Rust Syntax, Keywords, and Patterns Reference

A comprehensive reference for Rust's syntactic elements, reserved keywords, and idiomatic patterns.

Part 1: Keywords

Rust reserves keywords for current and future use. They cannot be used as identifiers (variable names, function names, etc.).

Currently Used Keywords

Keyword	Purpose
`as`	Type casting or renaming imports
`async`	Declare an asynchronous function or block
`await`	Suspend execution until an async result is ready
`break`	Exit a loop immediately
`const`	Define compile-time constants or constant pointers
`continue`	Skip to the next loop iteration
`crate`	Refer to the current crate root
`dyn`	Dynamic dispatch for trait objects
`else`	Fallback branch for `if` expressions

`enum`	Define an enumeration type
`extern`	Link external code or specify ABI
`false`	Boolean literal
`fn`	Define a function
`for`	Loop over an iterator
`if`	Conditional branching
`impl`	Implement inherent or trait methods
`in`	Part of `for` loop syntax
`let`	Bind a value to a variable
`loop`	Infinite loop construct
`match`	Pattern matching expression
`mod`	Define a module
`move`	Force a closure to take ownership
`mut`	Declare mutable binding or reference
`pub`	Make an item public
`ref`	Bind by reference in patterns
`return`	Return from a function
`Self`	Type alias for the implementing type
`self`	Current module or method receiver

`static`	Global variable or `'static` lifetime
`struct`	Define a structure type
`super`	Parent module
`trait`	Define a trait
`true`	Boolean literal
`type`	Define a type alias
`unsafe`	Mark code that bypasses safety checks
`use`	Bring paths into scope
`where`	Specify trait bounds on generics
`while`	Conditional loop

Reserved for Future Use

These keywords are reserved but have no current functionality:

```
abstract   become   box   do   final   macro   override
priv   try   typeof   unsized   virtual   yield
```

Part 2: Syntax Elements

Variable Bindings

```
let x = 5;              // Immutable binding
let mut y = 10;         // Mutable binding
let z: i32 = 15;        // Explicit type annotation
let (a, b) = (1, 2);    // Destructuring bind
let _ = unused_value(); // Discard with underscore
```

Type Annotations

```rust
let integer: i32 = 42;
let float: f64 = 3.14;
let boolean: bool = true;
let character: char = 'R';
let string_slice: &str = "hello";
let owned_string: String = String::from("world");
let array: [i32; 3] = [1, 2, 3];
let tuple: (i32, f64, char) = (42, 3.14, 'x');
let unit: () = ();
```

Functions

```rust
// Basic function
fn add(a: i32, b: i32) -> i32 {
    a + b  // Implicit return (no semicolon)
}

// No return value (returns unit type)
fn print_value(x: i32) {
    println!("{}", x);
}

// Early return
fn early_return(x: i32) -> i32 {
    if x < 0 {
        return 0;  // Explicit return
    }
    x * 2
}

// Generic function
fn largest<T: PartialOrd>(a: T, b: T) -> T {
    if a > b { a } else { b }
}

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

Control Flow

```
// if-else expression (returns a value)
let result = if condition { value_a } else { value_b };

// if-else chain
if x < 0 {
    println!("negative");
} else if x == 0 {
    println!("zero");
} else {
    println!("positive");
}

// if-let for single pattern matching
if let Some(value) = optional {
    println!("Got: {}", value);
}

// while loop
while condition {
    // body
}

// while-let
while let Some(item) = iterator.next() {
    process(item);
}

// for loop (iterates over anything implementing IntoIterator)
for item in collection {
    process(item);
}

// for with range
for i in 0..10 {      // 0 to 9
    println!("{}", i);
}

for i in 0..=10 {     // 0 to 10 inclusive
    println!("{}", i);
}

// loop (infinite, must break)
loop {
    if done {
        break;
    }
}

// loop with return value
let result = loop {
    if condition {
        break 42;  // Returns 42
    }
};
```

```
// labeled loops'outer: for i in 0..10 {    'inner: for j in 0..10 {        if condition {            break 'outer;  // Breaks outer loop        }    }}
```

Match Expressions

