Rust Language Cheat Sheet

08.02.2020

Contains clickable links to **The Book** BK, **Rust by Example** EX, **Std Docs** STD, **Nomicon** NOM, **Reference** REF. Other symbols used: largely **deprecated**, has a **minimum edition** '18, is **work in progress**, or **bad**.

Fira Code Ligatures (••=, ⇒) Night Mode

Data Structures

Data types and memory locations defined via keywords.

Example	Explanation
`struct S {}`	Define a **struct** BK EX STD REF with named fields.
`struct S { x: T }`	Define struct with named field `x` of type `T`.
`struct S(T);`	Define "tupled" struct with numbered field `.0` of type `T`.
`struct S;`	Define **zero sized** NOM unit struct. Occupies no space, optimized away.
`enum E {}`	Define an **enum** BK EX REF, *c.* algebraic data types, tagged unions.
`enum E { A, B(), C {} }`	Define variants of enum; can be unit- `A`, tuple- `B()` and struct-like `C{}`.
`enum E { A = 1 }`	If variants are only unit-like, allow discriminant values, e.g., for FFI.
`union U {}`	Unsafe C-like **union** REF for FFI compatibility.
`static X: T = T();`	**Global variable** BK EX REF with `'static` lifetime, single memory location.
`const X: T = T();`	Defines **constant** BK EX REF. Copied into a temporary when used.
`let x: T;`	Allocate `T` bytes on stack [1] bound as `x`. Assignable once, not mutable.
`let mut x: T;`	Like `let`, but allow for mutability and mutable borrow. [2]
`x = y;`	Moves `y` to `x`, invalidating `y` if `T` is not `Copy`, and copying `y` otherwise.

[1] They live on the stack for synchronous code. For `async` code these variables become part of the async's state machine which may ultimately reside on the heap.
[2] Note that technically *mutable* and *immutable* are a bit of a misnomer. Even if you have an immutable binding or shared reference, it might contain a Cell, which supports so called *interior mutability*.

Creating and accessing data structures; and some more *sigilic* types.

Example	Explanation
`S { x: y }`	Create `struct S {}` or `use`'ed `enum E::S {}` with field `x` set to `y`.
`S { x }`	Same, but use local variable `x` for field `x`.
`S { ..s }`	Fill remaining fields from `s`, esp. useful with Default.
`S { 0: x }`	Like `S(x)` below, but set field `.0` with struct syntax.
`S(x)`	Create `struct S(T)` or `use`'ed `enum E::S()` with field `.0` set to `x`.

Example	Explanation
`S`	If `S` is unit `struct S;` or `use`'ed `enum E::S` create value of `S`.
`E::C { x: y }`	Create enum variant `C`. Other methods above also work.
`()`	Empty tuple, both literal and type, aka **unit**. STD
`(x)`	Parenthesized expression.
`(x,)`	Single-element **tuple** expression. EX STD REF
`(S,)`	Single-element tuple type.
`[S]`	Array type of unspecified length, i.e., **slice**. STD EX REF Can't live on stack. *
`[S; n]`	**Array type** EX STD of fixed length `n` holding elements of type `S`.
`[x; n]`	Array instance with `n` copies of `x`. REF
`[x, y]`	Array instance with given elements `x` and `y`.
`x[0]`	Collection indexing. Overloadable Index, IndexMut
`x[..]`	Collection slice-like indexing via RangeFull, *c.* slices.
`x[a..]`	Collection slice-like indexing via RangeFrom.
`x[..b]`	Collection slice-like indexing RangeTo.
`x[a..b]`	Collection slice-like indexing via Range.
`a..b`	Right-exclusive **range** REF creation, also seen as `..b`.
`a..=b`	Inclusive range creation, also seen as `..=b`.
`s.x`	Named **field access**, REF might try to Deref if `x` not part of type `S`.
`s.0`	Numbered field access, used for tuple types `S(T)`.

* For now, see tracking issue and corresponding RFC 1909.

References & Pointers

Granting access to un-owned memory. Also see section on Generics & Constraints.

Example	Explanation
`&S`	Shared **reference** BK STD NOM REF (space for holding *any* `&S`).
`&[S]`	Special slice reference that contains (`address`, `length`).
`&str`	Special string reference that contains (`address`, `length`).
`&dyn S`	Special **trait object** BK reference that contains (`address`, `vtable`).
`&mut S`	Exclusive reference to allow mutability (also `&mut [S]`, `&mut dyn S`, ...)
`*const S`	Immutable **raw pointer type** BK STD REF w/o memory safety.
`*mut S`	Mutable raw pointer type w/o memory safety.
`&s`	Shared **borrow** BK EX STD (e.g., address, len, vtable, ... of *this* `s`, like `0x1234`).
`&mut s`	Exclusive borrow that allows **mutability**. EX
`ref s`	**Bind by reference**. EX
`*r`	**Dereference** BK STD NOM a reference `r` to access what it points to.
`*r = s;`	If `r` is a mutable reference, move or copy `s` to target memory.
`s = *r;`	Make `s` a copy of whatever `r` references, if that is `Copy`.
`s = *my_box;`	Special case for `Box` that can also move out Box'ed content if it isn't `Copy`.
`'a`	A **lifetime parameter**, BK EX NOM REF, duration of a flow in static analysis.

