# Rust Language Cheat Sheet

10. July 2023

Contains clickable links to **The Book** BK, **Rust by Example** EX, **Std Docs** STD, **Nomicon** NOM, **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 }` | If variants are only unit-like, allow **discriminant values**, REF e.g., for FFI. |
| `enum E {}` | Enum w/o variants is **uninhabited**, REF can't be instantiated, *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 memory location. |
| `const X: T = T();` | Defines **constant**, BK EX REF copied into a temporary when used. |
| `let x: T;` | Allocate `T` bytes on stack[1] bound as `x`. Assignable once, not mutable. |
| `let mut x: T;` | Like `let`, but allow for **mutability** BK EX and mutable borrow.[2] |
| `x = y;` | Moves `y` to `x`, invalidating `y` if `T` is not `Copy`, STD and copying `y` otherwise. |

[1] **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.
[2] 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 |

| Example | Explanation |
|---|---|
| `(x)` | Parenthesized expression. |
| `(x,)` | Single-element **tuple** expression. EX STD REF |
| `(S,)` | Single-element tuple type. |
| `[S]` | Array type of unspecified length, i.e., **slice**. EX STD REF Can't live on stack. * |
| `[S; n]` | **Array type** EX STD REF of fixed length `n` holding elements of type `S`. |
| `[x; n]` | **Array instance** REF (expression) with `n` copies of `x`. |
| `[x, y]` | Array instance with given elements `x` and `y`. |
| `x[0]` | Collection indexing, here w. `usize`. Implementable with **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 (`address`, `count`). |
|     `&str` | Special string slice reference that contains (`address`, `byte_length`). |
|     `&mut S` | Exclusive reference to allow mutability (also `&mut [S]`, `&mut dyn S`, …). |
|     `&dyn T` | Special **trait object** BK reference that contains (`address`, `vtable`). |
| `&s` | Shared **borrow** BK EX STD (e.g., address, 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 reference; *c.* `ptr:addr_of!()` STD 🚧 ▯ |
|     `&raw mut s` | Same, but mutable. 🚧 Raw ptrs. are 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;` | Mutable 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 place. |
|     `s = *my_box;` | Special case🔗 for **`Box`**STD that can also move out b'ed content that isn't `Copy`. |

| Example | Explanation |
|---|---|
| `'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 lifetime `'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 function**. RFC |
| `const fn f() {}` | Constant `fn` usable at compile time, e.g., `const X: u32 = f(Y)`. '18 |
| `async fn f() {}` | **Async** REF '18 function transformation, ↓ makes `f` return an `impl` **`Future`**. STD |
| `async fn f() -> S {}` | Same, but make `f` return an `impl Future<Output=S>`. |
| `async { x }` | Used within a function, make `{ x }` an `impl Future<Output=X>`. |
| `fn() -> S` | **Function references**,[1] BK STD REF memory holding address of a callable. |
| `Fn() -> S` | **Callable Trait** BK STD (also `FnMut`, `FnOnce`), implemented by closures, fn's … |
| `\|\| {}` | A **closure** BK EX REF that borrows its **captures**, ↓ 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. |
| `return \|\| true` | Closures sometimes look like logical ORs (here: return a closure). |
| `unsafe` | If you enjoy debugging segfaults Friday night; **unsafe code**. ↓ BK EX NOM REF |
| `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. |

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

## Control Flow

Control execution within a function.

