Rust Language Cheat Sheet

8. September 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. *

Example	Explanation
`[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.
`..`	**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 …
`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.

Example	Explanation
`'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.

Example	Explanation
`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.

Example	Explanation
`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.
`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`.
`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, [1]UTF-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.

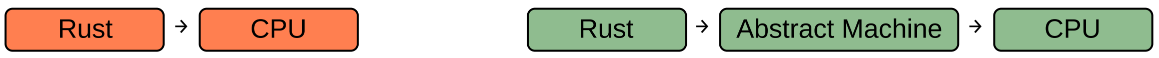
Behind the Scenes

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



🔴 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**.

Misconceptions

On the left things people may incorrectly assume they *should get away with* if Rust targeted CPU directly. On the right things you'd interfere with if in reality if you violate the AM contract.

Without AM	With AM
`0xffff_ffff` would make a valid `char`. 🛑	AM may exploit *'invalid'* bit patterns to pack unrelated data.
`0xff` and `0xff` are same pointer. 🛑	AM pointers can have **provenance** STD for optimization.
Any r/w on pointer `0xff` always fine. 🛑	AM may issue cache-friendly ops since *'no read possible'*.
Reading un-init just gives random value. 🛑	AM *'knows'* read impossible, may remove all related code.
Data race just gives random value. 🛑	AM may split R/W, produce *impossible* value. ↓
Null ref. is just `0x0` in some register. 🛑	Holding `0x0` in reference summons Cthulhu.

This table is only to outline what the AM does. Unlike C or C++, Rust never lets you do the wrong thing unless you force it with `unsafe`. ↓

Language Sugar

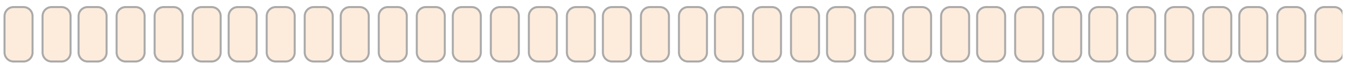
If something works that "shouldn't work now that you think about it", it might be due to one of these.

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

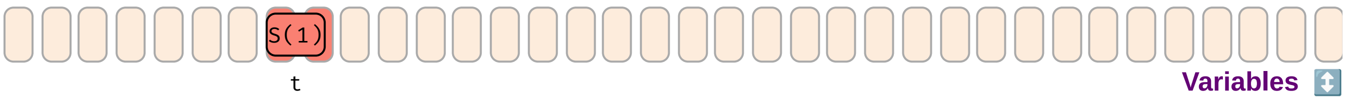
An illustrated guide to moves, references and lifetimes.



Application Memory ↕️

- Application memory is just array of bytes on low level.
- Operating environment usually segments that, amongst others, into:
  - **stack** (small, low-overhead memory,[1] most *variables* go here),
  - **heap** (large, flexible memory, but always handled via stack proxy like `Box<T>`),
  - **static** (most commonly used as resting place for `str` part of `&str`),
  - **code** (where bitcode of your functions reside).
- Most tricky part is tied to **how stack evolves**, which is **our focus**.

[1] For fixed-size values stack is trivially manageable: *take a few bytes more while you need them, discarded once you leave*. However, giving out pointers to these *transient* locations form the very essence of why *lifetimes* exist; and are the subject of the rest of this chapter.

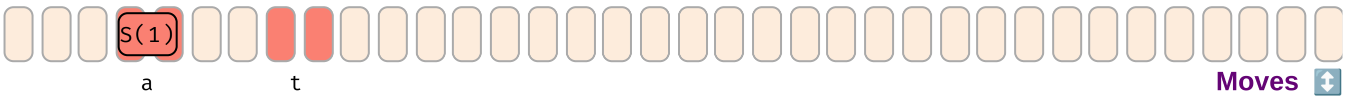


Variables ↕️

```
let t = S(1);
```

- Reserves memory location with name `t` of type `S` and the value `S(1)` stored inside.
- If declared with `let` that location lives on stack. [1]
- Note the **linguistic ambiguity**, in the term ***variable***, it can mean the:
  1. **name** of the location in the source file ("rename that variable"),
  2. **location** in a compiled app, `0x7` ("tell me the address of that variable"),
  3. **value** contained within, `S(1)` ("increment that variable").
- Specifically towards the compiler `t` can mean **location of** `t`, here `0x7`, and **value within** `t`, here `S(1)`.

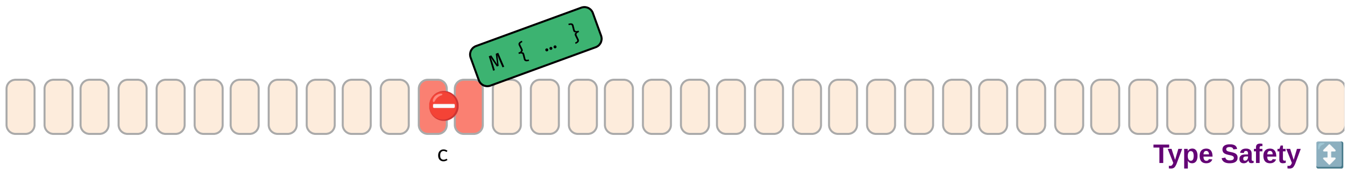
[1] Compare above,[↑] true for fully synchronous code, but `async` stack frame might placed it on heap via runtime.



Moves ↕️

```
let a = t;
```

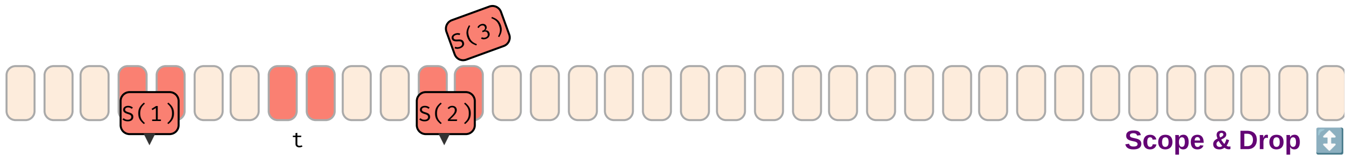
- This will **move** value within `t` to location of `a`, or copy it, if `S` is `Copy`.
- After move location `t` is **invalid** and cannot be read anymore.
  - Technically the bits at that location are not really *empty*, but *undefined*.
  - If you still had access to `t` (via `unsafe`) they might still *look* like valid `S`, but any attempt to use them as valid `S` is undefined behavior. [↓]
- We do not cover `Copy` types explicitly here. They change the rules a bit, but not much:
  - They won't be dropped.
  - They never leave behind an 'empty' variable location.



Type Safety

```
let c: S = M::new();
```

- The **type of a variable** serves multiple important purposes, it:
  1. dictates how the underlying bits are to be interpreted,
  2. allows only well-defined operations on these bits
  3. prevents random other values or bits from being written to that location.
- Here assignment fails to compile since the bytes of `M::new()` cannot be converted to form of type `S`.
- **Conversions between types will *always* fail** in general, **unless explicit rule allows it** (coercion, cast, …).

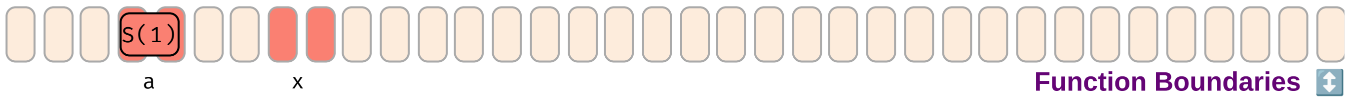


Scope & Drop

```
{
    let mut c = S(2);
    c = S(3);  // <- Drop called on `c` before assignment.
    let t = S(1);
    let a = t;
}   // <- Scope of `a`, `t`, `c` ends here, drop called on `a`, `c`.
```

- Once the 'name' of a non-vacated variable goes out of (drop-)**scope**, the contained value is **dropped**.
  - Rule of thumb: execution reaches point where name of variable leaves `{}`-block it was defined in
  - In detail more tricky, esp. temporaries, …
- Drop also invoked when new value assigned to existing variable location.
- In that case `Drop::drop()` is called on the location of that value.
  - In the example above `drop()` is called on `a`, twice on `c`, but not on `t`.
- Most non-`Copy` values get dropped most of the time; exceptions include `mem::forget()`, `Rc` cycles, `abort()`.

Call Stack

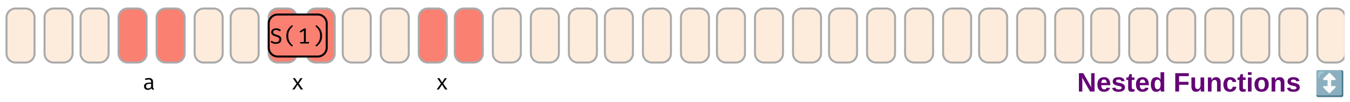


Function Boundaries

```
fn f(x: S) { … }

let a = S(1); // <- We are here
f(a);
```

- When a **function is called**, memory for parameters (and return values) are reserved on stack.[1]
- Here before `f` is invoked value in `a` is moved to 'agreed upon' location on stack, and during `f` works like 'local variable' `x`.

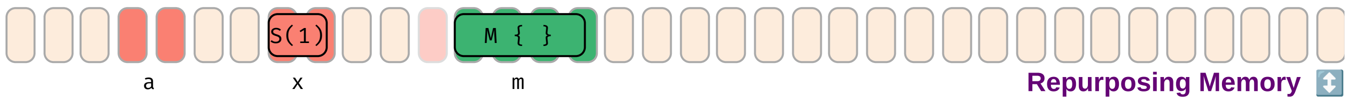
[1] Actual location depends on calling convention, might practically not end up on stack at all, but that doesn't change mental model.



Nested Functions ↕️

```
fn f(x: S) {
    if once() { f(x) } // ← We are here (before recursion)
}

let a = S(1);
f(a);
```

- **Recursively calling** functions, or calling other functions, likewise extends the stack frame.
- Nesting too many invocations (esp. via unbounded recursion) will cause stack to grow, and eventually to overflow, terminating the app.



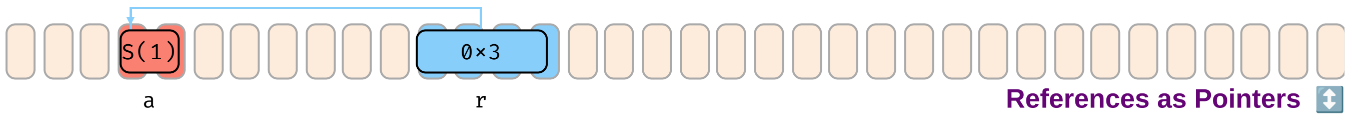
Repurposing Memory ↕️

```
fn f(x: S) {
    if once() { f(x) }
    let m = M::new() // ← We are here (after recursion)
}

let a = S(1);
f(a);
```

- Stack that previously held a certain type will be repurposed across (even within) functions.
- Here, recursing on `f` produced second `x`, which after recursion was partially reused for `m`.

Key take away so far, there are multiple ways how memory locations that previously held a valid value of a certain type stopped doing so in the meantime. As we will see shortly, this has implications for pointers.

References & Pointers

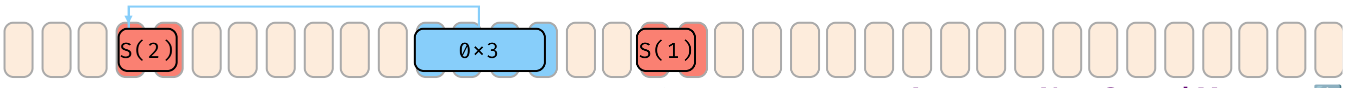


References as Pointers ↕️

```
let a = S(1);
let r: &S = &a;
```

- A **reference type** such as `&S` or `&mut S` can hold the **location of** some `s`.
- Here type `&S`, bound as name `r`, holds *location of* variable `a` (`0x3`), that must be type `S`, obtained via `&a`.
- If you think of variable `c` as *specific location*, reference **`r` is a *switchboard for locations***.
- The type of the reference, like all other types, can often be inferred, so we might omit it from now on:

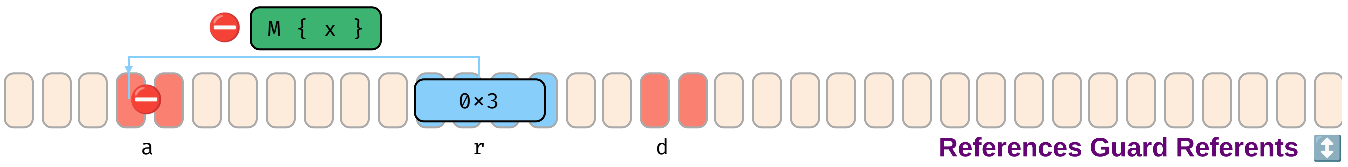
```
let r: &S = &a;
let r = &a;
```



Access to Non-Owned Memory ↕️

```
let mut a = S(1);
let r = &mut a;
let d = r.clone();  // Valid to clone (or copy) from r-target.
*r = S(2);          // Valid to set new S value to r-target.
```

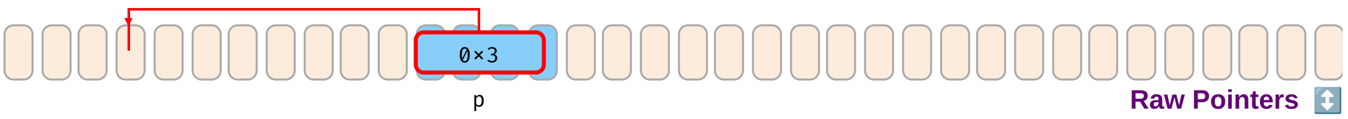
- References can **read from** (`&S`) and also **write to** (`&mut S`) locations they point to.
- The *dereference* `*r` means to use the **location `r` points to** (*not* the *location of* or *value within* `r` itself)
- In the example, clone `d` is created from `*r`, and `S(2)` written to `*r`.
    - We assume `S` implements `Clone`, and `r.clone()` clones target-of-`r`, not `r` itself.
    - On assignment `*r = …` old value in location also dropped (not shown above).



References Guard Referents ↕️

```
let mut a = …;
let r = &mut a;
let d = *r;       // Invalid to move out value, `a` would be empty.
*r = M::new();    // invalid to store non S value, doesn't make sense.
```

- While bindings guarantee to always *hold* valid data, references guarantee to always *point to* valid data.
- Esp. `&mut T` must provide same guarantees as variables, and some more as they can't dissolve the target:
    - They do **not allow writing invalid** data.
    - They do **not allow moving out** data (would leave target empty w/o owner knowing).



