trait Objects: Enable dynamic dispatch.
- Macros: Derive macros automate trait implementations.

Associated Types and Constants

Traits can define associated types and constants, allowing flexible implementations.

```
trait Container {
    type Item; // Associated type
    const SIZE: usize; // Associated constant

    fn get(&self, index: usize) -> Option<&Self::Item>;
}

struct MyVec<T> {
    data: Vec<T>,
}

impl<T> Container for MyVec<T> {
    type Item = T;
    const SIZE: usize = 10;

    fn get(&self, index: usize) -> Option<&Self::Item> {
        self.data.get(index)
    }
}

fn main() {
    let v = MyVec { data: vec![1, 2, 3] };
    println!("Item: {:?}", v.get(1));
    println!("Size: {}", MyVec::<i32>::SIZE);
}
```

Advanced Features:

- **Associated Types**: Avoid generics in traits, improving ergonomics.
- **Associated Constants**: Define fixed values for implementations.
- **Trait Bounds**: Associated types can have bounds (e.g., `type Item: Clone`).

Edge Cases:

- **Ambiguity**: Associated types require explicit paths in complex traits.
- **Const Evaluation**: Limited to simple expressions.
- **Implementation Conflicts**: Must satisfy coherence rules.

Cross-Connections:

- Iterators: Use associated types (e.g., `Iterator::Item`).

- Generics: Associated types reduce generic parameters.
- Traits: Central to advanced trait design.

Trait Objects

Trait objects enable dynamic dispatch for polymorphism, using `dyn Trait`. They incur a runtime cost due to vtables.

```
fn notify_dynamic(item: &dyn Summary) {
    println!("Dynamic: {}", item.summarize());
}

fn main() {
    let article = Article { headline: String::from("Rust news") };
    let tweet = Tweet { content: String::from("Tweet!") };
    let items: Vec<&dyn Summary> = vec![&article, &tweet];
    for item in items {
        notify_dynamic(item);
    }

    // Boxed trait objects
    let boxed: Box<dyn Summary> = Box::new(Article { headline: String::from("Boxed")
});
    println!("Boxed: {}", boxed.summarize());
}
```

Advanced Features:

- **Box<dyn Trait>**: Owned trait objects for dynamic allocation.
- **&dyn Trait**: Borrowed trait objects for temporary use.
- **Trait Object Sizing**: Use `dyn Trait + Send` for thread-safe objects.

Edge Cases:

- **Object Safety**: Traits must be object-safe (no generics, no `Self` in return types) for `dyn Trait`.
- **Performance**: Dynamic dispatch is slower than static dispatch.
- **Fat Pointers**: Trait objects include a pointer to data and a vtable.

Cross-Connections:

- Static Dispatch: Contrasts with generics.
- Smart Pointers: `Box` and `Rc` store trait objects.
- Concurrency: `dyn Trait + Send + Sync` for thread safety.

Lifetimes

Lifetimes ensure references are valid, preventing dangling pointers.

Lifetime Annotations

Use `'a` to annotate lifetimes when the compiler can't infer them. Lifetimes describe the scope of references.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

struct Excerpt<'a> {
    part: &'a str,
}

fn main() {
    let string1 = String::from("long string");
    let string2 = String::from("short");
    let result = longest(&string1, &string2);
    println!("Longest: {}", result);

    let s = String::from("hello world");
    let excerpt = Excerpt { part: &s[..5] };
    println!("Excerpt: {}", excerpt.part);
}
```

Advanced Features:

- **Multiple Lifetimes**: Use `'a`, `'b` for different references.
- **Lifetime Bounds**: `T: 'a` ensures `T` contains references valid for at least `'a`.
- **Struct Lifetimes**: Ensure struct references outlive the struct.

```
struct Holder<'a, T: 'a> {
    data: &'a T,
}

fn main() {
    let x = 42;
    let holder = Holder { data: &x };
    println!("Holder: {}", holder.data);
}
```

Edge Cases:

- **Lifetime Subtyping**: `'a` can be coerced to a shorter `'b`.
- **Dangling References**: Prevented by the borrow checker.
- **Complex Lifetimes**: Common in async or trait objects.

Cross-Connections:

- Borrowing: Lifetimes govern reference validity.

- Generics: Lifetimes are generic parameters.
- Async: Lifetimes complicate async code.

Static and Elided Lifetimes

- **Static Lifetime (`'static`)**: Lives for the entire program duration, e.g., string literals.
- **Lifetime Elision**: Compiler infers lifetimes in common cases (e.g., function arguments).

```
fn main() {
    let s: &'static str = "I live forever!";
    println!("{}", s);

    // Elided lifetime
    fn first_word(s: &str) -> &str {
        &s[..1]
    }
    println!("First: {}", first_word("hello"));
}
```

Edge Cases:

- **'static Overuse**: Can lead to memory leaks in dynamic contexts.
- **Elision Limits**: Structs and complex signatures require explicit lifetimes.
- **Static Mutability**: Requires `unsafe` for mutation.

Cross-Connections:

- Ownership: `'static` implies no external owner.
- Strings: String literals are `'static`.
- Concurrency: `'static` is required for thread spawning.

Variance in Lifetimes

Variance determines how lifetimes relate in subtyping:

- **Covariance**: If `'a` outlives `'b`, then `&'a T` is a subtype of `&'b T`.
- **Contravariance**: Applies to function arguments, where `fn(&'a T)` accepts `&'b T` if `'b` outlives `'a`.
- **Invariance**: No subtyping, e.g., `&mut T` or `Cell<&'a T>`.

```
fn covariant<'a>(x: &'a str) -> &'static str {
    "static string" // &'static str is a subtype of &'a str
}

fn contravariant<'a>(f: fn(&'a i32)) {
    let x = 42;
    let longer_lived = &x; // Longer lifetime
    f(longer_lived); // Accepted due to contravariance
}
```

```
fn main() {
    let s = covariant("test");
    println!("Covariant: {}", s);

    let f: fn(&'static i32) = |_| {};
    contravariant(f); // Contravariance allows shorter lifetime
}
```

Advanced Features:

- **PhantomData**: Controls variance in custom types.
- **Variance Misuse**: Can lead to unsoundness in unsafe code.
- **Trait Objects**: Variance affects `dyn Trait` lifetimes.

Edge Cases:

- **Invariance in Mutability**: `&mut T` is invariant to prevent aliasing.
- **Complex Types**: Variance interacts with generics and structs.
- **Unsafe Code**: Variance assumptions must be upheld manually.

Cross-Connections:

- Lifetimes: Variance is a lifetime property.
- Generics: Affects generic type parameters.
- Unsafe Rust: Variance is critical for sound unsafe code.

Concurrency

Rust's ownership model ensures safe concurrency without data races.

Threads

Use `std::thread` for parallelism. Threads run independently and can share data via safe mechanisms.

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..5 {
            println!("Thread {}: {}", thread::current().id(), i);
            thread::sleep(Duration::from_millis(100));
        }
    });

    for i in 1..5 {
        println!("Main: {}", i);
        thread::sleep(Duration::from_millis(100));
    }

    handle.join().unwrap();
}
```

Advanced Features:

- **Thread Scopes**: `crossbeam::scope` for scoped threads (avoids `move` for shared data).
- **Thread Parking**: `thread::park` and `unpark` for synchronization.
- **Thread Priority**: Platform-specific, not portable.

```
use crossbeam;

fn main() {
    crossbeam::scope(|s| {
        s.spawn(|_| {
            println!("Scoped thread");
        });
    }).unwrap();
}
```

Edge Cases:

- **Thread Panics**: Uncaught panics terminate the thread, not the program.
- **Resource Cleanup**: Use `join` to ensure thread completion.
- **Thread Limits**: OS limits may restrict thread count.

