Rust Language Cheat Sheet

12. August 2025

Contains clickable links to **The Book**,BK **Rust by Example**,EX **Std Docs**,STD **Nomicon**,NOM and **Reference**.REF ✚

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 }`	Enum with explicit **discriminant values**, REF e.g., for FFI.
`enum E {}`	Enum w/o variants is **uninhabited**, REF can't be created, *c.* 'never' ↓ ⚲
`union U {}`	Unsafe C-like **union** REF for FFI compatibility. ⚲
`static X: T = T();`	**Global variable** BK EX REF with `'static` lifetime, single 🔴[1] 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[2] bound as `x`. Assignable once, not mutable.
`let mut x: T;`	Like `let`, but allow for **mutability** BK EX and mutable borrow.[3]
`x = y;`	Moves `y` to `x`, inval. `y` if `T` is not **`Copy`**, STD and copying `y` otherwise.

[1] In *libraries* you might secretly end up with multiple instances of `X`, depending on how your crate is imported. 🖉
[2] **Bound variables** BK EX REF live on stack for synchronous code. In `async {}` they become part of async's state machine, may reside on heap.
[3] Technically *mutable* and *immutable* are misnomer. Immutable binding or shared reference may still contain Cell STD, giving *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::default()`. STD
`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`.
`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 unspec. length, i.e., **slice**. EX STD REF Can't live on stack. *
`[S; n]`	**Array type** EX STD REF of fixed length `n` holding elements of type `S`.
`[x; n]`	**Array instance** REF (expression) with `n` copies of `x`.
`[x, y]`	Array instance with given elements `x` and `y`.
`x[0]`	Collection indexing, here w. `usize`. Impl. via **Index**, **IndexMut**.
`x[..]`	Same, via range (here *full range*), also `x[a..b]`, `x[a..=b]`, … *c.* below.
`a..b`	**Right-exclusive range** STD REF creation, e.g., `1..3` means `1, 2`.
`..b`	Right-exclusive **range to** STD without starting point.
`..=b`	**Inclusive range to** STD without starting point.
`a..=b`	**Inclusive range**, STD `1..=3` means `1, 2, 3`.
`a..`	**Range from** STD without ending point.

Example	Explanation
`..`	**Full range**, STD usually means *the whole collection*.
`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, RFC pending completion of tracking issue.

References & Pointers

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

Example	Explanation
`&S`	Shared **reference** BK STD NOM REF (type; space for holding *any* `&s`).
`&[S]`	Special slice reference that contains (`addr`, `count`).
`&str`	Special string slice reference that contains (`addr`, `byte_len`).
`&mut S`	Exclusive reference to allow mutability (also `&mut [S]`, `&mut dyn S`, ...).
`&dyn T`	Special **trait object** BK REF ref. as (`addr`, `vtable`); `T` must be **object safe**. REF
`&s`	Shared **borrow** BK EX STD (e.g., addr., len, vtable, ... of *this* `s`, like `0x1234`).
`&mut s`	Exclusive borrow that allows **mutability**. EX
`*const S`	Immutable **raw pointer type** BK STD REF w/o memory safety.
`*mut S`	Mutable raw pointer type w/o memory safety.
`&raw const s`	Create raw pointer w/o going through ref.; *c.* `ptr:addr_of!()` STD 🝖
`&raw mut s`	Same, but mutable. 🚧 Needed for unaligned, packed fields. 🝖
`ref s`	**Bind by reference**, EX makes binding reference type. 🗑
`let ref r = s;`	Equivalent to `let r = &s`.
`let S { ref mut x } = s;`	Mut. ref binding (`let x = &mut s.x`), shorthand destructuring ↓ version.
`*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 = *r;`	Won't work 🛑 if `*r` is not `Copy`, as that would move and leave empty.
`s = *my_box;`	Special case💊 for `Box`STD that can move out b'ed content not `Copy`.
`'a`	A **lifetime parameter**, BK EX NOM REF duration of a flow in static analysis.
`&'a S`	Only accepts address of some `s`; address existing `'a` or longer.
`&'a mut S`	Same, but allow address content to be changed.
`struct S<'a> {}`	Signals this `S` will contain address with lt. `'a`. Creator of `S` decides `'a`.
`trait T<'a> {}`	Signals any `S`, which `impl T for S`, might contain address.
`fn f<'a>(t: &'a T)`	Signals this function handles some address. 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 types can adhere to.
`trait T : R {}`	`T` is subtrait of **supertrait** BK EX 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`; specifies *how exactly* `S` acts like `T`.
`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**, BK EX REF e.g., within an `impl S {}`.
`struct S(T);`	More arcanely, *also*↑ defines `fn S(x: T) -> S` **constructor fn**. RFC 🝖
`const fn f() {}`	Constant `fn` usable at compile time, e.g., `const X: u32 = f(Y)`. REF '18
`const { x }`	Used within a function, ensures `{ x }` evaluated during compilation. REF
`async fn f() {}`	**Async** REF '18 function transform, ↓ 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>`. REF
`async move { x }`	Moves captured variables into future, *c.* move closure. REF ↓
`fn() -> S`	**Function references**, 1 BK STD REF memory holding address of a callable.
`Fn() -> S`	**Callable trait** BK STD (also `FnMut`, `FnOnce`), impl. by closures, fn's ...

Example	Explanation
`AsyncFn() -> S`	**Callable async trait** STD (also `AsyncFnMut`, `AsyncFnOnce`), impl. by async *c.*
`|| {}`	A **closure** BK EX REF that borrows its **captures**, ↓ REF (e.g., a local variable).
`|x| {}`	Closure accepting one argument named `x`, body is block expression.
`|x| x + x`	Same, without block expression; may only consist of single expression.
`move |x| x + y`	**Move closure** REF taking ownership; i.e., `y` transferred into closure.
`async |x| x + x`	**Async closure.** REF Converts its result into an `impl Future<Output=X>`.
`async move |x| x + y`	**Async move closure**. Combination of the above.
`return || true`	Closures sometimes look like logical ORs (here: return a closure).
`unsafe`	If you enjoy debugging segfaults; **unsafe code.** ↓ BK EX NOM REF
`unsafe fn f() {}`	Means "*calling can cause UB,* ↓ ***YOU must check*** *requirements*".
`unsafe trait T {}`	Means "*careless impl. of `T` can cause UB*; **implementor must check**".
`unsafe { f(); }`	Guarantees to compiler "***I have checked*** *requirements, trust me*".
`unsafe impl T for S {}`	Guarantees *`S` is well-behaved w.r.t `T`*; people may use `T` on `S` safely.
`unsafe extern "abi" {}`	Starting with Rust 2024 `extern "abi" {}` blocks ↓ must be `unsafe`.
`pub safe fn f();`	Inside an `unsafe extern "abi" {}`, mark `f` is actually safe to call. RFC

[1] Most documentation calls them function **pointers**, but function **references** might be more appropriate🔗 as they can't be `null` and must point to valid target.

Control Flow

Control execution within a function.

Example	Explanation
`while x {}`	**Loop**, REF run while expression `x` is true.
`loop {}`	**Loop indefinitely** REF until `break`. Can yield value with `break x`.
`for x in collection {}`	Syntactic sugar to loop over **iterators**. BK STD REF
`↪ collection.into_iter()`	Effectively converts any **`IntoIterator`** STD type into proper iterator first.
`↪ iterator.next()`	On proper **`Iterator`** STD then `x = next()` until exhausted (first `None`).
`if x {} else {}`	**Conditional branch** REF if expression is true.
`'label: {}`	**Block label**, RFC can be used with `break` to exit out of this block. 1.65+
`'label: loop {}`	Similar **loop label**, EX REF useful for flow control in nested loops.
`break`	**Break expression** REF to exit a labelled block or loop.
`break 'label x`	Break out of block or loop named `'label` and make `x` its value.
`break 'label`	Same, but don't produce any value.
`break x`	Make `x` value of the innermost loop (only in actual `loop`).
`continue`	**Continue expression** REF to the next loop iteration of this loop.
`continue 'label`	Same but instead of this loop, enclosing loop marked with 'label.
`x?`	If `x` is Err or None, **return and propagate**. BK EX STD REF
`x.await`	Syntactic sugar to **get future, poll, yield**. REF '18 Only inside `async`.
`↪ x.into_future()`	Effectively converts any **`IntoFuture`** STD type into proper future first.
`↪ future.poll()`	On proper **`Future`** STD then `poll()` and yield flow if **`Poll::Pending`**. STD
`return x`	**Early return** REF from fn. More idiomatic is to end with expression.
`{ return }`	Inside normal `{}`-blocks `return` exits surrounding function.
`|| { return }`	Within closures `return` exits that *c.* only, i.e., closure is *s. fn.*
`async { return }`	Inside `async` a `return` **only** REF 🔴 exits that `{}`, i.e., `async {}` is *s. fn.*
`f()`	Invoke callable `f` (e.g., a function, closure, function pointer, `Fn`, ...).
`x.f()`	Call member fn, 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` in **crate root** '15 REF or **ext. prelude**; '18 REF **global path**. REF 🗑
`crate::b`	Search `b` in crate root. '18
`self::b`	Search `b` in current module.
`super::b`	Search `b` in 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` anon. into scope, useful for traits with conflicting names.
`use a::*;`	Bring everything from `a` in, only recomm. if `a` is some **prelude**. STD 🔗
`pub use a::b;`	Bring `a::b` into scope and reexport from here.
`pub T`	"Public if parent path is public" **visibility** BK REF for `T`.
`pub(crate) T`	Visible at most[1] in current crate.
`pub(super) T`	Visible at most[1] in parent.
`pub(self) T`	Visible at most[1] in current module (default, same as no `pub`).
`pub(in a::b) T`	Visible at most[1] in ancestor `a::b`.
`extern crate a;`	Declare dependency on external **crate**; BK 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.

[1] Items in child modules always have access to any item, regardless if `pub` or not.

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** BK REF in `fn f(self) {}`, e.g., akin to `fn f(self: Self) {}`.
`&self`	Same, but refers to self as borrowed, would equal `f(self: &Self)`
`&mut self`	Same, but mutably borrowed, would equal `f(self: &mut Self)`
`self: Box<Self>`	**Arbitrary self type**, add methods to smart ptrs (`my_box.f_of_self()`).
`<S as T>`	**Disambiguate** BK REF type `S` as trait `T`, e.g., `<S as T>::f()`.
`a::b as c`	In `use` of symbol, import `S` as `R`, e.g., `use a::S as R`.
`x as u32`	Primitive **cast**, EX REF may truncate and be a bit surprising. [1] NOM

[1] See **Type Conversions** below for all the ways to convert between types.