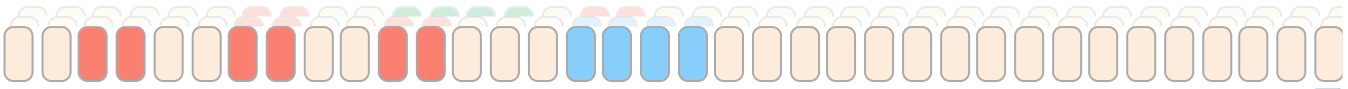
Raw Pointers ↕️

```
let p: *const S = questionable_origin();
```

- In contrast to references, pointers come with almost no guarantees.
- They may point to invalid or non-existent data.
- Dereferencing them is `unsafe`, and treating an invalid `*p` as if it were valid is undefined behavior. ↓

Lifetime Basics

Lifetimes in Functions

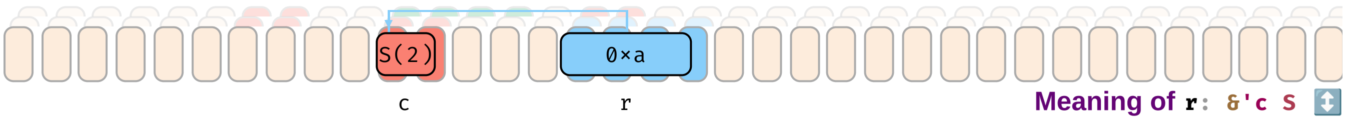


"Lifetime" of Things ↕️

- Every entity in a program has some (temporal / spatial) extent where it is relevant, i.e., *alive*.
- Loosely speaking, this *alive time* can be[1]
  1. the **LOC** (lines of code) where an **item is available** (e.g., a module name).
  2. the **LOC** between when a *location* is **initialized** with a value, and when the location is **abandoned**.
  3. the **LOC** between when a location is first **used in a certain way**, and when that **usage stops**.
  4. the **LOC (or actual time)** between when a *value* is created, and when that value is dropped.
- Within the rest of this section, we will refer to the items above as the:
  1. **scope** of that item, irrelevant here.
  2. **scope** of that variable or location.
  3. **lifetime**[2] of that usage.
  4. **lifetime** of that value, might be useful when discussing open file descriptors, but also irrelevant here.
- Likewise, lifetime parameters in code, e.g., `r: &'a S`, are
  - concerned with LOC any **location r *points to*** needs to be accessible or locked;
  - unrelated to the 'existence time' (as LOC) of `r` itself (well, it needs to exist shorter, that's it).
- `&'static S` means address must be *valid during all lines of code*.

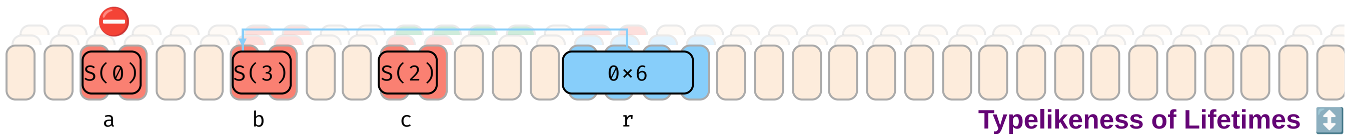
[1] There is sometimes ambiguity in the docs differentiating the various *scopes* and *lifetimes*. We try to be pragmatic here, but suggestions are welcome.

[2] *Live lines* might have been a more appropriate term …



Meaning of `r: &'c S` ↕️

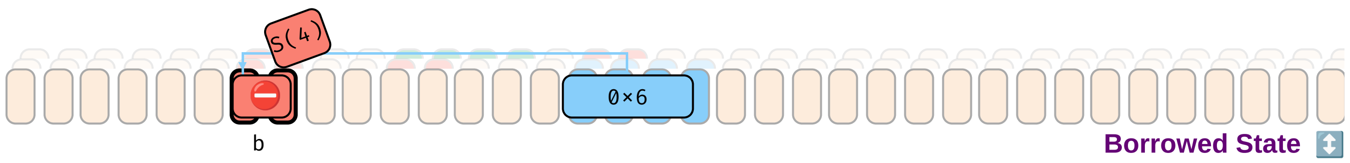
- Assume you got a `r: &'c S` from somewhere it means:
  - `r` holds an address of some `S`,
  - any address `r` points to must and will exist for at least `'c`,
  - the variable `r` itself cannot live longer than `'c`.



Typelikeness of Lifetimes ↕️

```
{
    let b = S(3);
    {
        let c = S(2);
        let r: &'c S = &c;      // Does not quite work since we can't name lifetimes of
local
        {                       // variables in a function body, but very same principle
applies
            let a = S(0);       // to functions next page.

            r = &a;             // Location of `a` does not live sufficient many lines →
not ok.
            r = &b;             // Location of `b` lives all lines of `c` and more → ok.
        }
    }
}
```

- Assume you got a `mut r: &mut 'c S` from somewhere.
    - That is, a mutable location that can hold a mutable reference.
- As mentioned, that reference must guard the targeted memory.
- However, the **`'c` part**, like a type, also **guards what is allowed into `r`**.
- Here assigning `&b` (`0×6`) to `r` is valid, but `&a` (`0×3`) would not, as only `&b` lives equal or longer than `&c`.

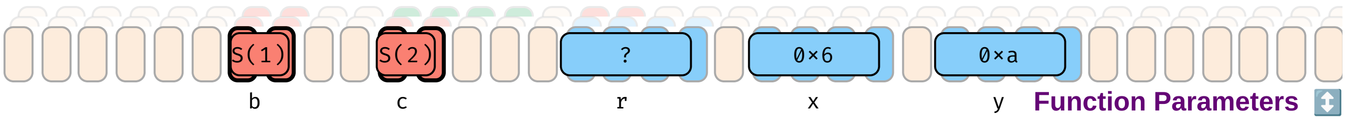


Borrowed State ↕️

```
let mut b = S(0);
let r = &mut b;

b = S(4);    // Will fail since `b` in borrowed state.

print_byte(r);
```

- Once the address of a variable is taken via `&b` or `&mut b` the variable is marked as **borrowed**.
- While borrowed, the content of the address cannot be modified anymore via original binding `b`.
- Once address taken via `&b` or `&mut b` stops being used (in terms of LOC) original binding `b` works again.

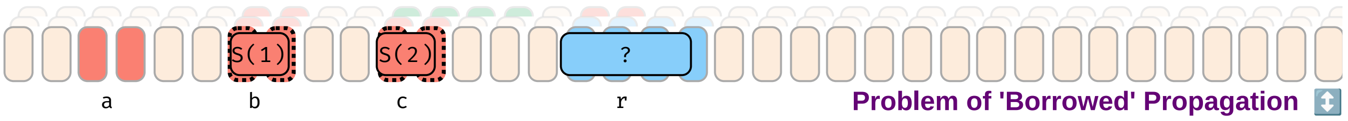
Advanced 🝖



Function Parameters

```
fn f(x: &S, y:&S) → &u8 { … }

let b = S(1);
let c = S(2);

let r = f(&b, &c);
```

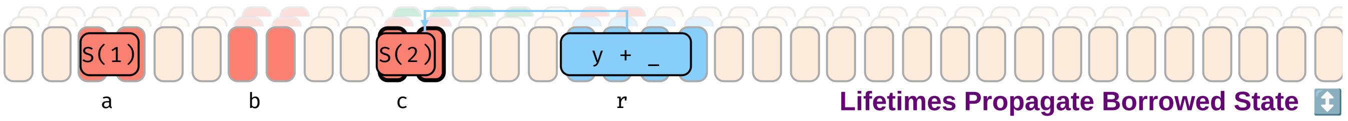
- When calling functions that take and return references two interesting things happen:
  - The used local variables are placed in a borrowed state,
  - But it is during compilation unknown which address will be returned.



Problem of 'Borrowed' Propagation

```
let b = S(1);
let c = S(2);

let r = f(&b, &c);

let a = b;   // Are we allowed to do this?
let a = c;   // Which one is _really_ borrowed?

print_byte(r);
```

- Since `f` can return only one address, not in all cases `b` and `c` need to stay locked.
- In many cases we can get quality-of-life improvements.
  - Notably, when we know one parameter *couldn't* have been used in return value anymore.

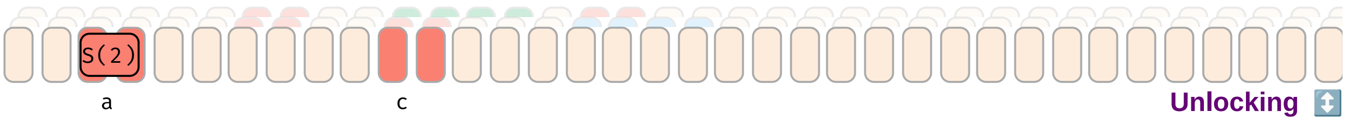


Lifetimes Propagate Borrowed State

```
fn f<'b, 'c>(x: &'b S, y: &'c S) → &'c u8 { … }

let b = S(1);
let c = S(2);

let r = f(&b, &c); // We know returned reference is `c`-based, which must stay locked,
                   // while `b` is free to move.

let a = b;

print_byte(r);
```

- Lifetime parameters in signatures, like `'c` above, solve that problem.
- Their primary purpose is:

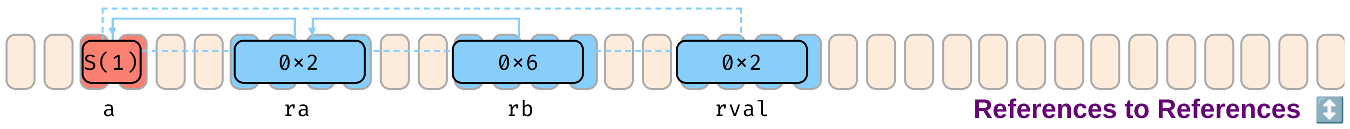
    - **outside the function**, to explain based on which input address an output address could be generated,
    - **within the function**, to guarantee only addresses that live at least `'c` are assigned.
- The actual lifetimes `'b`, `'c` are transparently picked by the compiler at **call site**, based on the borrowed variables the developer gave.
- They are **not** equal to the *scope* (which would be LOC from initialization to destruction) of `b` or `c`, but only a minimal subset of their scope called *lifetime*, that is, a minmal set of LOC based on how long `b` and `c` need to be borrowed to perform this call and use the obtained result.
- In some cases, like if `f` had `'c: 'b` instead, we still couldn't distinguish and both needed to stay locked.



Unlocking ↕️

```rust
let mut c = S(2);

let r = f(&c);
let s = r;
                    // ← Not here, `s` prolongs locking of `c`.

print_byte(s);

let a = c;          // ← But here, no more use of `r` or `s`.
```

- A variable location is *unlocked* again once the last use of any reference that may point to it ends.



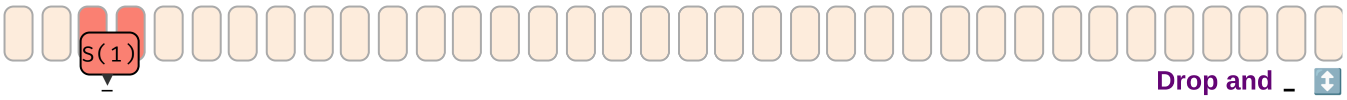
References to References ↕️

```
// Return short ('b) reference
fn f1sr<'b, 'a>(rb: &'b     &'a     S) -> &'b     S { *rb }
fn f2sr<'b, 'a>(rb: &'b     &'a mut S) -> &'b     S { *rb }
fn f3sr<'b, 'a>(rb: &'b mut &'a     S) -> &'b     S { *rb }
fn f4sr<'b, 'a>(rb: &'b mut &'a mut S) -> &'b     S { *rb }

// Return short ('b) mutable reference.
// f1sm<'b, 'a>(rb: &'b     &'a     S) -> &'b mut S { *rb } // M
// f2sm<'b, 'a>(rb: &'b     &'a mut S) -> &'b mut S { *rb } // M
// f3sm<'b, 'a>(rb: &'b mut &'a     S) -> &'b mut S { *rb } // M
fn f4sm<'b, 'a>(rb: &'b mut &'a mut S) -> &'b mut S { *rb }

// Return long ('a) reference.
fn f1lr<'b, 'a>(rb: &'b     &'a     S) -> &'a     S { *rb }
// f2lr<'b, 'a>(rb: &'b     &'a mut S) -> &'a     S { *rb } // L
fn f3lr<'b, 'a>(rb: &'b mut &'a     S) -> &'a     S { *rb }
// f4lr<'b, 'a>(rb: &'b mut &'a mut S) -> &'a     S { *rb } // L

// Return long ('a) mutable reference.
// f1lm<'b, 'a>(rb: &'b     &'a     S) -> &'a mut S { *rb } // M
// f2lm<'b, 'a>(rb: &'b     &'a mut S) -> &'a mut S { *rb } // M
// f3lm<'b, 'a>(rb: &'b mut &'a     S) -> &'a mut S { *rb } // M
// f4lm<'b, 'a>(rb: &'b mut &'a mut S) -> &'a mut S { *rb } // L

// Now assume we have a `ra` somewhere
let mut ra: &'a mut S = …;

let rval = f1sr(&&*ra);       // OK
let rval = f2sr(&&mut *ra);
let rval = f3sr(&mut &*ra);
let rval = f4sr(&mut ra);

//  rval = f1sm(&&*ra);       // Would be bad, since rval would be mutable
//  rval = f2sm(&&mut *ra);   // reference obtained from broken mutability
//  rval = f3sm(&mut &*ra);   // chain.
let rval = f4sm(&mut ra);

let rval = f1lr(&&*ra);
//  rval = f2lr(&&mut *ra);   // If this worked we'd have `rval` and `ra` …
let rval = f3lr(&mut &*ra);
//  rval = f4lr(&mut ra);     // … now (mut) aliasing `S` in compute below.

//  rval = f1lm(&&*ra);       // Same as above, fails for mut-chain reasons.
//  rval = f2lm(&&mut *ra);   //                    "
//  rval = f3lm(&mut &*ra);   //                    "
//  rval = f4lm(&mut ra);     // Same as above, fails for aliasing reasons.

// Some fictitious place where we use `ra` and `rval`, both alive.
compute(ra, rval);
```

Here (`M`) means compilation fails because mutability error, (`L`) lifetime error. Also, dereference `*rb` not strictly necessary, just added for clarity.

- `f_sr` cases always work, short reference (only living `'b`) can always be produced.
- `f_sm` cases usually fail simply because *mutable chain* to `S` needed to return `&mut S`.
- `f_lr` cases can fail because returning `&'a S` from `&'a mut S` to caller means there would now exist two references (one mutable) to same `S` which is illegal.
- `f_lm` cases always fail for combination of reasons above.

Note: This example is about the `f` functions, not `compute`. You can assume it to be defined as `fn compute(x: &S, y: &S) {}`. In that case the `ra` parameter would be automatically coerced ↓ from `&mut S` to `&S`, since you can't have a shared and a mutable reference to the same target.



Drop and _ ↕️

```
{
    let f = |x, y| (S(x), S(y)); // Function returning two 'Droppables'.

    let (     x1, y) = f(1, 4);  // S(1) - Scope   S(4) - Scope
    let (     x2, _) = f(2, 5);  // S(2) - Scope   S(5) - Immediately
    let (ref x3, _) = f(3, 6);   // S(3) - Scope   S(6) - Scope

    println!("…");
}
```

Here `Scope` means contained value lives until end of scope, i.e., past the `println!()`.

- Functions or expressions producing movable values must be handled by callee.
- Values stores in 'normal' bindings are kept until end of scope, then dropped.
- Values stored in _ bindings are usually dropped right away.
- However, sometimes references (e.g., `ref x3`) can keep value (e.g., the tuple `(S(3), S(6))`) around for longer, so `S(6)`, being part of that tuple can only be dropped once reference to its `S(3)` sibling disappears).

↕️ Examples expand by clicking.