Example	Explanation
`&'a S`	Only accepts a `S` with an address that lives `'a` or longer.
`&'a mut S`	Same, but allow content of address to be changed.
`struct S<'a> {}`	Signals `S` will contain address with lifetime `'a`. Creator of `S` decides `'a`.
`trait T<'a> {}`	Signals a `S` which `impl T for S` might contain address.
`fn f<'a>(t: &'a T)`	Same, for function. Caller decides `'a`.
`'static`	Special lifetime lasting the entire program execution.

Functions & Behavior

Define units of code and their abstractions.

Example	Explanation
`trait T {}`	Define a **trait**; BK EX REF common behavior others can implement.
`trait T : R {}`	`T` is subtrait of **supertrait** REF `R`. Any `S` must `impl R` before it can `impl T`.
`impl S {}`	**Implementation** REF of functionality for a type `S`, e.g., methods.
`impl T for S {}`	Implement trait `T` for type `S`.
`impl !T for S {}`	Disable an automatically derived **auto trait** NOM REF.
`fn f() {}`	Definition of a **function** BK EX REF; or associated function if inside `impl`.
`fn f() -> S {}`	Same, returning a value of type S.
`fn f(&self) {}`	Define a method, e.g., within an `impl S {}`.
`const fn f() {}`	Constant `fn` usable at compile time, e.g., `const X: u32 = f(Y)`. '18
`async fn f() {}`	**Async** '18 function transformation, makes `f` return an `impl Future`. STD
`async fn f() -> S {}`	Same, but make `f` return an `impl Future<Output=S>`.
`async { x }`	Used within a function, make `{ x }` an `impl Future<Output=X>`.
`fn() -> S`	**Function pointers**, BK STD REF, memory holding address of a callable.
`Fn() -> S`	**Callable Trait**, BK STD (also `FnMut`, `FnOnce`), implemented by closures, fn's ...
`|| {}`	A **closure** BK EX REF that borrows its captures.
`|x| {}`	Closure with a bound parameter `x`.
`|x| x + x`	Closure without block expression; may only consist of single expression.
`move |x| x + y`	Closure taking ownership of its captures.
`return || true`	Closures sometimes look like logical ORs (here: return a closure).
`unsafe {}`	If you enjoy debugging segfaults Friday night; **unsafe code**. BK EX NOM REF

Control Flow

Control execution within a function.

Example	Explanation
`while x {}`	**Loop** REF, run while expression `x` is true.
`loop {}`	**Loop infinitely** REF until `break`. Can yield value with `break x`.
`for x in iter {}`	Syntactic sugar to loop over **iterators**. BK STD REF
`if x {} else {}`	**Conditional branch** REF if expression is true.
`'label: loop {}`	**Loop label** EX REF, useful for flow control in nested loops.

Example	Explanation
`break`	**Break expression** REF to exit a loop.
`break x`	Same, but make `x` value of the loop expression (only in actual `loop`).
`break 'label`	Exit not only this loop, but the enclosing one marked with `'label`.
`continue`	**Continue expression** REF to the next loop iteration of this loop.
`continue 'label`	Same, but instead of enclosing loop marked with `'label`.
`x?`	If `x` is Err or None, **return and propagate**. BK EX STD REF
`x.await`	Only works inside `async`. Yield flow until Future or Stream ? `x` ready. '18
`return x`	Early return from function. More idiomatic way is to end with expression.
`f()`	Invoke callable `f` (e.g., a function, closure, function pointer, `Fn`, ...).
`x.f()`	Call member function, requires `f` takes `self`, `&self`, ... as first argument.
`X::f(x)`	Same as `x.f()`. Unless `impl Copy for X {}`, `f` can only be called once.
`X::f(&x)`	Same as `x.f()`.
`X::f(&mut x)`	Same as `x.f()`.
`S::f(&x)`	Same as `x.f()` if `X` derefs to `S`, i.e., `x.f()` finds methods of `S`.
`T::f(&x)`	Same as `x.f()` if `X impl T`, i.e., `x.f()` finds methods of `T` if in scope.
`X::f()`	Call associated function, e.g., `X::new()`.
`<X as T>::f()`	Call trait method `T::f()` implemented for `X`.

Organizing Code

Segment projects into smaller units and minimize dependencies.