| Example | Explanation |
|---|---|
| `while x {}` | **Loop**, REF run while expression `x` is true. |
| `loop {}` | **Loop indefinitely** REF until `break`. Can yield value with `break x`. |
| `for x in collection {}` | Syntactic sugar to loop over **iterators**. BK STD REF |
| `collection.into_iter()` | Effectively converts any `IntoIterator` STD type into proper iterator first. |
| `iterator.next()` | On proper `Iterator` STD then `x = next()` until exhausted (first `None`). |
| `if x {} else {}` | **Conditional branch** REF if expression is true. |
| `'label: {}` | **Block label**, RFC can be used with `break` to exit out of this block. 1.65+ |
| `'label: loop {}` | Similar **loop label**, EX REF useful for flow control in nested loops. |
| `break` | **Break expression** REF to exit a labelled block or loop. |
| `break 'label x` | Break out of block or loop named `'label` and make `x` its value. |
| `break 'label` | Same, but don't produce any value. |
| `break x` | Make `x` value of the innermost loop (only in actual `loop`). |
| `continue` | **Continue expression** REF to the next loop iteration of this loop. |
| `continue 'label` | Same but instead of this loop, enclosing loop marked with 'label. |
| `x?` | If `x` is Err or None, **return and propagate**. BK EX STD REF |
| `x.await` | Syntactic sugar to **get future, poll, yield**. REF '18 Only works 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 function. More idiomatic is to end with expression. |
| `{ return }` | Inside normal `{}`-blocks `return` exits surrounding function. |
| `\|\| { return }` | Within closures `return` exits that closure only, i.e., closure is *s.* function. |
| `async { return }` | Inside `async` a `return` **only** REF 🛑 exits that `{}`, i.e., `async {}` is *s.* function. |
| `f()` | Invoke callable `f` (e.g., a function, closure, function pointer, `Fn`, …). |
| `x.f()` | Call member function, requires `f` takes `self`, `&self`, … as first argument. |
| `X::f(x)` | Same as `x.f()`. Unless `impl Copy for X {}`, `f` can only be called once. |
| `X::f(&x)` | Same as `x.f()`. |
| `X::f(&mut x)` | Same as `x.f()`. |
| `S::f(&x)` | Same as `x.f()` if `X` derefs to `S`, i.e., `x.f()` finds methods of `S`. |
| `T::f(&x)` | Same as `x.f()` if `X impl T`, i.e., `x.f()` finds methods of `T` if in scope. |
| `X::f()` | Call associated function, e.g., `X::new()`. |
| `<X as T>::f()` | Call trait method `T::f()` implemented for `X`. |

## Organizing Code

Segment projects into smaller units and minimize dependencies.

| Example | Explanation |
|---|---|
| `mod m {}` | Define a **module**, BK EX REF get definition from inside `{}`. ↓ |
| `mod m;` | Define a module, get definition from `m.rs` or `m/mod.rs`. ↓ |
| `a::b` | Namespace **path** EX REF to element `b` within `a` (`mod`, `enum`, …). |
| `::b` | Search `b` in **crate root** '15 REF or **external prelude**; '18 REF **global path**. REF 🗑️ |
| `crate::b` | Search `b` in crate root. '18 |
| `self::b` | Search `b` in current module. |

| Example | Explanation |
|---|---|
| `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` anonymously into scope, useful for traits with conflicting names. |
| `use a::*;` | Bring everything from `a` in, only recommended if `a` is some **prelude**. STD 🔗 |
| `pub use a::b;` | Bring `a::b` into scope and reexport from here. |
| `pub T` | "Public if parent path is public" **visibility** BK REF for `T`. |
| `pub(crate) T` | Visible at most[1] in current crate. |
| `pub(super) T` | Visible at most[1] in parent. |
| `pub(self) T` | Visible at most[1] in current module (default, same as no `pub`). |
| `pub(in a::b) T` | Visible at most[1] in ancestor `a::b`. |
| `extern crate a;` | Declare dependency on external **crate**; BK REF 🗑️ just `use a::b` in '18. |
| `extern "C" {}` | *Declare* external dependencies and ABI (e.g., `"C"`) from **FFI**. BK EX NOM REF |
| `extern "C" fn f() {}` | *Define* function to be exported with ABI (e.g., `"C"`) to FFI. |

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

## Type Aliases and Casts

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

| Example | Explanation |
|---|---|
| `type T = S;` | Create a **type alias**, BK REF i.e., another name for `S`. |
| `Self` | Type alias for **implementing type**, REF e.g., `fn new() -> Self`. |
| `self` | Method subject 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 pointers (`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 |
|---|---|
| `$x:ty` | Macro capture, the `:...` **fragment** REF declares what is allowed for `$x`. [2] |
| `$x` | Macro substitution, e.g., use the captured `$x:ty` from above. |
| `$(x),*` | Macro **repetition** REF *zero or more times*. |

| Inside Macros [1] | Explanation |
|---|---|
| `$(x),?` | Same, but *zero or one time*. |
| `$(x),+` | Same, but *one or more times*. |
| `$(x)<<+` | In fact separators other than `,` are also accepted. Here: `<<`. |

[1] Applies to **'macros by example'**. REF
[2] See **Tooling Directives** below for all captures.