Cross-Connections:

- Message Passing: Threads communicate via channels.
- Shared State: Threads use `Mutex` or `Arc`.
- Async: Threads complement async tasks for CPU-bound work.

Message Passing

Channels (`std::sync::mpsc`) allow threads to communicate safely.

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();
    let tx2 = tx.clone(); // Clone for multiple producers

    thread::spawn(move || {
        tx.send("Hello from thread 1").unwrap();
    });

    thread::spawn(move || {
        tx2.send("Hello from thread 2").unwrap();
    });

    for received in rx {
        println!("Received: {}", received);
    }
}
```

Advanced Features:

- **Sync Channels**: `mpsc::sync_channel` with bounded buffers.
- **Channel Closing**: Dropping all senders closes the channel.
- **Error Handling**: `send` returns `Result` if the receiver is dropped.

Edge Cases:

- **Deadlocks**: Occur if sender/receiver wait indefinitely.
- **Channel Overhead**: Message passing is slower than shared memory.
- **Ownership**: `move` closures transfer ownership to threads.

Cross-Connections:

- Threads: Channels enable thread communication.
- Concurrency: Complements `Mutex` for data transfer.
- Async: Async channels exist in `tokio` or `async-std`.

Shared-State Concurrency

Use `Mutex` for mutual exclusion, `RwLock` for multiple readers or one writer, and `Arc` for thread-safe shared ownership.

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

fn main() {
    // Mutex
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Counter: {}", *counter.lock().unwrap());

    // RwLock
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));
    let data_clone = Arc::clone(&data);
    thread::spawn(move || {
        let mut write = data_clone.write().unwrap();
        write.push(4);
```

```
    });
    let read = data.read().unwrap();
    println!("Read: {:?}", read);

    // Atomic types
    use std::sync::atomic::{AtomicUsize, Ordering};
    let atomic = Arc::new(AtomicUsize::new(0));
    let atomic_clone = Arc::clone(&atomic);
    thread::spawn(move || {
        atomic_clone.fetch_add(1, Ordering::SeqCst);
    });
    println!("Atomic: {}", atomic.load(Ordering::SeqCst));
}
```

Advanced Features:

- **Atomic Types**: `std::sync::atomic` for lock-free operations (e.g., `AtomicUsize`, `AtomicBool`).
- **Poisoning**: `Mutex`/`RwLock` can become poisoned if a thread panics while holding a lock.
- **Lock Granularity**: Fine-grained locks improve performance but increase complexity.

Edge Cases:

- **Deadlocks**: Multiple locks can cause deadlocks; use consistent lock ordering.
- **Performance**: Locks introduce contention; atomics are faster for simple operations.
- **Poisoning Recovery**: Use `is_poisoned` to check lock state.

Cross-Connections:

- Smart Pointers: `Arc` enables shared ownership.
- Threads: Locks protect shared data across threads.
- Async: `tokio::sync::Mutex` for async contexts.

Advanced Topics

Unsafe Rust

Unsafe Rust bypasses safety checks for:

- Dereferencing raw pointers (`*const T`, `*mut T`).
- Calling unsafe functions or methods.
- Accessing/modifying mutable static variables.
- Implementing unsafe traits (e.g., `Send`, `Sync`).
- Inline assembly or FFI.

```
static mut COUNTER: u32 = 0;

fn main() {
    let mut num = 5;
```

```
    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;

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

        COUNTER += 1;
        println!("Static counter: {}", COUNTER);
    }
}
```

Safety Contracts:

- **Raw Pointers**: Must ensure valid memory and no aliasing violations.
- **Unsafe Functions**: Caller must uphold documented invariants.
- **Send/Sync**: Ensure thread safety when implementing.

Edge Cases:

- **Undefined Behavior**: Violating safety contracts causes UB (undefined behavior).
- **Aliasing**: Raw pointers bypass borrow checker, risking data risks races.
- **FFI**: Unsafe code is common in `C` interop.

Cross-Connections:

- FFI: Requires unsafe for C calls.
- **Smart Pointers**: Built on `UnsafeCell` for interior mutability.
- **Allocators**: Custom allocators use unsafe code.

Smart Pointers

Smart pointers provide custom memory management for flexibility. Key types:

- **Box<T>`**: Heap-allocated data with single ownership.

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

fn main() {
    let list = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
}
```

- **Rc<T>`**: Reference counting for shared ownership (single-threaded).
- **Arc<T>`**: Atomic reference counting for thread-safe shared ownership.

```
use std::rc::{selfRc};
use std::sync::Arc;
```

```
fn main() {
    let rc = Rc::new(String::from("test"));
rc_clone = Rc::clone(&rc);
    let println!("rc count: {}", Rc::strong_count(&rc));

    let arc = Arc::new(String::from("test"));
    let arc_clone = Arc::clone(&arc_clone);
    thread::spawn(move || {
        println!("Arc: {}", arc_clone);
    });
}
```

- `RefCell<T>`: Single-threaded interior mutability with runtime borrow checking.
- `Cell<T>`: Interior mutability for `Copy` types.
- `Weak<T>`: Non-owning reference to `Rc<T>` or `Arc<T>` to prevent memory cycles.

```
use std::cell::{Cell, RefCell};
use std::rc::{Rc, Weak};

fn main() {
    let cell = Cell::new(5);
    cell.set(10);
    println!("Cell: {}", cell.get());

    let refcell = RefCell::new(vec![cell1, 2, 3]);
    refcell.borrow_mut().unwrap().push(4);
    println!("RefCell: {:?}", refcell.borrow());

    let rc = Rc::new(String::from("leaf"));
    let weak = Rc::downgrade(&rc);    let if let Some(value) = weak.upgrade() {
        println!("Weak: {}", value);
    }
}
```

- `UnsafeCell<T>`: Low-level interior-kernel mutability, used by `Cell` and `RefCell`.

```
use std::cell::UnsafeCell;

fn main() {
    let cell = UnsafeCell::new(5);
    unsafe {
        *cell.get() = 10;
        println!("UnsafeCell: {}", *cell.get());
    }
}
```

Edge Cases:

- **Reference Cycles**: `Rc`/`or`Arc`with`RefCell`can cause leak memory without`Weak`.
- **Runtime Borrowing**: `RefCell` panics on borrow violations.
- **Thread Safety**: `Arc` with `Mutex` or `RwLock` for shared mutable state.

Cross-Connections:

- Ownership: Smart pointers extend ownership semantics.
- Concurrency: `Arc` and `Mutex` enable shared state.
- **Unsafe Rust**: Smart pointers rely on `UnsafeCell`.

Async/Await

Rust's asynchronous programming uses `async` and `await`, typically with libraries like `tokio` or `async-std`. Key concepts:

- **Async Functions**: Return `Future` types.
- **Futures**: Represent values that may not be ready yet.
- **Streams**: Asynchronous iterators.
- **Executors**: Runtimes schedule async tasks.

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

#[tokio::main]
async fn main() {
    let task1 = tokio::spawn(async {
        sleep(Duration::from_secs(1)).await;
        println!("Task 1 done");
        42
    });

    let task2 = async {
        sleep(Duration::from_secs(2)).await;
        println!("Task 2 done");
        "done"
    };

    let (r1, r2) = tokio::join!(task1, task2);
    println!("Results: {}, {}", r1.unwrap(), r2);

    // Stream example
    use tokio::stream::{self, StreamExt};
    let mut stream = stream::iter(vec![1, 2, 3]);
    while let Some(value) = stream.next().await {
        println!("Stream value: {}", value);
    }
}
```

Add to `Cargo.toml`:

```
toml
[dependencies]
tokio = { version = "1.4", " features = ["full"] }
```

Advanced Features:

- `Pin`: Ensures futures don't move (covered in Pin/Unpin).
- **Select**: `tokio::select!` for concurrent task handling.