Example	Explanation
`mod m {}`	Define a **module** BK EX REF, get definition from inside `{}`.
`mod m;`	Define a module, get definition from `m.rs` or `m/mod.rs`.
`a::b`	Namespace **path** EX REF to element `b` within `a` (`mod`, `enum`, ...).
`::b`	Search `b` relative to crate root.
`crate::b`	Search `b` relative to crate root. '18
`self::b`	Search `b` relative to current module.
`super::b`	Search `b` relative to parent module.
`use a::b;`	**Use** EX REF `b` directly in this scope without requiring `a` anymore.
`use a::{b, c};`	Same, but bring `b` and `c` into scope.
`use a::b as x;`	Bring `b` into scope but name `x`, like `use std::error::Error as E`.
`use a::b as _;`	Bring `b` anonymously into scope, useful for traits with conflicting names.
`use a::*;`	Bring everything from `a` into scope.
`pub use a::b;`	Bring `a::b` into scope and reexport from here.
`pub T`	"Public if parent path is public" **visibility** BK for `T`.
`pub(crate) T`	Visible at most in current crate.
`pub(self) T`	Visible at most in current module.
`pub(super) T`	Visible at most in parent.
`pub(in a::b) T`	Visible at most in `a::b`.

Example	Explanation
`extern crate a;`	Declare dependency on external **crate** BK EX REF 🗑 ; just `use a::b` in '18.
`extern "C" {}`	*Declare* external dependencies and ABI (e.g., `"C"`) from **FFI**. BK EX NOM REF
`extern "C" fn f() {}`	*Define* function to be exported with ABI (e.g., `"C"`) to FFI.

Type Aliases and Casts

Short-hand names of types, and methods to convert one type to another.

Example	Explanation
`type T = S;`	Create a **type alias** BK REF, i.e., another name for `S`.
`Self`	Type alias for **implementing type** REF, e.g. `fn new() -> Self`.
`self`	Method subject in `fn f(self) {}`, same as `fn f(self: Self) {}`.
`&self`	Same, but refers to self as borrowed, same as `f(self: &Self)`
`&mut self`	Same, but mutably borrowed, same as `f(self: &mut Self)`
`self: Box<Self>`	Arbitrary self type, add methods to smart pointers (`my_box.f_of_self()`).
`S as T`	**Disambiguate** BK REF type `S` as trait `T`, e.g., `<X as T>::f()`.
`S as R`	In `use` of symbol, import `S` as `R`, e.g., `use a::b as x`.
`x as u32`	Primitive **cast** EX REF, may truncate and be a bit surprising. NOM

Macros & Attributes

Code generation constructs expanded before the actual compilation happens.

Example	Explanation
`m!()`	**Macro** BK STD REF invocation, also `m!{}`, `m![]` (depending on macro).
`$x:ty`	Macro capture, also `$x:expr`, `$x:ty`, `$x:path`, ... see next table.
`$x`	Macro substitution in **macros by example**. BK EX REF
`$(x),*`	Macro repetition "zero or more times" in macros by example.
`$(x),?`	Same, but "zero or one time".
`$(x),+`	Same, but "one or more times".
`$(x)<<+`	In fact separators other than `,` are also accepted. Here: `<<`.
`$crate`	Special hygiene variable, crate where macros is defined. ?
`#[attr]`	Outer **attribute**. EX REF, annotating the following item.
`#![attr]`	Inner attribute, annotating the surrounding item.

In a `macro_rules!` implementation, the following macro captures can be used:

Macro Capture	Explanation
`$x:item`	An item, like a function, struct, module, etc.
`$x:block`	A block `{}` of statements or expressions, e.g., `{ let x = 5; }`
`$x:stmt`	A statement, e.g., `let x = 1 + 1;`, `String::new();` or `vec![];`
`$x:expr`	An expression, e.g., `x`, `1 + 1`, `String::new()` or `vec![]`
`$x:pat`	A pattern, e.g., `Some(t)`, `(17, 'a')` or `_`.
`$x:ty`	A type, e.g., `String`, `usize` or `Vec<u8>`.

Macro Capture	Explanation
`$x:ident`	An identifier, for example in `let x = 0;` the identifier is `x`.
`$x:path`	A path (e.g. `foo`, `::std::mem::replace`, `transmute::<_, int>`, ...).
`$x:literal`	A literal (e.g. `3`, `"foo"`, `b"bar"`, etc.).
`$x:meta`	A meta item; the things that go inside `#[...]` and `#![...]` attributes.
`$x:tt`	A single token tree, see here for more details.

Pattern Matching

Constructs found in `match` or `let` expressions, or function parameters.

Example	Explanation
`match m {}`	Initiate **pattern matching** BK EX REF, then use match arms, *c.* next table.
`let S(x) = get();`	Notably, `let` also pattern matches similar to the table below.
`let S { x } = s;`	Only `x` will be bound to value `s.x`.
`let (_, b, _) = abc;`	Only `b` will be bound to value `abc.1`.
`let (a, ..) = abc;`	Ignoring 'the rest' also works.
`let Some(x) = get();`	**Won't** work 🔴 if pattern can be **refuted** REF, use `if let` instead.
`if let Some(x) = get() {}`	Branch if pattern can actually be assigned (e.g., `enum` variant).
`fn f(S { x }: S)`	Function parameters also work like `let`, here `x` bound to `s.x` of `f(s)`.

