

Rust Quick Reference

Ownership, types, traits, pattern matching essentials

Basics

Hello World

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

Cargo Commands

```
cargo new my_project # create new project  
cargo build          # compile (debug)  
cargo build --release # compile (optimized)  
cargo run            # build and run  
cargo test           # run tests
```

Project Structure

```
Cargo.toml Project manifest (dependencies, metadata)  
src/main.rs Binary crate entry point  
src/lib.rs Library crate root  
tests/ Integration tests directory
```

Variables & Mutability

Binding & Mutability

```
let x = 5; // immutable by default  
let mut y = 10; // mutable  
y += 1;  
const MAX: u32 = 100; // compile-time constant
```

Shadowing

```
let x = 5;  
let x = x + 1; // shadows previous x  
let x = "now a string"; // can change type
```

Scalar Types

```
i8..i128, isize Signed integers  
u8..u128, usize Unsigned integers  
f32, f64 Floating point (f64 default)  
bool true / false  
char Unicode scalar value (4 bytes)
```

Compound Types

```
let tup: (i32, f64, char) = (42, 6.4, 'z');  
let (a, b, c) = tup; // destructure  
let arr: [i32; 3] = [1, 2, 3];  
let first = arr[0];
```

Functions

Definition

```
fn add(a: i32, b: i32) -> i32 {  
    a + b // no semicolon = return expression  
}
```

Closures

```
let double = |x: i32| x * 2;  
let sum: i32 = vec![1, 2, 3]  
    .iter()  
    .map(|x| x * 2)  
    .sum();
```

Function Pointers & Traits

```
fn(T) -> U Function pointer type  
Fn(T) -> U Closure that borrows  
FnMut(T) -> U Closure that mutably borrows  
FnOnce(T) -> U Closure that takes ownership
```

Control Flow

If / Else

```
let status = if score >= 90 { "A" }  
             else if score >= 80 { "B" }  
             else { "C" }; // if is an expression
```

Loops

```
loop { break; } // infinite  
while condition { } // while  
for item in &vec { } // iterator  
for i in 0..10 { } // range  
for (i, v) in vec.iter().enumerate() { }
```

Loop Labels

```
'outer: for i in 0..5 {  
    for j in 0..5 {  
        if i + j > 6 { break 'outer; }  
    }  
}
```

Ownership & Borrowing

Ownership Rules

1. Each value has exactly one owner.
2. When the owner goes out of scope, the value is dropped.
3. Values can be moved or cloned.

Move & Clone

```
let s1 = String::from("hello");  
let s2 = s1; // s1 is moved, no longer valid  
let s3 = s2.clone(); // deep copy, both valid
```

Borrowing

```
fn len(s: &String) -> usize { s.len() } // shared ref  
fn push(s: &mut String) { s.push('!'); } // mutable ref  
// Rule: many &T OR one &mut T, never both
```

Lifetimes

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

Structs & Enums

Struct

```
struct User {  
    name: String,  
    age: u32,  
    active: bool,  
}  
let u = User { name: String::from("Alice"), age: 30, active: true };
```

Impl Block

```
impl User {  
    fn new(name: &str, age: u32) -> Self {  
        Self { name: name.to_string(), age, active: true }  
    }  
    fn greeting(&self) -> String {  
        format!("Hi, {}!", self.name)  
    }  
}
```

Enums

```
enum Shape {  
    Circle(f64),  
    Rect { w: f64, h: f64 },  
    Point,  
}  
let s = Shape::Circle(5.0);
```

Pattern Matching

Match Expression

```
match shape {  
    Shape::Circle(r) => std::f64::consts::PI * r * r,  
    Shape::Rect { w, h } => w * h,  
    Shape::Point => 0.0,  
}
```

If Let & While Let

```
if let Some(val) = optional {  
    println!("{val}");  
}  
while let Some(top) = stack.pop() {  
    println!("{top}");  
}
```