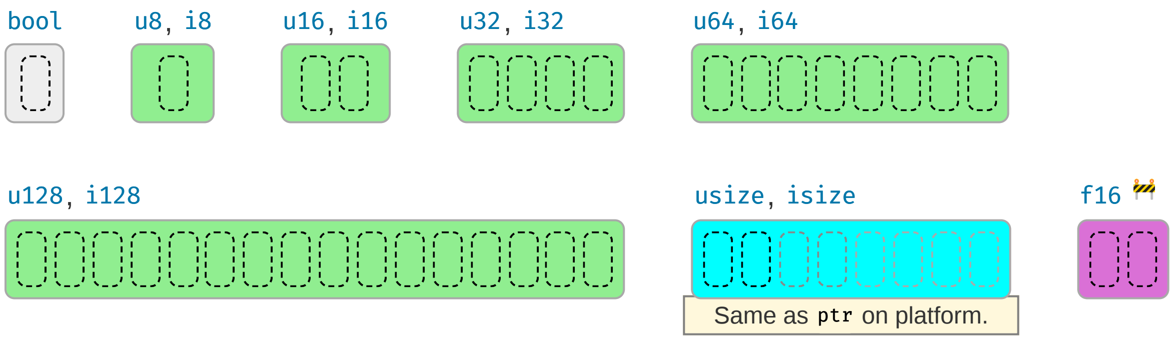
Memory Layout

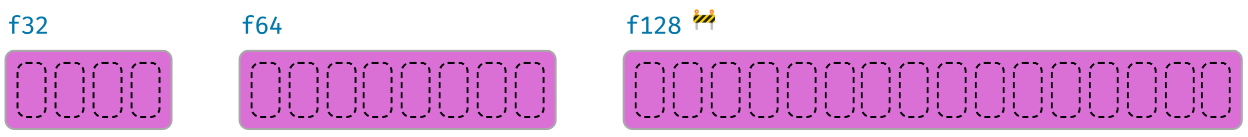
Byte representations of common types.

Basic Types

Essential types built into the core of the language.

Boolean REF **and Numeric Types** REF





Unsigned Types

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`.

Signed Types

Type	Max Value
`i8`	`127`
`i16`	`32_767`
`i32`	`2_147_483_647`
`i64`	`9_223_372_036_854_775_807`
`i128`	`170_141_183_460_469_231_731_687_303_715_884_105_727`
`isize`	Depending on platform pointer size, same as `i16`, `i32`, or `i64`.

Type	Min Value
`i8`	`-128`
`i16`	`-32_768`
`i32`	`-2_147_483_648`
`i64`	`-9_223_372_036_854_775_808`
`i128`	`-170_141_183_460_469_231_731_687_303_715_884_105_728`
`isize`	Depending on platform pointer size, same as `i16`, `i32`, or `i64`.

Float Types

Type	Max value	Min pos value	Max lossless integer[1]
`f16` 🚧	65504.0	$6.10 \cdot 10^{-5}$	`2048`
`f32`	$3.40 \cdot 10^{38}$	$3.40 \cdot 10^{-38}$	`16_777_216`
`f64`	$1.79 \cdot 10^{308}$	$2.23 \cdot 10^{-308}$	`9_007_199_254_740_992`
`f128` 🚧	$1.19 \cdot 10^{4932}$	$3.36 \cdot 10^{-4932}$	$2.07 \cdot 10^{34}$

[1] The maximum integer `M` so that all other integers `0 ⩽ X ⩽ M` can be losslessly represented in that type. In other words, there might be larger integers that could still be represented losslessly (e.g., `65504` for `f16`), but up until that value a lossless representation is guaranteed.

Float values approximated for visual clarity. Negative limits are values multipled with -1.

Float Internals

Sample bit representation* for a `f32`:

```
S | E E E E E E E E | F F F F F F F F F F F F F F F F F F F F F F F
```

Explanation:

f32	S (1)	E (8)	F (23)	Value
Normalized number	±	1 to 254	any	$\pm(1.F)_2 * 2^{E-127}$
Denormalized number	±	0	non-zero	$\pm(0.F)_2 * 2^{-126}$
Zero	±	0	0	±0
Infinity	±	255	0	±∞
NaN	±	255	non-zero	NaN

Similarly, for `f64` types this would look like:

f64	S (1)	E (11)	F (52)	Value
Normalized number	±	1 to 2046	any	$\pm(1.F)_2 * 2^{E-1023}$
Denormalized number	±	0	non-zero	$\pm(0.F)_2 * 2^{-1022}$
Zero	±	0	0	±0
Infinity	±	2047	0	±∞
NaN	±	2047	non-zero	NaN

* Float types follow IEEE 754-2008 and depend on platform endianness.

Casting Pitfalls

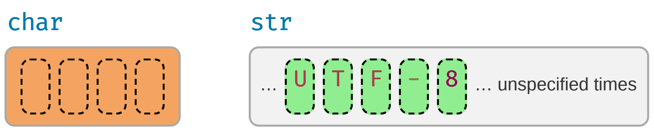
Cast[1]	Gives	Note
`3.9_f32 as u8`	`3`	Truncates, consider `x.round()` first.
`314_f32 as u8`	`255`	Takes closest available number.
`f32::INFINITY as u8`	`255`	Same, treats `INFINITY` as *really* large number.
`f32::NAN as u8`	`0`	-
`_314 as u8`	`58`	Truncates excess bits.
`_257 as i8`	`1`	Truncates excess bits.
`_200 as i8`	`-56`	Truncates excess bits, MSB might then also signal negative.

Arithmetic Pitfalls

Operation[1]	Gives	Note
`200_u8 / 0_u8`	Compile error.	-
`200_u8 / _0` [d, r]	Panic.	Regular math may panic; here: division by zero.
`200_u8 + 200_u8`	Compile error.	-
`200_u8 + _200` [d]	Panic.	Consider `checked_`, `wrapping_`, ... instead. STD
`200_u8 + _200` [r]	`144`	In release mode this will overflow.
`-128_i8 * -1`	Compile error.	Would overflow (`128_i8` doesn't exist).
`-128_i8 * _1neg` [d]	Panic.	-
`-128_i8 * _1neg` [r]	`-128`	Overflows back to `-128` in release mode.
`1_u8 / 2_u8`	`0`	Other integer division truncates.
`0.8_f32 + 0.1_f32`	`0.90000004`	-
`1.0_f32 / 0.0_f32`	`f32::INFINITY`	-
`0.0_f32 / 0.0_f32`	`f32::NAN`	-
`x < f32::NAN`	`false`	`NAN` comparisons always return false.
`x > f32::NAN`	`false`	`NAN` comparisons always return false.
`f32::NAN == f32::NAN`	`false`	Use `f32::is_nan()` STD instead.

[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



Any Unicode scalar.

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

Basics

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**.

Usage

Chars	Description
`let c = 'a';`	Often a `char` (unicode scalar) can coincide with your intuition of *character*.
`let c = '❤';`	It can also hold many Unicode symbols.
`let c = '❤️';`	But not always. Given emoji is **two** `char` (see Encoding) and **can't** 🛑 be held by `c`.[1]
`c = 0xffff_ffff;`	Also, chars are **not allowed** 🛑 to hold arbitrary bit patterns.

[1] Fun fact, due to the Zero-width joiner (⨝) what the user *perceives as a character* can get even more unpredictable: 👨‍👩‍👧 is in fact 5 chars 👨⨝👩⨝👧, and rendering engines are free to either show them fused as one, or separately as three, depending on their abilities.

Strings	Description
`let s = "a";`	A `str` is usually never held directly, but as `&str`, like `s` here.
`let s = "❤❤️";`	It can hold arbitrary text, has variable length per *c.*, and is hard to index.

Encoding

```
let s = "I ❤ Rust";
let t = "I ❤️ Rust";
```

Variant	Memory Representation[2]
`s.as_bytes()`	`49 20 e2 9d a4 20 52 75 73 74` [3]
`t.as_bytes()`	`49 20 e2 9d a4 ef b8 8f 20 52 75 73 74` [4]
`s.chars()`[1]	`49 00 00 00 20 00 00 00 64 27 00 00 20 00 00 00 52 00 00 00 75 00 00 00 73 00 …`
`t.chars()`[1]	`49 00 00 00 20 00 00 00 64 27 00 00 0f fe 01 00 20 00 00 00 52 00 00 00 75 00 …`

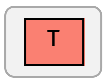
[1] Result then collected into array and transmuted to bytes.
[2] Values given in hex, on x86.
[3] Notice how ❤, having Unicode Code Point (U+2764), is represented as **64 27 00 00** inside the `char`, but got UTF-8 encoded to **e2 9d a4** in the `str`.
[4] Also observe how the emoji Red Heart ❤️, is a combination of ❤ and the U+FE0F Variation Selector, thus `t` has a higher char count than `s`.

⚠️ For what seem to be browser bugs Safari and Edge render the hearts in Footnote 3 and 4 wrong, despite being able to differentiate them correctly in `s` and `t` above.

Custom Types

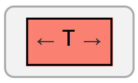
Basic types definable by users. Actual **layout** REF is subject to **representation**; REF padding can be present.

`T`



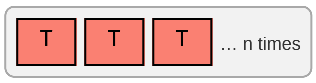
Sized type.

`T: ?Sized`



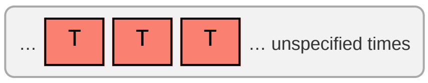
Maybe sized.

`[T; n]`



Fixed array of `n` elements.

`[T]`

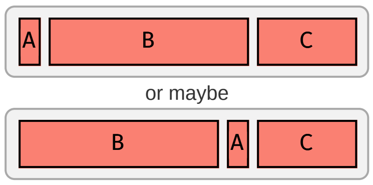


Slice type of unknown-many elements. Neither `Sized` (nor carries `len` information), and most often lives behind reference as `&[T]`. ↓

`struct S;`

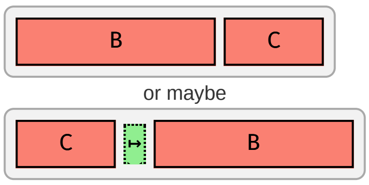
Zero-sized type.

`(A, B, C)`



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

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

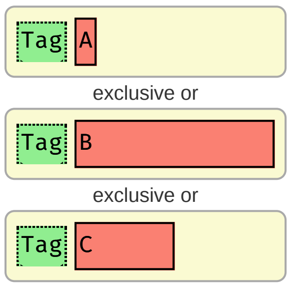


Compiler may also add padding.

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.

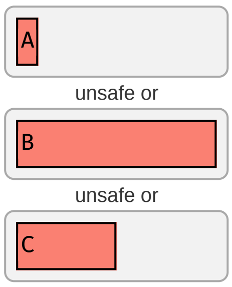
These **sum types** hold a value of one of their sub types:

`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 { … }`

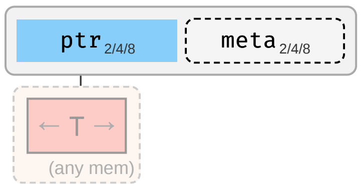


Can unsafely reinterpret memory. Result might be undefined.

References & Pointers

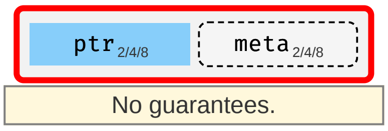
References give safe access to 3rd party memory, raw pointers `unsafe` access. The corresponding `mut` types have an identical data layout to their immutable counterparts.

`&'a T`



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

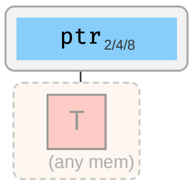
`*const T`



Pointer Meta

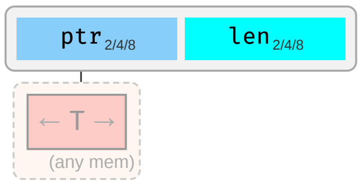
Many reference and pointer types can carry an extra field, **pointer metadata**. STD It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

`&'a T`



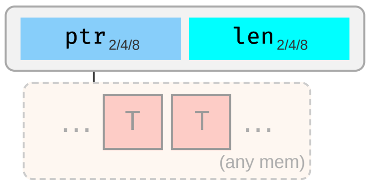
No meta for sized target. (pointer is thin).

`&'a T`



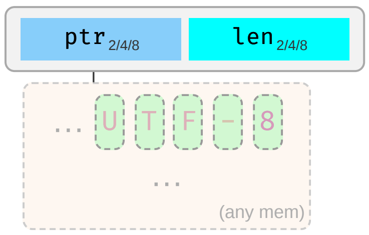
If `T` is a DST `struct` such as `S { x: [u8] }` meta field `len` is count of dyn. sized content.

`&'a [T]`



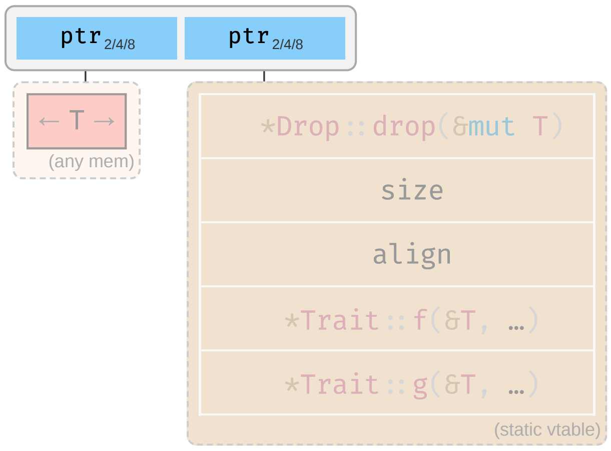
Regular **slice reference** (i.e., the reference type of a slice type `[T]`) [↑] often seen as `&[T]` if `'a` elided.