Pattern matching arms in `match` expressions. The left side of these arms can also be found in `let` expressions.

Example	Explanation
`E::A => {}`	Match enum variant `A`, *c.* **pattern matching**. BK EX REF
`E::B ( .. ) => {}`	Match enum tuple variant `B`, wildcard any index.
`E::C { .. } => {}`	Match enum struct variant `C`, wildcard any field.
`S { x: 0, y: 1 } => {}`	Match struct with specific values (only accepts `s` with `s.x` of `0` and `s.y` of `1`).
`S { x: a, y: b } => {}`	Match struct with *any*(!) values and bind `s.x` to `a` and `s.y` to `b`.
`S { x, y } => {}`	Same, but shorthand with `s.x` and `s.y` bound as `x` and `y` respectively.
`S { .. } => {}`	Match struct with any values.
`D => {}`	Match enum variant `E::D` if `D` in `use`.
`D => {}`	Match anything, bind `D`; possibly false friend 🔴 of `E::D` if `D` not in `use`.
`_ => {}`	Proper wildcard that matches anything / "all the rest".
`(a, 0) => {}`	Match tuple with any value for `a` and `0` for second.
`[a, 0] => {}`	**Slice pattern**, REF 🔗 match array with any value for `a` and `0` for second.
`[1, ..] => {}`	Match array starting with `1`, any value for rest; **subslice pattern**. 🚧 ?
`[2, .., 5] => {}`	Match array starting with `1`, ending with `5`. 🚧
`[2, x @ .., 5] => {}`	Same, but also bind `x` to slice representing middle (*c.* next entry). 🚧
`x @ 1..=5 => {}`	Bind matched to `x`; **pattern binding**, BK EX REF here `x` would be `1`, `2`, ... or `5`.
`0 | 1 => {}`	Pattern alternatives (or-patterns).
`E::A | E::Z`	Same, but on enum variants.
`E::C {x} | E::D {x}`	Same, but bind `x` if all variants have it.

Example	Explanation
`S { x } if x > 10`	Pattern **match guards**, BK EX REF condition must be true as well to match.

Generics & Constraints

Generics combine with many other constructs such as `struct S<T>`, `fn f<T>()`, ...

Example	Explanation
`S<T>`	A **generic** BK EX type with a type parameter (`T` is placeholder name here).
`S<T: R>`	Type short hand **trait bound** BK EX specification (`R` *must* be actual trait).
`T: R, P: S`	**Independent trait bounds** (here one for `T` and one for `P`).
`T: R, S`	Compile error, you probably want compound bound `R + S` below.
`T: R + S`	**Compound trait bound** BK EX, `T` must fulfill `R` and `S`.
`T: R + 'a`	Same, but w. lifetime. `T` must fulfill `R`, if `T` has lifetimes, must outlive `'a`.
`T: ?Sized`	Opt out of a pre-defined trait bound, here `Sized`. ?
`T: 'a`	Type **lifetime bound** EX; if T has references, they must outlive `'a`.
`'b: 'a`	Lifetime `'b` must live at least as long as (i.e., *outlive*) `'a` bound.
`S<T> where T: R`	Same as `S<T: R>` but more pleasant to read for longer bounds.
`S<T = R>`	**Default type parameter** BK for associated type.
`S<'_>`	Inferred **anonymous lifetime**; asks compiler to *'figure it out'* if obvious.
`S<_>`	Inferred **anonymous type**, e.g., as `let x: Vec<_> = iter.collect()`
`S::<T>`	**Turbofish** STD call site type disambiguation, e.g. `f::<u32>()`.
`trait T<X> {}`	A trait generic over `X`. Can have multiple `impl T for S` (one per `X`).
`trait T { type X; }`	Defines **associated type** BK REF `X`. Only one `impl T for S` possible.
`type X = R;`	Set associated type within `impl T for S { type X = R; }`.
`impl<T> S<T> {}`	Implement functionality for any `T` in `S<T>`.
`impl S<T> {}`	Implement functionality for exactly `S<T>` (e.g., `S<u32>`).
`fn f() -> impl T`	**Existential types** BK, returns an unknown-to-caller `S` that `impl T`.
`fn f(x: &impl T)`	Trait bound,"**impl traits**" BK, somewhat similar to `fn f<S:T>(x: &S)`.
`fn f(x: &dyn T)`	Marker for **dynamic dispatch** BK REF, `f` will not be monomorphized.
`fn f() where Self: R`	In a `trait T {}`, mark `f` as accessible only on types that also `impl R`.
`for<'a>`	**Higher-ranked trait bounds.** NOM REF
`trait T: for<'a> R<'a> {}`	Any `S` that `impl T` would also have to fulfill `R` for any lifetime.