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).
`#[attr]`	Outer **attribute**, EX REF annotating the following item.
`#![attr]`	Inner attribute, annotating the *upper*, surrounding item.

Inside Macros [1]	Explanation
`$x:ty`	Macro capture, the `:ty` **fragment specifier** REF ,[2] declares what `$x` may be.
`$x`	Macro substitution, e.g., use the captured `$x:ty` from above.
`$(x),*`	Macro **repetition** REF *zero or more times*.
`$(x),+`	Same, but *one or more times*.
`$(x)?`	Same, but *zero or one time* (separator doesn't apply).
`$(x)<<+`	In fact separators other than `,` are also accepted. Here: `<<`.

[1] Applies to **'macros by example'**. REF

[2] See **Tooling Directives** below for all fragment specifiers.

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 **destructures** EX 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 (.., a, b) = (1, 2);`	Specific bindings take precedence over 'the rest', here `a` is `1`, `b` is `2`.
`let s @ S { x } = get();`	Bind `s` to `S` while `x` is bnd. to `s.x`, **pattern binding**, BK EX REF *c.* below
`let w @ t @ f = get();`	Stores 3 copies of `get()` result in each `w`, `t`, `f`.
`let (|x| x) = get();`	Pathological or-pattern,↓ **not** closure.🛑 Same as `let x = get();`
`let Ok(x) = f();`	**Won't** work 🛑 if p. can be **refuted**, REF use `let else` or `if let` instead.
`let Ok(x) = f();`	But can work if alternatives uninhabited, e.g., `f` returns `Result<T, !>` 1.82+
`let Ok(x) = f() else {};`	Try to assign RFC if not `else {}` w. must `break`, `return`, `panic!`, … 1.65+ 🔥
`if let Ok(x) = f() {}`	Branch if pattern can be assigned (e.g., `enum` variant), syntactic sugar. *
`if let … && let … { }`	**Let chains**, REF use more than binding w.o. nesting. '24
`while let Ok(x) = f() {}`	Equiv.; here keep calling `f()`, run `{}` as long as *p.* can be assigned.
`fn f(S { x }: S)`	Function param. also work like `let`, here `x` bound to `s.x` of `f(s)`.

* Desugars to `match get() { Some(x) => {}, _ => () }`.

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

Within Match Arm	Explanation
`E::A => {}`	Match enum variant `A`, *c.* **pattern matching**. BK EX REF
`E::B ( .. ) => {}`	Match enum tuple variant `B`, ignoring any index.
`E::C { .. } => {}`	Match enum struct variant `C`, ignoring any field.
`S { x: 0, y: 1 } => {}`	Match *s.* with specific values (only `s` with `s.x` of `0` and `s.y` of `1`).
`S { x: a, y: b } => {}`	Match *s.* 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".
`0 | 1 => {}`	Pattern alternatives, **or-patterns**. RFC
`E::A | E::Z => {}`	Same, but on enum variants.
`E::C {x} | E::D {x} => {}`	Same, but bind `x` if all variants have it.
`Some(A | B) => {}`	Same, can also match alternatives deeply nested.
`|x| x => {}`	**Pathological or-pattern**,↑🛑 leading `|` ignored, is just `x | x`, thus `x`.
`|x => {}`	Similar, leading `|` ignored.
`(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**. REF RFC
`[1, .., 5] => {}`	Match array starting with `1`, ending with `5`.
`[1, x @ .., 5] => {}`	Same, but also bind `x` to slice representing middle (*c.* pattern binding).
`[a, x @ .., b] => {}`	Same, but match any first, last, bound as `a`, `b` respectively.
`1 .. 3 => {}`	**Range pattern**, BK REF here matches `1` and `2`; partially unstable. 🚧
`1 ..= 3 => {}`	Inclusive range pattern, matches `1`, `2` and `3`.
`1 .. => {}`	Open range pattern, matches `1` and any larger number.
`x @ 1..=5 => {}`	Bind matched to `x`; **pattern binding**, BK EX REF here `x` would be `1` … `5`.
`Err(x @ Error {..}) => {}`	Also works nested, here `x` binds to `Error`, esp. useful with `if` below.
`S { x } if x > 10 => {}`	Pattern **match guards**, BK EX REF condition must be true as well to match.

Generics & Constraints

Generics combine with type constructors, traits and functions to give your users more flexibility.

Example	Explanation
`struct S<T> …`	A **generic** BK EX type with a type parameter (`T` is placeholder here).
`S<T> where T: R`	**Trait bound**, BK EX REF limits allowed `T`, guarantees `T` has trait `R`.
`where T: R, P: S`	**Independent trait bounds**, here one for `T` and one for (not shown) `P`.
`where T: R, S`	Compile error, 🛑 you probably want compound bound `R + S` below.

Example	Explanation
`where T: R + S`	**Compound trait bound**, BK EX `T` must fulfill `R` and `S`.
`where T: R + 'a`	Same, but w. lifetime. `T` must fulfill `R`, if `T` has *lt.*, must outlive `'a`.
`where T: ?Sized`	Opt out of a pre-defined trait bound, here `Sized`. ?
`where T: 'a`	Type **lifetime bound**; EX if T has references, they must outlive `'a`.
`where T: 'static`	Same; does *not* mean value `t` *will* 🛑 live `'static`, only that it could.
`where 'b: 'a`	Lifetime `'b` must live at least as long as (i.e., *outlive*) `'a` bound.
`where u8: R<T>`	Can also make conditional statements involving *other* types. 🝖
`S<T: R>`	Short hand bound, almost same as above, shorter to write.
`S<const N: usize>`	**Generic const bound**; REF user of type `S` can provide constant value `N`.
`S<10>`	Where used, const bounds can be provided as primitive values.
`S<{5+5}>`	Expressions must be put in curly brackets.
`S<T = R>`	**Default parameters**; BK makes `S` a bit easier to use, but keeps flexible.
`S<const N: u8 = 0>`	Default parameter for constants; e.g., in `f(x: S) {}` param `N` is `0`.
`S<T = u8>`	Default parameter for types, e.g., in `f(x: S) {}` param `T` is `u8`.
`S<'_>`	Inferred **anonymous lt.**; 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>()`.
`E::<T>::A`	Generic enums can receive their type parameters on their type `E` ...
`E::A::<T>`	... or at the variant (`A` here); allows `Ok::<R, E>(r)` and similar.
`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 RFC `X`. Only one `impl T for S` possible.
`trait T { type X<G>; }`	Defines **generic associated type** (GAT), RFC `X` can be generic `Vec<>`.
`trait T { type X<'a>; }`	Defines a GAT generic over a lifetime.
`type X = R;`	Set associated type within `impl T for S { type X = R; }`.
`type X<G> = R<G>;`	Same for GAT, e.g., `impl T for S { type X<G> = Vec<G>; }`.
`impl<T> S<T> {}`	Impl. `fn`'s for any `T` in `S<T>` ***generically***, REF here `T` ty. parameter.
`impl S<T> {}`	Impl. `fn`'s for exactly `S<T>` ***inherently***, REF here `T` specific type, e.g., `u8`.
`fn f() -> impl T`	**Existential types** (aka *RPIT*), BK returns an unknown-to-caller `S` that `impl T`.
`-> impl T + 'a`	Signals the hidden type lives at least as long as `'a`. RFC
`-> impl T + use<'a>`	Signals instead the hidden type captured lifetime `'a`, **use bound**. 🔗 ?
`-> impl T + use<'a, R>`	Also signals the hidden type may have captured lifetimes from `R`.
`-> S<impl T>`	The `impl T` part can also be used inside type arguments.
`fn f(x: &impl T)`	Trait bound via "**impl traits**", BK similar to `fn f<S: T>(x: &S)` below.
`fn f(x: &dyn T)`	Invoke `f` via **dynamic dispatch**, BK REF `f` will not be instantiated for `x`.
`fn f<X: T>(x: X)`	Fn. generic over `X`, `f` will be instantiated ('monomorphized') per `X`.
`fn f() where Self: R;`	In `trait T {}`, make `f` accessible only on types known to also `impl R`.
`fn f() where Self: Sized;`	Using `Sized` can opt `f` out of trait object vtable, enabling `dyn T`.
`fn f() where Self: R {}`	Other `R` useful w. dflt. fn. (non dflt. would need be impl'ed anyway).

Higher-Ranked Items 🝖

Actual types and traits, abstract over something, usually lifetimes.

Example	Explanation
`for<'a>`	Marker for **higher-ranked bounds.** NOM REF 🝖
`trait T: for<'a> R<'a> {}`	Any `S` that `impl T` would also have to fulfill `R` for any lifetime.
`fn(&'a u8)`	Function pointer type holding fn callable with **specific** lifetime `'a`.
`for<'a> fn(&'a u8)`	**Higher-ranked type**[1] 🔗 holding fn call. with **any** *lt.*; subtype↓ of above.
`fn(&'_ u8)`	Same; automatically expanded to type `for<'a> fn(&'a u8)`.
`fn(&u8)`	Same; automatically expanded to type `for<'a> fn(&'a u8)`.
`dyn for<'a> Fn(&'a u8)`	Higher-ranked (trait-object) type, works like `fn` above.
`dyn Fn(&'_ u8)`	Same; automatically expanded to type `dyn for<'a> Fn(&'a u8)`.
`dyn Fn(&u8)`	Same; automatically expanded to type `dyn for<'a> Fn(&'a u8)`.

[1] Yes, the `for<>` is part of the type, which is why you write `impl T for for<'a> fn(&'a u8)` below.

Implementing Traits	Explanation
`impl<'a> T for fn(&'a u8) {}`	For fn. pointer, where call accepts **specific** *lt.* `'a`, impl trait `T`.
`impl T for for<'a> fn(&'a u8) {}`	For fn. pointer, where call accepts **any** *lt.*, impl trait `T`.

Implementing Traits	Explanation
`impl T for fn(&u8) {}`	Same, short version.

Strings & Chars

Rust has several ways to create textual values.

