# Rust Language quick reference

**START HERE**

`let x = 5;`
Immutable binding

`let mut x = 5;`
Mutable binding

`&x / &mut x`
Borrow shared / exclusive

`match v { ... }`
Exhaustive branching

`Option<T> / Result<T, E>`
No null, no exceptions

`expr?`
Propagate the error

`impl Trait for Type`
Implement behaviour

`#[derive(Debug)]`
Generate trait impls

**Owner** one variable owns the value → `move` **New owner** old binding unusable → `&` **Borrow** many &T or one &mut T → `scope ends` **Drop** memory freed, no GC

Ownership: move, borrow, drop

## CONCEPTS

Every value has exactly one owner. Assigning or passing it moves it; borrow with & to keep using it.

Borrow rule: any number of &T or exactly one &mut T at a time, never both, and no reference outlives its owner.

Expressions have values: a block ends in its last expression without a semicolon, and that is its return value.

## BINDINGS AND TYPES

`let <x> = <v>;`
Immutable binding (shadowing allowed)

`let mut <x> = <v>;`
Mutable binding

`let <x>: <T> = <v>;`
Annotate the type

`i32 u8 usize f64 bool char`
Common scalar types

`&[<T>]`
Slice: borrowed view of a sequence

`&str / String`
Borrowed text / owned growable text

`Vec<T>`
Growable list

## FUNCTIONS AND CONTROL FLOW

`fn <name>(<a>: <T>) -> <R> { ... }`
Define a function

`|<x>| <expr>`
Closure

`move |<x>| <expr>`
Closure that takes ownership of captures

`if <c> { ... } else { ... }`
Conditional, usable as an expression

`if let <pat> = <v> { ... }`
Run when one pattern matches

`for <x> in <iter> { ... }`
Iterate a collection or range

`0..<n> / 0..=<n>`
Range, end exclusive / inclusive

## STRUCTS AND ENUMS

`struct <S> { <f>: <T> }`
Struct with named fields

`enum <E> { <A>, <B>(<T>), <C> { <f>: <U> } }`
Enum with data per variant

`impl <S> { fn <new>() -> Self { ... } }`
Add methods and associated functions

## STRUCTS AND ENUMS (cont.)

`&self / &mut self / self`
Method receiver: borrow / mutate / consume

`#[derive(Debug, Clone, PartialEq)]`
Auto-implement common traits

## PATTERN MATCHING

`match <v> { <pat> => <expr>, _ => <expr> }`
Exhaustive branching

`Some(<x>) / None`
Option variants

`Ok(<v>) / Err(<e>)`
Result variants

## OWNERSHIP AND REFERENCES

`&<x>`
Shared borrow (read only)

`&mut <x>`
Exclusive borrow (read and write)

`<x>.clone()`
Deep copy, explicit

`<'a>`
Lifetime parameter, as in fn f<'a>(x: &'a str)

`Box<T>`
Heap allocation, single owner

## OWNERSHIP AND REFERENCES (cont.)

`Rc<T> / Arc<T>`
Shared ownership, single / multi thread

## OPTION AND RESULT

`<opt>.unwrap_or(<d>)`
Value or a default

`<x>.map(|<v>| ...)`
Transform the inner value

`<x>.and_then(|<v>| ...)`
Chain a step that may fail

`<x>.unwrap() / .expect("<msg>")`
Panic if None or Err

`<res>.map_err(|<e>| ...)`
Convert the error type

`<x>?`
Return early on None or Err

## TRAITS AND GENERICS

`trait <T> { fn <m>(&self) -> <R>; }`
Define shared behaviour

`impl <Tr> for <S> { ... }`
Implement a trait for a type

`fn <f><T: <Tr>>(<x>: T)`
Generic function with a bound

`fn <f>(<x>: &impl <Tr>)`
Shorthand for a trait bound

## TRAITS AND GENERICS (cont.)

`where <T>: <A> + <B>,`
Bounds on a separate line

`struct <S><T> { <f>: T }`
Generic struct

`Box<dyn <Tr>>`
Trait object, dynamic dispatch

## COLLECTIONS AND ITERATORS

`vec![<a>, <b>]`
Vec literal

`<v>.push(<x>) / <v>.pop()`
Append / remove last

`<v>[<i>] / <v>.get(<i>)`
Index (panics) / checked (Option)

`<v>.iter() / .iter_mut() / .into_iter()`
Borrow / mutably borrow / consume

`<it>.map(|<x>| ...)`
Transform each item

`<it>.filter(|<x>| ...)`
Keep matching items

`<it>.collect::<Vec<_>>()`
Gather into a collection

`<it>.enumerate()`
Pair items with an index

`HashMap::new() / .insert(<k>, <v>)`
Key-value map

## COLLECTIONS AND ITERATORS (cont.)

`<m>.entry(<k>).or_insert(<d>)`
Insert if missing, get a &mut

## STRINGS

`String::from("<s>") / "<s>".to_string()`
Owned String from a literal

`format!("{} {}", <a>, <b>)`
Build a String

`println!("{<x>}")`
Print with inline variable

`{:?} / {:#?}`
Debug / pretty Debug formatting

## MODULES AND CRATES

`mod <name>;`
Declare a module (src/<name>.rs)

`use <path>::<Item>;`
Import a name

## TESTING AND DOCS

`#[test] fn <name>() { ... }`
Mark a test function

`assert!(<c>) / assert_eq!(<a>, <b>)`
Fail the test on mismatch