## Pattern Matching

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

| Example | Explanation |
|---|---|
| `match m {}` | Initiate **pattern matching**, BK EX REF then use match arms, *c*. next table. |
| `let S(x) = get();` | Notably, `let` also **destructures** EX similar to the table below. |
| `let S { x } = s;` | Only `x` will be bound to value `s.x`. |
| `let (_, b, _) = abc;` | Only `b` will be bound to value `abc.1`. |
| `let (a, ..) = abc;` | Ignoring 'the rest' also works. |
| `let (.., a, b) = (1, 2);` | Specific bindings take precedence over 'the rest', here `a` is `1`, `b` is `2`. |
| `let s @ S { x } = get();` | Bind `s` to `S` while `x` is bound 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 Some(x) = get();` | **Won't** work 🔴 if pattern can be **refuted**, REF use `let else` or `if let` instead. |
| `let Some(x) = get() else {};` | Assign if possible,RFC if not `else {}` w. must `break`, `return`, `panic!`, … 1.65+ 🔥 |
| `if let Some(x) = get() {}` | Branch if pattern can be assigned (e.g., `enum` variant), syntactic sugar. * |
| `while let Some(x) = get() {}` | Equiv.; here keep calling `get()`, run `{}` as long as pattern can be assigned. |
| `fn f(S { x }: S)` | Function parameters 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 struct with specific values (only accepts `s` with `s.x` of `0` and `s.y` of `1`). |
| `S { x: a, y: b } => {}` | Match struct with *any* 🔴 values and bind `s.x` to `a` and `s.y` to `b`. |
| `S { x, y } => {}` | Same, but shorthand with `s.x` and `s.y` bound as `x` and `y` respectively. |
| `S { .. } => {}` | Match struct with any values. |
| `D => {}` | Match enum variant `E::D` if `D` in `use`. |
| `D => {}` | Match anything, bind `D`; possibly false friend 🔴 of `E::D` if `D` not in `use`. |
| `_ => {}` | Proper wildcard that matches anything / "all the rest". |
| `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`, therefore `x`. ▯ |

| Within Match Arm | Explanation |
|---|---|
| `(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`, `2`, ... or `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 name here). |
| `S<T> where T: R` | **Trait bound**, BK EX REF limits allowed `T`, guarantees `T` has `R`; `R` must be trait. |
| `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 lifetimes, 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 esp. *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>` | Also allows you to 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 it 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 lifetime**; asks compiler to *'figure it out'* if obvious. |
| `S<_>` | Inferred **anonymous type**, e.g., as `let x: Vec<_> = iter.collect()` |
| `S::<T>` | **Turbofish** STD call site type disambiguation, e.g., `f::<u32>()`. |
| `trait T<X> {}` | A trait generic over `X`. Can have multiple `impl T for S` (one per `X`). |
| `trait T { type X; }` | Defines **associated type** BK REF RFC `X`. Only one `impl T for S` possible. |
| `trait T { type X<G>; }` | Defines **generic associated type** (GAT), RFC e.g., `X` can be generic `Vec<>`. 1.65+ |
| `trait T { type X<'a>; }` | Defines a GAT generic over a lifetime. |

| Example | Explanation |
|---|---|
| `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> {}` | Implement `fn`'s for any `T` in `S<T>` ***generically***, REF here `T` type parameter. |
| `impl S<T> {}` | Implement `fn`'s for exactly `S<T>` ***inherently***, REF here `T` specific type, e.g., `u8`. |
| `fn f() -> impl T` | **Existential types**, BK returns an unknown-to-caller `S` that `impl T`. |
| `fn f(x: &impl T)` | Trait bound via "**impl traits**", BK somewhat like `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)` | Function 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 `dyn T` trait object vtable, enabling trait obj. |
| `fn f() where Self: R {}` | Other `R` useful w. dflt. methods (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 callable 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 UTF-8, will interpret the following 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 □ . |
| `r"..."` | **Raw string literal**. REF, 1UTF-8, but won't interpret any escape above. |
| `r#"..."#` | Raw string literal, UTF-8, but can also contain `"`. Number of `#` can vary. |
| `b"..."` | **Byte string literal**; REF, 1 constructs ASCII `[u8]`, not a string. |

| Example | Explanation |
|---|---|
| `br"..."`, `br#"..."#` | Raw byte string literal, ASCII `[u8]`, combination of the above. |
| `'🦀'` | **Character literal**, REF fixed 4 byte unicode '**char**'. STD |
| `b'x'` | ASCII **byte literal**, REF a single `u8` byte. |

[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 types, traits, functions, … |
| `//!` | Inner line doc comment, mostly used at start of file to document module. |
| `//` | 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 returns; compat. with any type 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 assignment is no-op, does **not** 🛑 move out `x` or preserve scope! |
| `_ = x;` | You can assign *anything* to `_` without `let`, i.e., `_ = ignore_error();` 1.59+ 🔥 |
| `_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. |
| `r#foo` | A **raw identifier** BK EX 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.