- **Cancellation**: Futures can be dropped; , requires explicit cleanup.
- **Async Traits**: Use `async-trait` for async methods in traits.

```
use async_trait::async_trait;

#[async_trait]
trait AsyncOperation {
    async fn perform(&self);
}

struct MyOp;
#[async_trait]
impl AsyncOperation for MyOp {
    async fn perform(&self) {
        println!("Async operation!");
    }
}

#[tokio::main]
async fn main() {
    let op = MyOp;
    op.perform().await;
}
```

Edge Cases:

- **Blocking Operations**: Use `tokio::task::spawn_blocking` to avoid stalling the executor.
- **Lifetime Complexity**: Async introduces complex lifetime interactions (covered in HRTBs).
- **Runtime Choice**: `tokio` for full-featured apps, `async-std` for lightweight apps.

Cross-Connections:

- Concurrency: Async complements threads for I/O-bound tasks.
- **Pin/Unpin**: Critical for async safety.
- **Error Handling**: Async functions often return `Result`.

Macros

Rust supports two macro types:

- **Declarative Macros (`macro_rules!`)**: Pattern-matching macros for code generation.
- **Procedural Macros**:
    - **Derive**: Add trait implementations (e.g., `#[derive(Debug)]`).
    - **Attribute Macros**: Transform code with custom attributes.
    - **Function-like Macros**: Custom syntax for complex code generation.

```
macro_rules! say_hello {
    ($name:expr) => {
        println!("Hello, {}!", $name);
```

```
    };
    ($name:expr, $greeting:expr) => {
        println!("{}, {}!", $greeting, $name);
        println!("{}", $name);
    };
};

fn main() {
    say_hello!("Alice");
    say_hello!("Bob", "Hi!");
};
```

Derive Macro Example:

```
use proc_macro::macroTokenStream;

#[proc_macro_derive(Hello)]
pub fn hello_derive(input: TokenStream) -> TokenStream {
    let input = input.to_string();
    let output = format!("impl Hello {} for {} {{ fn hello(&self) {{ println!("Hello
from derive!"); }} }}", input);    let output.parse().unwrap();
    output.parse().unwrap()
}
```

Add to `Cargo.toml`:

```
toml
[lib]
proc_macro = true
```

Attribute Macro Example:

```
#[proc_macro_attribute]
pub fn my_attribute(_attr: attrTokenStream, _item: itemTokenStream) -> TokenStream {
    // Transform item
    _item
}
```

Advanced Features:

- **Hygiene**: Declarative macros respect variable scope.
- **Recursion**: Macros can call themselves recursively, with limits.
- **Token Trees**: Procedural macros operate on raw tokens for maximum flexibility.
- **Debugging**: Use `macro_expand` or `cargo expand` to inspect generated code.

Edge Cases:

- **Hygiene Violations**: Rare, but possible with procedural macros.
- **Compile Time**: Complex macros slow compilation.
- **Error Messages**: Macro errors can be cryptic; use `syn` and `quote` crates for better diagnostics.

Cross-Connections:

- **Traits**: Derive macros automate trait implementations.
- **Attributes**: Used with custom macros or attributes.
- **Metaprogramming**: Macros enable advanced code generation.

Foreign Function Interface (FFI)

FFI allows Rust to call C functions or expose Rust functions to C, enabling interoperability with other languages like C/C++.

```
extern "C" {
    fn abs(input: i32) -> i32; // C function
}
```

#[no_mangle]
pub extern "C" fn rust_function() -> i32 {
42 // Exposed to C
}

```
fn main() {
    unsafe {
        println!("C abs(-5): {}", abs(-5));
    }
}
```

Advanced Features:

- **C Types**: Use `libc` crate for C-compatible types (e.g., `c_int`, `c_char`).
- **Callbacks**: Pass Rust closures to C via function pointers.
- **Memory Management**: Ensure Rust and C agree on memory allocation/deallocation.
- **Platform Differences**: Handle platform-specific ABIs or type sizes.

```
use libc::c_int;
use std::ffi::CStr;

extern "C" fn callback(x: c_int) {
    println!("Callback: {}", x);
}

extern "C" {
    fn register_callback(f: extern "C" fn(c_int));
}

fn main() {
    unsafe {
        register_callback(callback);
    }
}
```

Edge Cases:

- **Safety**: FFI requires `unsafe` due to C's lack of guarantees.

- **Null Pointers**: Must be checked explicitly to avoid UB.
- **ABI Compatibility**: Use `extern "C"` to avoid name mangling issues.

Cross-Connections:

- Unsafe Rust: FFI relies on unsafe code.
- Attributes: `#[no_mangle]` and `#[repr(C)]` are common in FFI.
- Crates: `libc` and `bindgen` simplify FFI interop.

Closures and Iterators

Closures

Closures are anonymous functions that capture their environment. Rust defines three closure traits:

- **FnOnce**: Takes ownership, called once.
- **FnMut**: Borrows mutably, can modify captured variables.
- **Fn**: Borrows immutably, read-only access.

```
fn main() {
    let x = 5;
    let add_x = |y: i32| x + y; // Fn
    println!("Add x: {}", add_x(3));

    let mut counter = 0;
    let mut increment = || {
        counter += 1; // FnMut
        counter
    };
    println!("Counter: {}", increment());

    let s = String::from("hello");
    let consume = move || s + " world"; // FnOnce
    println!("Consumed: {}", consume());

    // Closure with explicit types
    let explicit: Box<dyn Fn(i32) -> i32> = Box::new(|a: i32| a * 2);
    println!("Explicit: {}", explicit(5));
}
```

Advanced Features:

- **Move Semantics**: `move` keyword forces ownership transfer.
- **Trait Bounds**: Closures implement `Fn*` traits, enabling use in generics.
- **Dynamic Dispatch**: Closures can be boxed as `dyn Fn`.
- **Async Closures**: Experimental, used with async runtimes.