`&'a str`



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

`&'a dyn Trait`



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

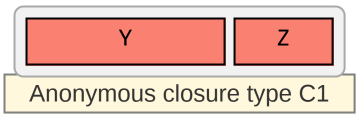
Closures

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

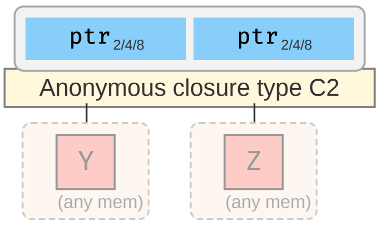
```
let y = ...;
let z = ...;

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:

`move |x| x + y.f() + z`



`|x| x + y.f() + z`



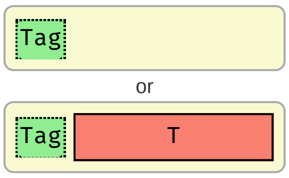
Also produces anonymous `fn` such as `f_c1(C1, X)` or `f_c2(&C2, X)`. Details depend on which `FnOnce`, `FnMut`, `Fn` ... is supported, based on properties of captured types.

[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

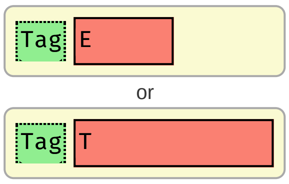
Rust's standard library combines the above primitive types into useful types with special semantics, e.g.:

`Option<T>` STD



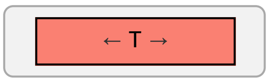
Tag may be omitted for certain T, e.g., `NonNull`.STD

`Result<T, E>` STD



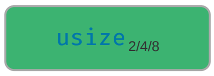
Either some error `E` or value of `T`.

`ManuallyDrop<T>` STD



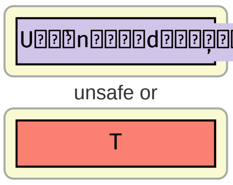
Prevents `T::drop()` from being called.

`AtomicUsize` STD



Other atomic similarly.

`MaybeUninit<T>`STD

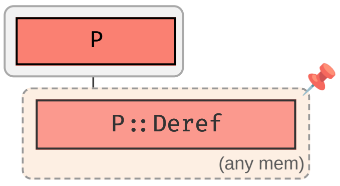


Uninitialized memory or some `T`. Only legal way to work with uninit data.

`PhantomData<T>` STD

Zero-sized helper to hold otherwise unused lifetimes.

`Pin<P>` STD

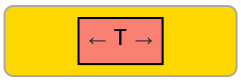


Signals tgt. of `P` is pinned 'forever' even past lt. of `Pin`. Value within may not be moved out (but new one moved in), unless `Unpin`.STD

🛑 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.

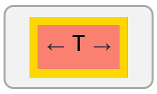
Cells

`UnsafeCell<T>` STD



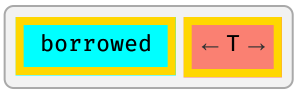
Magic type allowing aliased mutability.

`Cell<T>` STD



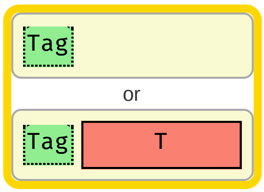
Allows `T`'s to move in and out.

`RefCell<T>` STD



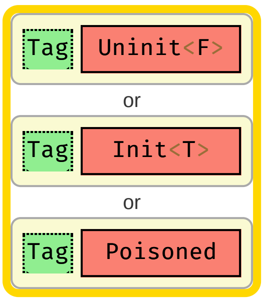
Also support dynamic borrowing of `T`. Like `Cell` this is `Send`, but not `Sync`.

`OnceCell<T>` STD



Initialized at most once.

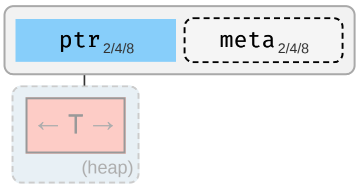
`LazyCell<T, F>` STD



Initialized on first access.

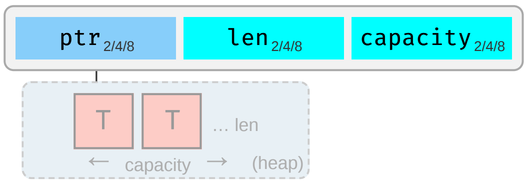
Order-Preserving Collections

`Box<T>` STD



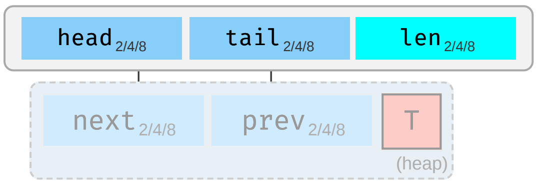
For some `T` stack proxy may carry meta† (e.g., `Box<[T]>`).

`Vec<T>` STD



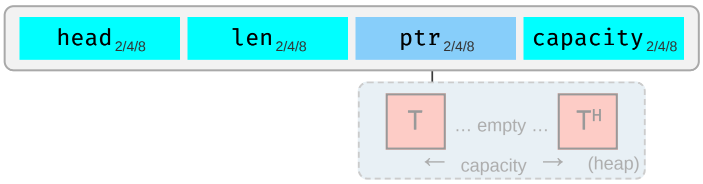
Regular *growable array* vector of single type.

`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.

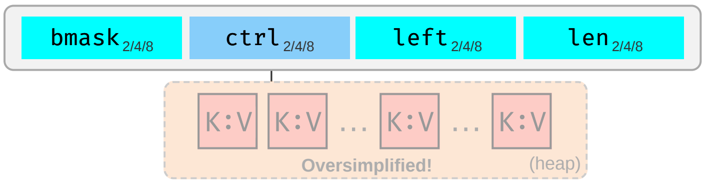
`VecDeque<T>` STD



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

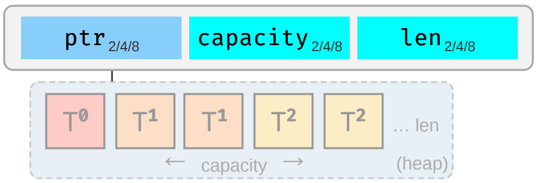
Other Collections

`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.

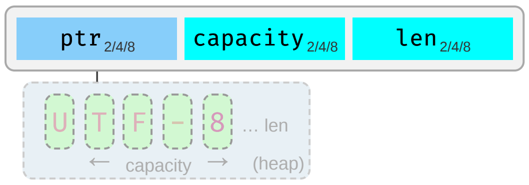
`BinaryHeap<T>` STD



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.

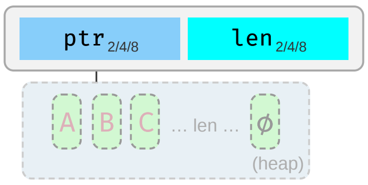
Owned Strings

`String` STD



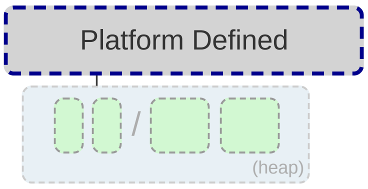
Observe how `String` differs from `&str` and `&[char]`.

`CString` STD



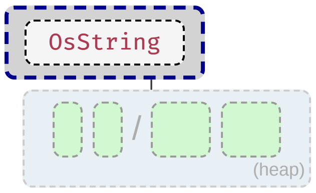
NUL-terminated but w/o NUL in middle.

`OsString` STD



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

`PathBuf` STD

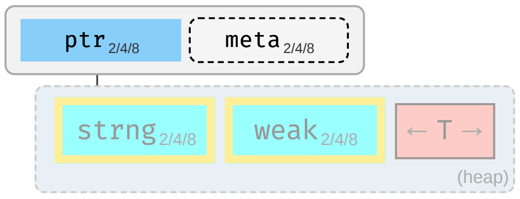


Encapsulates how operating system represents paths.

Shared Ownership

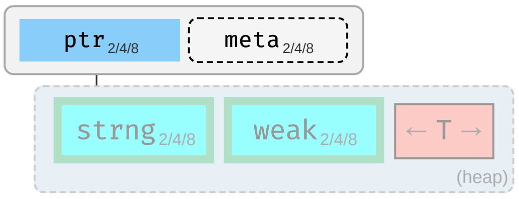
If the type does not contain a `Cell` for `T`, these are often combined with one of the `Cell` types above to allow shared de-facto mutability.

`Rc<T>` STD



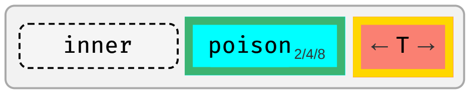
Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell`to allow mutation. Is neither `Send` nor `Sync`.

`Arc<T>` STD



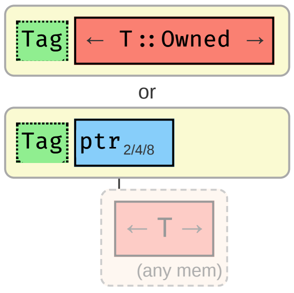
Same, but allow sharing between threads IF contained `T` itself is `Send` and `Sync`.

`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

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.

I/O

Intent	Snippet
Create a new file STD	`File::create(PATH)?`
Same, via OpenOptions	`OpenOptions::new().create(true).write(true).truncate(true).open(PATH)?`
Read file as `String` STD	`read_to_string(path)?`

Macros

Intent	Snippet
Macro w. variable arguments	`macro_rules! var_args { ($($args:expr),*) => {{ }} }`
Using `args`, e.g., calling `f` multiple times.	`$( f($args); )*`

Transforms 🔥

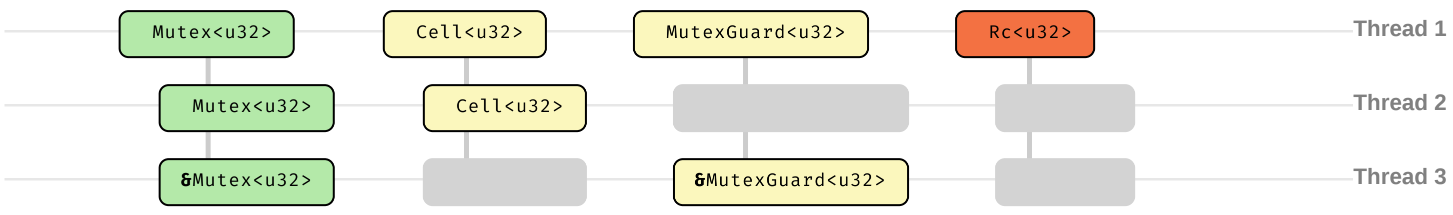
Starting Type	Resource
`Option<T> -> …`	See the Type-Based Cheat Sheet
`Result<T, R> -> …`	See the Type-Based Cheat Sheet
`Iterator<Item=T> -> …`	See the Type-Based Cheat Sheet
`&[T] -> …`	See the Type-Based Cheat Sheet
`Future<T> -> …`	See the Futures Cheat Sheet

Esoterics

Intent	Snippet
Cleaner closure captures	`wants_closure({ let c = outer.clone(); move || use_clone(c) })`
Fix inference in '`try`' closures	`iter.try_for_each(|x| { Ok::<(), Error>(()) })?;`
Iterate *and* edit `&mut [T]` if `T` Copy.	`Cell::from_mut(mut_slice).as_slice_of_cells()`
Get subslice with length.	`&original_slice[offset..][..length]`
Canary so trait `T` is **object safe**. REF	`const _: Option<&dyn T> = None;`
Semver trick to unify types. 🔗	`my_crate = "next.version"` in `Cargo.toml` + re-export types.
Use macro inside own crate. 🔗	`macro_rules! internal_macro {}` with `pub(crate) use internal_macro;`

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.

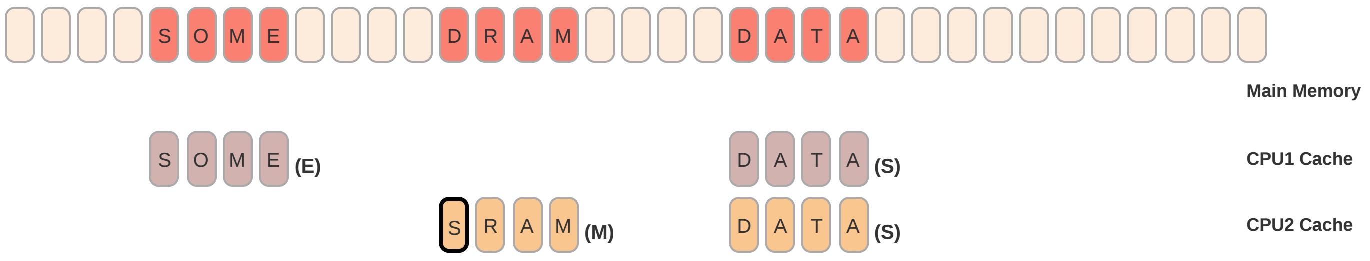
When is ...	... Send?
`T`	All contained fields are `Send`, or `unsafe` impl'ed.
`struct S { ... }`	All fields are `Send`, or `unsafe` impl'ed.
`struct S<T> { ... }`	All fields are `Send` and T is `Send`, or `unsafe` impl'ed.
`enum E { ... }`	All fields in all variants are `Send`, or `unsafe` impl'ed.
`&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`.

When 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`.

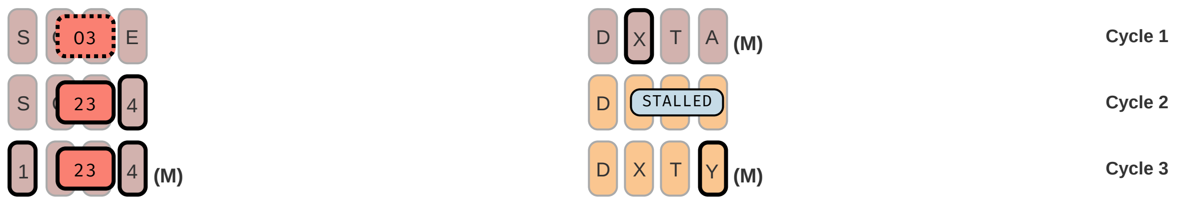
[1] This is a bit of pseudo-code to get the point across, the idea is to have an `Rc` before an `.await` point and keep using it beyond that point.

Atomics & Cache

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

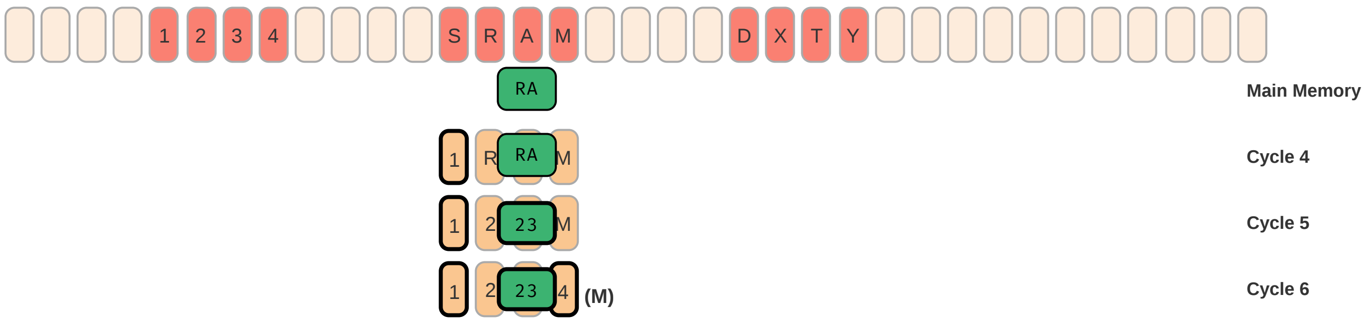


Modern CPUs don't accesses memory directly, only their cache. Each CPU has its own cache, 100x faster than RAM, but much smaller. It comes in **cache lines**, some *sliced* window of bytes, which track if it's an exclusive (E), shared (S) or modified (M) view of the main memory. Caches talk to each other 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

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.

Obtaining

Basics

Assume you have a collection `c` of type `C` you want to use:

- **`c.into_iter()`**[1] — Turns collection `c` into an **`Iterator`** STD `i` and **consumes**[2] `c`. *Std.* way to get iterator.
- **`c.iter()`** — Courtesy method **some** collections provide, returns **borrowing** Iterator, doesn't consume `c`.
- **`c.iter_mut()`** — Same, but **mutably borrowing** Iterator that allow collection to be changed.

The Iterator

Once you have an `i`:

- **`i.next()`** — Returns `Some(x)` next element `c` provides, or `None` if we're done.

For Loops

- **`for x in c {}`** — Syntactic sugar, calls `c.into_iter()` and loops `i` until `None`.

[1] Requires **`IntoIterator`** STD for `C` to be implemented. Type of item depends on what `C` was.

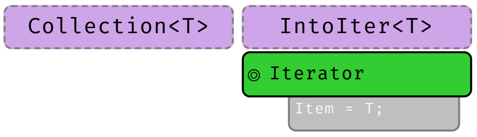
[2] If it looks as if it doesn't consume `c` that's because type was `Copy`. For example, if you call `(&c).into_iter()` it will invoke `.into_iter()` on `&c` (which will consume a *copy* of the reference and turn it into an Iterator), but the original `c` remains untouched.

Creating

Essentials

Let's assume you have a `struct Collection<T> {}` you authored. You should also implement:

- `struct IntoIter<T> {}` — Create a struct to hold your iteration status (e.g., an index) for value iteration.
- `impl Iterator for IntoIter<T> {}` — Implement `Iterator::next()` so it can produce elements.



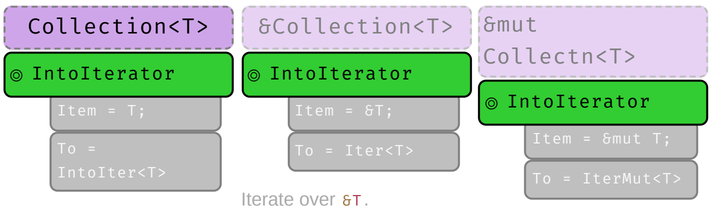
At this point you have something that can behave as an **Iterator**, STD but no way of actually obtaining it. See the next tab for how that usually works.

For Loops

Native Loop Support

Many users would expect your collection to *just work* in `for` loops. You need to implement:

- `impl IntoIterator for Collection<T> {}` — Now `for x in c {}` works.
- `impl IntoIterator for &Collection<T> {}` — Now `for x in &c {}` works.
- `impl IntoIterator for &mut Collection<T> {}` — Now `for x in &mut c {}` works.



Iterate over `T`.

Iterate over `&T`.

Iterate over `&mut T`.

As you can see, the **IntoIterator** STD trait is what actually connects your collection with the **IntoIter** struct you created in the previous tab. The two siblings of **IntoIter** (**Iter** and **IterMut**) are discussed in the next tab.

Borrowing

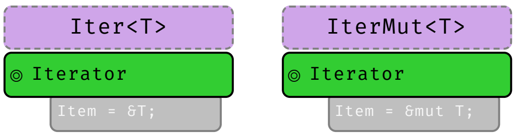
Shared & Mutable Iterators

In addition, if you want your collection to be useful when borrowed you should implement:

- `struct Iter<T> {}` — Create struct holding `&Collection<T>` state for shared iteration.
- `struct IterMut<T> {}` — Similar, but holding `&mut Collection<T>` state for mutable iteration.
- `impl Iterator for Iter<T> {}` — Implement shared iteration.
- `impl Iterator for IterMut<T> {}` — Implement mutable iteration.

Also you might want to add convenience methods:

- `Collection::iter(&self) -> Iter`,
- `Collection::iter_mut(&mut self) -> IterMut`.



The code for borrowing interator support is basically just a repetition of the previous steps with a slightly different types, e.g., `&T` vs `T`.

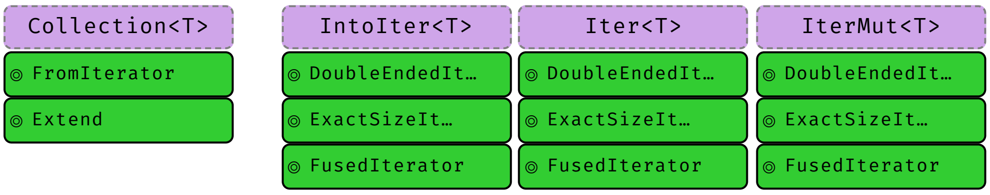
Interoperability

Iterator Interoperability

To allow 3rd **party iterators** to 'collect into' your collection implement:

- `impl FromIterator for Collection<T> {}` — Now `some_iter.collect::<Collection<_>>()` works.
- `impl Extend for Collection<T> {}` — Now `c.extend(other)` works.

In addition, also consider adding the extra traits from `std::iter` STD to your previous structs:



Writing collections can be work. The good news is, if you followed all these steps your collections will feel like *first class citizens*.

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

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()`

CString

If you have `x` of type …	Use this …
`String`	`CString::new(x)?`
`CString`	`x`
`OsString`	`CString::new(x.to_str()?)?`
`PathBuf`	`CString::new(x.to_str()?)?`
`Vec<u8>` 1	`CString::new(x)?`
`&str`	`CString::new(x)?`
`&CStr`	`x.to_owned()` i
`&OsStr`	`CString::new(x.to_os_string().into_string()?)?`
`&Path`	`CString::new(x.to_str()?)?`
`&[u8]` 1	`CString::new(Vec::from(x))?`
`*mut c_char` 2	`unsafe { CString::from_raw(x) }`

OsString

If you have `x` of type …	Use this …
`String`	`OsString::from(x)` i
`CString`	`OsString::from(x.to_str()?)`
`OsString`	`x`
`PathBuf`	`x.into_os_string()`
`Vec<u8>` 1	`unsafe { OsString::from_encoded_bytes_unchecked(x) }`

If you have `x` of type ...	Use this ...
`&str`	`OsString::from(x)` i
`&CStr`	`OsString::from(x.to_str()?)`
`&OsStr`	`OsString::from(x)` i
`&Path`	`x.as_os_str().to_owned()`
`&[u8]` 1	`unsafe { OsString::from_encoded_bytes_unchecked(x.to_vec()) }`

PathBuf

If you have `x` of type ...	Use this ...
`String`	`PathBuf::from(x)` i
`CString`	`PathBuf::from(x.to_str()?)`
`OsString`	`PathBuf::from(x)` i
`PathBuf`	`x`
`Vec<u8>` 1	`unsafe { PathBuf::from(OsString::from_encoded_bytes_unchecked(x)) }`
`&str`	`PathBuf::from(x)` i
`&CStr`	`PathBuf::from(x.to_str()?)`
`&OsStr`	`PathBuf::from(x)` i
`&Path`	`PathBuf::from(x)` i
`&[u8]` 1	`unsafe { PathBuf::from(OsString::from_encoded_bytes_unchecked(x.to_vec())) }`

Vec<u8>

If you have `x` of type ...	Use this ...
`String`	`x.into_bytes()`
`CString`	`x.into_bytes()`
`OsString`	`x.into_encoded_bytes()`
`PathBuf`	`x.into_os_string().into_encoded_bytes()`
`Vec<u8>` 1	`x`
`&str`	`Vec::from(x.as_bytes())`
`&CStr`	`Vec::from(x.to_bytes_with_nul())`
`&OsStr`	`Vec::from(x.as_encoded_bytes())`
`&Path`	`Vec::from(x.as_os_str().as_encoded_bytes())`
`&[u8]` 1	`x.to_vec()`

&str

If you have `x` of type ...	Use this ...
`String`	`x.as_str()`
`CString`	`x.to_str()?`

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

`&CStr`

If you have `x` of type …	Use this …
`String`	`CString::new(x)?.as_c_str()`
`CString`	`x.as_c_str()`
`OsString`	`x.to_str()?`
`PathBuf`	?,3
`Vec<u8>` [1,4]	`CStr::from_bytes_with_nul(&x)?`
`&str`	?,3
`&CStr`	`x`
`&OsStr`	?
`&Path`	?
`&[u8]` [1,4]	`CStr::from_bytes_with_nul(x)?`
`*const c_char` [1]	`unsafe { CStr::from_ptr(x) }`

`&OsStr`

If you have `x` of type …	Use this …
`String`	`OsStr::new(&x)`
`CString`	?
`OsString`	`x.as_os_str()`
`PathBuf`	`x.as_os_str()`
`Vec<u8>` [1]	`unsafe { OsStr::from_encoded_bytes_unchecked(&x) }`
`&str`	`OsStr::new(x)`
`&CStr`	?
`&OsStr`	`x`
`&Path`	`x.as_os_str()`
`&[u8]` [1]	`unsafe { OsStr::from_encoded_bytes_unchecked(x) }`

`&Path`

If you have `x` of type ...	Use this ...
`String`	`Path::new(x)` r
`CString`	`Path::new(x.to_str()?)`
`OsString`	`Path::new(x.to_str()?)` r
`PathBuf`	`Path::new(x.to_str()?)` r
`Vec<u8>` 1	`unsafe { Path::new(OsStr::from_encoded_bytes_unchecked(&x)) }`
`&str`	`Path::new(x)` r
`&CStr`	`Path::new(x.to_str()?)`
`&OsStr`	`Path::new(x)` r
`&Path`	`x`
`&[u8]` 1	`unsafe { Path::new(OsStr::from_encoded_bytes_unchecked(x)) }`

`&[u8]`

If you have `x` of type ...	Use this ...
`String`	`x.as_bytes()`
`CString`	`x.as_bytes()`
`OsString`	`x.as_encoded_bytes()`
`PathBuf`	`x.as_os_str().as_encoded_bytes()`
`Vec<u8>` 1	`&x`
`&str`	`x.as_bytes()`
`&CStr`	`x.to_bytes_with_nul()`
`&OsStr`	`x.as_encoded_bytes()`
`&Path`	`x.as_os_str().as_encoded_bytes()`
`&[u8]` 1	`x`

Other

You want	And have `x`	Use this ...
`*const c_char`	`CString`	`x.as_ptr()`

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

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.

Printable Types

In `format!` and friends, types convert via trait `Display` `"{}"` STD or `Debug` `"{:?}"` STD , non exhaustive list:

Type	Implements
`String`	`Debug, Display`
`CString`	`Debug`
`OsString`	`Debug`
`PathBuf`	`Debug`
`Vec<u8>`	`Debug`
`&str`	`Debug, Display`
`&CStr`	`Debug`
`&OsStr`	`Debug`
`&Path`	`Debug`
`&[u8]`	`Debug`
`bool`	`Debug, Display`
`char`	`Debug, Display`
`u8` … `i128`	`Debug, Display`
`f32, f64`	`Debug, Display`
`!`	`Debug, Display`
`()`	`Debug`

In short, pretty much everything is `Debug`; more *special* types might need special handling or conversion ↑ to `Display`.

Formatting

Each argument designator in format macro is either empty `{}`, `{argument}`, or follows a basic **syntax**:

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

Element	Meaning
`argument`	Number (`0`, `1`, ...), variable '21 or name,'18 e.g., `print!("{x}")`.
`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` STD formatter `?` 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` STD (`?`) formatting, hex (`x`), binary (`b`), octal (`o`), pointer (`p`), exp (`e`) ... see more.

Format Example	Explanation
`{}`	Print the next argument using `Display`. STD
`{x}`	Same, but use variable `x` from scope. '21
`{:?}`	Print the next argument using `Debug`. STD
`{2:#?}`	Pretty-print the 3rd argument with `Debug` STD 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`).

Full Example	Explanation
`println!("{}", x)`	Print `x` using `Display` STD on std. out and append new line. '15 🗑️
`println!("{x}")`	Same, but use variable `x` from scope. '21
`format!("{a:.3} {b:?}")`	Convert `a` with 3 digits, add space, `b` with `Debug` STD, return `String`. '21

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.

Entry	Code
`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

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

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

Libraries