```
let x = 5;

match x {
    1 => println!("one"),
    2 | 3 => println!("two or three"),  // Multiple patterns
    4..=9 => println!("four through nine"),  // Range pattern
    n if n > 100 => println!("big: {}", n),  // Guard
    _ => println!("something else"),  // Wildcard
}

// Match with destructuring
match point {
    (0, 0) => println!("origin"),
    (x, 0) => println!("on x-axis at {}", x),
    (0, y) => println!("on y-axis at {}", y),
    (x, y) => println!("at ({}, {})", x, y),
}

// Match returns a value
let description = match value {
    0 => "zero",
    1 => "one",
    _ => "many",
};
```

Structs

```
// Named struct
struct Point {
    x: f64,
    y: f64,
}

// Tuple struct
struct Color(u8, u8, u8);

// Unit struct
struct Marker;

// Instantiation
let p = Point { x: 1.0, y: 2.0 };
let c = Color(255, 128, 0);
let m = Marker;

// Field shorthand (when variable name matches field name)
let x = 1.0;
let y = 2.0;
let p = Point { x, y };

// Struct update syntax
let p2 = Point { x: 3.0, ..p };  // y comes from p
```

Enums

```rust
// Simple enum
enum Direction {
    North,
    South,
    East,
    West,
}

// Enum with data
enum Message {
    Quit,                       // No data
    Move { x: i32, y: i32 },    // Named fields
    Write(String),              // Tuple variant
    ChangeColor(u8, u8, u8),    // Multiple values
}

// Usage
let msg = Message::Move { x: 10, y: 20 };

match msg {
    Message::Quit => println!("quit"),
    Message::Move { x, y } => println!("move to ({}, {})", x, y),
    Message::Write(text) => println!("write: {}", text),
    Message::ChangeColor(r, g, b) => println!("color: {},{},{}", r, g, b),
}
```

Impl Blocks

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    // Associated function (no self) - called with ::
    fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }

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

    // Mutable method
    fn scale(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }

    // Consuming method (takes ownership)
    fn into_square(self) -> Rectangle {
        let side = self.width.max(self.height);
        Rectangle { width: side, height: side }
    }
}

// Usage
let r = Rectangle::new(10, 20);  // Associated function
let area = r.area();             // Method
```

Traits

```rust
// Define a trait
trait Summary {
    // Required method
    fn summarize(&self) -> String;

    // Default implementation
    fn preview(&self) -> String {
        format!("Read more: {}", self.summarize())
    }
}

// Implement for a type
impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{} by {}", self.title, self.author)
    }
}

// Trait bounds
fn notify<T: Summary>(item: &T) {
    println!("News: {}", item.summarize());
}

// Multiple bounds
fn process<T: Summary + Display>(item: &T) { /* ... */ }

// Where clause (cleaner for complex bounds)
fn complex<T, U>(t: &T, u: &U) -> i32
where
    T: Summary + Clone,
    U: Debug + PartialOrd,
{
    // ...
}

// impl Trait syntax
fn make_summary() -> impl Summary {
    Article { /* ... */ }
}
```

Generics

```rust
// Generic struct
struct Container<T> {
    value: T,
}

// Generic enum
enum Option<T> {
    Some(T),
    None,
}

// Multiple type parameters
struct Pair<T, U> {
    first: T,
    second: U,
}

// Generic impl
impl<T> Container<T> {
    fn new(value: T) -> Self {
        Self { value }
    }
}

// Impl for specific type
impl Container<i32> {
    fn is_positive(&self) -> bool {
        self.value > 0
    }
}
```

Lifetimes

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

// Struct with lifetime
struct Excerpt<'a> {
    part: &'a str,
}

// Multiple lifetimes
fn complex<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    x
}

// Static lifetime
let s: &'static str = "I live forever";
```

Closures

```
// Basic closure
let add_one = |x| x + 1;

// With type annotations
let add_one: fn(i32) -> i32 = |x: i32| -> i32 { x + 1 };

// Multi-line closure
let process = |x| {
    let y = x * 2;
    y + 1
};

// Capturing environment
let factor = 2;
let multiply = |x| x * factor;  // Borrows factor

// move closure (takes ownership)
let data = vec![1, 2, 3];
let closure = move || {
    println!("{:?}", data);  // Owns data now
};
```

Error Handling

```rust
// Result type
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("division by zero"))
    } else {
        Ok(a / b)
    }
}