Strings & Chars

Rust has several ways to create string or char literals, depending on your needs.

Example	Explanation
`" ... "`	**String literal**, REF UTF-8, will interpret `\n` as *line break* `0×A`, ...
`r" ... "`,	**Raw string literal**. REF UTF-8, won't interpret `\n`, ...
`r#" ... "#`, etc.	Raw string literal, UTF-8, but can also contain `"`.
`b" ... "`	**Byte string literal**; REF constructs ASCII `[u8]`, not a string.
`br" ... "`, `br#" ... "#`, etc.	Raw byte string literal, ASCII `[u8]`, combination of the above.

Example	Explanation
`'🦀'`	**Character literal**, REF fixed 4 byte unicode '**char**'. STD
`b'x'`	ASCII **byte literal**. REF

Comments

No comment.

Example	Explanation
`//`	Line comment, use these to document code flow or *internals*.
`//!`	Inner line **doc comment** BK EX REF for auto generated documentation.
`///`	Outer line doc comment, use these on types.
`/* ... */`	Block comment.
`/*! ... */`	Inner block doc comment.
`/** ... */`	Outer block doc comment.
`` ```rust ... ``` ````	In doc comments, include a doc test (doc code running on `cargo test`).
`#`	In doc tests, hide line from documentation (```` ``` # use x::hidden; ``` ````).

Miscellaneous

These sigils did not fit any other category but are good to know nonetheless.

Example	Explanation
`!`	Always empty **never type**. 🚧 BK EX STD REF
`_`	Unnamed variable binding, e.g., `|x, _| {}`.
`_x`	Variable binding explicitly marked as unused.
`1_234_567`	Numeric separator for visual clarity.
`1_u8`	Type specifier for **numeric literals** EX REF (also `i8`, `u16`, ...).
`0×BEEF`, `0o777`, `0b1001`	Hexadecimal (`0x`), octal (`0o`) and binary (`0b`) integer literals.
`r#foo`	A **raw identifier** BK EX for edition compatibility.
`x;`	**Statement** REF terminator, *c.* **expressions** EX REF

Common Operators

Rust supports all common operators you would expect to find in a language (`+`, `*`, `%`, `=`, `==`...). Since they behave no differently in Rust we do not list them here. For some of them Rust also supports **operator overloading**. STD

Standard Library

🚧 This section is **WORK IN PROGRESS**. Comments, issues and PRs are very welcome. 🚧

Traits

Common Markers

These traits mark **special properties** of the underlying type.

A type `T` marked ...	Means `T` can ...
`Copy`	Be copied bitwise; cheap & fast. Also means `x = t` will copy t instead of move.
`Clone`	Be explicitly duplicated via `.clone()`. Might be expensive.
`Send`*	Be moved between threads safely.
`Sync`*	Have its reference `&T` sent between threads safely.
`Sized`*	Live on the stack and has a size known at compilation time.

* Automatically implemented by compiler where appropriate.

String Conversions

If you **want** a string of type ...

`String` | `CString` | `OSString` | `Vec<u8>` | `&str` | `&CStr` | `&[u8]` | `&[u16]` | `*const c_char`

If you have `x` of type ...	Use this ...
`String`	`x.clone()`
`CString`	`x.into_string()?`
`OSString`	`x.to_str()?.into()`
`Vec<u8>`[1]	`String::from_utf8(x)?`
`&str`	`x.into()`
`&CStr`	`x.to_str()?.into()`
`&OSStr`	`x.to_str()?.into()`
`&[u8]`[1]	`String::from_utf8_lossy(x).into()`

[1] You should or must (if `unsafe` calls are involved) ensure the raw data comes with a valid representation for the string type (e.g., being UTF-8 encoded data for a `String`).

Guides

Project Anatomy

Basic project layout, and common files and folders, as used by Rust tooling.

Entry	Code
`benches/`	Benchmarks for your crate, run via `cargo bench`, requires nightly by default. *
`examples/`	Examples how to use your crate, run via `cargo run --example my_example`.
`src/`	Actual source code for your project.

Entry	Code
`build.rs`	Pre-build script, e.g., when compiling C / FFI, needs to be specified in `Cargo.toml`.
`main.rs`	Default entry point for applications, this is what `cargo run` uses.
`lib.rs`	Default entry point for libraries. This is where lookup for `my_crate::f` starts.
🗀 `tests/`	Integration tests go here, invoked via `cargo test`. Unit tests often stay in `src/` file.
`.rustfmt.toml`	In case you want to customize how `cargo fmt` works.
`.clippy.toml`	Special configuration for certain clippy lints.
`Cargo.toml`	Main project configuration. Defines dependencies, artifacts ...
`Cargo.lock`	Dependency details for reproducible builds, recommended to `git` for apps, not for libs.