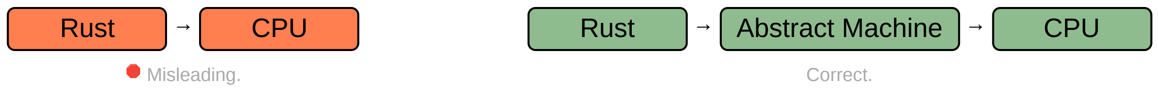
# Behind the Scenes

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**



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 *'domain'* attached for optimization. |
| Any r/w on pointer `0xff` always fine. 🔴 | AM may issue cache-friendly ops trusting *'no read can happen'*. |
| Reading un-init just gives random value. 🔴 | AM *'knows'* read impossible, may remove all related bitcode. |
| Data race just gives random value. 🔴 | AM may split R/W, produce *impossible* value, see above. |
| Null reference 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.

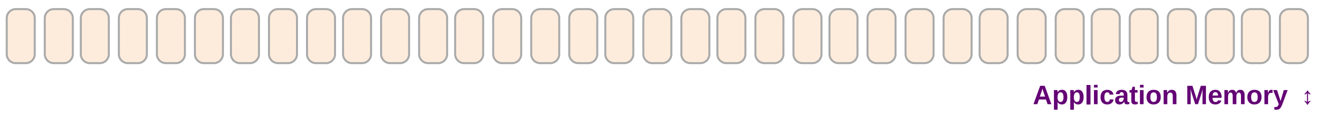
| Name | Description |
|---|---|
| **Coercions** NOM | *Weakens* types to match signature, e.g., `&mut T` to `&T`; *c. type conversions*. ↓ |
| **Deref** NOM 🔗 | Derefs `x: T` until `*x`, `**x`, … compatible with some target `S`. |
| **Prelude** STD | Automatic import of basic items, e.g., `Option`, `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. |
| **Method Resolution** REF | Derefs or borrow `x` until `x.f()` works. |
| **Match Ergonomics** RFC | Repeatedly dereferences scrutinee and adds `ref` and `ref mut` to bindings. |
| **Rvalue Static Promotion** RFC | Makes references to constants `'static`, e.g., `&42`, `&None`, `&mut []`. |
| **Dual Definitions** RFC | Defining one thing (e.g., `struct S(u8)`) implicitly def. another (e.g., `fn S`). |

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

## Memory & Lifetimes

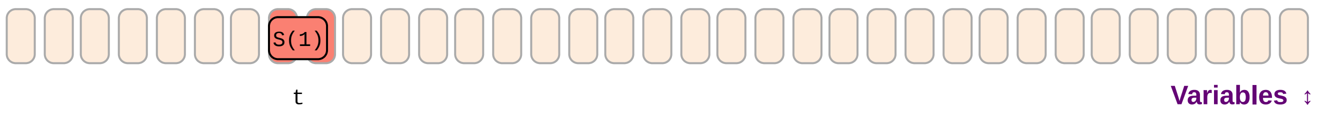
An illustrated guide to moves, references and lifetimes.

**Types & Moves**



- 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 managable: *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.



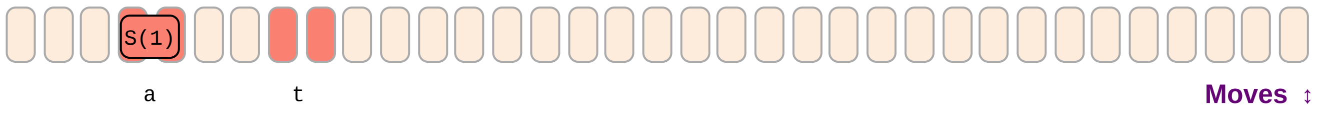
```rust
let t = S(1);
```

- Reserves memory location with name `t` of type `S` and the value `S(1)` stored inside.