Example	Explanation
`"..."`	**String literal**, REF, 1 a UTF-8 `&'static str`, STD supporting these escapes:
`"\n\r\t\0\\"`	**Common escapes** REF, e.g., `"\n"` becomes *new line*.
`"\x36"`	**ASCII *e.*** REF up to `7f`, e.g., `"\x36"` would become `6`.
`"\u{7fff}"`	**Unicode *e.*** REF up to 6 digits, e.g., `"\u{7fff}"` becomes 翿.
`r"..."`	**Raw string literal**. REF, 1UTF-8, but won't interpret any escape above.
`r#"..."#`	Raw string literal, UTF-8, but can also contain `"`. Number of `#` can vary.
`c"..."`	**C string literal**, REF a NUL-terminated `&'static CStr`, STD for FFI. 1.77+
`cr"..."`, `cr#"..."#`	Raw C string literal, combination analog to above.
`b"..."`	**Byte string literal**; REF, 1 constructs ASCII-only `&'static [u8; N]`.
`br"..."`, `br#"..."#`	Raw byte string literal, combination analog to above.
`b'x'`	ASCII **byte literal**, REF a single `u8` byte.
`'🦀'`	**Character literal**, REF fixed 4 byte unicode '**char**'. STD

[1] Supports multiple lines out of the box. Just keep in mind `Debug`↓ (e.g., `dbg!(x)` and `println!("{x:?}")`) might render them as `\n`, while `Display`↓ (e.g., `println!("{x}")`) renders them *proper*.

Documentation

Debuggers hate him. Avoid bugs with this one weird trick.

Example	Explanation
`///`	Outer line **doc comment**,[1] BK EX REF use these on ty., traits, fn's, ...
`//!`	Inner line doc comment, mostly used at top of file.
`//`	Line comment, use these to document code flow or *internals*.
`/* … */`	Block comment. [2] 🗑
`/** … */`	Outer block doc comment. [2] 🗑
`/*! … */`	Inner block doc comment. [2] 🗑

[1] Tooling Directives outline what you can do inside doc comments.

[2] Generally discouraged due to bad UX. If possible use equivalent line comment instead with IDE support.

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
`fn f() -> ! {}`	Function that never ret.; compat. with any *ty.* e.g., `let x: u8 = f();`
`fn f() -> Result<(), !> {}`	Function that must return `Result` but signals it can never `Err`. 🚧
`fn f(x: !) {}`	Function that exists, but can never be called. Not very useful. 🝖 🚧
`_`	Unnamed **wildcard** REF variable binding, e.g., `|x, _| {}`.
`let _ = x;`	Unnamed assign. is no-op, does **not** 🛑 move out `x` or preserve scope!
`_ = x;`	You can assign *anything* to `_` without `let`, i.e., `_ = ignore_rval();` 🔥
`_x`	Variable binding that won't emit *unused variable* warnings.
`1_234_567`	Numeric separator for visual clarity.
`1_u8`	Type specifier for **numeric literals** EX REF (also `i8`, `u16`, ...).
`0xBEEF`, `0o777`, `0b1001`	Hexadecimal (`0x`), octal (`0o`) and binary (`0b`) integer literals.
`12.3e4`, `1E-8`	**Scientific notation** for floating-point literals. REF
`r#foo`	A **raw identifier** BK EX for edition compatibility. 🝖
`'r#a`	A **raw lifetime label** ? for edition compatibility. 🝖
`x;`	**Statement** REF terminator, *c.* **expressions** EX REF

Common Operators

Rust supports most operators you would expect (`+`, `*`, `%`, `=`, `==`, ...), including **overloading**. STD Since they behave no differently in Rust we do not list them here.

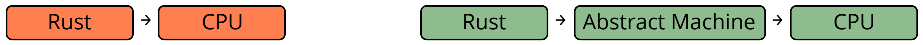
Behind the Scenes

Arcane knowledge that may do terrible things to your mind, highly recommended.

The Abstract Machine

Like `C` and `C++`, Rust is based on an *abstract machine*.

Overview | **Misconceptions**



● Misleading.

Correct.

With rare exceptions you are never 'allowed to reason' about the actual CPU. You write code for an *abstracted* CPU. Rust then (sort of) understands what you want, and translates that into actual RISC-V / x86 / ... machine code.

This *abstract machine*

- is not a runtime, and does not have any runtime overhead, but is a *computing model abstraction*,
- contains concepts such as memory regions (*stack*, ...), execution semantics, ...
- *knows* and *sees* things your CPU might not care about,
- is de-facto a contract between you and the compiler,
- and **exploits all of the above for optimizations**.

Language Sugar

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

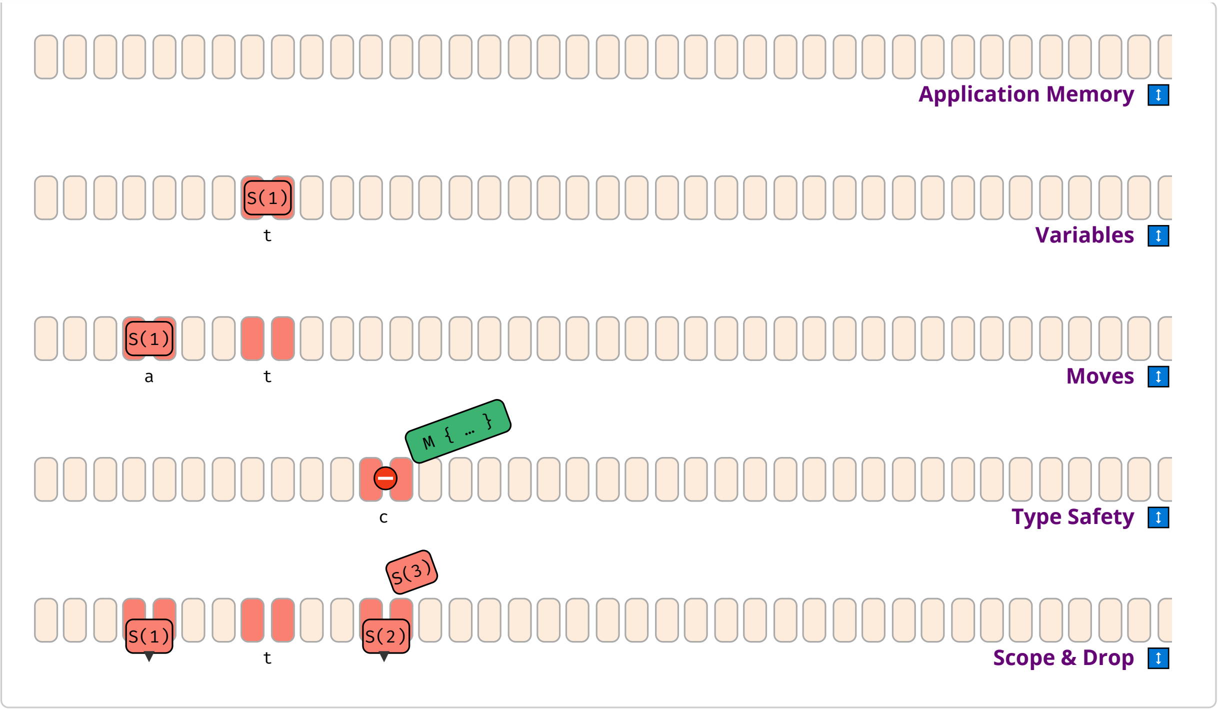
Name	Description
Coercions NOM	*Weakens* types to match signature, e.g., `&mut T` to `&T`; *c. type conv.* ↓
Deref NOM 🔗	Derefs `x: T` until `*x`, `**x`, ... compatible with some target `S`.
Prelude STD	Automatic import of basic items, e.g., `Option`, `drop()`, ...
Reborrow 🔗	Since `x: &mut T` can't be copied; moves new `&mut *x` instead.
Lifetime Elision BK NOM REF	Allows you to write `f(x: &T)`, instead of `f<'a>(x: &'a T)`, for brevity.
Lifetime Extensions 🔗 REF	In `let x = &tmp().f` and similar hold on to temporary past line.
Method Resolution REF	Derefs or borrow `x` until `x.f()` works.
Match Ergonomics RFC	Repeatedly deref. scrutinee and adds `ref` and `ref mut` to bindings.
Rvalue Static Promotion RFC 🝖	Makes refs. to constants `'static`, e.g., `&42`, `&None`, `&mut []`.
Dual Definitions RFC 🝖	Defining one (e.g., `struct S(u8)`) implicitly def. another (e.g., `fn S`).
Drop Hidden Flow REF 🝖	At end of blocks `{ ... }` or `_` assignment, may call `T::drop()`. STD
Drop Not Callable STD 🝖	Compiler forbids explicit `T::drop()` call, must use `mem::drop()`. STD
Auto Traits REF	Always impl'ed for your types, closures, futures if possible.

Opinion 💬 — These features make your life easier *using* Rust, but stand in the way of *learning* it. If you want to develop a *genuine understanding*, spend some extra time exploring them.

Memory & Lifetimes

An illustrated guide to moves, references and lifetimes.

Types & Moves | **Call Stack** | **References & Pointers** | **Lifetime Basics** | **Lifetimes in Functions** | **Advanced** 🝖



Examples expand by clicking.

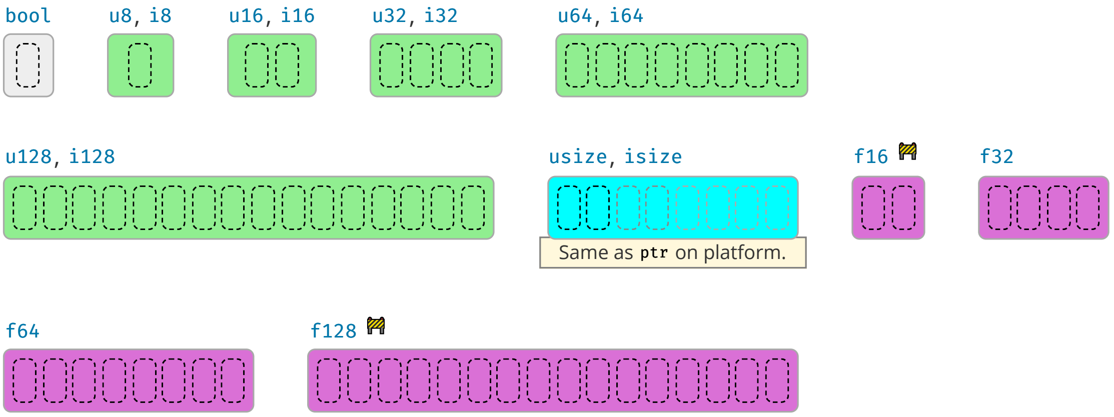
Memory Layout

Byte representations of common types.