* On stable consider Criterion.

Idiomatic Rust

If you are used to programming Java or C, consider these.

Idiom	Code
Think in Expressions	`x = if x { a } else { b };`
	`x = loop { break 5 };`
	`fn f() -> u32 { 0 }`
Think in Iterators	`(1..10).map(f).collect()`
	`names.iter().filter(|x| x.starts_with("A"))`
Handle Absence with `?`	`x = try_something()?;`
	`get_option()?.run()?`
Use Strong Types	`enum E { Invalid, Valid { ... } }` over `ERROR_INVALID = -1`
	`enum E { Visible, Hidden }` over `visible: bool`
	`struct Charge(f32)` over `f32`
Provide Builders	`Car::new("Model T").hp(20).run();`
Split Implementations	Generic types `S<T>` can have a separate `impl` per `T`.
	Rust doesn't have OO, but with separate `impl` you can get specialization.
Unsafe	Avoid `unsafe {}`, often safer, faster solution without it. Exception: FFI.
Implement Traits	`#[derive(Debug, Copy, ...)]` and custom `impl` where needed.
Tooling	With **clippy** you can improve your code quality.
	Formatting with **rustfmt** helps others to read your code.
	Add **unit tests** [BK] (`#[test]`) to ensure your code works.
	Add **doc tests** [BK] (```` ``` my_api::f() ``` ````) to ensure docs match code.
Documentation	Annotate your APIs with doc comments that can show up on **docs.rs**.
	Don't forget to include a **summary sentence** and the **Examples** heading.
	If applicable: **Panics**, **Errors**, **Safety**, **Abort** and **Undefined Behavior**.

🔥 We **highly** recommend you also follow the **API Guidelines** (**Checklist**) for any shared project! 🔥

Async-Await 101

If you are familiar with async / await in C# or TypeScript, here are some things to keep in mind:

Construct	Explanation
`async`	Anything declared `async` always returns an `impl Future<Output=_>`. [STD]
`async fn f() {}`	Function `f` returns an `impl Future<Output=()>`.
`async fn f() -> S {}`	Function `f` returns an `impl Future<Output=S>`.
`async { x }`	Transforms `{ x }` into an `impl Future<Output=X>`.
`let sm = f();`	Calling `f()` that is `async` will **not** execute `f`, but produce state machine `sm`. [1]
`sm = async { g() };`	Likewise, does **not** execute the `{ g() }` block; produces state machine.
`runtime.block_on(sm);` [2]	Outside an `async {}`, schedules `sm` to actually run. Would execute `g()`.
`sm.await`	Inside an `async {}`, run `sm` until complete. Yield to runtime if `sm` not ready.

[1] Technically `async` transforms the following code into an anonymous, compiler-generated state machine type, and `f()` instantiates that machine. The state machine always `impl Future`, possibly `Send` & co, depending on types you used inside `async`. State machine driven by worker thread invoking `Future::poll()` via runtime directly, or parent `.await` indirectly.
[2] Right now Rust doesn't come with its own runtime. Use external crate instead, such as async-std or tokio 0.2+. Also, Futures in Rust are an MPV. There is **much** more utility stuff in the futures crate.

At each `x.await`, state machine passes control to subordinate state machine `x`. At some point a low-level state machine invoked via `.await` might not be ready. In that the case worker thread returns all the way up to runtime so it can drive another Future. Some time later the runtime:

- **might** resume execution. It usually does, unless `sm` / `Future` dropped.
- **might** resume with the previous worker **or another** worker thread (depends on runtime).

Simplified diagram for code written inside an `async` block :

```
       consecutive_code();          consecutive_code();          consecutive_code();
START --------------------> x.await --------------------> y.await --------------------> READY
// ^                          ^     ^                            Future<Output=X> ready -^
// Invoked via runtime        |     |
// or an external .await      |     This might resume on another thread (next best available),
//                            |     or NOT AT ALL if Future was dropped.
//                            |
//                           Execute `x`. If ready: just continue execution; if not, return
//                           this thread to runtime.
```

This leads to the following considerations when writing code inside an `async` construct:

Constructs [1]	Explanation
`sleep_or_block();`	Definitely bad 🛑, never halt current thread, clogs executor.
`set_TL(a); x.await; TL();`	Definitely bad 🛑, `await` may return from other thread, thread local invalid.
`s.no(); x.await; s.go();`	Maybe bad 🛑, `await` will not return if `Future` dropped while waiting. [2]
`Rc::new(); x.await; rc();`	Non-`Send` types prevent `impl Future` from being `Send`; less compatible.

[1] Here we assume `s` is any non-local that could temporarily be put into an invalid state; `TL` is any thread local storage, and that the `async {}` containing the code is written without assuming executor specifics.
[2] Since Drop is run in any case when `Future` is dropped, consider using drop guard that cleans up / fixes application state if it has to be left in bad condition across `.await` points.