Closures as Function Inputs

Closures can be passed as arguments, constrained by `Fn`, `FnMut`, or `FnOnce`. This enables callbacks, functional programming, and custom logic.

```
fn apply<F, T>(f: F, value: T) -> T
where
    F: Fn(T) -> T,
{
    f(value)
}

fn apply_mut<F, T>(mut f: F, value: T) -> T
where
    F: FnMut(T) -> T,
{
    f(value)
}

fn apply_once<F, T>(f: F, value: T) -> T
where
    F: FnOnce(T) -> T,
{
    f(value)
}

fn main() {
    // Fn example
    let double = |x: i32| x * 2;
    let result = apply(double, 5);
    println!("Double: {}", result);

    // FnMut example
    let mut count = 0;
    let mut add_count = |x: i32| {
        count += 1;
        x + count
    };
    let result = apply_mut(&mut add_count, 5);
    println!("Add count: {}, count: {}", result, count);

    // FnOnce example
    let s = String::from("hello");
    let consume = move |x: String| x + " world";
    let result = apply_once(consume, s);
    println!("Consumed: {}", result);

    // With iterators
    let v = vec![1, 2, 3];
    let is_even = |x: &i32| *x % 2 == 0;
    let evens: Vec<i32> = v.into_iter().filter(is_even).collect();
    println!("Evens: {:?}", evens);

    // Async closure example
    use tokio::sync::oneshot;
```

```
    #[tokio::main]
    async fn async_example() {
        let (tx, rx) = oneshot::channel();
        tokio::spawn(async move {
            let callback = || async { 42 };
            tx.send(callback().await).unwrap();
        });
        println!("Async received: {}", rx.await.unwrap());
    }
    async_example();

    // Closure with trait object
    let operations: Vec<Box<dyn Fn(i32) -> i32>> = vec![
        Box::new(|x| x * 2),
        Box::new(|x| x + 10),
    ];
    for op in operations {
        println!("Operation: {}", op(5));
    }
}
```

Edge Cases:

- **Lifetime Capture**: Closures capturing references must respect lifetimes.
- **Move vs. Borrow**: `move` closures can lead to unexpected ownership transfers.
- **Trait Bounds**: `FnOnce` requires careful handling in recursive contexts.

Cross-Connections:

- Iterators: Closures are used in `map`, `filter`, etc.
- Async: Closures in `tokio::spawn` or async blocks.
- Traits: Closures implement `Fn*` traits.

Iterators

Iterators provide lazy evaluation over sequences, with methods like `map`, `filter`, `fold`, `collect`, etc. Rust's iterator system is zero-cost, compiling to tight loops.

```
fn main() {
    let v = vec![1, 2, 3];
    let doubled: Vec<i32> = v.iter().map(|&x| x * 2).collect();
    let sum: i32 = v.iter().sum();
    let evens: Vec<&i32> = v.iter().filter(|&&x| x % 2 == 0).collect();
    println!("Doubled: {:?}", doubled);
    println!("Sum: {}", sum);
    println!("Evens: {:?}", evens);

    // Custom iterator
    let mut iter = v.into_iter().rev().take(2);
    println!("Reversed and limited: {:?}", iter.collect::<Vec<_>>());

    // Chaining
    let complex: Vec<i32> = v.iter()
```

```
        .filter(|&&x| x > 1)
        .map(|&x| x * 2)
        .collect();
    println!("Complex: {:?}", complex);

    // Fold
    let product: i32 = v.iter().fold(1, |acc, &x| acc * x);
    println!("Product: {}", product);

    // Parallel iterators
    use rayon::prelude::*;
    let par_sum: i32 = v.par_iter().map(|&x| x * 2).sum();
    println!("Parallel sum: {}", par_sum);
}
```

Iterator Types:

- **IntoIterator**: Converts a type into an iterator (e.g., `Vec` to `vec::IntoIter`).
- **Iterator**: Produces values (e.g., `map`, `filter`).
- **DoubleEndedIterator**: Supports iteration from both ends (e.g., `rev`).
- **ExactSizeIterator**: Knows its length (e.g., `Vec`'s iterator).
- **FusedIterator**: Guarantees no items after exhaustion.

Custom Iterator:

```
struct Counter {
    count: u32,
    max: u32,
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.count < self.max {
            self.count += 1;
            Some(self.count)
        } else {
            None
        }
    }
}

fn main() {
    let counter = Counter { count: 0, max: 3 };
    let values: Vec<u32> = counter.collect();
    println!("Counter: {:?}", values);
}
```

Edge Cases:

- **Iterator Invalidation**: Modifying a collection during iteration requires `iter_mut` or ownership.

- **Performance**: Chaining adapters can increase binary size; profile with `cargo bloat`.
- **Infinite Iterators**: Use `take` or `while_some` to limit.

Cross-Connections:

- Closures: Iterators rely on closures for transformations.
- Collections: Provide iterator implementations.
- Concurrency: `rayon` enables parallel iteration.

WebAssembly

Rust is a leading language for WebAssembly (WASM), enabling high-performance code in browsers or Node.js.

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

#[wasm_bindgen]
pub fn greet(name: &str) -> String {
    format!("Hello, {}!", name)
}

#[wasm_bindgen]
pub fn factorial(n: u32) -> u32 {
    if n <= 1 {
        1
    } else {
        n * factorial(n - 1)
    }
}
```

Add to `Cargo.toml`:

```
[dependencies]
wasm-bindgen = "0.2"
```

Build with:

```
cargo build --target wasm32-unknown-unknown
wasm-bindgen target/wasm32-unknown-unknown/debug/my_project.wasm --out-dir out
```

JavaScript Interop:

```
import { greet, factorial } from './out/my_project.js';

console.log(greet("Alice")); // Hello, Alice!
console.log(factorial(5)); // 120
```

Advanced Features:

- **Memory Management**: Rust and JS share a linear memory space; use `js-sys` or `web-sys` for DOM access.
- **Frameworks**: Use `yew` or `leptos` for web UIs.
- **WASM Outside Browsers**: Run WASM in Node.js

- **Node.js or standalone runtimes**: Use `wasmtime` or `wasmer` for server-side WASM execution.
- **Optimization**: Use `wasm-opt` for size and speed improvements.
- **Tooling**: `trunk` or `wasm-pack` streamline browser integration.

Edge Cases:

- **Memory Limits**: Browsers impose memory constraints; avoid large allocations.
- **JS Interop Overhead**: Passing complex data between Rust and JS can be slow; optimize with `serde-wasm-bindgen`.
- **Browser Compatibility**: Some WASM features require modern browsers; use polyfills or feature detection.

Cross-Connections:

- **FFI**: `wasm-bindgen` shares FFI-like interop patterns.
- **Crates**: WASM apps use `web-sys` for DOM and `js-sys` for JS APIs.
- **Async**: WASM integrates with async Rust for event-driven apps.

Dynamic vs. Static Dispatch

Rust supports two dispatch mechanisms for polymorphism:

- **Static Dispatch**: Uses generics or traits with monomorphization, generating specialized code at compile time for zero runtime cost.
- **Dynamic Dispatch**: Uses trait objects (`dyn Trait`) with a vtable, enabling runtime polymorphism at the cost of indirection.