Pattern Syntax

<code>_</code>	Wildcard, matches anything
<code>x @ 1..=5</code>	Bind matched range to x
<code>(a, b, ..)</code>	Destructure tuple, ignore rest
<code>Some(x) if x > 0</code>	Match guard
<code>Foo { x, .. }</code>	Struct, ignore other fields

Error Handling

Result & Option

```
enum Result<T, E> { Ok(T), Err(E) }  
enum Option<T> { Some(T), None }
```

The ? Operator

```
fn read_file(path: &str) -> Result<String, io::Error> {  
    let mut s = String::new();  
    File::open(path)?.read_to_string(&mut s);  
    Ok(s)  
}
```

Handling Errors

```
match result {  
    Ok(val) => println!("{val}"),  
    Err(e) => eprintln!("Error: {e}"),  
}  
let val = result.unwrap_or(0);  
let val = result.unwrap_or_else(|_| default());
```

Common Methods

<code>.unwrap()</code>	Get value or panic
<code>.expect(msg)</code>	Get value or panic with message
<code>.unwrap_or(default)</code>	Get value or use default
<code>.map(f)</code>	Transform the Ok/Some value
<code>.and_then(f)</code>	Chain operations (flatmap)
<code>.is_ok() / .is_some()</code>	Boolean check

Rust Quick Reference

Traits

Defining & Implementing

```
trait Summary {
    fn summarize(&self) -> String;
    fn preview(&self) -> String { // default impl
        format!("{}", ..., &self.summarize()[..20])
    }
}
impl Summary for User {
    fn summarize(&self) -> String { self.name.clone() }
}
```

Trait Bounds

```
fn notify(item: &impl Summary) { }
fn notify<T: Summary + Display>(item: &T) { }
fn notify(item: &(impl Summary + Display)) { }
```

Common Traits

Display	User-facing string formatting
Debug	Debug formatting <code>{:?}",</code>
Clone, Copy	Duplication (deep / bitwise)
PartialEq, Eq	Equality comparison
PartialOrd, Ord	Ordering comparison
Iterator	<code>next()</code> for iteration
From, Into	Type conversions
Default	Default value constructor

Collections

Vec

```
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
v.pop(); // returns Option<i32>
let first = &v[0]; // panics if empty
let first = v.get(0); // returns Option<&i32>
```

HashMap

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("key", 42);
m.entry("key").or_insert(0);
if let Some(val) = m.get("key") { }
```

String

```
let s = String::from("hello");
let s = "hello".to_string();
let combined = format!("{}", s, "world");
for c in s.chars() { } // iterate characters
```

Iterators

```
let sum: i32 = vec![1, 2, 3].iter().sum();
let doubled: Vec<_> = v.iter().map(|x| x * 2).collect();
let evens: Vec<_> = v.iter().filter(|x| *x % 2 == 0).collect();
```

Concurrency

Threads

```
use std::thread;
let handle = thread::spawn(|| {
    println!("from spawned thread");
});
handle.join().unwrap();
```

Channels

```
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(42).unwrap();
let val = rx.recv().unwrap();
```

Shared State

Arc<T>	Atomic reference counting (thread-safe Rc)
Mutex<T>	Mutual exclusion, lock to access inner value
RwLock<T>	Multiple readers or one writer
Send	Trait: safe to transfer between threads
Sync	Trait: safe to share references between threads

Macros & Attributes

Common Macros

println!()	Print with newline
format!()	Return formatted String
vec![]	Create Vec from literals
todo!()	Placeholder, panics at runtime
assert!(expr)	Panic if expr is false
assert_eq!(a, b)	Panic if a != b

Derive Attributes

```
#[derive(Debug, Clone, PartialEq)]
struct Point { x: f64, y: f64 }
// Auto-implements Debug, Clone, PartialEq
```

Testing Attributes

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() { assert_eq!(add(2, 2), 4); }
    #[test]
    #[should_panic]
    fn it_panics() { panic!("boom"); }
}
```