Closures in APIs

There is a subtrait relationship `Fn` : `FnMut` : `FnOnce`. That means, a closure that implements `Fn`, also implements `FnMut` and `FnOnce`. Likewise, a closure that implements `FnMut`, also implements `FnOnce`.

From a call site perspective that means:

Signature	Function `g` can call ...	Function `g` accepts ...
`g<F: FnOnce()>(f: F)`	... `f()` once.	`Fn`, `FnMut`, `FnOnce`
`g<F: FnMut()>(mut f: F)`	... `f()` multiple times.	`Fn`, `FnMut`
`g<F: Fn()>(f: F)`	... `f()` multiple times.	`Fn`

Notice how **asking** for a `Fn` closure as a function is most restrictive for the caller; but **having** a `Fn` closure as a caller is most compatible with any function.

From the perspective of someone defining a closure:

Closure	Implements*	Comment
`|| { moved_s; }`	`FnOnce`	Caller must give up ownership of `moved_s`.
`|| { &mut s; }`	`FnOnce`, `FnMut`	Allows `g()` to change caller's local state `s`.
`|| { &s; }`	`FnOnce`, `FnMut`, `Fn`	May not mutate state; but can share and reuse `s`.

* Rust prefers capturing by reference (resulting in the most "compatible" `Fn` closures from a caller perspective), but can be forced to capture its environment by copy or move via the `move || {}` syntax.

That gives the following advantages and disadvantages:

Requiring	Advantage	Disadvantage
`F: FnOnce`	Easy to satisfy as caller.	Single use only, `g()` may call `f()` just once.
`F: FnMut`	Allows `g()` to change caller state.	Caller may not reuse captures during `g()`.
`F: Fn`	Many can exist at same time.	Hardest to produce for caller.

A Guide to Reading Lifetimes

Lifetimes can be overwhelming at times. Here is a simplified guide on how to read and interpret constructs containing lifetimes if you are familiar with C.

Construct	How to read
`let s: S = S(0)`	A location that is `S`-sized, named `s`, and contains the value `S(0)`.
	If declared with `let`, that location lives on the stack. [1]
	Generally, `s` can mean *location of* `s`, and *value within* `s`.
	As a location, `s = S(1)` means, assign value `S(1)` to location `s`.
	As a value, `f(s)` means call `f` with value inside of `s`.
	To explicitly talk about its location (address) we do `&s`.
	To explicitly talk about a location that can hold such a location we do `&S`.
`&'a S`	A `&S` is a **location that can hold** (at least) **an address**, called reference.
	Any address stored in here must be that of a valid `S`.
	Any address stored must be proven to exist for at least (*outlive*) duration `'a`.
	In other words, the `&S` part sets bounds for what any address here must contain.
	While the `&'a` part sets bounds for how long any such address must at least live.

Construct	How to read
	The lifetime our containing location is unrelated, but naturally always shorter.
	Duration of `'a` is purely compile time view, based on static analysis.
`&S`	Sometimes `'a` might be elided (or can't be specified) but it still exists.
	Within methods bodies, lifetimes are determined automatically.
	Within signatures, lifetimes may be 'elided' (annotated automatically).
`&s`	This will produce the **actual address of location** `s`, called 'borrow'.
	The moment `&s` is produced, location `s` is put into a **borrowed state**.
	Checking if in borrowed state is based on compile-time analysis.
	This analysis is based on all possible address propagation paths.
	As long as **any** `&s` could be around, `s` cannot be altered directly.
	For example, in `let a = &s; let b = a;`, also `b` needs to go.
	Borrowing of `s` stops once last `&s` is last used, not when `&s` dropped.
`&mut s`	Same, but will produce a mutable borrow.
	A `&mut` will allow the *owner of the borrow* (address) to change `s` content.
	This reiterates that not the value in `s`, but location of `s` is borrowed.

[1] Compare Data Structures section above: while true for synchronous code, an `async` 'stack frame' might actually be placed on to the heap by the used async runtime.

When reading function or type signatures in particular:

Construct	How to read
`S<'a> {}`	Signals that `S` will contain* at least one address (i.e., reference).
	`'a` will be determined automatically by the user of this struct.
	`'a` will be chosen as small as possible.
`f<'a>(x: &'a T)`	Signals this function will accept an address (i.e., reference).
`→ &'a S`	... and that it returns one.
	`'a` will be determined automatically by the caller.
	`'a` will be chosen as small as possible.
	`'a` will be picked so that it **satisfies input and output** at call site.
	More importantly, **propagate borrow state** according to lifetime names!
	So while result address with `'a` is used, input address with `'a` is locked.
	Here: while `s` from `let s = f(&x)` is around, `x` counts as 'borrowed'.
`<'a, 'b: 'a>`	The lifetimes declared in `S<>` and `f<>` can also have bounds.
	The `<'a, 'b>` part means the type will handle at least 2 addresses.
	The `'b: 'a` part is a **lifetime bound**, and means `'b` must **outlive** `'a`.
	Any address in an `&'b X` must exist at least as long as any in an `&'a Y`.