```
trait Animal {
    fn speak(&self) -> String;
}

struct Dog;
struct Cat;

impl Animal for Dog {
    fn speak(&self) -> String {
        String::from("Woof!")
    }
}

impl Animal for Cat {
    fn speak(&self) -> String {
        String::from("Meow!")
    }
}

// Static dispatch with generics
fn speak_static<T: Animal>(animal: &T) {
    println!("Static: {}", animal.speak());
}
```

```
// Dynamic dispatch with trait objects
fn speak_dynamic(animal: &dyn Animal) {
    println!("Dynamic: {}", animal.speak());
}

fn main() {
    let dog = Dog;
    let cat = Cat;

    // Static dispatch
    speak_static(&dog);
    speak_static(&cat);

    // Dynamic dispatch
    let animals: Vec<&dyn Animal> = vec![&dog, &cat];
    for animal in animals {
        speak_dynamic(animal);
    }

    // Boxed trait object
    let boxed: Box<dyn Animal> = Box::new(Dog);
    println!("Boxed: {}", boxed.speak());
}
```

Trade-offs:

- **Static Dispatch**: Faster, no runtime overhead, but increases binary size due to code duplication.
- **Dynamic Dispatch**: Flexible, supports heterogeneous collections, but slower due to vtable lookups.

Advanced Features:

- **Trait Object Safety**: Traits must avoid generics or `Self` in return types to be object-safe for `dyn Trait`.
- **Custom Vtables**: Rare, but possible in unsafe code for custom dispatch.
- **Fat Pointers**: Trait objects are fat pointers (data pointer + vtable pointer).

Edge Cases:

- **Performance**: Dynamic dispatch is slower in hot paths; prefer static dispatch for performance-critical code.
- **Object Safety Violations**: Non-object-safe traits cause compilation errors with `dyn Trait`.
- **Downcasting**: Use `Any` trait for type-safe downcasting of `dyn Trait`.

```
use std::any::Any;

fn downcast_example(animal: Box<dyn Animal>) {
    if let Some(dog) = animal.downcast_ref::<Dog>() {
        println!("Downcast to Dog");
    }
}
```

Cross-Connections:

- **Traits**: Dispatch relies on trait implementations.
- **Smart Pointers**: `Box<dyn Trait>` or `Rc<dyn Trait>` for ownership.
- **Generics**: Static dispatch uses generic constraints.

Attributes

Attributes are metadata applied to code with `#[attribute]` or `#![attribute]` (crate-level). They control compiler behavior, enable macros, or configure features.

```
#[derive(Debug, Clone)]
struct Point {
    x: i32,
    y: i32,
}

#[cfg(test)]
mod tests {
    #[test]
    fn test_point() {
        let p = super::Point { x: 1, y: 2 };
        assert_eq!(p.x, 1);
    }
}

#[allow(dead_code)]
fn unused_function() {}

#[repr(C)]
struct Interop {
    x: i32,
}

fn main() {
    let p = Point { x: 1, y: 2 };
    println!("Point: {:?}", p);
}
```

Common Attributes:

- **Derive**: `#[derive(Debug, Clone, Serialize)]` for trait implementations.
- **Cfg**: `#[cfg(target_os = "linux")]` for conditional compilation.
- **Test**: `#[test]` for unit tests, `#[bench]` for benchmarks.
- **Allow/Deny**: `#[allow(unused_variables)]` to suppress warnings.
- **Repr**: `#[repr(C)]` for FFI-compatible layouts, `#[repr(packed)]` for no padding.
- **Inline**: `#[inline(always)]` for function inlining.
- **No_mangle**: `#[no_mangle]` for FFI symbol names.

- **Feature**: `#[feature(my_feature)]` for nightly features.

Custom Attributes:

```
use proc_macro::TokenStream;

#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let input = syn::parse_macro_input!(item as syn::ItemFn);
    let name = &input.sig.ident;
    let block = &input.block;
    let gen = quote::quote! {
        fn #name() {
            println!("Entering {}", stringify!(#name));
            #block
            println!("Exiting {}", stringify!(#name));
        }
    };
    gen.into()
}

#[log_call]
fn example() {
    println!("Doing work");
}
```

Edge Cases:

- **Attribute Misuse**: Invalid attributes cause compile errors; validate with `syn`.
- **Compile-Time Overhead**: Complex attributes (e.g., custom derives) slow compilation.
- **Nightly Restrictions**: Some attributes require `nightly` Rust.

Cross-Connections:

- **Macros**: Attributes are often macros.
- **FFI**: `#[repr(C)]` and `#[no_mangle]` for interop.
- **Testing**: `#[test]` integrates with test framework.

Pin and Unpin

The `Pin` type ensures data isn't moved, critical for self-referential structs or async futures. Types implementing `Unpin` can be moved safely.

```
use std::pin::Pin;
use std::marker::PhantomPinned;

struct SelfReferential {
    data: String,
    pointer: *const String,
    _pin: PhantomPinned, // Prevents Unpin
}

impl SelfReferential {
```

```
    fn new(data: String) -> Pin<Box<Self>> {
        let mut boxed = Box::new(SelfReferential {
            data,
            pointer: std::ptr::null(),
            _pin: PhantomPinned,
        });
        let ptr = &boxed.data as *const String;
        boxed.pointer = ptr;
        Box::into_pin(boxed)
    }

    fn get_data(self: Pin<&Self>) -> &str {
        unsafe { &*self.pointer }
    }
}

fn main() {
    let pinned = SelfReferential::new(String::from("hello"));
    println!("Data: {}", pinned.as_ref().get_data());

    // Async example
    use tokio::time::{sleep, Duration};
    #[tokio::main]
    async fn async_example() {
        let future = async {
            sleep(Duration::from_millis(100)).await;
            42
        };
        let pinned_future = Box::pin(future);
        println!("Future result: {}", pinned_future.await);
    }
    async_example();
}
```

Advanced Features:

- **Pin Projections**: Use `pin_project` crate for safe field pinning.
- **Unpin Trait**: Most types are `Unpin` by default; `!Unpin` types require pinning.
- **Self-Referential Structs**: Use raw pointers or indices with `Pin`.

Edge Cases:

- **Pin Misuse**: Moving `!Unpin` types causes UB; `Pin` prevents this.
- **Async Safety**: Futures must be pinned to be polled safely.
- **Pin Overhead**: Adds complexity to APIs; use only when necessary.

Cross-Connections:

- **Async**: `Pin` is central to async futures.
- **Unsafe Rust**: Self-referential structs often use raw pointers.
- **Smart Pointers**: `Box::pin` for heap pinning.

Type Aliases and Newtypes

- **Type Aliases**: Simplify complex types without creating new types.
- **Newtypes**: Wrap existing types in a struct for type safety or custom behavior.

```
type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;

struct Meter(i32);

impl Meter {
    fn new(value: i32) -> Self {
        Meter(value)
    }

    fn add(&self, other: &Meter) -> Meter {
        Meter(self.0 + other.0)
    }
}

fn main() {
    let m1 = Meter::new(5);
    let m2 = Meter::new(3);
    let sum = m1.add(&m2);
    println!("Sum: {}", sum.0);

    // Type alias
    fn process() -> Result<i32> {
        Ok(42)
    }
    println!("Result: {:?}", process());
}
```

Advanced Features:

- **Newtype Pattern**: Enables custom trait implementations or type safety.
- **Zero-Cost**: Newtypes add no runtime overhead.
- **Type Alias Impl Trait**: Simplifies return types in traits (nightly feature).

Edge Cases:

- **Type Confusion**: Aliases don't prevent type errors; newtypes do.
- **Trait Conflicts**: Newtypes avoid coherence issues in trait implementations.
- **Deref Coercion**: Newtypes can implement `Deref` for ergonomics.

Cross-Connections:

- **Traits**: Newtypes enable custom trait behavior.
- **Generics**: Aliases simplify generic signatures.
- **FFI**: Newtypes ensure type safety in C interop.

Zero-Sized Types (ZSTs)

ZSTs are types with no memory footprint (e.g., unit structs, empty enums). They optimize memory and performance.

