Complete Rust Cheatsheet

~ Developer Shaurya

Important Links

Rust Cheatsheet: https://developershaurya.com/rust-cheatsheet/

Website: https://developershaurya.com/

All CheatSheets:

https://developershaurya.com/cheat-sheets/

Youtube: https://www.youtube.com/@DeveloperShaurya

Developer Shaurya Community:

https://t.me/developer_shaurya (Telegram)

CheatSheet

RUST CHEATSHEET – A Quick Guide for Beginners & Developers

by Developer Shaurya • November 26, 2025 • 0

(Beginner → Advanced → Expert → System-level)

Content Table

1. BASIC SYNTAX

Print

```
println!("Hello {}", name);
dbg!(value);
```

Comments

```
// single line
/* multi line */
```

Variables

```
let x = 5;       // immutable
let mut y = 10;  // mutable
const PI: f32 = 3.14;
static GLOBAL: i32 = 5;    // global
```

Shadowing

```
let x = 5;
let x = x + 1;
```

2. DATA TYPES

Scalars

```
i8 i16 i32 i64 i128 isize
u8 u16 u32 u64 u128 usize
f32 f64
bool char
```

Compound Types

```
let tup: (i32, bool) = (10, true);
let arr: [i32; 4] = [1, 2, 3, 4];
let slice = &arr[1..3];
```

String Types

```
&str                // string slice
String              // heap string
Cow<'_, str>        // clone-on-write
OsString, OsStr     // OS-safe
CString, CStr       // C interop
```

3. OWNERSHIP & BORROWING

Move

```
let a = String::from("hi");
let b = a;   // move, a invalid
```

Clone

```
let b = a.clone();
```

Borrow

```
fn foo(s: &String) {}
fn bar(s: &mut String) {}
```

Borrowing Rules

✓ One mutable reference **OR**
✓ Many shared references
✗ Not both at the same time

4. CONTROL FLOW

```
if condition { }
else { }

match x {
    1 => println!("one"),
    2..=5 => println!("range"),
    _ => (),
}

if let Some(v) = option {}
while let Some(v) = iter.next() {}
```

5. FUNCTIONS & CLOSURES

Normal

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

Closures

```
let f = |x| x + 1;
let g = move |x| x + y;
```

Closure Traits

- `Fn` – read-only
- `FnMut` – mutable captures
- `FnOnce` – consumes values

6. STRUCTS

```
struct User { name: String, age: u32 }

impl User {
    fn greet(&self) {}
    fn grow(&mut self) { self.age += 1; }
}
```

Tuple Struct

```
struct Color(u8, u8, u8);
```

Unit struct

```
struct Marker;
```

7. ENUMS + PATTERN MATCHING

```
enum ResultType {
    Success(String),
    Error(i32),
}
```

Patterns

```
let (a, b) = (1, 2);
let Point { x, y } = p;
let Some(v) = opt else { return };
```

Guards

```
match x {
    x if x > 10 => {}
    _ => {}
}
```

8. TRAITS (Interfaces)

Basic

```
trait Speak { fn say(&self); }
impl Speak for Dog { fn say(&self) {} }
```

Trait Objects

```
let obj: Box<dyn Speak> = Box::new(Dog);
```

Associated Types

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Default Type Params

```
trait A<T = u32> {}
```

9. GENERICS

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

Common Bounds

```
T: Clone + Debug + Copy + Send + Sync + 'static
```

10. MODULES

File Structure

```
src/
 ├── main.rs
 ├── lib.rs
 └── utils/
       └── mod.rs
```

Visibility

```
pub
pub(crate)
pub(super)
pub(in crate::mod)
```

Re-exports

```
pub use crate::utils::helper;
```

11. COLLECTIONS

Vector

```
let mut v = vec![1,2,3];
v.push(10);
v.pop();
```

HashMap

```
let mut m = HashMap::new();
m.insert("a", 1);
```

HashSet, BTreeMap, VecDeque

```
use std::collections::*;
```

12. ITERATORS

Methods

```
map, filter, find, position,
fold, collect, flatten,
enumerate, zip
```

Custom Iterator

```
impl Iterator for Counter {
    type Item = i32;
    fn next(&mut self) -> Option<Self::Item> { ... }
}
```