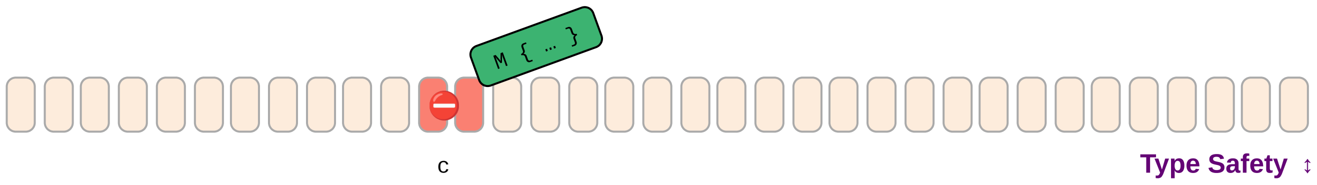
1

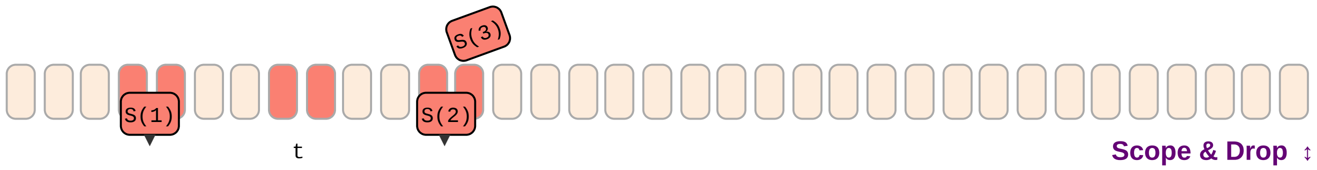
2

1 ↑



↓

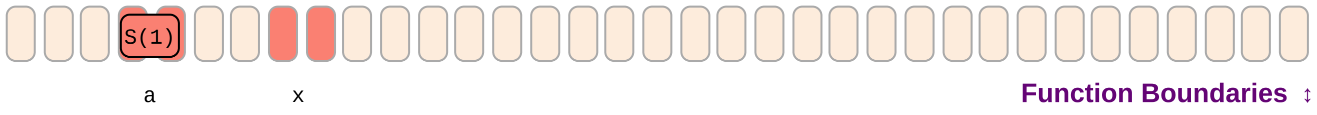




**Scope & Drop** ↕

mem::