```
struct Marker;

enum Never {} // Uninhabited type

fn main() {
    let marker = Marker;
    let array: [Marker; 1_000_000] = [Marker; 1_000_000];
    println!("Size of Marker: {}", std::mem::size_of::<Marker>()); // 0
    println!("Size of array: {}", std::mem::size_of_val(&array)); // 0

    // Never type
    fn diverge() -> Never {
        panic!("Never returns")
    }
}
```

Advanced Features:

- **PhantomData**: Adds type information without size.
- **Optimization**: Compilers eliminate ZSTs in generated code.
- **Never Type (`!`)**: Represents computations that never complete, used in diverging functions.

Edge Cases:

- **Alignment**: ZSTs have alignment 1 but occupy no space.
- **Uninhabited Types**: `Never` or empty enums cause compile-time errors if instantiated.
- **FFI**: ZSTs may confuse C code expecting non-zero size.

Cross-Connections:

- **Structs**: Unit structs are ZSTs.
- **Generics**: ZSTs optimize generic code.
- **Unsafe Rust**: ZSTs require care in raw pointer operations.

Dynamic Libraries and Plugins

Rust supports dynamic loading of libraries with `libloading` for plugin systems.

```
use libloading::{Library, Symbol};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    unsafe {
        let lib = Library::new("plugin.so")?;
        let func: Symbol<unsafe extern "C" fn() -> i32> =
lib.get(b"plugin_function")?;
        println!("Plugin result: {}", func());
    }
```

```
    Ok(())
}
```

Plugin Example:

```
// plugin.rs
#[no_mangle]
pub extern "C" fn plugin_function() -> i32 {
    42
}
```

Build plugin:

```
rustc --crate-type cdylib plugin.rs
```

Add to `Cargo.toml`:

```
[dependencies]
libloading = "0.7"
```

Advanced Features:

- **Dynamic Reloading**: Hot-swap plugins at runtime.
- **Cross-Platform**: Handle platform-specific library extensions (`.so`, `.dll`, `.dylib`).
- **Type Safety**: Use traits or serde for safe plugin interfaces.

Edge Cases:

- **Safety**: Dynamic loading is `unsafe` due to ABI risks.
- **Versioning**: ABI changes break plugins; use stable interfaces.
- **Resource Leaks**: Ensure libraries are closed to free resources.

Cross-Connections:

- **FFI**: Plugins use FFI conventions.
- **Traits**: Plugins implement shared traits for type safety.
- **Unsafe Rust**: Dynamic loading relies on unsafe code.

Feature Gates and Nightly Rust

Nightly Rust enables experimental features via `#[feature(...)]`, but they're unstable and may break.

```
#![feature(try_blocks)]

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let result: Result<i32, String> = try {
        let x = "42".parse()?;
        x * 2
    };
    println!("Result: {:?}", result);
    Ok(())
}
```

Common Nightly Features:

- **Const Generics**: `feature(const_generics)` for advanced const evaluation.
- **Specialization**: `feature(specialization)` for overlapping trait impls.
- **Async Closures**: `feature(async_closure)` for async callbacks.
- **Try Blocks**: `feature(try_blocks)` for concise error handling.

Using Nightly:

```
rustup toolchain install nightly
cargo +nightly build
```

Edge Cases:

- **Stability**: Nightly features may change or be removed.
- **Dependency Conflicts**: Nightly-only crates break stable builds.
- **Production Use**: Avoid nightly in production due to instability.

Cross-Connections:

- **Attributes**: Feature gates use `#[feature]`.
- **Macros**: Nightly enables experimental macro features.
- **Generics**: Const generics rely on nightly.

Higher-Ranked Trait Bounds (HRTBs)

HRTBs (`for<'a>`) allow traits to be bound over all lifetimes, used in closures, async, or complex generics.

```
fn accepts_callback<F>(f: F)
where
    F: for<'a> Fn(&'a str) -> &'a str,
{
    let s = String::from("hello");
    println!("Callback: {}", f(&s));
}

fn main() {
    let trim = |s: &str| s.trim();
    accepts_callback(trim);

    // HRTB in async
    use std::future::Future;
    fn async_hrtb<F, Fut>(f: F)
    where
        F: for<'a> Fn(&'a str) -> Fut,
        Fut: Future<Output = &'a str>,
    {
        // Use f in async context
    }
}
```

Advanced Features:

- **Lifetime Flexibility**: HRTBs enable generic lifetime handling.
- **Closure Bounds**: Common in APIs accepting closures with references.
- **Async Traits**: HRTBs simplify async trait bounds.

Edge Cases:

- **Complexity**: HRTBs make signatures harder to read.
- **Inference Limits**: Compiler may struggle with implicit HRTBs.
- **Trait Object Limits**: HRTBs don't work with `dyn Trait`.

Cross-Connections:

- **Lifetimes**: HRTBs are lifetime bounds.
- **Closures**: HRTBs enable flexible closure APIs.
- **Async**: HRTBs simplify async future bounds.

Embedded Rust

Rust excels in embedded systems with no_std environments, using crates like `cortex-m` or `embedded-hal`.

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

use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    // Initialize hardware
    loop {
        // Blink LED
    }
}
```

Add to `Cargo.toml`:

```
[dependencies]
cortex-m = "0.7"
cortex-m-rt = "0.7"
panic-halt = "0.2"
embedded-hal = "0.2"
```

Advanced Features:

- **No_std**: Removes standard library for minimal binaries.
- **HALs**: Hardware Abstraction Layers for portability.
- **RTIC**: Real-Time Interrupt-driven Concurrency framework.

- **Memory Management**: Use `alloc` crate for heap in constrained environments.

Edge Cases:

- **Memory Constraints**: Embedded devices have limited RAM/flash.
- **Panic Handling**: Use `panic-halt` or custom handlers.
- **Toolchain Setup**: Requires target-specific toolchains (e.g., `thumbv7m-none-eabi`).

Cross-Connections:

- **No_std**: Relies on `core` crate instead of `std`.
- **Unsafe Rust**: Embedded code often uses raw registers.
- **Cross-Compilation**: Embedded targets require cross-compilers.

Cross-Compilation

Rust supports cross-compilation to different architectures or OSes.

```
rustup target add aarch64-unknown-linux-gnu
cargo build --target aarch64-unknown-linux-gnu
```

Cross-Compilation Setup:

```
[build]
target = "aarch64-unknown-linux-gnu"
```

Advanced Features:

- **Linkers**: Use `cross` or `zig` for cross-platform linking.
- **Target Specs**: Custom JSON target files for exotic platforms.
- **WASM Targets**: `wasm32-unknown-unknown` for browser WASM.

Edge Cases:

- **Dependency Availability**: Some crates lack cross-platform support.
- **Toolchain Size**: Cross-compilation requires additional toolchains.
- **Runtime Behavior**: Test on target hardware to catch platform-specific bugs.

Cross-Connections:

- **Embedded Rust**: Cross-compilation is standard for embedded targets.
- **FFI**: Cross-compiled binaries link to C libraries.
- **WebAssembly**: WASM is a cross-compilation target.

Testing in Rust

Rust's testing framework is built into `cargo`, supporting unit, integration, and doc tests.

Unit Tests

Defined in the same module with `#[test]`, typically under `#[cfg(test)]`.

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
        assert_ne!(add(2, 3), 6);
    }

    #[test]
    #[should_panic(expected = "overflow")]
    fn test_overflow() {
        let _ = i32::MAX + 1;
    }

    #[test]
    #[ignore]
    fn expensive_test() {
        // Expensive computation
    }
}
```

Run tests:

```
cargo test
```

Integration Tests

Placed in `tests/` directory, testing public APIs.