Basic Types

Essential types built into the core of the language.

Boolean REF **and Numeric Types** REF



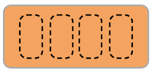
Unsigned Types | Signed Types | Float Types | Float Internals | Casting Pitfalls | Arithmetic Pitfalls

Type	Max Value
u8	255
u16	65_535
u32	4_294_967_295
u64	18_446_744_073_709_551_615
u128	340_282_366_920_938_463_463_374_607_431_768_211_455
usize	Depending on platform pointer size, same as u16, u32, or u64.

[1] Expression `_100` means anything that might contain the value `100`, e.g., `100_i32`, but is opaque to compiler.
[d] Debug build.
[r] Release build.

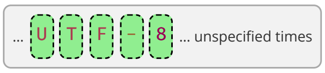
Textual Types REF

`char`



Any Unicode scalar.

`str`



Rarely seen alone, but as `&str` instead.

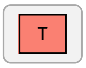
Basics | Usage | Encoding

Type	Description
`char`	Always 4 bytes and only holds a single Unicode **scalar value** 🔗.
`str`	An `u8`-array of unknown length guaranteed to hold **UTF-8 encoded code points**.

Custom Types

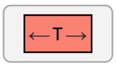
Basic types definable by users. Actual **layout** REF is subject to **representation**; REF padding can be present.

`T`



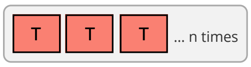
Sized type.

`T: ?Sized`



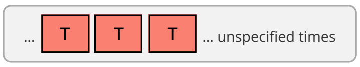
Maybe sized.

`[T; n]`



Fixed array of `n` elements.

`[T]`

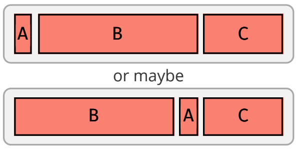


Slice type of unknown-many elements. Neither `Sized` (nor carries `len` information), and most often lives behind reference as `&[T]`. ↓

`struct S;`

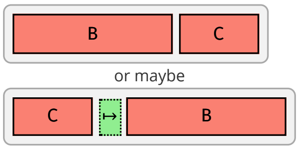
Zero-sized type.

`(A, B, C)`



Unless a representation is forced (e.g., via `#[repr(C)]`), type layout unspecified.

`struct S { b: B, c: C }`

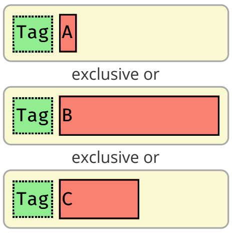


Compiler may also add padding.

Also note, two types `A(X, Y)` and `B(X, Y)` with exactly the same fields can still have differing layout; never `transmute()` STD without representation guarantees.

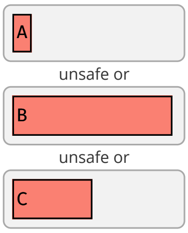
These **sum types** hold a value of one of their sub types:

`enum E { A, B, C }`



Safely holds A or B or C, also called 'tagged union', though compiler may squeeze tag into 'unused' bits.

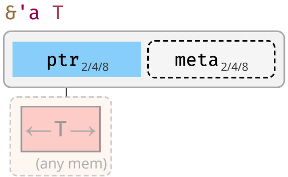
`union { … }`



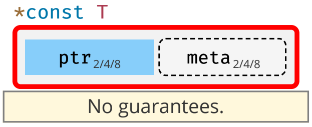
Can unsafely reinterpret memory. Result might be undefined.

References & Pointers

References give safe access to 3rd party memory, raw pointers `unsafe` access. The corresponding `mut` types have an identical data layout to their immutable counterparts.

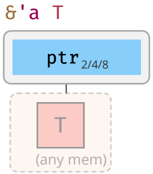


Must target some valid `t` of `T`, and any such target must exist for at least `'a`.

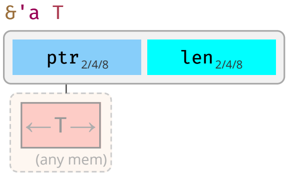


Pointer Meta

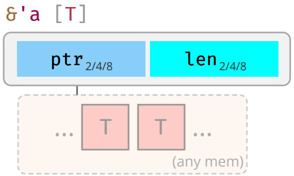
Many reference and pointer types can carry an extra field, **pointer metadata**. STD It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.



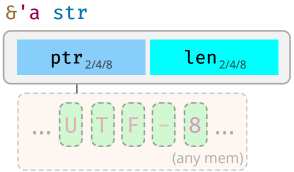
No meta for sized target. (pointer is thin).



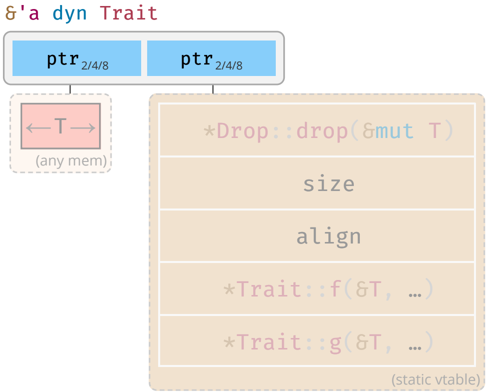
If `T` is a DST `struct` such as `S { x: [u8] }` meta field `len` is count of dyn. sized content.



Regular **slice reference** (i.e., the reference type of a slice type `[T]`) ↑ often seen as `&[T]` if `'a` elided.



String slice reference (i.e., the reference type of string type `str`), with meta `len` being byte length.



Meta points to vtable, where `*Drop::drop()`, `*Trait::f()`, ... are pointers to their respective `impl` for `T`.

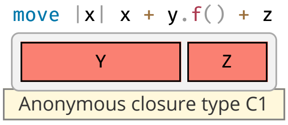
Closures

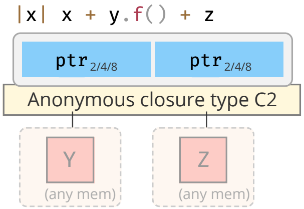
Ad-hoc functions with an automatically managed data block **capturing** REF, [1] environment where closure was defined. For example, if you had:

```
let y = ...;
let z = ...;

with_closure(move |x| x + y.f() + z); // y and z are moved into closure instance (of type C1)
with_closure(     |x| x + y.f() + z); // y and z are pointed at from closure instance (of type C2)
```

Then the generated, anonymous closures types `C1` and `C2` passed to `with_closure()` would look like:





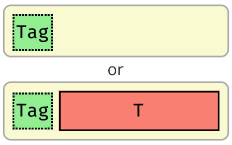
Also produces anonymous `fn` such as f_{c1}`(C1, X)` or f_{c2}`(&C2, X)`. Details depend on which `FnOnce`, `FnMut`, `Fn` ... is supported, based on properties of captured types.

[1] A bit oversimplified a closure is a convenient-to-write 'mini function' that accepts parameters *but also* needs some local variables to do its job. It is therefore a type (containing the needed locals) and a function. *'Capturing the environment'* is a fancy way of saying that and how the closure type holds on to these locals, either *by moved value*, or *by pointer*. See **Closures in APIs** ↓ for various implications.

Standard Library Types

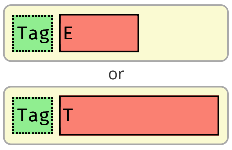
Rust's standard library combines the above primitive types into useful types with special semantics, e.g.:

`Option<T>` STD



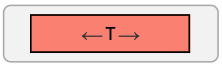
Tag may be omitted for certain T, e.g., `NonNull`.STD

`Result<T, E>` STD



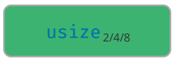
Either some error `E` or value of `T`.

`ManuallyDrop<T>` STD



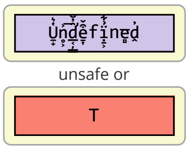
Prevents `T::drop()` from being called.

`AtomicUsize` STD



Other atomic similarly.

`MaybeUninit<T>`STD

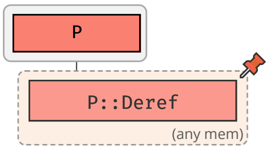


Uninitialized memory or some `T`. Only legal way to work with uninit data.

`PhantomData<T>` STD

Zero-sized helper to hold otherwise unused lifetimes.

`Pin<P>` STD

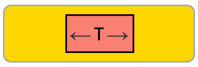


Signals tgt. of `P` is pinned 'forever' even past lt. of `Pin`. Value within may not be moved out (but new one moved in), unless `Unpin`.STD

● All depictions are for **illustrative** purposes only. The fields should exist in latest `stable`, but Rust makes no guarantees about their layouts, and you must not attempt to *unsafely* access anything unless the docs allow it.

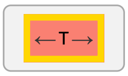
Cells

`UnsafeCell<T>` STD



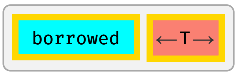
Magic type allowing aliased mutability.

`Cell<T>` STD



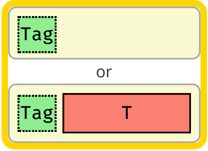
Allows `T`'s to move in and out.

`RefCell<T>` STD



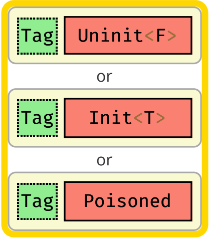
Also support dynamic borrowing of `T`. Like `Cell` this is `Send`, but not `Sync`.

`OnceCell<T>` STD



Initialized at most once.

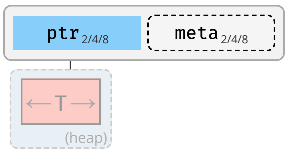
`LazyCell<T, F>` STD



Initialized on first access.