Call Stack

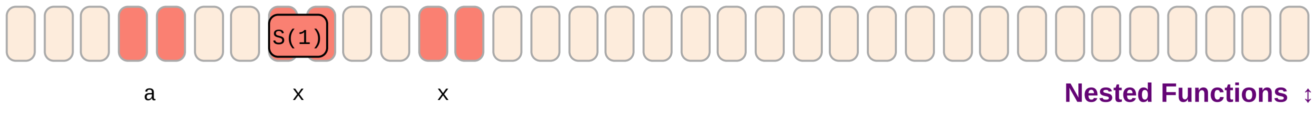


**Function Boundaries** ↕

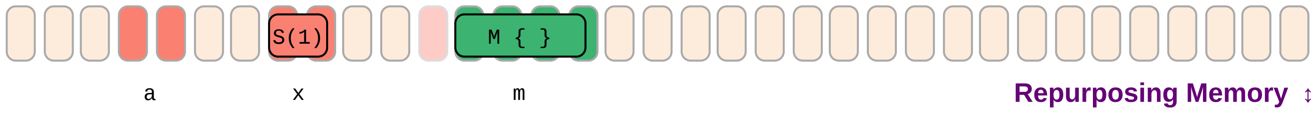
```
fn f(x: S) { … }

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.

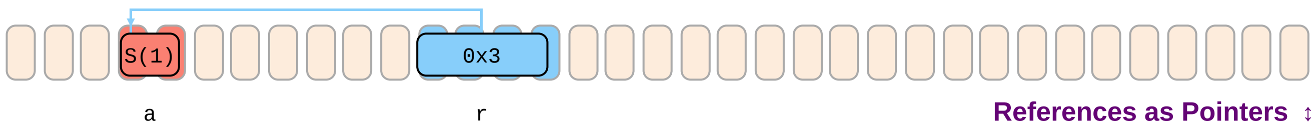


**Nested Functions** ↕



**Repurposing Memory** ↕

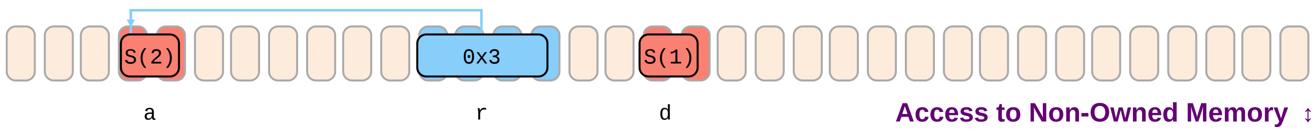
**References & Pointers**



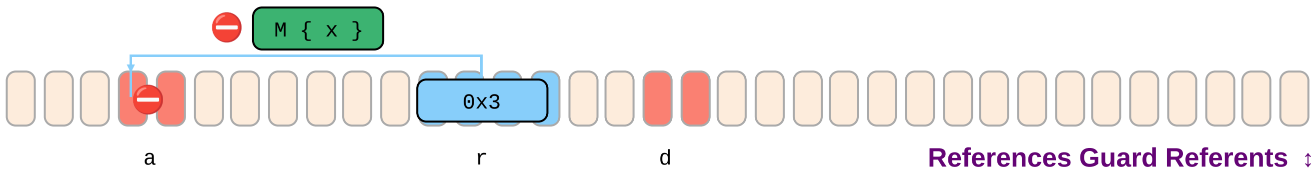
**References as Pointers** ↕

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

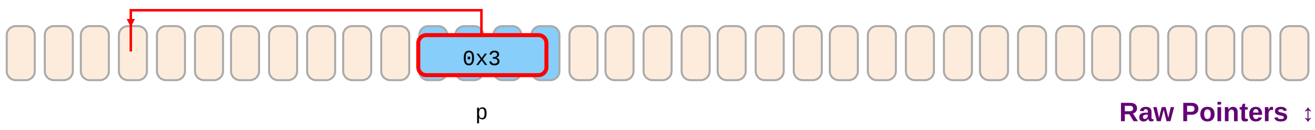
- A **reference type** such as `&S` or `&mut S` can hold the **location of** some `s`.
- Here type `&S`, bound as name `r`, holds *location of* variable `a` (`0x3`), that must be type `S`, obtained via `&a`.



**Access to Non-Owned Memory** ↕

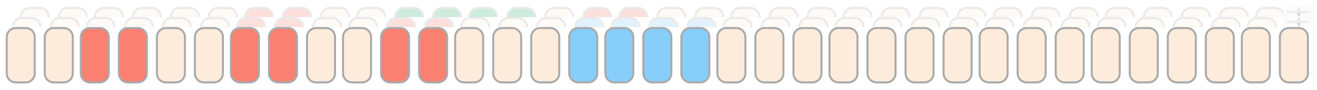


**References Guard Referents** ↕



**Raw Pointers** ↕

Lifetime Basics

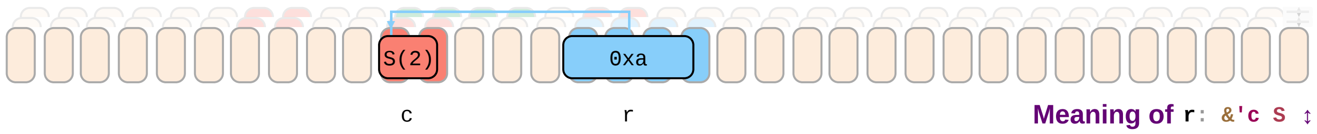