```
// src/lib.rs (default library entry point)

pub fn f() {}      // Is a public item in root, so it's accessible from the outside.

mod m {
    pub fn g() {}  // No public path (`m` not public) from root, so `g`
}                  // is not accessible from the outside of the crate.
```

Unit Tests

Integration Tests

Benchmarks🚧

Build Scripts

```
// src/my_module.rs (any file of your project)

fn f() → u32 { 0 }

#[cfg(test)]
mod test {
    use super::f;           // Need to import items from parent module. Has
                            // access to non-public members.
    #[test]
    fn ff() {
        assert_eq!(f(), 0);
    }
}
```

```
// tests/sample.rs (sample integration test)

#[test]
fn my_sample() {
    assert_eq!(my_crate::f(), 123); // Integration tests (and benchmarks) 'depend' to the
crate like
}                                   // a 3rd party would. Hence, they only see public items.
```

```
// benches/sample.rs (sample benchmark)

#![feature(test)]   // #[bench] is still experimental

extern crate test;  // Even in '18 this is needed for … reasons.
                    // Normally you don't need this in '18 code.

use test::{black_box, Bencher};

#[bench]
fn my_algo(b: &mut Bencher) {
    b.iter(|| black_box(my_crate::f())); // `black_box` prevents `f` from being optimized
away.
}
```

```
// build.rs (sample pre-build script)

fn main() {
    // You need to rely on env. vars for target; `#[cfg(…)]` are for host.
    let target_os = env::var("CARGO_CFG_TARGET_OS");
}
```

*See here for list of environment variables set.

Proc Macros

```rust
// src/lib.rs (default entry point for proc macros)

extern crate proc_macro;  // Apparently needed to be imported like this.

use proc_macro::TokenStream;

#[proc_macro_attribute]   // Crates can now use `#[my_attribute]`
pub fn my_attribute(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

```toml
// Cargo.toml

[package]
name = "my_crate"
version = "0.1.0"

[lib]
proc-macro = true
```

Module trees and imports:

Module Trees

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! 🛑

Namespaces

Rust has three kinds of **namespaces**:

Namespace *Types*	Namespace *Functions*	Namespace *Macros*
`mod X {}`	`fn X() {}`	`macro_rules! X { … }`
`X` (crate)	`const X: u8 = 1;`	
`trait X {}`	`static X: u8 = 1;`	
`enum X {}`		
`union X {}`		

Namespace *Types*	Namespace *Functions*	Namespace *Macros*
`struct X {}`		
`struct X;`[1]	`struct X;`[1]	
`struct X();`[2]	`struct X();`[2]	

[1] Counts in *Types* and in *Functions*, defines type `X` *and* constant `X`.
[2] Counts in *Types* and in *Functions*, defines type `X` *and* function `X`.

- In any given scope, for example within a module, only one item per namespace can exist, e.g.,
  - `enum X {}` and `fn X() {}` can coexist
  - `struct X;` and `const X` cannot coexist
- With a `use my_mod::X;` all items called `X` will be imported.

Due to naming conventions (e.g., `fn` and `mod` are lowercase by convention) and *common sense* (most developers just don't name all things `X`) you won't have to worry about these *kinds* in most cases. They can, however, be a factor when designing macros.

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

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.

Within Macros	Explanation
`$x:lifetime`	A lifetime (e.g., `'a`, `'static`, etc.).
`$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. ?

Documentation

Inside a **doc comment** BK EX REF these work:

Within Doc Comments	Explanation
`` ```…``` ``	Include a **doc test** (doc code running on `cargo test`).
`` ```X,Y …``` ``	Same, and include optional configurations; with `X`, `Y` being …
`rust`	Make it explicit test is written in Rust; implied by Rust tooling.
`-`	Compile test. Run test. Fail if panic. **Default behavior**.
`should_panic`	Compile test. Run test. Execution should panic. If not, fail test.
`no_run`	Compile test. Fail test if code can't be compiled, Don't run test.
`compile_fail`	Compile test but fail test if code *can* be compiled.
`ignore`	Do not compile. Do not run. Prefer option above instead.
`edition2018`	Execute code as Rust '18; default is '15.
`#`	Hide line from documentation (`` ``` # use x::hidden; ``` ``).
`` [`S`] ``	Create a link to struct, enum, trait, function, … `S`.
`` [`S`](crate::S) ``	Paths can also be used, in the form of markdown links.