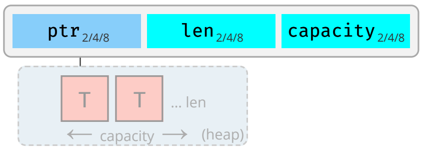
Order-Preserving Collections

`Box<T>` STD



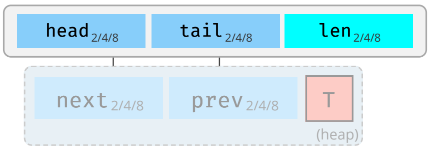
For some `T` stack proxy may carry meta† (e.g., `Box<[T]>`).

`Vec<T>` STD



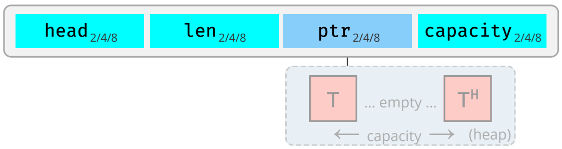
Regular *growable array* vector of single type.

`LinkedList<T>` STD



Elements `head` and `tail` both `null` or point to nodes on the heap. Each node can point to its `prev` and `next` node. Eats your cache (just look at the thing!); don't use unless you evidently must. ●

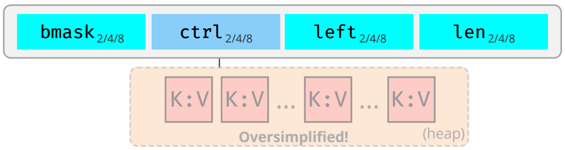
`VecDeque<T>` STD



Index `head` selects in array-as-ringbuffer. This means content may be non-contiguous and empty in the middle, as exemplified above.

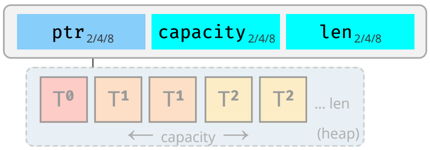
Other Collections

`HashMap<K, V>` STD



Stores keys and values on heap according to hash value, SwissTable implementation via hashbrown. `HashSet` STD identical to `HashMap`, just type `V` disappears. Heap view grossly oversimplified. 🔴

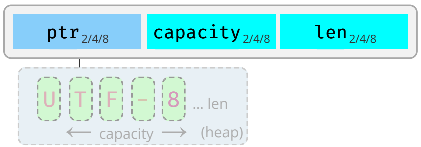
`BinaryHeap<T>` STD



Heap stored as array with 2^N elements per layer. Each `T` can have 2 children in layer below. Each `T` larger than its children.

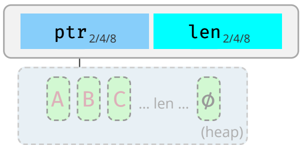
Owned Strings

`String` STD



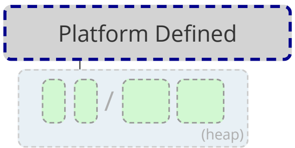
Observe how `String` differs from `&str` and `&[char]`.

`CString` STD



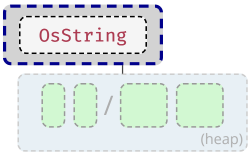
NUL-terminated but w/o NUL in middle.

`OsString` STD



Encapsulates how operating system represents strings (e.g., WTF-8 on Windows).

`PathBuf` STD

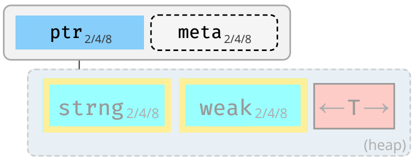


Encapsulates how operating system represents paths.

Shared Ownership

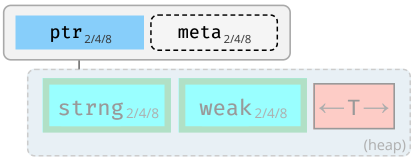
If the type does not contain a `Cell` for `T`, these are often combined with one of the `Cell` types above to allow shared de-facto mutability.

`Rc<T>` STD



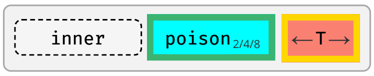
Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell`to allow mutation. Is neither `Send` nor `Sync`.

`Arc<T>` STD



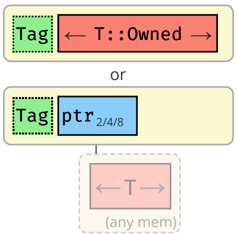
Same, but allow sharing between threads IF contained `T` itself is `Send` and `Sync`.

`Mutex<T>` STD / `RwLock<T>` STD



Inner fields depend on platform. Needs to be held in `Arc` to be shared between decoupled threads, or via `scope()` STD for scoped threads.

`Cow<'a, T>` STD



Holds read-only reference to some `T`, or owns its `ToOwned` STD analog.

Standard Library

One-Liners

Snippets that are common, but still easy to forget. See **Rust Cookbook** 🔗 for more.

Strings | I/O | Macros | Transforms 🔥 | Esoterics 🝖

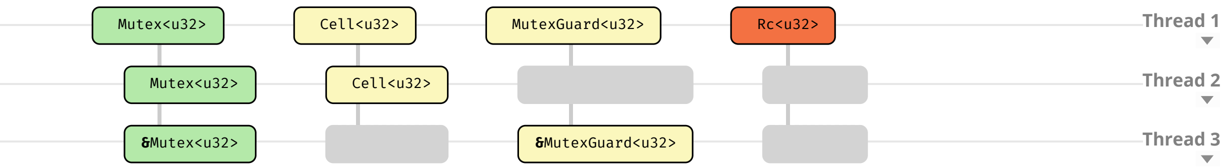
Intent	Snippet
Concatenate strings (any `Display`↓ that is). STD [1] '21	`format!("{x}{y}")`
Append string (any `Display` to any `Write`). '21 STD	`write!(x, "{y}")`
Split by separator pattern. STD 🔗	`s.split(pattern)`
... with `&str`	`s.split("abc")`
... with `char`	`s.split('/')`
... with closure	`s.split(char::is_numeric)`
Split by whitespace. STD	`s.split_whitespace()`
Split by newlines. STD	`s.lines()`
Split by regular expression. 🔗 [2]	`Regex::new(r"\s")?.split("one two three")`

[1] Allocates; if `x` or `y` are not going to be used afterwards consider using `write!` or `std::ops::Add`.

[2] Requires regex crate.

Thread Safety

Assume you hold some variables in Thread 1, and want to either **move** them to Thread 2, or pass their **references** to Thread 3. Whether this is allowed is governed by **`Send`**STD and **`Sync`**STD respectively:



Example	Explanation
`Mutex<u32>`	Both `Send` and `Sync`. You can safely pass or lend it to another thread.
`Cell<u32>`	`Send`, not `Sync`. Movable, but its reference would allow concurrent non-atomic writes.
`MutexGuard<u32>`	`Sync`, but not `Send`. Lock tied to thread, but reference use could not allow data race.
`Rc<u32>`	Neither since it is easily clonable heap-proxy with non-atomic counters.

Trait	`Send`	`!Send`
`Sync`	*Most types* ... `Arc<T>`[1,2], `Mutex<T>`[2]	`MutexGuard<T>`[1], `RwLockReadGuard<T>`[1]
`!Sync`	`Cell<T>`[2], `RefCell<T>`[2]	`Rc<T>`, `&dyn Trait`, `*const T`[3]

[1] If `T` is `Sync`.

[2] If `T` is `Send`.

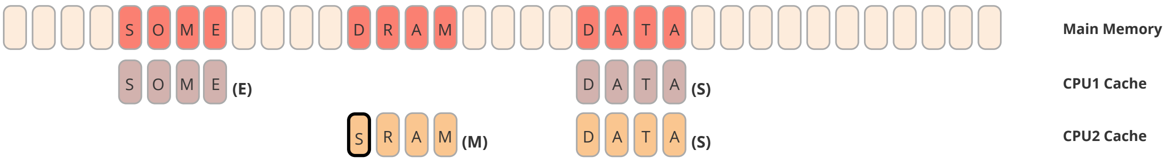
[3] If you need to send a raw pointer, create newtype `struct Ptr(*const u8)` and `unsafe impl Send for Ptr {}`. Just ensure you *may* send it.

When is ...	... Send?
`T`	All contained fields are `Send`, or `unsafe` impl'ed.
`struct S { ... }`	All fields are `Send`, or `unsafe` impl'ed.
`struct S<T> { ... }`	All fields are `Send` and T is `Send`, or `unsafe` impl'ed.
`enum E { ... }`	All fields in all variants are `Send`, or `unsafe` impl'ed.
`&T`	If `T` is `Sync`.
`|| {}`	Closures are `Send` if all *captures* are `Send`.
`|x| { }`	`Send`, regardless of `x`.
`|x| { Rc::new(x) }`	`Send`, since still nothing captured, despite `Rc` not being `Send`.
`|x| { x + y }`	Only `Send` if `y` is `Send`.
`async { }`	Futures are `Send` if no `!Send` is held over `.await` points.
`async { Rc::new() }`	`Future` is `Send`, since the `!Send` type `Rc` is not held over `.await`.
`async { rc; x.await; rc; }` [1]	`Future` is `!Send`, since `Rc` used across the `.await` point.
`async || { }` 🚧	Async *cl.* `Send` if all cpts. `Send`, res. `Future` if also no `!Send` inside.
`async |x| { x + y }` 🚧	Async closure `Send` if `y` is `Send`. Future `Send` if `x` and `y` `Send`.

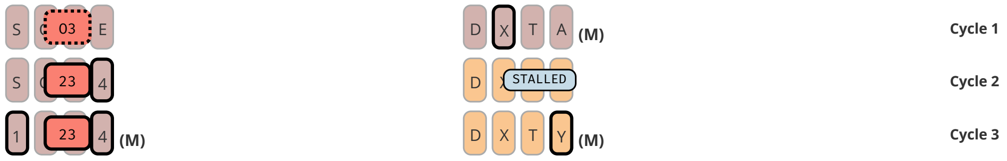
[1] This is a bit of pseudo-code to get the point across, the idea is to have an `Rc` before an `.await` point and keep using it beyond that point.

Atomics & Cache 🝖

CPU cache, memory writes, and how atomics affect it.