// ? operator (early return on error)
fn process() -> Result<i32, Error> {
    let file = File::open("data.txt")?;  // Returns Err if fails
    let data = read_data(&file)?;
    Ok(data.len() as i32)
}

// unwrap and expect
let value = result.unwrap();            // Panics on Err
let value = result.expect("failed!");  // Panics with message

// Option handling
let maybe = Some(5);
let value = maybe.unwrap_or(0);         // Default on None
let value = maybe.unwrap_or_else(|| compute_default());
```

References and Borrowing

```rust
let s = String::from("hello");

let r1 = &s;          // Immutable borrow
let r2 = &s;          // Multiple immutable borrows OK

let r3 = &mut s;      // Mutable borrow (only one allowed)

// Dereferencing
let x = 5;
let r = &x;
assert_eq!(*r, 5);   // Dereference with *
```

Smart Pointers

```rust
use std::rc::Rc;
use std::cell::RefCell;

// Box: heap allocation
let boxed: Box<i32> = Box::new(42);

// Rc: reference counting
let shared: Rc<String> = Rc::new(String::from("shared"));
let clone = Rc::clone(&shared);

// RefCell: interior mutability
let cell: RefCell<i32> = RefCell::new(5);
*cell.borrow_mut() += 1;
```

Attributes

```rust
// Function attributes
#[inline]
#[must_use]
fn important() -> i32 { 42 }

// Derive macros
#[derive(Debug, Clone, PartialEq)]
struct Point { x: i32, y: i32 }

// Conditional compilation
#[cfg(target_os = "linux")]
fn linux_only() { /* ... */ }

// Test attribute
#[test]
fn test_something() {
    assert_eq!(2 + 2, 4);
}

// Allow/deny lints
#[allow(dead_code)]
#[deny(unused_variables)]
fn example() { /* ... */ }

// Documentation
/// This is a doc comment for the following item
///
/// # Examples
/// ```
/// let x = my_function();
/// ```
fn my_function() { }
```

Modules

```rust
// Declare module
mod math {
    pub fn add(a: i32, b: i32) -> i32 { a + b }

    fn private_helper() { /* ... */ }

    pub mod advanced {
        pub fn sqrt(x: f64) -> f64 { x.sqrt() }
    }
}

// Use module items
use math::add;
use math::advanced::sqrt;

// Re-exports
pub use math::add;

// Glob import (generally discouraged)
use std::collections::*;

// Renaming
use std::io::Result as IoResult;
```

Part 3: Patterns

Patterns are used in `match`, `if let`, `while let`, `let`, and function parameters for destructuring and matching values.

Literal Patterns

Match exact values.

```rust
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("other"),
}
```

Variable Patterns

Bind matched value to a name.

```
let x = 5;

match x {
    n => println!("got {}", n),  // n binds to x's value
}
```

Wildcard Pattern

Matches anything, discards value.

```
let pair = (1, 2);

match pair {
    (0, _) => println!("first is zero"),
    (_, 0) => println!("second is zero"),
    (_, _) => println!("neither is zero"),
}
```

Rest Pattern

Match remaining elements.

```
let numbers = (1, 2, 3, 4, 5);

match numbers {
    (first, .., last) => println!("{} to {}", first, last),
}

let arr = [1, 2, 3, 4, 5];
match arr {
    [first, second, ..] => println!("starts with {}, {}", first, second),
}
```

Range Patterns

Match a range of values.

```
let x = 5;

match x {
    1..=5 => println!("one through five"),
    6..=10 => println!("six through ten"),
    _ => println!("something else"),
}

// Also works with char
match c {
    'a'..='z' => println!("lowercase"),
    'A'..='Z' => println!("uppercase"),
    _ => println!("other"),
}
```

Or Patterns

Match multiple alternatives.

```
let x = 2;

match x {
    1 | 2 | 3 => println!("one, two, or three"),
    4 | 5 => println!("four or five"),
    _ => println!("other"),
}
```

Struct Patterns

Destructure structs.

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

let p = Point { x: 0, y: 7 };

match p {
    Point { x: 0, y } => println!("on y-axis at {}", y),
    Point { x, y: 0 } => println!("on x-axis at {}", x),
    Point { x, y } => println!("at ({}, {})", x, y),
}

// Shorthand when variable name matches field
let Point { x, y } = p;

// Ignore fields with ..
match p {
    Point { x, .. } => println!("x is {}", x),
}
```

Tuple Patterns

Destructure tuples.

```
let tuple = (1, "hello", 3.14);

let (a, b, c) = tuple;

match tuple {
    (1, _, _) => println!("starts with 1"),
    (_, "hello", _) => println!("says hello"),
    _ => println!("other"),
}
```

Enum Patterns

Match and destructure enum variants.