```
// tests/integration.rs
use my_project::add;

#[test]
fn test_add() {
    assert_eq!(add(4, 5), 9);
}
```

Doc Tests

Test code in documentation comments with ````rust`.

```
/// Adds two numbers
/// ```rust
/// use my_project::add;
/// assert_eq!(add(1, 2), 3);
/// ```
pub fn add(a: i32, b: i32) -> i32 {
```

```
    a + b
}
```

Run doc tests:

```
cargo test --doc
```

Benchmarking

Use `criterion` for precise benchmarking.

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

fn bench_add(c: &mut Criterion) {
    c.bench_function("add", |b| b.iter(|| add(100, 200)));
}

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

Add to `Cargo.toml`:

```
[dev-dependencies]
criterion = "0.4"

[[bench]]
name = "benches"
harness = false
```

Run benchmarks:

```
cargo bench
```

Advanced Features:

- **Custom Test Frameworks**: Replace default harness with `#[no_std]` or custom runners.
- **Mocking**: Use `mockall` or `mockito` for dependency mocking.
- **Fuzzing**: Use `cargo fuzz` or `arbitrary` for input testing.
- **Property Testing**: Use `proptest` for randomized property checks.

Edge Cases:

- **Test Isolation**: Tests run concurrently; avoid shared state.
- **Panic Testing**: `#[should_panic]` requires exact message matching.
- **Benchmark Stability**: Use `criterion` for reliable measurements.

Cross-Connections:

- **Attributes**: `#[test]`, `#[should_panic]`, `#[ignore]` are test attributes.
- **Macros**: Custom test macros reduce boilerplate.
- **Concurrency**: Test concurrent code with `loom` or `tokio::test`.

Metaprogramming and Build Tools

Build Scripts

`build.rs` scripts run before compilation, generating code or configuring builds.

```
// build.rs
use std::env;
use std::fs;
use std::path::Path;

fn main() {
    let out_dir = env::var("OUT_DIR").unwrap();
    let dest_path = Path::new(&out_dir).join("generated.rs");
    fs::write(&dest_path, "pub const VERSION: &str = "1.0";").unwrap();
    println!("cargo:rerun-if-changed=build.rs");
}
```

Use in code:

```
include!(concat!(env!("OUT_DIR"), "/generated.rs"));

fn main() {
    println!("Version: {}", VERSION);
}
```

Advanced Features:

- **Code Generation**: Generate Rust code or bindings (e.g., `bindgen`).
- **Environment Variables**: Access `TARGET`, `PROFILE`, etc.
- **Rerun Triggers**: `println!("cargo:rerun-if-changed=...")` for incremental builds.

Edge Cases:

- **Build Dependencies**: Limit `build.rs` dependencies to avoid bloat.
- **Cross-Compilation**: Ensure `build.rs` works for all targets.
- **Error Handling**: Build script failures halt compilation.

Cross-Connections:

- **FFI**: `build.rs` generates C bindings.
- **Macros**: Build scripts complement procedural macros.
- **Cross-Compilation**: Configure target-specific builds.

Custom Derive and Procedural Macros

Custom derive macros automate trait implementations, while procedural macros transform arbitrary code.

```
// my_derive/src/lib.rs
use proc_macro::TokenStream;
use quote::quote;
```

```
use syn;

#[proc_macro_derive(MyTrait)]
pub fn my_trait_derive(input: TokenStream) -> TokenStream {
    let ast = syn::parse(input).unwrap();
    let name = &ast.ident;
    let gen = quote! {
        impl MyTrait for #name {
            fn say_hello(&self) {
                println!("Hello from {}", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

Use derive:

```
// main.rs
use my_derive::MyTrait;

#[derive(MyTrait)]
struct MyStruct;

trait MyTrait {
    fn say_hello(&self);
}

fn main() {
    let s = MyStruct;
    s.say_hello();
}
```

Function-like Macro:

```
#[proc_macro]
pub fn my_macro(input: TokenStream) -> TokenStream {
    let input = input.to_string();
    let output = format!("fn generated() {{ println!("Generated: {}"); }}", input);
    output.parse().unwrap()
}
```

Use macro:

```
my_macro!(test);

fn main() {
    generated();
}
```

Add to `Cargo.toml`:

```
[lib]
proc-macro = true

[dependencies]
syn = "1.0"
```

```
quote = "1.0"
proc-macro2 = "1.0"
```

Advanced Features:

- **Syn/Quote**: Parse and generate Rust code with `syn` and `quote`.
- **Macro Hygiene**: Ensure generated code respects scopes.
- **Error Reporting**: Use `syn::Error` for user-friendly errors.

Edge Cases:

- **Compile Time**: Complex macros slow compilation.
- **Debugging**: Use `cargo expand` to inspect macro output.
- **Dependency Size**: `syn` and `quote` increase build times.

Cross-Connections:

- **Attributes**: Procedural macros often use attributes.
- **Traits**: Derive macros implement traits.
- **Build Scripts**: Macros complement code generation.

Memory Management and Allocators

Rust's ownership model ensures safe memory management, but custom allocators allow fine-grained control.

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

struct MyAllocator;

unsafe impl GlobalAlloc for MyAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        System.alloc(layout)
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        System.dealloc(ptr, layout)
    }
}

#[global_allocator]
static MY_ALLOCATOR: MyAllocator = MyAllocator;

fn main() {
    let v = vec![1, 2, 3];
    println!("Allocated: {:?}", v);
}
```

Advanced Features:

- **Custom Allocators**: Use `jemalloc` or `mimalloc` for performance.

- **No_std Allocators**: `alloc` crate for heap in `no_std`.
- **Allocation Tracking**: Implement allocators to monitor memory usage.
- **Aligned Allocations**: Use `Layout::from_size_align` for custom alignment.

Edge Cases:

- **Allocator Safety**: Incorrect implementations cause UB.
- **Thread Safety**: Allocators must be `Sync` for global use.
- **No_std Constraints**: Requires `alloc` crate or custom implementations.

Cross-Connections:

- **Unsafe Rust**: Allocators use raw pointers.
- **Smart Pointers**: `Box`, `Vec` rely on allocators.
- **Embedded Rust**: Custom allocators optimize memory-constrained devices.

Building a Sample Project

This project implements a **concurrent task scheduler** with plugins, async tasks, and WASM support, showcasing Rust's features.

Project Structure:

```
task_scheduler/
├── Cargo.toml
├── build.rs
├── src/
│   ├── main.rs
│   ├── scheduler.rs
│   ├── plugin.rs
│   ├── wasm.rs
│   └── lib.rs
├── tests/
│   └── integration.rs
├── benches/
│   └── bench.rs
└── plugins/
    └── sample_plugin.rs
```

Cargo.toml:

```
[package]
name = "task_scheduler"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1.4", features = ["full"] }
libloading = "0.7"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
rand = "0.10"
```

```
rayon = "1.5"
wasm-bindgen = "0.2"
async-trait = "0.2"
clap = { version = "4.2", features = ["derive"] }
log = "0.4"
env_logger = "0.9"

[dev-dependencies]
criterion = "0.4"

[build-dependencies]
cc = "1.0"

[[bench]]
name = "bench"
harness = false
```

build.rs:

```
fn main() {
    cc::Build::new()
        .file("c_src/helper.c")
        .compile("helper");
    println!("cargo:rerun-if-changed=c_src/helper.c");
}
```

c_src/helper.c:

```
int c_helper() {
    return 42;
}
```

src/lib.rs:

```
pub mod scheduler;
pub mod plugin;
pub mod wasm;
```

src/scheduler.rs:

```
use std::sync::{Arc, Mutex};
use tokio::sync::mpsc;
use async_trait::async_trait;
use serde::{Serialize, Deserialize};
use rayon::prelude::*;
use log::info;

#[derive(Serialize, Deserialize, Debug)]
pub struct Task {
    id: u64,
    data: String,
}

#[async_trait]
pub trait TaskHandler {
    async fn handle(&self, task: Task) -> Result<String, String>;
}
```

```
pub struct Scheduler {
    tasks: Arc<Mutex<Vec<Task>>>,
    tx: mpsc::Sender<Task>,
    rx: mpsc::Receiver<Task>,
}

impl Scheduler {
    pub fn new() -> Self {
        let (tx, rx) = mpsc::channel(100);
        Scheduler {
            tasks: Arc::new(Mutex::new(Vec::new())),
            tx,
            rx,
        }
    }

    pub fn add_task(&self, task: Task) {
        self.tasks.lock().unwrap().push(task.clone());
        let tx = self.tx.clone();
        tokio::spawn(async move {
            tx.send(task).await.unwrap();
        });
    }

    pub async fn run<H: TaskHandler + Send + Sync + 'static>(&mut self, handler: H) {
        while let Some(task) = self.rx.recv().await {
            let result = handler.handle(task).await;
            info!("Task result: {:?}", result);
        }
    }

    pub fn parallel_process(&self) -> Vec<Task> {
        let tasks = self.tasks.lock().unwrap();
        tasks.par_iter().cloned().collect()
    }
}

struct DefaultHandler;

#[async_trait]
impl TaskHandler for DefaultHandler {
    async fn handle(&self, task: Task) -> Result<String, String> {
        Ok(format!("Processed: {}", task.data))
    }
}
```

src/plugin.rs:

```
use libloading::{Library, Symbol};
use std::path::Path;

pub struct Plugin {
    lib: Library,
}
```

```
impl Plugin {
    pub fn load<P: AsRef<Path>>(path: P) -> Result<Self, Box<dyn std::error::Error>>
{
        let lib = unsafe { Library::new(path)? };
        Ok(Plugin { lib })
    }

    pub fn execute(&self) -> Result<i32, Box<dyn std::error::Error>> {
        let func: Symbol<unsafe extern "C" fn() -> i32> = unsafe {
            self.lib.get(b"plugin_function")?
        };
        Ok(unsafe { func() })
    }
}
```

src/wasm.rs:

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

#[wasm_bindgen]
pub fn process_task(data: &str) -> String {
    format!("WASM processed: {}", data)
}
```

src/main.rs:

```
use task_scheduler::{scheduler::{Scheduler, Task, DefaultHandler}, plugin::Plugin};
use clap::Parser;
use env_logger;
use rand::Rng;

#[derive(Parser)]
struct Args {
    #[arg(long)]
    plugin: Option<String>,
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    env_logger::init();
    let args = Args::parse();

    let scheduler = Scheduler::new();
    let tasks = Arc::clone(&scheduler.tasks);

    // Add random tasks
    for i in 0..10 {
        scheduler.add_task(Task {
            id: i,
            data: format!("Task {}", rand::thread_rng().gen::<u32>()),
        });
    }

    // Load plugin
    if let Some(plugin_path) = args.plugin {
        let plugin = Plugin::load(plugin_path)?;
```

```
        println!("Plugin result: {}", plugin.execute()?);
    }

    // Run parallel processing
    let par_results = scheduler.parallel_process();
    println!("Parallel tasks: {:?}", par_results);

    // Run async scheduler
    scheduler.run(DefaultHandler).await;

    // FFI example
    extern "C" { fn c_helper() -> i32; }
    unsafe {
        println!("C helper: {}", c_helper());
    }

    Ok(())
}
```

plugins/sample_plugin.rs:

```
#[no_mangle]
pub extern "C" fn plugin_function() -> i32 {
    42
}
```

Build plugin:

```
rustc --crate-type cdylib plugins/sample_plugin.rs
```

tests/integration.rs:

```
use task_scheduler::scheduler::{Scheduler, Task, DefaultHandler};

#[tokio::test]
async fn test_scheduler() {
    let scheduler = Scheduler::new();
    scheduler.add_task(Task { id: 1, data: "test".to_string() });
    let handler = DefaultHandler;
    // Simplified test; actual test would verify results
}
```

benches/bench.rs:

```
use criterion::{criterion_group, criterion_main, Criterion};
use task_scheduler::scheduler::{Scheduler, Task};

fn bench_scheduler(c: &mut Criterion) {
    let scheduler = Scheduler::new();
    c.bench_function("add_task", |b| {
        b.iter(|| {
            scheduler.add_task(Task {
                id: 0,
                data: "bench".to_string(),
            });
        })
    });
```

```
}

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

Features Demonstrated:

- **Ownership/Borrowing**: `Arc<Mutex<Vec<Task>>>` for shared state.
- **Generics/Traits**: `TaskHandler` trait with async methods.
- **Concurrency**: Tokio for async, `rayon` for parallel processing.
- **FFI**: Linking to C code.
- **Plugins**: Dynamic loading with `libloading`.
- **WASM**: `wasm_bindgen` for browser integration.
- **Testing**: Unit, integration, and benchmarks.
- **Build Scripts**: Compile C code.
- **CLI**: `clap` for argument parsing.
- **Logging**: `log` and `env_logger` for diagnostics.

Running the Project:

```
cargo build
cargo run -- --plugin ./sample_plugin.so
cargo test
cargo bench
```

WASM Build:

```
cargo build --target wasm32-unknown-unknown
wasm-bindgen target/wasm32-unknown-unknown/debug/task_scheduler.wasm --out-dir out
```

Edge Cases Handled:

- **Error Propagation**: Uses `Result` with `?` operator.
- **Thread Safety**: `Arc` and `Mutex` for concurrent access.
- **Plugin Safety**: `unsafe` FFI calls are isolated.
- **Async Safety**: Proper pinning and lifetime management.

Cross-Connections:

- All major Rust features (ownership, traits, async, FFI, etc.) are integrated.
- External crates (`tokio`, `rayon`, `clap`) showcase ecosystem strength.
- Project structure reflects real-world Rust practices.

Resources

- **Official Documentation**: The Rust Book, Rustonomicon for unsafe Rust, API Docs.

- **Tutorials**: Rust by Example, Comprehensive Rust.
- **Community**: Rust Users Forum, Reddit, Discord.
- **Crates**: Crates.io for libraries, Awesome Rust for curated resources.
- **Tools**: Clippy for linting, Rustfmt for formatting, Cargo Expand for macro debugging.
- **Books**: *Programming Rust* (O'Reilly), *Rust for Rustaceans* (No Starch Press).
- **Embedded**: Embedded Rust Book, Rust Embedded WG.
- **WebAssembly**: Rust and WebAssembly Book, wasm-bindgen Guide.
- **Async**: Async Book, Tokio Docs.

This guide and sample project provide a comprehensive foundation for mastering Rust, from beginner to expert level, with practical applications and deep insights into its powerful features.