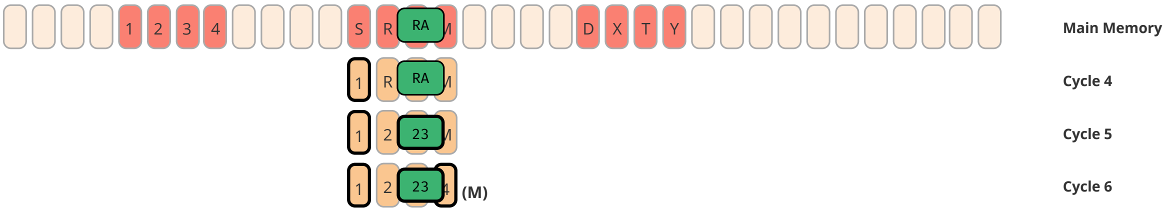


Modern CPUs don't accesses memory directly, only their cache. Each CPU has its own cache, 100x faster than RAM, but much smaller. It comes in **cache lines**,🔗 some *sliced* window of bytes, which track if it's an exclusive (E), shared (S) or modified (M) 🔗 view of the main memory. Caches talk to each other to ensure **coherence**,🔗 i.e., 'small-enough' data will be 'immediately' seen by all other CPUs, but that may stall the CPU.



Left: Both compiler *and* CPUs are free to **re-order** 🔗 and split R/W memory access. Even if you explicitly said `write(1); write(23); write(4)`, your compiler might think it's a good idea to write `23` first; in addition your CPU might insist on splitting the write, doing `3` before `2`. Each of these steps could be observable (even the *impossible* `03`) by CPU2 via an `unsafe` *data race*. Reordering is also fatal for locks.

Right: Semi-related, even when two CPUs do not attempt to access each other's data (e.g., update 2 independent variables), they might still experience a significant performance loss if the underlying memory is mapped by 2 cache lines (**false sharing**).🔗



Atomics address the above issues by doing two things, they

- make sure a read / write / update is not partially observable by temporarily locking cache lines in other CPUs,
- force both the compiler and the CPU to not re-order *'unrelated'* access around it (i.e., act as a **fence** STD). Ensuring multiple CPUs agree on the relative order of these other ops is called **consistency**. 🔗 This also comes at a cost of missed performance optimizations.

Note — The above section is greatly simplified. While the issues of coherence and consistency are universal, CPU architectures differ a lot in how they implement caching and atomics, and in their performance impact.

A. Ordering	Explanation
`Relaxed` STD	Full reordering. Unrelated R/W can be freely shuffled around the atomic.
`Release` STD, 1	When writing, ensure other data loaded by 3rd party `Acquire` is seen after this write.
`Acquire` STD, 1	When reading, ensures other data written before 3rd party `Release` is seen after this read.
`SeqCst` STD	No reordering around atomic. All unrelated reads and writes stay on proper side.

[1] To be clear, when synchronizing memory access with 2+ CPUs, *all* must use `Acquire` or `Release` (or stronger). The writer must ensure that all other data it wishes to *release* to memory are put before the atomic signal, while the readers who wish to *acquire* this data must ensure that their other reads are only done after the atomic signal.

Iterators

Processing elements in a collection.

Basics | **Obtaining** | **Creating** | **For Loops** | **Borrowing** | **Interoperability**

There are, broadly speaking, four *styles* of collection iteration:

Style	Description
`for x in c { ... }`	*Imperative*, useful w. side effects, interdepend., or need to break flow early.
`c.iter().map().filter()`	*Functional*, often much cleaner when only results of interest.
`c_iter.next()`	*Low-level*, via explicit `Iterator::next()` STD invocation.
`c.get(n)`	*Manual*, bypassing official iteration machinery.

Opinion 💬 — Functional style is often easiest to follow, but don't hesitate to use `for` if your `.iter()` chain turns messy. When implementing containers iterator support would be ideal, but when in a hurry it can sometimes be more practical to just implement `.len()` and `.get()` and move on with your life.

Number Conversions

As-**correct**-as-it-currently-gets number conversions.

↓ Have / Want →	`u8 … i128`	`f32 / f64`	String
`u8 … i128`	`u8::try_from(x)?` [1]	`x as f32` [3]	`x.to_string()`
`f32 / f64`	`x as u8` [2]	`x as f32`	`x.to_string()`
`String`	`x.parse::<u8>()?`	`x.parse::<f32>()?`	`x`

[1] If type true subset `from()` works directly, e.g., `u32::from(my_u8)`.
[2] Truncating (`11.9_f32 as u8` gives `11`) and saturating (`1024_f32 as u8` gives `255`); *c.* below.
[3] Might misrepresent number (`u64::MAX as f32`) or produce `Inf` (`u128::MAX as f32`).

Also see **Casting-** and **Arithmetic Pitfalls** ↑ for more things that can go wrong working with numbers.

String Conversions

If you **want** a string of type …

`String` `CString` `OsString` `PathBuf` `Vec<u8>` `&str` `&CStr` `&OsStr` `&Path` `&[u8]` **Other**

If you have `x` of type …	Use this …
`String`	`x`
`CString`	`x.into_string()?`
`OsString`	`x.to_str()?.to_string()`
`PathBuf`	`x.to_str()?.to_string()`
`Vec<u8>` [1]	`String::from_utf8(x)?`
`&str`	`x.to_string()` [i]
`&CStr`	`x.to_str()?.to_string()`
`&OsStr`	`x.to_str()?.to_string()`
`&Path`	`x.to_str()?.to_string()`
`&[u8]` [1]	`String::from_utf8_lossy(x).to_string()`

[i] Short form `x.into()` possible if type can be inferred.
[r] Short form `x.as_ref()` possible if type can be inferred.
[1] You must ensure `x` comes with a valid representation for the string type (e.g., UTF-8 data for a `String`).
[2] The `c_char` **must** have come from a previous `CString`. If it comes from FFI see `&CStr` instead.
[3] No known shorthand as `x` will lack terminating `0x0`. Best way to probably go via `CString`.
[4] Must ensure `x` actually ends with `0x0`.

String Output

How to convert types into a `String`, or output them.

APIs **Printable Types** **Formatting**

Rust has, among others, these APIs to convert types to stringified output, collectively called *format* macros:

Macro	Output	Notes
`format!(fmt)`	`String`	Bread-and-butter "to `String`" converter.
`print!(fmt)`	Console	Writes to standard output.
`println!(fmt)`	Console	Writes to standard output.
`eprint!(fmt)`	Console	Writes to standard error.
`eprintln!(fmt)`	Console	Writes to standard error.
`write!(dst, fmt)`	Buffer	Don't forget to also `use std::io::Write;`
`writeln!(dst, fmt)`	Buffer	Don't forget to also `use std::io::Write;`

Method	Notes
`x.to_string()` STD	Produces `String`, implemented for any `Display` type.

Here `fmt` is string literal such as `"hello {}"`, that specifies output (compare "Formatting" tab) and additional parameters.

Tooling

Project Anatomy

Basic project layout, and common files and folders, as used by `cargo`. ↓

Entry	Code
📁 `.cargo/`	**Project-local cargo configuration**, may contain `config.toml`. 🔗 🝖
📁 `benches/`	Benchmarks for your crate, run via **`cargo bench`**, requires nightly by default. * 🚧
📁 `examples/`	Examples how to use your crate, they see your crate like external user would.
`my_example.rs`	Individual examples are run like **`cargo run --example my_example`**.
📁 `src/`	Actual source code for your project.
`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.
📁 `src/bin/`	Place for additional binaries, even in library projects.
`extra.rs`	Additional binary, run with `cargo run --bin extra`.
📁 `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**, utilized via **`cargo clippy`** 🝖
`build.rs`	**Pre-build script**, 🔗 useful when compiling C / FFI, ...
`Cargo.toml`	Main **project manifest**, 🔗 Defines dependencies, artifacts ...
`Cargo.lock`	For reproducible builds. Add to git for apps, consider not for libs. 💬 🔗 🔗
`rust-toolchain.toml`	Define **toolchain override**🔗 (channel, components, targets) for this project.

* On stable consider Criterion.

Minimal examples for various entry points might look like:

Applications | Libraries | Unit Tests | Integration Tests | Benchmarks🚧 | Build Scripts | Proc Macros🝖

```
// src/main.rs (default application entry point)

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

Module trees and imports:

Module Trees | Namespaces🝖

Modules BK EX REF and **source files** work as follows:

- **Module tree** needs to be explicitly defined, is **not** implicitly built from **file system tree**. 🔗
- **Module tree root** equals library, app, ... entry point (e.g., `lib.rs`).

Actual **module definitions** work as follows:

- A `mod m {}` defines module in-file, while `mod m;` will read `m.rs` or `m/mod.rs`.
- Path of `.rs` based on **nesting**, e.g., `mod a { mod b { mod c; }}}` is either `a/b/c.rs` or `a/b/c/mod.rs`.

- Files not pathed from module tree root via some `mod m;` won't be touched by compiler! 🛑

Cargo

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 doc --no-deps --open`	Locally generate documentation for your code.
`cargo run`	Run your project, if a binary is produced (main.rs).
`cargo run --bin b`	Run binary `b`. Unifies feat. with other dependents (can be confusing).
`cargo run --package w`	Run main of sub-worksp. `w`. Treats features more sanely.
`cargo … --timings`	Show what crates caused your build to take so long. 🔥
`cargo tree`	Show dependency graph, all crates used by project, transitively.
`cargo tree -i foo`	Inverse dependency lookup, explain why `foo` is used.
`cargo info foo`	Show crate metadata for `foo` (by default for version used by this project).
`cargo +{nightly, stable} …`	Use given toolchain for command, e.g., for 'nightly only' tools.
`cargo +1.85.0 …`	Also accepts a specific version directly.
`cargo +nightly …`	Some nightly-only commands (substitute … with command below)
`rustc -- -Zunpretty=expanded`	Show expanded macros. 🚧
`rustup doc`	Open offline Rust documentation (incl. the books), good on a plane!

Here `cargo build` means you can either type `cargo build` or just `cargo b`; and `--release` means it can be replaced with `-r`.

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**.

Cross Compilation

🔘 Check target is supported.

🔘 Install target via **`rustup target install aarch64-linux-android`** (for example).

🔘 Install native toolchain (required to *link*, depends on target).

Get from target vendor (Google, Apple, …), might not be available on all hosts (e.g., no iOS toolchain on Windows).

Some toolchains require additional build steps (e.g., Android's `make-standalone-toolchain.sh`).

🔘 Update **`~/.cargo/config.toml`** like this:

```
[target.aarch64-linux-android]
linker = "[PATH_TO_TOOLCHAIN]/aarch64-linux-android/bin/aarch64-linux-android-clang"
```

or

```
[target.aarch64-linux-android]
linker = "C:/[PATH_TO_TOOLCHAIN]/prebuilt/windows-x86_64/bin/aarch64-linux-android21-clang.cmd"
```

🔘 Set **environment variables** (optional, wait until compiler complains before setting):

```
set CC=C:\[PATH_TO_TOOLCHAIN]\prebuilt\windows-x86_64\bin\aarch64-linux-android21-clang.cmd
set CXX=C:\[PATH_TO_TOOLCHAIN]\prebuilt\windows-x86_64\bin\aarch64-linux-android21-clang.cmd
set AR=C:\[PATH_TO_TOOLCHAIN]\prebuilt\windows-x86_64\bin\aarch64-linux-android-ar.exe
…
```

Whether you set them depends on how compiler complains, not necessarily all are needed.

Some platforms / configurations can be **extremely sensitive** how paths are specified (e.g., `\` vs `/`) and quoted.

✓ Compile with `cargo build --target=aarch64-linux-android`

Tooling Directives

Special tokens embedded in source code used by tooling or preprocessing.

Macro Fragments | **Documentation** | `#![globals]` | `#[code]` | `#[quality]` | `#[macros]` | `#[cfg]` | `build.rs`

Inside a **declarative** BK **macro by example** BK EX REF `macro_rules!` implementation these **fragment specifiers** REF work:

Within Macros	Explanation
`$x:ty`	Macro capture (here a `$x` is the capture and `ty` means `x` must be type).
`$x:block`	A block `{}` of statements or expressions, e.g., `{ let x = 5; }`
`$x:expr`	An expression, e.g., `x`, `1 + 1`, `String::new()` or `vec![]`
`$x:expr_2021`	An expression that matches the behavior of Rust '21 RFC
`$x:ident`	An identifier, for example in `let x = 0;` the identifier is `x`.
`$x:item`	An item, like a function, struct, module, etc.
`$x:lifetime`	A lifetime (e.g., `'a`, `'static`, etc.).
`$x:literal`	A literal (e.g., `3`, `"foo"`, `b"bar"`, etc.).
`$x:meta`	A meta item; the things that go inside `#[…]` and `#![…]` attributes.
`$x:pat`	A pattern, e.g., `Some(x)`, `(17, 'a')` or `x|x`.
`$x:pat_param`	Subset of patterns without top-level `|`, e.g., `Some(x)` or `x`.
`$x:path`	A path (e.g., `foo`, `::std::mem::replace`, `transmute::<_, int>`).
`$x:stmt`	A statement, e.g., `let x = 1 + 1;`, `String::new();` or `vec![];`
`$x:tt`	A single token tree, see here for more details.
`$x:ty`	A type, e.g., `String`, `usize` or `Vec<u8>`.
`$x:vis`	A visibility modifier; `pub`, `pub(crate)`, etc.
`$crate`	Special hygiene variable, crate where macros is defined. ?

For the *On* column in attributes:
`C` means on crate level (usually given as `#![my_attr]` in the top level file).
`M` means on modules.
`F` means on functions.
`S` means on static.
`T` means on types.
`X` means something special.
`!` means on macros.
`*` means on almost any item.

Working with Types

Types, Traits, Generics

Allowing users to *bring their own types* and avoid code duplication.

Types & Traits | **Generics** | **Advanced Concepts**?

Types

Type Equivalence and Conversions

Implementations — `impl S { }`

Traits — `trait T { }`

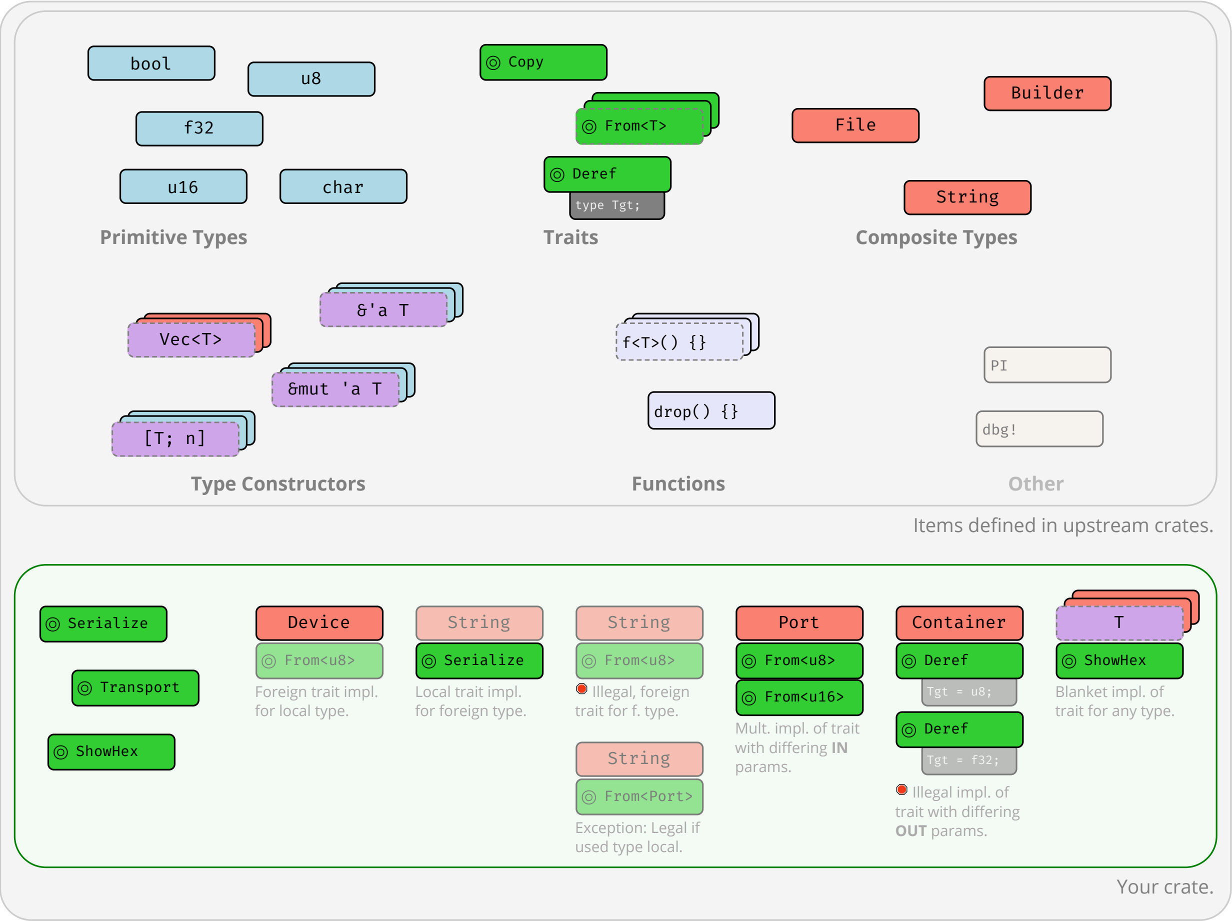
Implementing Traits for Types — `impl T for S { }`

Traits vs. Interfaces

Examples expand by clicking.

Foreign Types and Traits

A visual overview of types and traits in your crate and upstream.



Examples of traits and types, and which traits you can implement for which type.

Type Conversions

How to get `B` when you have `A`?

Intro | **Computation (Traits)** | **Casts** | **Coercions** | **Subtyping** | **Variance**