`#![globals]`

Attributes affecting the whole crate or app:

Opt-Out's	On	Explanation
`#![no_std]`	`C`	Don't (automatically) import **`std`** STD ; use **`core`** STD instead. REF
`#![no_implicit_prelude]`	`CM`	Don't add **`prelude`** STD, need to manually import `None`, `Vec`, … REF
`#![no_main]`	`C`	Don't emit `main()` in apps if you do that yourself. REF

Opt-In's	On	Explanation
`#![feature(a, b, c)]`	`C`	Rely on f. that may not get stabilized, *c.* **Unstable Book**. 🚧

Builds	On	Explanation
`#![crate_name = "x"]`	C	Specify current crate name, e.g., when not using `cargo`. ? REF
`#![crate_type = "bin"]`	C	Specify current crate type (`bin`, `lib`, `dylib`, `cdylib`, …). REF
`#![recursion_limit = "123"]`	C	Set *compile-time* recursion limit for deref, macros, … REF
`#![type_length_limit = "456"]`	C	Limits maximum number of type substitutions. REF
`#![windows_subsystem = "x"]`	C	On Windows, make a `console` or `windows` app. REF

Handlers	On	Explanation
`#[alloc_error_handler]`	F	Make some `fn(Layout) -> !` the **allocation fail. handler**. 🔗 🚧
`#[global_allocator]`	S	Make static item impl. `GlobalAlloc` STD **global allocator**. REF
`#[panic_handler]`	F	Make some `fn(&PanicInfo) -> !` app's **panic handler**. REF

`#[code]`

Attributes primarily governing emitted code:

Developer UX	On	Explanation
`#[non_exhaustive]`	T	Future-proof `struct` or `enum`; hint it may grow in future. REF
`#[path = "x.rs"]`	M	Get module from non-standard file. REF
`#[diagnostic::on_unimplemented]`	X	Give better error messages when trait not implemented. RFC

Codegen	On	Explanation
`#[cold]`	F	Hint that function probably isn't going to be called. REF
`#[inline]`	F	Nicely suggest compiler should inline function at call sites. REF
`#[inline(always)]`	F	Emphatically threaten compiler to inline call, or else. REF
`#[inline(never)]`	F	Instruct compiler to feel sad if it still inlines the function. REF
`#[repr(X)]`[1]	T	Use another representation instead of the default **`rust`** REF one:
`#[target_feature(enable="x")]`	F	Enable CPU feature (e.g., `avx2`) for code of `unsafe fn`. REF
`#[track_caller]`	F	Allows `fn` to find **`caller`** STD for better panic messages. REF
`#[repr(C)]`	T	Use a C-compatible (f. FFI), predictable (f. `transmute`) layout. REF
`#[repr(C, u8)]`	`enum`	Give `enum` discriminant the specified type. REF
`#[repr(transparent)]`	T	Give single-element type same layout as contained field. REF
`#[repr(packed(1))]`	T	Lower align. of struct and contained fields, mildly UB prone. REF
`#[repr(align(8))]`	T	Raise alignment of struct to given value, e.g., for SIMD types. REF

[1] Some representation modifiers can be combined, e.g., `#[repr(C, packed(1))]`.

Linking	On	Explanation
`#[unsafe(no_mangle)]`	*	Use item name directly as symbol name, instead of mangling. REF
`#[unsafe(export_name = "foo")]`	FS	Export a `fn` or `static` under a different name. REF

Linking	On	Explanation
`#[unsafe(link_section = ".x")]`	FS	Section name of object file where item should be placed. REF
`#[link(name="x", kind="y")]`	X	Native lib to link against when looking up symbol. REF
`#[link_name = "foo"]`	F	Name of symbol to search for resolving `extern fn`. REF
`#[no_link]`	X	Don't link `extern crate` when only wanting macros. REF
`#[used]`	S	Don't optimize away `static` variable despite it looking unused. REF

#[quality]

Attributes used by Rust tools to improve code quality:

Code Patterns	On	Explanation
`#[allow(X)]`	*	Instruct `rustc` / `clippy` to ign. class `X` of possible issues. REF
`#[expect(X)]` [1]	*	Warn if a lint doesn't trigger. REF
`#[warn(X)]` [1]	*	… emit a warning, mixes well with `clippy` lints. 🔥 REF
`#[deny(X)]` [1]	*	… fail compilation. REF
`#[forbid(X)]` [1]	*	… fail compilation and prevent subsequent `allow` overrides. REF
`#[deprecated = "msg"]`	*	Let your users know you made a design mistake. REF
`#[must_use = "msg"]`	FTX	Makes compiler check return value is *processed* by caller. 🔥 REF

[1] 💬 There is some debate which one is the *best* to ensure high quality crates. Actively maintained multi-dev crates probably benefit from more aggressive `deny` or `forbid` lints; less-regularly updated ones probably more from conservative use of `warn` (as future compiler or `clippy` updates may suddenly break otherwise working code with minor issues).

Tests	On	Explanation
`#[test]`	F	Marks the function as a test, run with `cargo test`. 🔥 REF
`#[ignore = "msg"]`	F	Compiles but does not execute some `#[test]` for now. REF
`#[should_panic]`	F	Test must `panic!()` to actually succeed. REF
`#[bench]`	F	Mark function in `bench/` as benchmark for `cargo bench`. 🚧 REF

Formatting	On	Explanation
`#[rustfmt::skip]`	*	Prevent `cargo fmt` from cleaning up item. 🔗
`#![rustfmt::skip::macros(x)]`	CM	… from cleaning up macro `x`. 🔗
`#![rustfmt::skip::attributes(x)]`	CM	… from cleaning up attribute `x`. 🔗

Documentation	On	Explanation
`#[doc = "Explanation"]`	*	Same as adding a `///` doc comment. 🔗
`#[doc(alias = "other")]`	*	Provide other name for search in docs. 🔗
`#[doc(hidden)]`	*	Prevent item from showing up in docs. 🔗
`#![doc(html_favicon_url = "")]`	C	Sets the `favicon` for the docs. 🔗
`#![doc(html_logo_url = "")]`	C	The logo used in the docs. 🔗
`#![doc(html_playground_url = "")]`	C	Generates `Run` buttons and uses given service. 🔗
`#![doc(html_root_url = "")]`	C	Base URL for links to external crates. 🔗

Documentation	On	Explanation
`#![doc(html_no_source)]`	`C`	Prevents source from being included in docs. 🔗

`#[macros]`

Attributes related to the creation and use of macros:

Macros By Example	On	Explanation
`#[macro_export]`	`!`	Export `macro_rules!` as `pub` on crate level REF
`#[macro_use]`	`MX`	Let macros persist past mod.; or import from `extern crate`. REF

Proc Macros	On	Explanation
`#[proc_macro]`	`F`	Mark `fn` as **function-like** procedural *m.* callable as `m!()`. REF
`#[proc_macro_derive(Foo)]`	`F`	Mark `fn` as **derive macro** which can `#[derive(Foo)]`. REF
`#[proc_macro_attribute]`	`F`	Mark `fn` as **attribute macro** for new `#[x]`. REF

Derives	On	Explanation
`#[derive(X)]`	`T`	Let some proc macro provide a goodish `impl` of `trait X`. 🔥 REF

`#[cfg]`

Attributes governing conditional compilation:

Config Attributes	On	Explanation
`#[cfg(X)]`	`*`	Include item if configuration `X` holds. REF
`#[cfg(all(X, Y, Z))]`	`*`	Include item if all options hold. REF
`#[cfg(any(X, Y, Z))]`	`*`	Include item if at least one option holds. REF
`#[cfg(not(X))]`	`*`	Include item if `X` does not hold. REF
`#[cfg_attr(X, foo = "msg")]`	`*`	Apply `#[foo = "msg"]` if configuration `X` holds. REF

⚠️ Note, options can generally be set multiple times, i.e., the same key can show up with multiple values. One can expect `#[cfg(target_feature = "avx")]` **and** `#[cfg(target_feature = "avx2")]` to be true at the same time.

Known Options	On	Explanation
`#[cfg(debug_assertions)]`	`*`	Whether `debug_assert!()` & co. would panic. REF
`#[cfg(feature = "foo")]`	`*`	When your crate was compiled with *f.* `foo`. 🔥 REF
`#[cfg(target_arch = "x86_64")]`	`*`	The CPU architecture crate is compiled for. REF
`#[cfg(target_env = "msvc")]`	`*`	How DLLs and functions are interf. with on OS. REF
`#[cfg(target_endian = "little")]`	`*`	Main reason your new zero-cost prot. fails. REF
`#[cfg(target_family = "unix")]`	`*`	Family operating system belongs to. REF
`#[cfg(target_feature = "avx")]`	`*`	Whether a particular class of instructions is avail. REF

Known Options	On	Explanation
`#[cfg(target_os = "macos")]`	*	Operating system your code will run on. REF
`#[cfg(target_pointer_width = "64")]`	*	How many bits ptrs, `usize` and words have. REF
`#[cfg(target_vendor = "apple")]`	*	Manufacturer of target. REF
`#[cfg(panic = "unwind")]`	*	Whether `unwind` or `abort` will happen on panic. ?
`#[cfg(proc_macro)]`	*	Whether crate compiled as proc macro. REF
`#[cfg(test)]`	*	Whether compiled with `cargo test`. 🔥 REF

`build.rs`

Environment variables and outputs related to the pre-build script. Consider **build-rs**🔗 instead.

Input Environment	Explanation 🔗
`CARGO_FEATURE_X`	Environment variable set for each feature `x` activated.
`CARGO_FEATURE_SOMETHING`	If feature `something` were enabled.
`CARGO_FEATURE_SOME_FEATURE`	If *f.* `some-feature` were enabled; dash `-` converted to `_`.
`CARGO_CFG_X`	Exposes cfg's; joins mult. opts. by `,` and converts `-` to `_`.
`CARGO_CFG_TARGET_OS=macos`	If `target_os` were set to `macos`.
`CARGO_CFG_TARGET_FEATURE=avx,avx2`	If `target_feature` were set to `avx` and `avx2`.
`OUT_DIR`	Where output should be placed.
`TARGET`	Target triple being compiled for.
`HOST`	Host triple (running this build script).
`PROFILE`	Can be `debug` or `release`.

Available in `build.rs` via `env::var()?`. List not exhaustive.

Output String	Explanation 🔗
`cargo::rerun-if-changed=PATH`	(Only) run this `build.rs` again if `PATH` changed.
`cargo::rerun-if-env-changed=VAR`	(Only) run this `build.rs` again if environment `VAR` changed.
`cargo::rustc-cfg=KEY[="VALUE"]`	Emit given `cfg` option to be used for later compilation.
`cargo::rustc-cdylib-link-arg=FLAG`	When building a `cdylib`, pass linker flag.
`cargo::rustc-env=VAR=VALUE`	Emit var accessible via `env!()` in crate during compilation.
`cargo::rustc-flags=FLAGS`	Add special flags to compiler. ?
`cargo::rustc-link-lib=[KIND=]NAME`	Link native library as if via `-l` option.
`cargo::rustc-link-search=[KIND=]PATH`	Search path for native library as if via `-L` option.
`cargo::warning=MESSAGE`	Emit compiler warning.

Emitted from `build.rs` via `println!()`. List not exhaustive.

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

Types

`u8` `String` `Device`

- Set of values with given semantics, layout, …

Type	Values
`u8`	{ 0_{u8}, 1_{u8}, …, 255_{u8} }
`char`	{ 'a', 'b', … '🦀' }
`struct S(u8, char)`	{ (0_{u8}, 'a'), … (255_{u8}, '🦀') }

Sample types and sample values.

Type Equivalence and Conversions

`u8` `&u8` `&mut u8` `[u8; 1]` `String`

- It may be obvious but `u8`, `&u8`, `&mut u8`, are entirely different from each other
- Any `t: T` only accepts values from exactly `T`, e.g.,
  - `f(0_u8)` can't be called with `f(&0_u8)`,
  - `f(&mut my_u8)` can't be called with `f(&my_u8)`,
  - `f(0_u8)` can't be called with `f(0_i8)`.

Yes, $0 \neq 0$ (in a mathematical sense) when it comes to types! In a language sense, the operation $==(0_{u8}, 0_{u16})$ just isn't defined to prevent happy little accidents.

Type	Values
`u8`	{ 0_{u8}, 1_{u8}, …, 255_{u8} }
`u16`	{ 0_{u16}, 1_{u16}, …, 65_535_{u16} }
`&u8`	{ $0\times ffaa_{\&u8}$, $0\times ffbb_{\&u8}$, … }
`&mut u8`	{ $0\times ffaa_{\&mut\ u8}$, $0\times ffbb_{\&mut\ u8}$, … }

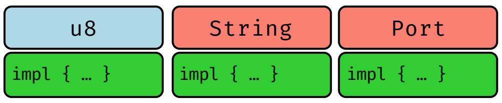
How values differ between types.

- However, Rust might sometimes help to **convert between types**[1]
  - **casts** manually convert values of types, `0_i8 as u8`
  - **coercions** automatically convert types if safe[2], `let x: &u8 = &mut 0_u8;`

[1] Casts and coercions convert values from one set (e.g., `u8`) to another (e.g., `u16`), possibly adding CPU instructions to do so; and in such differ from **subtyping**, which would imply type and subtype are part of the same set (e.g., `u8` being subtype of `u16` and `0_u8` being the same as `0_u16`) where such a conversion would be purely a compile time check. Rust does not use subtyping for regular types (and `0_u8` *does* differ from `0_u16`) but sort-of for lifetimes.

[2] Safety here is not just physical concept (e.g., `&u8` can't be coerced to `&u128`), but also whether 'history has shown that such a conversion would lead to programming errors'.

Implementations — `impl S { }`

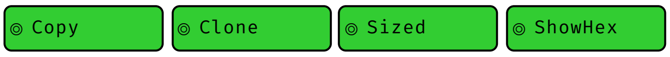


```
impl Port {
    fn f() { … }
}
```

- Types usually come with **inherent implementations**, REF e.g., `impl Port {}`, behavior *related* to type:
  - **associated functions** `Port::new(80)`
  - **methods** `port.close()`

What's considered *related* is more philosophical than technical, nothing (except good taste) would prevent a `u8::play_sound()` from happening.

Traits — `trait T { }`



- **Traits** …
  - are way to "abstract" behavior,
  - trait author declares semantically *this trait means X*,
  - other can implement ("subscribe to") that behavior for their type.
- Think about trait as "membership list" for types:

Copy Trait
`Self`
`u8`
`u16`
…

Clone Trait
`Self`
`u8`
`String`
…

Sized Trait
`Self`
`char`
`Port`
…

Traits as membership tables, `Self` refers to the type included.

- **Whoever is part of that membership list will adhere to behavior of list.**
- Traits can also include associated methods, functions, …

```
trait ShowHex {
    // Must be implemented according to documentation.
    fn as_hex() -> String;

    // Provided by trait author.
    fn print_hex() {}
}
```

```
trait Copy { }
```

- Traits without methods often called **marker traits**.
- `Copy` is example marker trait, meaning *memory may be copied bitwise*.

- Some traits entirely outside explicit control
- `Sized` provided by compiler for types with *known size*; either this is, or isn't

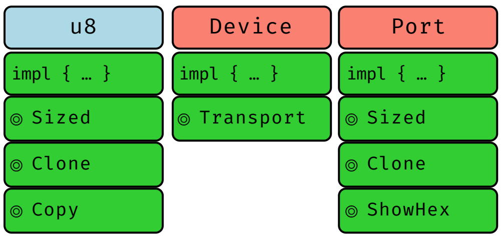
Implementing Traits for Types — `impl T for S { }`

```
impl ShowHex for Port { … }
```

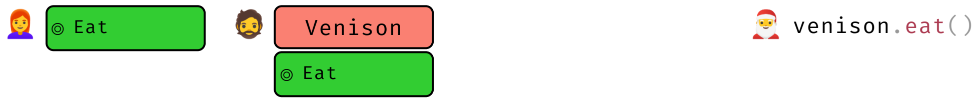
- Traits are implemented for types 'at some point'.
- Implementation `impl A for B` add type `B` to the trait membership list:

ShowHex Trait
Self
Port

- Visually, you can think of the type getting a "badge" for its membership:



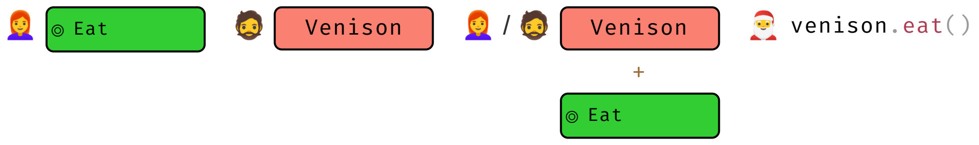
Traits vs. Interfaces



Interfaces

- In **Java**, Alice creates interface `Eat`.
- When Bob authors `Venison`, he must decide if `Venison` implements `Eat` or not.
- In other words, all membership must be exhaustively declared during type definition.
- When using `Venison`, Santa can make use of behavior provided by `Eat`:

```
// Santa imports `Venison` to create it, can `eat()` if he wants.
import food.Venison;

new Venison("rudolph").eat();
```



Traits

- In **Rust**, Alice creates trait `Eat`.
- Bob creates type `Venison` and decides not to implement `Eat` (he might not even know about `Eat`).
- Someone[*] later decides adding `Eat` to `Venison` would be a really good idea.
- When using `Venison` Santa must import `Eat` separately:

```rust
// Santa needs to import `Venison` to create it, and import `Eat` for trait method.
use food::Venison;
use tasks::Eat;

// Ho ho ho
Venison::new("rudolph").eat();
```

[*] To prevent two persons from implementing `Eat` differently Rust limits that choice to either Alice or Bob; that is, an `impl Eat for Venison` may only happen in the crate of `Venison` or in the crate of `Eat`. For details see coherence. [?]