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

let msg = Message::Move { x: 10, y: 20 };

match msg {
    Message::Quit => println!("quit"),
    Message::Move { x, y } => println!("move to ({}, {})", x, y),
    Message::Write(text) => println!("write: {}", text),
    Message::ChangeColor(r, g, b) => println!("color: {}/{}/{}", r, g, b),
}
```

Reference Patterns

Match references and bind by reference.

```rust
let reference = &4;

match reference {
    &val => println!("got value {}", val),  // Dereference in pattern
}

// Bind by reference (avoid moving)
let value = String::from("hello");

match value {
    ref r => println!("got reference: {}", r),  // r is &String
}

// Mutable reference binding
let mut value = 5;
match value {
    ref mut m => *m += 10,
}
```

Guards

Add conditions to pattern arms.

```rust
let pair = (2, -2);

match pair {
    (x, y) if x == y => println!("equal"),
    (x, y) if x + y == 0 => println!("sum to zero"),
    (x, _) if x % 2 == 0 => println!("first is even"),
    _ => println!("no match"),
}
```

@ Bindings

Bind a name while also testing a pattern.

```rust
let x = 5;

match x {
    n @ 1..=5 => println!("got {} in range 1-5", n),
    n @ 6..=10 => println!("got {} in range 6-10", n),
    n => println!("got {} outside ranges", n),
}

// With enums
enum Message {
    Hello { id: i32 },
}

match msg {
    Message::Hello { id: id_var @ 3..=7 } => {
        println!("found id in range: {}", id_var)
    }
    Message::Hello { id } => println!("other id: {}", id),
}
```

Slice Patterns

Match slices and arrays.

```rust
let arr = [1, 2, 3];

match arr {
    [1, _, _] => println!("starts with 1"),
    [_, 2, _] => println!("2 in middle"),
    [.., 3] => println!("ends with 3"),
    _ => println!("other"),
}

// With variable binding
match arr {
    [first, rest @ ..] => println!("first: {}, rest: {:?}", first, rest),
}

// Exact length matching
match slice {
    [] => println!("empty"),
    [single] => println!("one element: {}", single),
    [first, second] => println!("two elements"),
    [first, middle @ .., last] => println!("many elements"),
}
```

Box Patterns (Nightly)

Match through Box pointers.

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

let boxed = Box::new(5);

match boxed {
    box 0 => println!("zero"),
    box n => println!("got {}", n),
}
```

Part 4: Common Idioms

The Builder Pattern

Construct complex objects step by step.

```rust
struct ServerConfig {
    host: String,
    port: u16,
    max_connections: u32,
}

struct ServerConfigBuilder {
    host: String,
    port: u16,
    max_connections: u32,
}

impl ServerConfigBuilder {
    fn new() -> Self {
        Self {
            host: String::from("localhost"),
            port: 8080,
            max_connections: 100,
        }
    }

    fn host(mut self, host: &str) -> Self {
        self.host = host.to_string();
        self
    }

    fn port(mut self, port: u16) -> Self {
        self.port = port;
        self
    }

    fn build(self) -> ServerConfig {
        ServerConfig {
            host: self.host,
            port: self.port,
            max_connections: self.max_connections,
        }
    }
}

// Usage
let config = ServerConfigBuilder::new()
    .host("0.0.0.0")
    .port(3000)
    .build();
```

The Newtype Pattern

Wrap a type to provide distinct semantics.

```rust
struct Meters(f64);
struct Feet(f64);

impl Meters {
    fn to_feet(&self) -> Feet {
        Feet(self.0 * 3.28084)
    }
}

// Prevents mixing up units at compile time
fn calculate_area(width: Meters, height: Meters) -> f64 {
    width.0 * height.0
}
```