```
fn f(x: A) -> B {
    // How can you obtain B from A?
}
```

Method	Explanation
Identity	Trivial case, `B` **is exactly** `A`.
Computation	Create and manipulate instance of `B` by **writing code** transforming data.
Casts	**On-demand** conversion between types where caution is advised.
Coercions	**Automatic** conversion within *'weakening ruleset'*.[1]
Subtyping	**Automatic** conversion within *'same-layout-different-lifetimes ruleset'*.[1]

[1] While both convert `A` to `B`, **coercions** generally link to an *unrelated* `B` (a type "one could reasonably expect to have different methods"), while **subtyping** links to a `B` differing only in lifetimes.

Coding Guides

Idiomatic Rust

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

Idiom	Code
Think in Expressions	`y = if x { a } else { b };`
	`y = loop { break 5 };`
	`fn f() -> u32 { 0 }`
Think in Iterators	`(1..10).map(f).collect()`
	`names.iter().filter(|x| x.starts_with("A"))`
Test Absence with `?`	`y = 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`
Illegal State: Impossible	`my_lock.write().unwrap().guaranteed_at_compile_time_to_be_locked = 10;` [1]
	`thread::scope(|s| { /* Threads can't exist longer than scope() */ });`
Avoid *Global* State	Being depended on in multiple versions can secretly duplicate statics. 🛑 🔗
Provide Builders	`Car::new("Model T").hp(20).build();`
Make it Const	Where possible mark fns. `const`; where feasible run code inside `const {}`.
Don't Panic	Panics are *not* exceptions, they suggest immediate process abortion!
	Only panic on programming error; use `Option<T>` STD or `Result<T,E>` STD otherwise.
	If clearly user requested, e.g., calling `obtain()` vs. `try_obtain()`, panic ok too.
	Inside `const { NonZero::new(1).unwrap() }` p. becomes compile error, ok too.
Generics in Moderation	A simple `<T: Bound>` (e.g., `AsRef<Path>`) can make your APIs nicer to use.
	Complex bounds make it impossible to follow. If in doubt don't be creative with *g*.
Split Implementations	Generics like `Point<T>` can have separate `impl` per `T` for some specialization.
	`impl<T> Point<T> { /* Add common methods here */ }`
	`impl Point<f32> { /* Add methods only relevant for Point<f32> */ }`
Unsafe	Avoid `unsafe {}`,↓ often safer, faster solution without it.
Implement Traits	`#[derive(Debug, Copy, …)]` and custom `impl` where needed.
Tooling	Run **clippy** regularly to significantly improve your code quality. 🔥
	Format your code with **rustfmt** for consistency. 🔥
	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**.

[1] In most cases you should prefer `?` over `.unwrap()`. In the case of locks however the returned `PoisonError` signifies a panic in another thread, so unwrapping it (thus propagating the panic) is often the better idea.

> 🔥 We **highly** recommend you also follow the **API Guidelines** and the **Pragmatic Rust Guidelines** 🔥

Performance Tips

"My code is slow" sometimes comes up when porting microbenchmarks to Rust, or after profiling.

Rating	Name	Description
🚀🍼	**Release Mode** BK 🔥	Always do `cargo build --release` for massive speed boost.
🍼 ⚠️	**Target Native CPU** 🔗	Add `rustflags = ["-Ctarget-cpu=native"]` to `config.toml`.↑
🍼⚖️	**Codegen Units** 🔗	Codegen units `1` may yield faster code, slower compile.
🍼	**Reserve Capacity** STD	Pre-allocation of collections reduces allocation pressure.
🍼	**Recycle Collections** STD	Calling `x.clear()` and reusing `x` prevents allocations.
🍼	**Append to Strings** STD	Using `write!(&mut s, "{}")` can prevent extra allocation.
🍼⚖️	**Global Allocator** STD	On some platforms ext. allocator (e.g., **mimalloc** 🔗) faster.

Rating	Name	Description
	Bump Allocations 🔗	Cheaply gets *temporary*, dynamic memory, esp. in hot loops.
	Batch APIs	Design APIs to handle multiple similar elements at once, e.g., slices.
⚖️	**SoA / AoSoA** 🔗	Beyond that consider *struct of arrays* (SoA) and similar.
🚀 ⚖️	**SIMD** STD 🚧	Inside (math heavy) batch APIs using SIMD can give 2x - 8x boost.
	Reduce Data Size	Small types (e.g, `u8` vs `u32`, niches?) and data have better cache use.
	Keep Data Nearby 🔗	Storing often-used data *nearby* can improve memory access times.
	Pass by Size 🔗	Small (2-3 words) structs best passed by value, larger by reference.
⚖️	**Async-Await** 🔗	If *parallel waiting* happens a lot (e.g., server I/O) `async` good idea.
	Threading STD	Threads allow you to perform *parallel work* on mult. items at once.
🚀	**... in app**	Often good for apps, as lower wait times means better UX.
⚖️	**... inside libs**	Opaque *t.* use *inside* lib often not good idea, can be too opinionated.
🚀	**... for lib callers**	However, allowing *your user* to process *you* in parallel excellent idea.
⚖️	**Avoid Locks**	Locks in multi-threaded code kills parallelism.
⚖️	**Avoid Atomics**	Needless atomics (e.g., `Arc` vs `Rc`) impact other memory access.
⚖️	**Avoid False Sharing** 🔗	Make sure data R/W by different CPUs at least 64 bytes apart. 🔗
🚀 🍼	**Buffered I/O** STD 🔥	Raw `File` I/O highly inefficient w/o buffering.
🍼 ⚠️	**Faster Hasher** 🔗	Default `HashMap` STD hasher DoS attack-resilient but slow.
🍼 ⚠️	**Faster RNG**	If you use a crypto RNG consider swapping for non-crypto.
⚖️	**Avoid Trait Objects** 🔗	T.O. reduce code size, but increase memory indirection.
⚖️	**Defer Drop** 🔗	Dropping *heavy* objects in dump-thread can free up current one.
🍼 ⚠️	**Unchecked APIs** STD	If you are 100% confident, `unsafe { unchecked_ }` skips checks.

Entries marked 🚀 often come with a massive (> 2x) performance boost, 🍼 are easy to implement even after-the-fact, ⚖️ might have costly side effects (e.g., memory, complexity), ⚠️ have special risks (e.g., security, correctness).

Profiling Tips 💬

Profilers are indispensable to identify hot spots in code. For the best experience add this to your `Cargo.toml`:

```
[profile.release]
debug = true
```

Then do a `cargo build --release` and run the result with **Superluminal** (Windows) or **Instruments** (macOS). That said, there are many performance opportunities profilers won't find, but that need to be *designed in*.

Async-Await 101

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

Basics | **Execution Flow** | **Caveats** 🛑

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] [2]
`sm = async { g() };`	Likewise, does **not** execute the `{ g() }` block; produces state machine.
`runtime.block_on(sm);`	Outside an `async {}`, schedules `sm` to actually run. Would execute `g()`. [3] [4]
`sm.await`	Inside an `async {}`, run `sm` until complete. Yield to runtime if `sm` not ready.

[1] Technically `async` transforms following code into anonymous, compiler-generated state machine type; `f()` instantiates that machine.

[2] The state machine always `impl Future`, possibly `Send` & co, depending on types used inside `async`.

[3] State machine driven by worker thread invoking `Future::poll()` via runtime directly, or parent `.await` indirectly.

[4] Rust doesn't come with runtime, need external crate instead, e.g., tokio. Also, more helpers in futures crate.

Closures in APIs

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

From a call site perspective that means:

Signature	Function `g` can call …	Function `g` accepts …
`g<F: FnOnce()>(f: F)`	… `f()` at most 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.

Unsafe, Unsound, Undefined

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

Safe Code | Unsafe Code | Undefined Behavior | Unsound Code

Safe Code

- *Safe* has narrow meaning in Rust, vaguely 'the *intrinsic* prevention of undefined behavior (UB)'.
- Intrinsic means the language won't allow you to use *itself* to cause UB.
- Making an airplane crash or deleting your database is not UB, therefore 'safe' from Rust's perspective.
- Writing to `/proc/[pid]/mem` to self-modify your code is also 'safe', resulting UB not caused *intrinsincally*.

```
let y = x + x; // Safe Rust only guarantees the execution of this code is consistent with
print(y);      // 'specification' (long story …). It does not guarantee that y is 2x
               // (X::add might be implemented badly) nor that y is printed (Y::fmt may panic).
```

Responsible use of Unsafe

- Do not use `unsafe` unless you absolutely have to.
- Follow the Nomicon, Unsafe Guidelines, **always** follow **all** safety rules, and **never** invoke UB.
- Minimize the use of `unsafe` and encapsulate it in small, sound modules that are easy to review.
- Never create unsound abstractions; if you can't encapsulate `unsafe` properly, don't do it.
- Each `unsafe` unit should be accompanied by plain-text reasoning outlining its safety.

Adversarial Code

Adversarial code is *safe* 3rd party code that compiles but does not follow API *expectations*, and might interfere with your own (safety) guarantees.

You author	User code may possibly …
`fn g<F: Fn()>(f: F) { … }`	Unexpectedly panic.
`struct S<X: T> { … }`	Implement `T` badly, e.g., misuse `Deref`, …
`macro_rules! m { … }`	Do all of the above; call site can have *weird* scope.

Risk Pattern	Description
`#[repr(packed)]`	Packed alignment can make reference `&s.x` invalid.
`impl std::… for S {}`	Any trait `impl`, esp. `std::ops` may be broken. In particular …
`impl Deref for S {}`	May randomly `Deref`, e.g., `s.x != s.x`, or panic.
`impl PartialEq for S {}`	May violate equality rules; panic.
`impl Eq for S {}`	May cause `s != s`; panic; must not use `s` in `HashMap` & co.
`impl Hash for S {}`	May violate hashing rules; panic; must not use `s` in `HashMap` & co.
`impl Ord for S {}`	May violate ordering rules; panic; must not use `s` in `BTreeMap` & co.
`impl Index for S {}`	May randomly index, e.g., `s[x] != s[x]`; panic.
`impl Drop for S {}`	May run code or panic end of scope `{}`, during assignment `s = new_s`.
`panic!()`	User code can panic *any* time, resulting in abort or unwind.
`catch_unwind(|| s.f(panicky))`	Also, caller might force observation of broken state in `s`.
`let … = f();`	Variable name can affect order of `Drop` execution. [1] 🔴

[1] Notably, when you rename a variable from `_x` to `_` you will also change Drop behavior since you change semantics. A variable named `_x` will have `Drop::drop()` executed at the end of its scope, a variable named `_` can have it executed immediately on 'apparent' assignment ('apparent' because a binding named `_` means **wildcard** REF *discard this*, which will happen as soon as feasible, often right away)!

Implications

- Generic code **cannot be safe if safety depends on type cooperation** w.r.t. most (`std::`) traits.
- If type cooperation is needed you must use `unsafe` traits (prob. implement your own).
- You must consider random code execution at unexpected places (e.g., re-assignments, scope end).
- You may still be observable after a worst-case panic.

As a corollary, *safe*-but-deadly code (e.g., `airplane_speed<T>()`) should probably also follow these guides.

API Stability

When updating an API, these changes can break client code.RFC Major changes (🔴) are **definitely breaking**, while minor changes (🟡) **might be breaking**:

Crates
🔴 Making a crate that previously compiled for *stable* require *nightly*.
🔴 Removing Cargo features.
🟡 Altering existing Cargo features.

Modules
🔴 Renaming / moving / removing any public items.
🟡 Adding new public items, as this might break code that does `use your_crate::*`.

Structs
🔴 Adding private field when all current fields public.
🔴 Adding public field when no private field exists.
🟡 Adding or removing private fields when at least one already exists (before and after the change).
🟡 Going from a tuple struct with all private fields (with at least one field) to a normal struct, or vice versa.

Enums
🔴 Adding new variants; can be mitigated with early `#[non_exhaustive]` REF
🔴 Adding new fields to a variant.

Traits
🔴 Adding a non-defaulted item, breaks all existing `impl T for S {}`.
🔴 Any non-trivial change to item signatures, will affect either consumers or implementors.
🔴 Implementing any "fundamental" trait, as *not* implementing a fundamental trait already was a promise.
🟡 Adding a defaulted item; might cause dispatch ambiguity with other existing trait.
🟡 Adding a defaulted type parameter.
🟡 Implementing any non-fundamental trait; might also cause dispatch ambiguity.

Inherent Implementations
🟡 Adding any inherent items; might cause clients to prefer that over trait fn and produce compile error.

Signatures in Type Definitions
🔴 Tightening bounds (e.g., `<T>` to `<T: Clone>`).
🟡 Loosening bounds.
🟡 Adding defaulted type parameters.
🟡 Generalizing to generics.
Signatures in Functions
🔴 Adding / removing arguments.
🟡 Introducing a new type parameter.
🟡 Generalizing to generics.

Behavioral Changes
🔴 / 🟡 *Changing semantics might not cause compiler errors, but might make clients do wrong thing.*

Ralf Biedert, 2025 — cheats.rs