Generics

Type Constructors — `Vec<>`

`Vec<u8>` `Vec<char>`

- `Vec<u8>` is type "vector of bytes"; `Vec<char>` is type "vector of chars", but what is `Vec<>`?

Construct	Values
`Vec<u8>`	`{ [], [1], [1, 2, 3], … }`
`Vec<char>`	`{ [], ['a'], ['x', 'y', 'z'], … }`
`Vec<>`	-

Types vs type constructors.

`Vec<>`

- `Vec<>` is no type, does not occupy memory, can't even be translated to code.
- `Vec<>` is **type constructor**, a "template" or "recipe to create types"
  - allows 3rd party to construct concrete type via parameter,
  - only then would this `Vec<UserType>` become real type itself.

Generic Parameters — `<T>`

`Vec<T>` `[T; 128]` `&T` `&mut T` `S<T>`

- Parameter for `Vec<>` often named `T` therefore `Vec<T>`.
- `T` "variable name for type" for user to plug in something specific, `Vec<f32>`, `S<u8>`, …

Type Constructor	Produces Family
`struct Vec<T> {}`	`Vec<u8>`, `Vec<f32>`, `Vec<Vec<u8>>`, …
`[T; 128]`	`[u8; 128]`, `[char; 128]`, `[Port; 128]` …
`&T`	`&u8`, `&u16`, `&str`, …

Type vs type constructors.

```rust
// S<> is type constructor with parameter T; user can supply any concrete type for T.
struct S<T> {
    x: T
}

// Within 'concrete' code an existing type must be given for T.
fn f() {
    let x: S<f32> = S::new(0_f32);
}
```

Const Generics — `[T; N]` and `S<const N: usize>`

`[T; n]` `S<const N>`

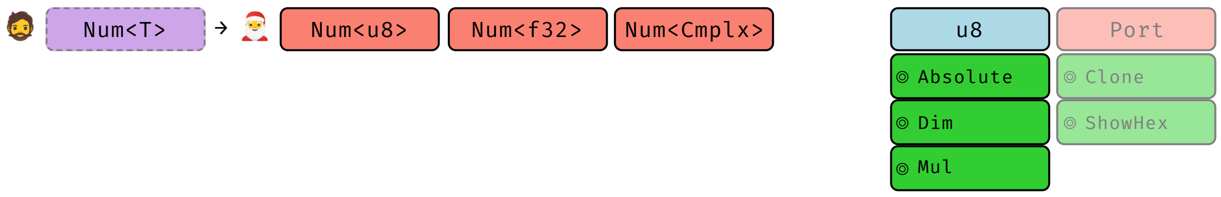
- Some type constructors not only accept specific type, but also **specific constant**.
- `[T; n]` constructs array type holding `T` type `n` times.
- For custom types declared as `MyArray<T, const N: usize>`.

Type Constructor	Produces Family
`[u8; N]`	`[u8; 0]`, `[u8; 1]`, `[u8; 2]`, …
`struct S<const N: usize> {}`	`S<1>`, `S<6>`, `S<123>`, …

Type constructors based on constant.

```rust
let x: [u8; 4]; // "array of 4 bytes"
let y: [f32; 16]; // "array of 16 floats"

// `MyArray` is type constructor requiring concrete type `T` and
// concrete usize `N` to construct specific type.
struct MyArray<T, const N: usize> {
    data: [T; N],
}
```

Bounds (Simple) — `where T: X`



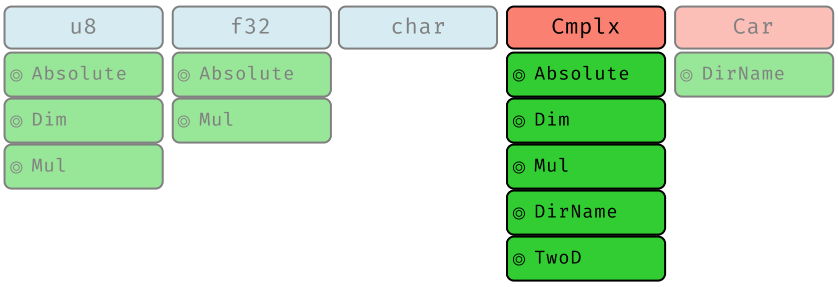
- If `T` can be any type, how can we *reason* about (write code) for such a `Num<T>`?
- Parameter **bounds**:
  - limit what types (**trait bound**) or values (**const bound** ?) allowed,
  - we now can make use of these limits!
- Trait bounds act as "membership check":

```rust
// Type can only be constructed for some `T` if that
// T is part of `Absolute` membership list.
struct Num<T> where T: Absolute {
    …
}
```

Absolute Trait
Self
u8
u16
…

We add bounds to the struct here. In practice it's nicer add bounds to the respective impl blocks instead, see later this section.

Bounds (Compound) — `where T: X + Y`



```rust
struct S<T>
where
    T: Absolute + Dim + Mul + DirName + TwoD
{ … }
```

- Long trait bounds can look intimidating.
- In practice, each `+ X` addition to a bound merely cuts down space of eligible types.

Implementing Families — `impl<>`

When we write:

```rust
impl<T> S<T> where T: Absolute + Dim + Mul {
    fn f(&self, x: T) { … };
}
```

It can be read as:

- here is an implementation recipe for any type `T` (the `impl <T>` part),
- where that type must be member of the `Absolute + Dim + Mul` traits,
- you may add an implementation block to the type family `S<>`,
- containing the methods …

You can think of such `impl<T> … {}` code as **abstractly implementing a family of behaviors**. REF Most notably, they allow 3rd parties to transparently materialize implementations similarly to how type constructors materialize types:

```rust
// If compiler encounters this, it will
// - check `0` and `x` fulfill the membership requirements of `T`
// - create two new version of `f`, one for `char`, another one for `u32`.
// - based on "family implementation" provided
s.f(0_u32);
s.f('x');
```

Blanket Implementations — `impl<T> X for T { … }`

Can also write "family implementations" so they apply trait to many types:

```
// Also implements Serialize for any type if that type already implements ToHex
impl<T> Serialize for T where T: ToHex { … }
```

These are called **blanket implementations**.

ToHex
`Self`
`Port`
`Device`
…

→ Whatever was in left table, may be added to right table, based on the following recipe (`impl`) →

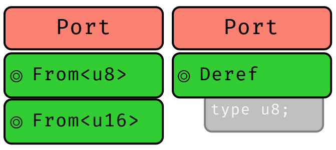
Serialize Trait
`Self`
`u8`
`Port`
…

They can be neat way to give foreign types functionality in a modular way if they just implement another interface.

Advanced Concepts

Trait Parameters — `Trait<In> { type Out; }`

Notice how some traits can be "attached" multiple times, but others just once?



Why is that?

- Traits themselves can be generic over two **kinds of parameters**:
    - `trait From<I> {}`
    - `trait Deref { type O; }`
- Remember we said traits are "membership lists" for types and called the list `Self`?
- Turns out, parameters `I` (for **input**) and `O` (for **output**) are just more *columns* to that trait's list:

```
impl From<u8> for u16 {}
impl From<u16> for u32 {}
impl Deref for Port { type O = u8; }
impl Deref for String { type O = str; }
```

From	
`Self`	**`I`**
`u16`	`u8`
`u32`	`u16`
…	

Deref	
`Self`	**`O`**
`Port`	`u8`
`String`	`str`
…	

Input and output parameters.

Now here's the twist,

- **any output `O` parameters must be uniquely determined by input parameters `I`**,
- (in the same way as a relation `X Y` would represent a function),
- `Self` counts as an input.

A more complex example:

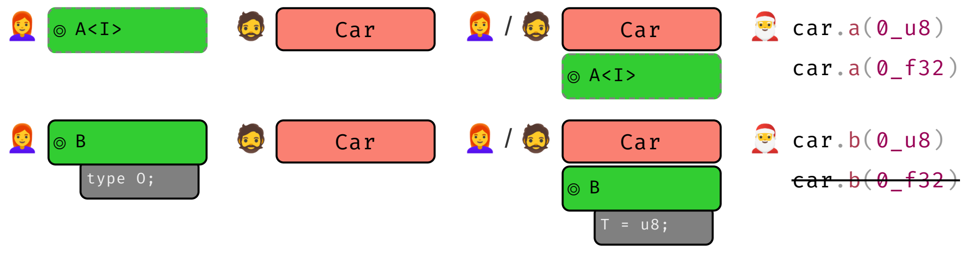
```
trait Complex<I1, I2> {
    type O1;
    type O2;
}
```

- this creates a relation of types named `Complex`,
- with 3 inputs (`Self` is always one) and 2 outputs, and it holds `(Self, I1, I2) => (O1, O2)`

Complex				
Self [I]	**I1**	**I2**	**O1**	**O2**
`Player`	`u8`	`char`	`f32`	`f32`
`EvilMonster`	`u16`	`str`	`u8`	`u8`
`EvilMonster`	`u16`	`String`	`u8`	`u8`
`NiceMonster`	`u16`	`String`	`u8`	`u8`
`NiceMonster`●	`u16`	`String`	`u8`	`u16`

Various trait implementations. The last one is not valid as `(NiceMonster, u16, String)` has already uniquely determined the outputs.

Trait Authoring Considerations (Abstract)



- Parameter choice (input vs. output) also determines who may be allowed to add members:
    - `I` parameters allow "familes of implementations" be forwarded to user (Santa),
    - `O` parameters must be determined by trait implementor (Alice or Bob).

```
trait A<I> { }
trait B { type O; }

// Implementor adds (X, u32) to A.
impl A<u32> for X { }

// Implementor adds family impl. (X, …) to A, user can materialze.
impl<T> A<T> for Y { }

// Implementor must decide specific entry (X, O) added to B.
impl B for X { type O = u32; }
```

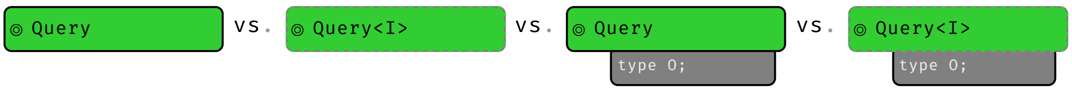
A	
Self	**I**
X	u32
Y	…

Santa may add more members by providing his own type for `T`.

B	
Self	**O**
Player	String
X	u32

For given set of inputs (here `Self`), implementor must pre-select `O`.

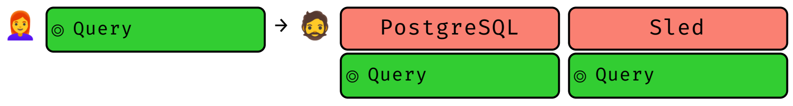
Trait Authoring Considerations (Example)



Choice of parameters goes along with purpose trait has to fill.

No Additional Parameters

```
trait Query {
    fn search(&self, needle: &str);
}

impl Query for PostgreSQL { … }
impl Query for Sled { … }

postgres.search("SELECT …");
```

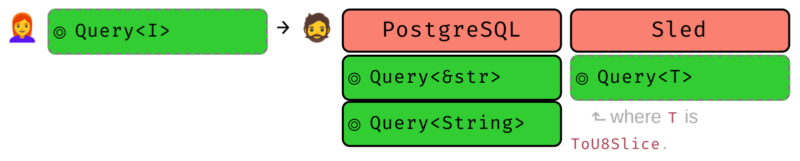


Trait author assumes:

- neither implementor nor user need to customize API.

Input Parameters

```
trait Query<I> {
    fn search(&self, needle: I);
}

impl Query<&str> for PostgreSQL { … }
impl Query<String> for PostgreSQL { … }
impl<T> Query<T> for Sled where T: ToU8Slice { … }

postgres.search("SELECT …");
postgres.search(input.to_string());
sled.search(file);
```

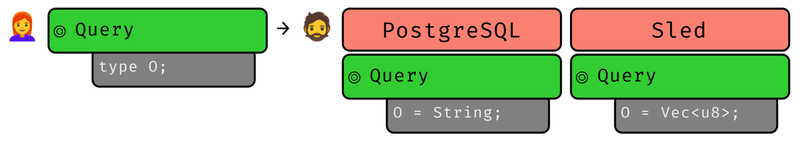


Trait author assumes:

- implementor would customize API in multiple ways for same `Self` type,
- users may want ability to decide for which `I`-types behavior should be possible.

Output Parameters

```
trait Query {
    type O;
    fn search(&self, needle: Self::O);
}

impl Query for PostgreSQL { type O = String; …}
impl Query for Sled { type O = Vec<u8>; … }

postgres.search("SELECT …".to_string());
sled.search(vec![0, 1, 2, 4]);
```



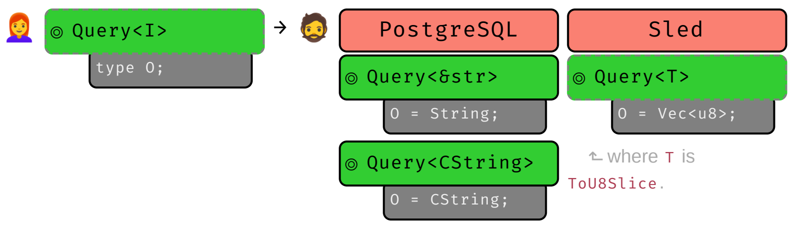
Trait author assumes:

- implementor would customize API for `Self` type (but in only one way),
- users do not need, or should not have, ability to influence customization for specific `Self`.

As you can see here, the term **input** or **output** does **not** (necessarily) have anything to do with whether `I` or `O` are inputs or outputs to an actual function!

Multiple In- and Output Parameters

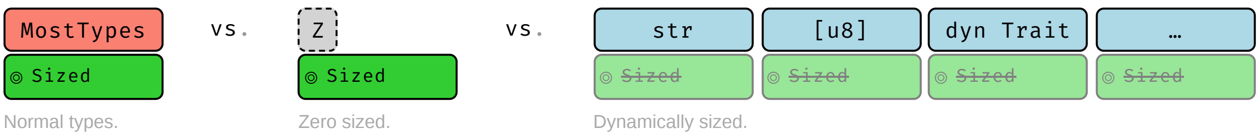
```
trait Query<I> {
    type O;
    fn search(&self, needle: I) -> Self::O;
}

impl Query<&str> for PostgreSQL { type O = String; … }
impl Query<CString> for PostgreSQL { type O = CString; … }
impl<T> Query<T> for Sled where T: ToU8Slice { type O = Vec<u8>; … }

postgres.search("SELECT …").to_uppercase();
sled.search(&[1, 2, 3, 4]).pop();
```



Like examples above, in particular trait author assumes:

- users may want ability to decide for which `I`-types ability should be possible,
- for given inputs, implementor should determine resulting output type.