13. ERROR HANDLING

Option

```
opt.unwrap();
opt.unwrap_or(0);
opt.ok_or("error")?;
```

Result

```
let x: Result<i32, String> = Ok(10);
? operator for propagation
```

Custom Errors

```
thiserror
anyhow
```

14. FILE I/O

```
fs::read_to_string("a.txt")?;
fs::write("b.txt", data)?;
```

15. MACROS (COMPLETE)

Declarative (`macro_rules!`)

```
macro_rules! add {
    ($a:expr, $b:expr) => { $a + $b };
}
```

Procedural Macros

- **Derive macros**
- **Function-like macros**
- **Attribute macros**

```
#[proc_macro_derive(MyDerive)]
pub fn my_derive(input: TokenStream) -> TokenStream {}
```

Built-in macros

```
println!
format!
vec!
macro_rules!
todo!
unreachable!
unimplemented!
```

16. ADVANCED PATTERN MATCHING

- Nested matches
- By-reference patterns
- Slices in match

```
match slice {
    [a, b, rest @ ..] => {}
}
```

17. MEMORY & POINTERS

Smart pointers

- `Box<T>` – heap alloc
- `Rc<T>` – shared ownership
- `Arc<T>` – thread-safe reference counter
- `RefCell<T>` – runtime borrow checking
- `Cell<T>` – copy-type mutability
- `Mutex<T>` / `RwLock<T>` – thread locks

Raw Pointers (Unsafe)

```
let a = 10;
let p = &a as *const i32;
```

18. UNSAFE RUST (FULL)

When needed:

- Raw pointer dereferencing
- FFI
- Calling unsafe functions
- Implementing unsafe traits
- Accessing mutable static variables
- Using `union`

Unsafe block

```
unsafe {
    *ptr = 10;
}
```

Foreign Function Interface

```
extern "C" {
    fn printf(...);
}
```

19. ASYNC & FUTURES

Basic async

```
async fn foo() -> i32 { 10 }
foo().await;
```

Future trait

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context);
}
```

Pinning

```
use std::pin::Pin;
```

Send & Sync

```
T: Send     // thread-safe move
T: Sync     // reference shared safely across threads
```

Tokio

```
#[tokio::main]
async fn main() {}
```

20. CONCURRENCY (ADVANCED)

Channels

```
mpsc::channel();
sync_channel();
crossbeam_channel;
```

Atomics

```
AtomicUsize, AtomicBool
```

Threads

```
thread::spawn(|| {});
```

21. TESTING

Unit Tests

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

Integration Tests

```
tests/test1.rs
```

Benchmarks

```
cargo bench
```

22. CARGO (FULL)

Commands

```
cargo new, build, run, fmt, clippy, doc
```

Features

```
[features]
default = ["json"]
json = []
```

Workspaces

```
[workspace]
members = ["core", "app"]
```

Build scripts

```
build.rs
```

23. CRATES YOU MUST KNOW

- **serde** – serialize/deserialize
- **tokio / async-std** – async runtime
- **reqwest** – HTTP client
- **tracing** – logging
- **anyhow** – easy errors
- **thiserror** – ergonomic custom errors
- **chrono** – date/time
- **rayon** – parallel iterators
- **dashmap** – concurrent HashMap

24. RUST COMPILER / LIFETIME ADVANCED

Lifetime elision rules

1. Each parameter gets its own lifetime
2. One input → output same lifetime
3. For `&self` methods → output = `self`

'static lifetime

```
let x: &'static str = "hello";
```

25. LOW-LEVEL RUST

Unions

```
union MyUnion {
    f: f32,
    i: i32,
}
```

Memory Layout

```
#[repr(C)]
struct CCompatible { a: i32, b: f32 }
```

Bitflags

```
bitflags::bitflags! {
    struct Flags: u32 {
        const A = 0b0001;
    }
}
```

26. ADVANCED MATCHING + GENERATORS

Rust has experimental:

- `async fn generators`
- `yield` statements (nightly)

27. WASM

```
wasm-bindgen
web-sys
js-sys
```

28. BUILDING CLI APPS

```
clap = "4"
```

29. RUST STANDARDS & BEST PRACTICES

✓ Prefer `&str` over `String` in APIs
✓ Avoid `clone()` unless needed
✓ Don't use `unwrap()` in production
✓ Use `?` for error propagation
✓ Use `clippy` for linting
✓ Write small modules
✓ Avoid `Rc<RefCell<T>>` in multithreaded code
✓ Prefer `Arc<Mutex<T>>` or channels

30. RUST PHILOSOPHY

- Zero-cost abstractions
- Fearless concurrency
- Memory safety without garbage collection