Rust Cheatsheet

Project Layout & Tooling Cargo essentials

- Create: `cargo new myproj` (or `-lib`)
- Build/Run: `cargo build`, `cargo run`
- Check fast: `cargo check`
- Tests: `cargo test`
- Format/Lint: `cargo fmt`, `cargo clippy`
- Docs: `cargo doc -open`

Common files

- `Cargo.toml`: deps, features, metadata
- `src/main.rs`: binary entry
- `src/lib.rs`: library root
- `mod.rs`: older module layout (still seen)

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

Bindings

- Immutable by default: `let x = 5;`
- Mutable: `let mut x = 5; x += 1;`
- Constants: `const MAX: u32 = 100;`
- Shadowing: `let x = x + 1; let x = "now str";`

Primitive types

- Integers: `i8 i16 i32 i64 i128 isize`, `u*`
- Floats: `f32 f64`
- Bool: `bool`; Char (Unicode scalar): `char`
- Unit: `()` (empty tuple)

Tuples & arrays

- Tuple: `let t: (i32, &str) = (1,"a"); let a = t.1;`
- Array (fixed): `let a = [1,2,3]; let z = [0; 10];`

Functions & Control Flow Functions

- Signature: `fn add(a: i32, b: i32) -> i32 { a + b }`
- Last expr returns; `return` optional

If as expression `let x = if cond { 1 } else { 2 };`

Loops

- `loop { ... break; }`
- `while cond { ... }`
- `for x in iter { ... }`
- Break with value: `let v = loop { break 42; };`

Match

- Exhaustive: `match x { 0 => ..., 1|2 => ..., _ => ...}`
- `Guards: n if n > 10 => ...`
- `Destructure: tuples/structs/enums`

Ownership, Borrowing, Lifetimes Ownership

quick rules

- Values have a single owner; drop at scope end
- Move by default for non-`Copy` types (e.g., `String`, `Vec`)
- Borrow with references: `&T` (shared), `&mut T` (exclusive)

Move vs Copy

- `Copy`: small stack types (ints, bool, char, tuples of Copy)
- `let a = 5; let b = a;` OK (Copy)
- `let s = String::from("hi"); let t = s;` move; `s` invalid

Borrowing

- Shared: many `&T` at once
- Mutable: only one `&mut T` at a time
- No mutable+shared overlap for same value

Slices

- `&[T]` from arrays/vectors
- `&str` is a string slice
- `let s = &string[0..3];` (careful with UTF-8 boundaries)

Lifetimes (practical)

- Usually inferred
- When returning refs: `fn first<'a>(s: 'a str) -> 'a str { ... }`
- Struct holding refs: `struct S<'a>{ r: 'a str }`

Strings, Collections String vs &str

- `&str`: borrowed view
- `String`: owned, growable

Common string ops

- Make: `let s = String::from("hi");`
- Push: `s.push('!'); s.push`$_s tr$(*"there"*);
- `Format: let x = format!("_", a, b);`
- `Iterate chars: for c in s.chars() {...}`
- `Bytes: s.as`$_b ytes$()

Vec

- `let mut v = vec![1,2,3]; v.push(4);`
- Index (panics if OOB): `v[0]`
- Safe get: `v.get(0)` returns `Option<&T>`
- Iterate: `for x in &v {...}` / `for x in v {...}` (moves)

HashMap

- `use std::collections::HashMap;`
- `let mut m = HashMap::new(); m.insert("k", 1);`
- Entry API: `*m.entry(k).or`$_i nsert(0)+ = 1$;

Structs, Enums, Pattern Matching Struct

- `struct User { id: u64, name: String }`
- Init: `User { id, name }`
- Update: `User { name, ..u }` (moves fields)

Impl blocks

- `impl User { fn new(id:u64)->Self { ... } }`
- Methods: `fn rename(&mut self, n:String) { self.name=n; }`

Enum

- `enum Msg { Quit, Move{x:i32,y:i32}, Text(String)}`
- Match destructure:

```
match m { Msg::Quit => ..., Msg::Move{x,y} =>
..., Msg::Text(s) => ... }
```

Option & Result

- `Option<T>`: `Some(T)` or `None`
- `Result<T,E>`: `Ok(T)` or `Err(E)`

Useful combinators

- Option: `map`, `and_then`, `unwrap_or`, `ok_or`
- Result: `map`, `map_err`, `and_then`, `unwrap_or_else`