The Typestate Pattern

Use the type system to enforce valid state transitions.

```rust
struct Locked;
struct Unlocked;

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

impl Door<Locked> {
    fn unlock(self) -> Door<Unlocked> {
        println!("Unlocking...");
        Door { _state: std::marker::PhantomData }
    }
}

impl Door<Unlocked> {
    fn lock(self) -> Door<Locked> {
        println!("Locking...");
        Door { _state: std::marker::PhantomData }
    }

    fn open(&self) {
        println!("Opening door");
    }
}

// Can only open an unlocked door - enforced at compile time
let door: Door<Locked> = Door { _state: std::marker::PhantomData };
let door = door.unlock();
door.open();
```

RAII (Resource Acquisition Is Initialization)

Resources are tied to object lifetime.

```
struct FileHandle {
    path: String,
}

impl FileHandle {
    fn new(path: &str) -> Self {
        println!("Opening file: {}", path);
        Self { path: path.to_string() }
    }
}

impl Drop for FileHandle {
    fn drop(&mut self) {
        println!("Closing file: {}", self.path);
    }
}

// File automatically closed when handle goes out of scope
{
    let handle = FileHandle::new("data.txt");
    // use handle...
}  // Automatically calls drop() here
```

The Iterator Pattern

Process sequences lazily.

```rust
let numbers = vec![1, 2, 3, 4, 5];

// Chained iterator methods
let result: Vec<i32> = numbers
    .iter()
    .filter(|&x| x % 2 == 0)
    .map(|x| x * x)
    .collect();

// Implementing 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
        }
    }
}
```

Option/Result Combinators

Chain operations without explicit matching.

```rust
// Option combinators
let maybe_number: Option<i32> = Some(5);

let result = maybe_number
    .map(|n| n * 2)           // Some(10)
    .filter(|&n| n > 5)       // Some(10)
    .and_then(|n| Some(n + 1)) // Some(11)
    .unwrap_or(0);            // 11

// Result combinators
let result: Result<i32, &str> = Ok(5);

let final_value = result
    .map(|n| n * 2)
    .map_err(|e| format!("Error: {}", e))
    .and_then(|n| if n > 0 { Ok(n) } else { Err("negative".into()) })
    .unwrap_or_default();
```

Default Trait Pattern

Provide sensible defaults.

```
#[derive(Default)]
struct Config {
    debug: bool,
    timeout: u32,
    name: String,
}

// Custom Default implementation
impl Default for Config {
    fn default() -> Self {
        Self {
            debug: false,
            timeout: 30,
            name: String::from("default"),
        }
    }
}

// Usage
let config = Config::default();
let custom = Config { timeout: 60, ..Default::default() };
```

From/Into Conversions

Type-safe conversions.

```rust
struct Celsius(f64);
struct Fahrenheit(f64);

impl From<Celsius> for Fahrenheit {
    fn from(c: Celsius) -> Self {
        Fahrenheit(c.0 * 9.0 / 5.0 + 32.0)
    }
}

// From automatically provides Into
let c = Celsius(100.0);
let f: Fahrenheit = c.into();  // Uses Into, which uses From

// Common pattern with strings
impl From<&str> for MyType {
    fn from(s: &str) -> Self {
        MyType { name: s.to_string() }
    }
}
```

Deref Coercion Pattern

Smart pointers that act like references.

```rust
use std::ops::Deref;

struct MyBox<T>(T);

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Now MyBox<String> can be used where &str is expected
fn print_str(s: &str) {
    println!("{}", s);
}

let boxed = MyBox(String::from("hello"));
print_str(&boxed);  // Deref coercion: &MyBox<String> -> &String -> &str
```

Quick Reference: Operators

Operator	Meaning
`::`	Path separator (modules, types)
`->`	Function return type
`=>`	Match arm
`..`	Range (exclusive end)
`..=`	Range (inclusive end)
`..`	Struct update / rest pattern
`?`	Error propagation
`&`	Borrow / reference type
`*`	Dereference / raw pointer type
`'a`	Lifetime annotation
`<>`	Generics
`|`	Closure parameters / or-pattern
`_`	Wildcard / ignored binding
`@`	Pattern binding
`#[]`	Outer attribute
`#![]`	Inner attribute
`!`	Never type / macro invocation

This reference covers Rust 1.70+ syntax and patterns.