* Technically the struct may not hold any data (e.g., when using the `'a` only for PhantomData or function pointers) but still make use of the `'a` for communicating and requiring that some of its functions require reference of a certain lifetime.

Invisible Sugar

If something works that "shouldn't work now that you think about it", it might be due to one of these.

Name	Description
Coercions NOM	'Weaken' types to match signature, e.g., `&mut T` to `&T`.
Deref NOM	Deref `x: T` until `*x`, `**x`, ... compatible with some target `S`.
Prelude STD	Automatic import of basic types.
Reborrow	Since `x: &mut T` can't be copied; move new `&mut *x` instead.
Lifetime Elision BK NOM REF	Automatically annotate `f(x: &T)` to `f<'a>(x: &'a T)`.
Method Resolution REF	Deref or borrow `x` until `x.f()` works.

Unsafe, Unsound, Undefined

Unsafe leads to unsound. Unsound leads to undefined. Undefined leads to the dark side of the force.

Unsafe Code | Undefine Behavior | Unsound Code

Unsafe Code

- Code marked `unsafe` has special permissions, e.g., to deref raw pointers, or invoke other `unsafe` functions.
- Along come special **promises the author *must* uphold to the compiler**, and the compiler *will* trust you.
- By itself `unsafe` code is not bad, but dangerous, and needed for FFI or exotic data structures.

```
// `x` must always point to race-free, valid, aligned, initialized u8 memory.
unsafe fn unsafe_f(x: *mut u8) {
    my_native_lib(x);
}
```

Responsible use of Unsafe

- Do not use `unsafe` unless you absolutely have to.
- Follow the Nomicon, Unsafe Guidelines, **always** uphold **all** safety invariants, and **never** invoke UB.
- Minimize the use of `unsafe` and encapsulate it in the small, sound modules that are easy to review.
- Each `unsafe` unit should be accompanied by plain-text reasoning outlining its safety.

Formatting Strings

Formatting applies to `print!`, `eprint!`, `write!` (and their `-ln` siblings like `println!`). Each format argument is either empty `{}`, `{argument}`, or follows a basic **syntax**:

```
{ [argument] ':' [[fill] align] [sign] ['#'] [width [$]] ['.' precision [$]] [type] }
```

Element	Meaning

Element	Meaning
`argument`	Number (`0`, `1`, ...) or argument name, e.g., `print!("{x}", x = 3)`.
`fill`	The character to fill empty spaces with (e.g., `0`), if `width` is specified.
`align`	Left (`<`), center (`^`), or right (`>`), if width is specified.
`sign`	Can be `+` for sign to always be printed.
`#`	Alternate formatting, e.g. prettify Debug `?` or prefix hex with `0x`.
`width`	Minimum width (≥ 0), padding with `fill` (default to space). If starts with `0`, zero-padded.
`precision`	Decimal digits (≥ 0) for numerics, or max width for non-numerics.
`$`	Interpret `width` or `precision` as argument identifier instead to allow for dynamic formatting.
`type`	**Debug** (`?`) formatting, hex (`x`), binary (`b`), octal (`o`), pointer (`p`), exp (`e`) ... see more.

Example	Explanation
`{:?}`	Print the next argument using Debug.
`{2:#?}`	Pretty-print the 3rd argument with Debug formatting.
`{val:^2$}`	Center the `val` named argument, width specified by the 3rd argument.
`{:<10.3}`	Left align with width 10 and a precision of 3.
`{val:#x}`	Format `val` argument as hex, with a leading `0x` (alternate format for `x`).

Tooling

Some commands and tools that are good to know.

Command	Description
`cargo init`	Create a new project for the latest edition.
`cargo build`	Build the project in debug mode (`--release` for all optimization).
`cargo check`	Check if project would compile (much faster).
`cargo test`	Run tests for the project.
`cargo run`	Run your project, if a binary is produced (main.rs).
`cargo doc --open`	Locally generate documentation for your code and dependencies.
`cargo rustc -- -Zunpretty=X`	Show more desugared Rust code, in particular with X being:
`expanded`	Show with expanded macros, ...
`cargo +{nightly, stable} ...`	Runs command with given toolchain, e.g., for 'nightly only' tools.
`rustup docs`	Open offline Rust documentation (incl. the books), good on a plane!

A command like `cargo build` means you can either type `cargo build` or just `cargo b`.

These are optional `rustup` components. Install them with `rustup component add [tool]`.

Tool	Description
`cargo clippy`	Additional (lints) catching common API misuses and unidiomatic code.
`cargo fmt`	Automatic code formatter (`rustup component add rustfmt`).

A large number of additional cargo plugins **can be found here**.