Error Handling Patterns `?` operator

- Propagates errors: `fn f() -> Result<T,E> { let x = g()?; Ok(x)}`
- Also works for `Option`: returns `None` early

Custom error (simple)

- `enum MyErr { Io(std::io::Error), BadInput }`
- Convert: `From` impl or manual mapping with `map_err`

Crash vs handle

- `unwrap()/expect()` for prototypes/tests or when truly impossible
- Prefer returning `Result` in libs and fallible code paths

Traits, Generics, Common Bounds Generics

- `fn id<T>(x:T)->T { x }`
- Struct: `struct Boxed<T>(T);`

Trait bounds

- `fn f<T: Clone + Debug>(t: T) {...}`
- Where clause:

```
fn f<T>(t:T) where T: Clone + std::fmt::Debug
{...}
```

Common traits

- `Debug` (`#[derive(Debug)]`) for printing with `{:?}`
- `Clone` vs `Copy`
- `Eq`/`PartialEq`, `Ord`/`PartialOrd`, `Hash`
- `Default`, `From`/`Into`, `AsRef`
- `Send`/`Sync` (thread safety markers)

Trait objects (dynamic dispatch)

- `Box<dyn Trait>` for heterogenous collections / plugin-y design
- Often needs `Trait + Send + Sync + 'static` in async/concurrency

Modules, Visibility, Imports Modules

- Declare: `mod foo;` (loads `foo.rs` or `foo/mod.rs`)
- Public: `pub` items; re-export: `pub use path::Item;`

Use paths

- `use crate::foo::Bar;`
- `use std::io::self, Read;`
- Alias: `use long::path as lp;`

Closures, Iterators, Common FP-ish Patterns Closures

- `|x| x + 1`
- Capture by ref/mut/move: `move |x| ...`

Iterator pipeline

- `v.iter().map(...).filter(...).collect::<Vec<_>>();`
- `iter() borrows, into_iter() moves, iter_mut() mutable refs`

Useful iterator methods

- `map`, `filter`, `filter_map`, `flat_map`
- `fold`, `reduce`
- `any`, `all`, `find`, `position`
- `enumerate`, `zip`, `chain`

Smart Pointers & Interior Mutability `Box`

- Heap allocation; recursive types

`Rc/Arc`

- Shared ownership (ref counting)
- `Rc` single-thread; `Arc` thread-safe

`RefCell/Mutex`

- Interior mutability with runtime checks: `RefCell<T>`
- Shared mutable across threads: `Arc<Mutex<T>>`

Typical combos

- Single-thread graph: `Rc<RefCell<Node>>`
- Multithread shared state: `Arc<Mutex<State>>` (or `RwLock`)

Concurrency (Std) & Async Notes Threads

- `std::thread::spawn(|| { ... })`
- Join: `let h = spawn(...); h.join().unwrap();`

Channels

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

Async (ecosystem)

- Needs runtime (often Tokio/async-std)
- `async fn f() -> Result<T,E>`
- `.await` waits a future

Macros, Attributes, Testing Macros

- Common: `println!`, `format!`, `vec!`, `dbg!`
- Derive: `#[derive(Debug, Clone, PartialEq)]`

Attributes

- `#[allow(dead_code)]`, `#[cfg(test)]`, `#[cfg(feature="x")]`

Testing

- `#[test] fn it_works() { assert_eq!(2+2,4); }`
- Panic test: `#[should_panic]`
- Result test: `fn t() -> Result<(),E> { ...; Ok(()) }`

Common Idioms (Quick Recipes)

Builder-ish struct init `let cfg = Cfg { a:1, ..Default::default() };`

Early return with match `let v = match opt { Some(x)=>x, None=>return };`

Borrow then mutate (pattern)

- Use scopes to end borrows before mutation
- Or clone small data; or use `split_at_mut` for slices

Parse string `let n: i32 = s.parse()?;`

Read file (simple) `let txt = std::fs::read_to_string("a.txt"`

Serde (common crate)

- `#[derive(Serialize, Deserialize)]`
- JSON: `serde_json::to_string(&v)?;`

Mini Reference

- Print debug: `println!(":?", x);`
- Pattern `if let`: `if let Some(x)=opt {...}`
- `while let`: `while let Some(x)=iter.next() {...}`
- Ranges: `0..n`, `0..=n`
- Ownership hint: prefer passing `&T` unless you need to take ownership