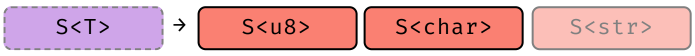
Dynamic / Zero Sized Types



- A type `T` is **`Sized`** STD if at compile time it is known how many bytes it occupies, `u8` and `&[u8]` are, `[u8]` isn't.
- Being `Sized` means `impl Sized for T {}` holds. Happens automatically and cannot be user impl'ed.
- Types not `Sized` are called **dynamically sized types** BK NOM REF (DSTs), sometimes **unsized**.
- Types without data are called **zero sized types** NOM (ZSTs), do not occupy space.

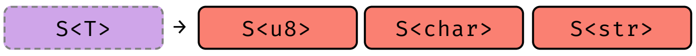
Example	Explanation
`struct A { x: u8 }`	Type `A` is sized, i.e., `impl Sized for A` holds, this is a 'regular' type.
`struct B { x: [u8] }`	Since `[u8]` is a DST, `B` in turn becomes DST, i.e., does not `impl Sized`.
`struct C<T> { x: T }`	Type params **have** implicit `T: Sized` bound, e.g., `C<A>` is valid, `C<B>` is not.
`struct D<T: ?Sized> { x: T }`	Using **`?Sized`** REF allows opt-out of that bound, i.e., `D<B>` is also valid.
`struct E;`	Type `E` is zero-sized (and also sized) and will not consume memory.
`trait F { fn f(&self); }`	Traits **do not have** an implicit `Sized` bound, i.e., `impl F for B {}` is valid.
`trait F: Sized {}`	Traits can however opt into `Sized` via supertraits.↑
`trait G { fn g(self); }`	For `Self`-like params DST `impl` may still fail as params can't go on stack.

?Sized



```
struct S<T> { … }
```

- `T` can be any concrete type.
- However, there exists invisible default bound `T: Sized`, so `S<str>` is not possible out of box.
- Instead we have to add `T : ?Sized` to opt-out of that bound:



```
struct S<T> where T: ?Sized { … }
```

Generics and Lifetimes — `<'a>`

`S<'a>` `&'a f32` `&'a mut u8`

- Lifetimes act* as type parameters:
  - user must provide specific `'a` to instantiate type (compiler will help within methods),
  - `S<'p>` and `S<'q>` are different types, just like `Vec<f32>` and `Vec<u8>` are
  - meaning you can't just assign value of type `S<'a>` to variable expecting `S<'b>` (exception: subtype relationship for lifetimes, i.e., `'a` outlives `'b`).

`S<'a>` → `S<'auto>` `S<'static>`

- `'static` is only globally available type of the lifetimes *kind*.

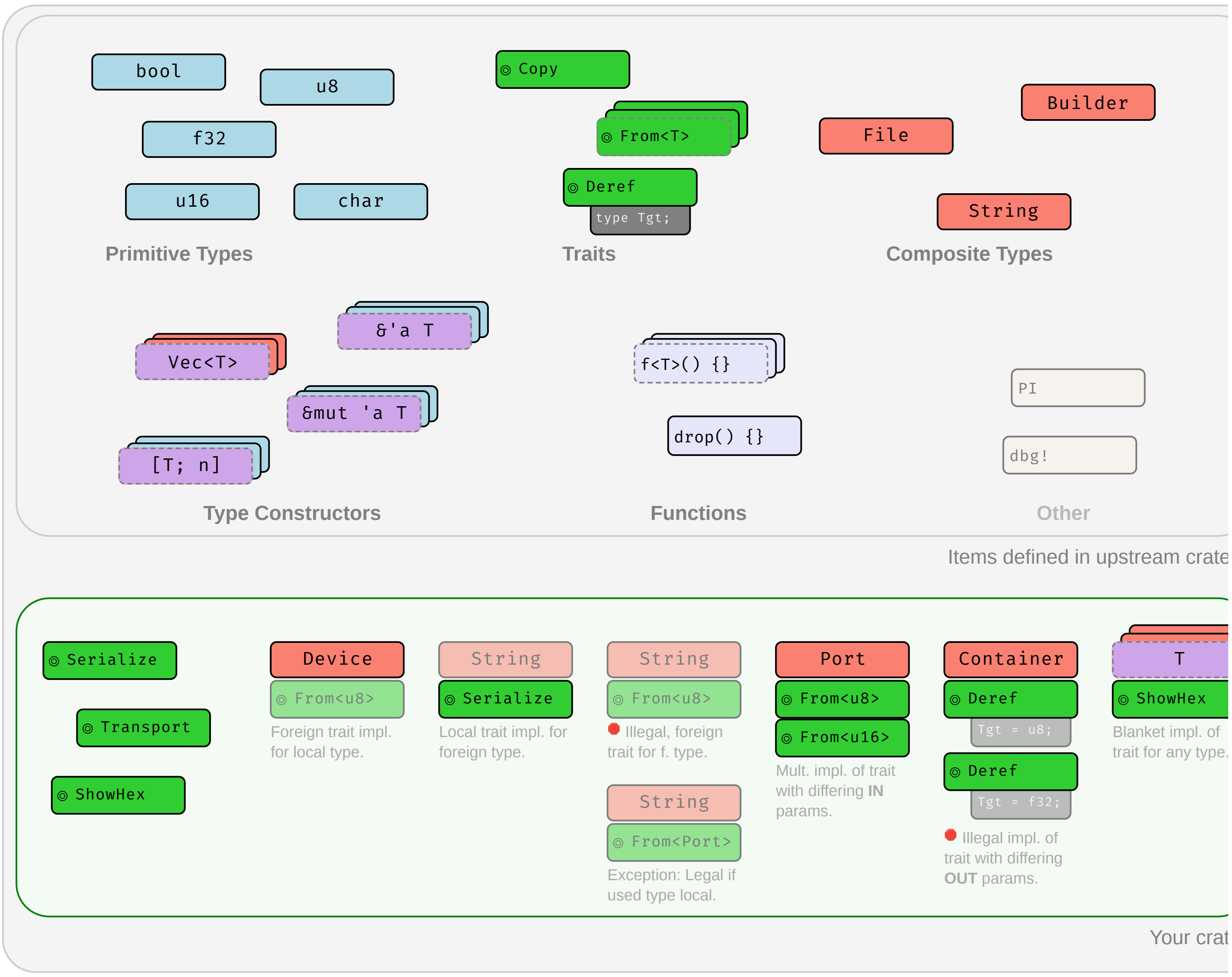
```rust
// `'a is free parameter here (user can pass any specific lifetime)
struct S<'a> {
    x: &'a u32
}

// In non-generic code, 'static is the only nameable lifetime we can explicitly put in here.
let a: S<'static>;

// Alternatively, in non-generic code we can (often must) omit 'a and have Rust determine
// the right value for 'a automatically.
let b: S;
```

* There are subtle differences, for example you can create an explicit instance `0` of a type `u32`, but with the exception of `'static` you can't really create a lifetime, e.g., "lines 80 - 100", the compiler will do that for you.

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

```
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.

Computation (Traits)

```
fn f(x: A) -> B {
    x.into()
}
```

Bread and butter way to get `B` from `A`. Some traits provide canonical, user-computable type relations:

Trait	Example	Trait implies …
`impl From<A> for B {}`	`a.into()`	*Obvious*, always-valid relation.
`impl TryFrom<A> for B {}`	`a.try_into()?`	*Obvious*, sometimes-valid relation.
`impl Deref for A {}`	`*a`	`A` is smart pointer carrying `B`; also enables coercions.
`impl AsRef<B> for A {}`	`a.as_ref()`	`A` can be *viewed* as `B`.
`impl AsMut<B> for A {}`	`a.as_mut()`	`A` can be mutably viewed as `B`.
`impl Borrow<B> for A {}`	`a.borrow()`	`A` has borrowed *analog* `B` (behaving same under `Eq`, …).
`impl ToOwned for A { … }`	`a.to_owned()`	`A` has owned analog `B`.

Casts

```
fn f(x: A) -> B {
    x as B
}
```

Convert types **with keyword `as`** if conversion *relatively obvious* but **might cause issues**. NOM

A	B	Example	Explanation
`Pointer`	`Pointer`	`device_ptr as *const u8`	If `*A`, `*B` are `Sized`.
`Pointer`	`Integer`	`device_ptr as usize`	
`Integer`	`Pointer`	`my_usize as *const Device`	
`Number`	`Number`	`my_u8 as u16`	Often surprising behavior. ↑
`enum` w/o fields	`Integer`	`E::A as u8`	
`bool`	`Integer`	`true as u8`	
`char`	`Integer`	`'A' as u8`	
`&[T; N]`	`*const T`	`my_ref as *const u8`	
`fn(…)`	`Pointer`	`f as *const u8`	If `Pointer` is `Sized`.
`fn(…)`	`Integer`	`f as usize`	

Where `Pointer`, `Integer`, `Number` are just used for brevity and actually mean:

- `Pointer` any `*const T` or `*mut T`;
- `Integer` any countable `u8` … `i128`;

- `Number` any `Integer`, `f32`, `f64`.

Opinion 💬 — Casts, esp. `Number - Number`, can easily go wrong. If you are concerned with correctness, consider more explicit methods instead.

Coercions

```rust
fn f(x: A) -> B {
    x
}
```

Automatically **weaken** type `A` to `B`; types can be *substantially*[1] different. NOM

A	B	Explanation
`&mut T`	`&T`	**Pointer weakening**.
`&mut T`	`*mut T`	-
`&T`	`*const T`	-
`*mut T`	`*const T`	-
`&T`	`&U`	**Deref**, if `impl Deref<Target=U> for T`.
`T`	`U`	**Unsizing**, if `impl CoerceUnsized<U> for T`.[2] 🚧
`T`	`V`	**Transitivity**, if `T` coerces to `U` and `U` to `V`.
`|x| x + x`	`fn(u8) -> u8`	**Non-capturing closure**, to equivalent `fn` pointer.

[1] *Substantially* meaning one can regularly expect a coercion result `B` to be *an entirely different type* (i.e., have entirely different methods) than the original type `A`.

[2] Does not quite work in example above as unsized can't be on stack; imagine `f(x: &A) -> &B` instead. Unsizing works by default for:

- `[T; n]` to `[T]`
- `T` to `dyn Trait` if `impl Trait for T {}`.
- `Foo<…, T, …>` to `Foo<…, U, …>` under arcane 🔗 circumstances.

Subtyping 🝖

```rust
fn f(x: A) -> B {
    x
}
```

Automatically converts `A` to `B` for types **only differing in lifetimes** NOM - subtyping **examples**:

A(subtype)	B(supertype)	Explanation
`&'static u8`	`&'a u8`	Valid, *forever*-pointer is also *transient*-pointer.
`&'a u8`	`&'static u8`	🛑 Invalid, transient should not be forever.

A(subtype)	B(supertype)	Explanation
`&'a &'b u8`	`&'a &'b u8`	Valid, same thing. **But now things get interesting. Read on.**
`&'a &'static u8`	`&'a &'b u8`	Valid, `&'static u8` is also `&'b u8`; **covariant** inside `&`.
`&'a mut &'static u8`	`&'a mut &'b u8`	🛑 Invalid and surprising; **invariant** inside `&mut`.
`Box<&'static u8>`	`Box<&'a u8>`	Valid, `Box` with forever is also box with transient; covariant.
`Box<&'a u8>`	`Box<&'static u8>`	🛑 Invalid, `Box` with transient may not be with forever.
`Box<&'a mut u8>`	`Box<&'a u8>`	🛑 ⚡ Invalid, see table below, `&mut u8` never *was a* `&u8`.
`Cell<&'static u8>`	`Cell<&'a u8>`	🛑 Invalid, `Cell` are **never** something else; invariant.
`fn(&'static u8)`	`fn(&'a u8)`	🛑 If `fn` needs forever it may choke on transients; **contravar.**
`fn(&'a u8)`	`fn(&'static u8)`	But sth. that eats transients **can be**(!) sth. that eats forevers.
`for<'r> fn(&'r u8)`	`fn(&'a u8)`	Higher-ranked type `for<'r> fn(&'r u8)` is also `fn(&'a u8)`.

In contrast, these are **not**🛑 examples of subtyping:

A	B	Explanation
`u16`	`u8`	🛑 **Obviously invalid**; `u16` should never automatically be `u8`.
`u8`	`u16`	🛑 Invalid **by design**; types w. different data still never subtype even if they *could*.
`&'a mut u8`	`&'a u8`	🛑 Trojan horse, not subtyping; but coercion (still works, just not subtyping).

Variance

```
fn f(x: A) -> B {
    x
}
```

Automatically converts `A` to `B` for types **only differing in lifetimes** NOM - subtyping **variance rules**:

- A longer lifetime `'a` that outlives a shorter `'b` is a subtype of `'b`.
- Implies `'static` is subtype of all other lifetimes `'a`.
- Whether types with parameters (e.g., `&'a T`) are subtypes of each other the following variance table is used:

Construct[1]	`'a`	`T`	`U`
`&'a T`	covariant	covariant	
`&'a mut T`	covariant	invariant	
`Box<T>`		covariant	
`Cell<T>`		invariant	
`fn(T) -> U`		**contra**variant	covariant
`*const T`		covariant	
`*mut T`		invariant	

Covariant means if `A` is subtype of `B`, then `T[A]` is subtype of `T[B]`.
Contravariant means if `A` is subtype of `B`, then **`T[B]`** is subtype of `T[A]`.

Invariant means even if `A` is subtype of `B`, neither `T[A]` nor `T[B]` will be subtype of the other.

[1] Compounds like `struct S<T> {}` obtain variance through their used fields, usually becoming invariant if multiple variances are mixed.

💡 **In other words**, 'regular' types are never subtypes of each other (e.g., `u8` is not subtype of `u16`), and a `Box<u32>` would never be sub- or supertype of anything. However, generally a `Box<A>`, *can* be subtype of `Box<B>` (via covariance) if `A` is a subtype of `B`, which can only happen if `A` and `B` are 'sort of the same type that only differed in lifetimes', e.g., `A` being `&'static u32` and `B` being `&'a u32`.

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::builder().name("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.

Idiom	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.
	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`:

```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

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.

Execution Flow

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.
```

Caveats 🔴

With the execution flow in mind, some 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` 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

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).
```

Unsafe 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);
}
```

Undefined Behavior

Undefined Behavior (UB)

- As mentioned, `unsafe` code implies special promises to the compiler (it wouldn't need be `unsafe` otherwise).
- Failure to uphold any promise makes compiler produce fallacious code, execution of which leads to UB.
- After triggering undefined behavior *anything* can happen. Insidiously, the effects may be 1) subtle, 2) manifest far away from the site of violation or 3) be visible only under certain conditions.
- A seemingly *working* program (incl. any number of unit tests) is no proof UB code might not fail on a whim.

- Code with UB is objectively dangerous, invalid and should never exist.

```
if maybe_true() {
    let r: &u8 = unsafe { &*ptr::null() };   // Once this runs, ENTIRE app is undefined.
Even if
} else {                                     // line seemingly didn't do anything, app might
now run
    println!("the spanish inquisition");     // both paths, corrupt database, or anything
else.
}
```

Unsound Code

Unsound Code

- Any *safe* Rust that could (even only theoretically) produce UB for any user input is always **unsound**.
- As is `unsafe` code that may invoke UB on its own accord by violating above-mentioned promises.
- Unsound code is a stability and security risk, and violates basic assumption many Rust users have.

```
fn unsound_ref<T>(x: &T) -> &u128 {      // Signature looks safe to users. Happens to be
    unsafe { mem::transmute(x) }         // ok if invoked with an &u128, UB for practically
}                                        // everything else.
```

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.

Risk Pattern	Description
`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.

Traits
🟡 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