## "Lifetime" of Things ↕

- Every entity in a program has some (temporal / spatial) room 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*.

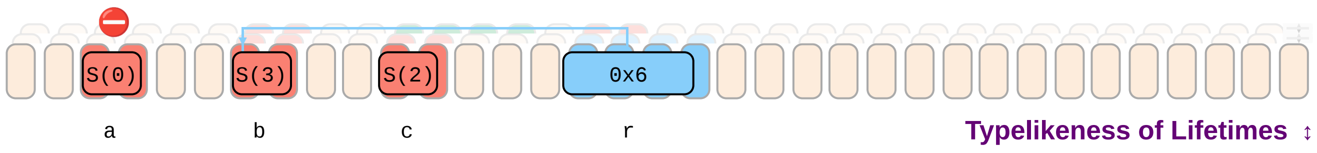
[1] There is sometimes ambiguity in the docs differentiating the various *scopes* and *lifetimes*. We try to be pragmatic here, but suggestions are welcome.

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

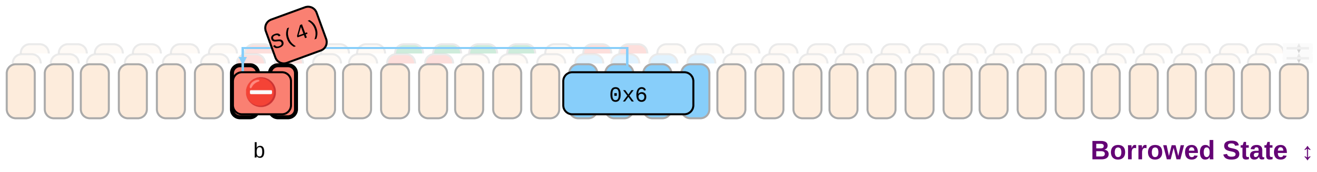


## Meaning of `r: &'c S` ↕

- 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`.

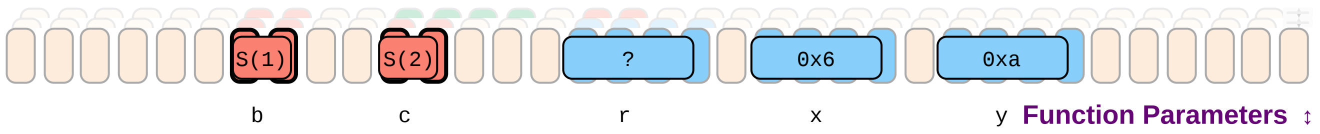


**Typelikeness of Lifetimes** ↕

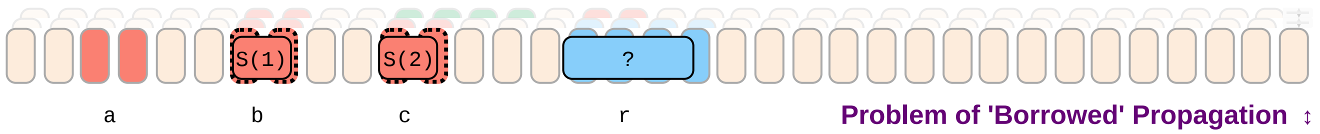


**Borrowed State** ↕

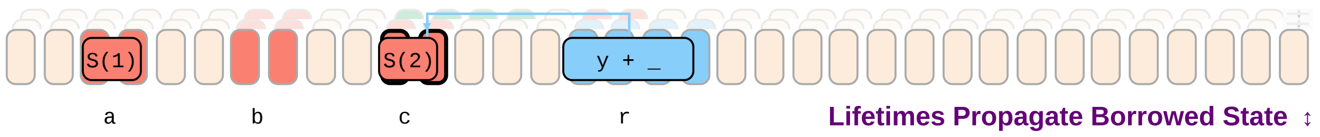
Lifetimes in Functions



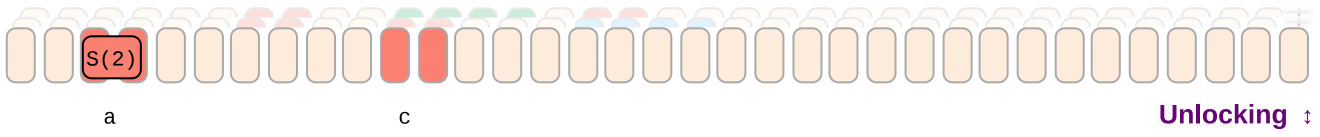
**Function Parameters** ↕



**Problem of 'Borrowed' Propagation** ↕

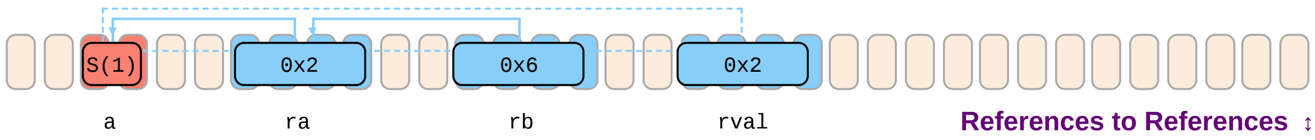


**Lifetimes Propagate Borrowed State** ↕

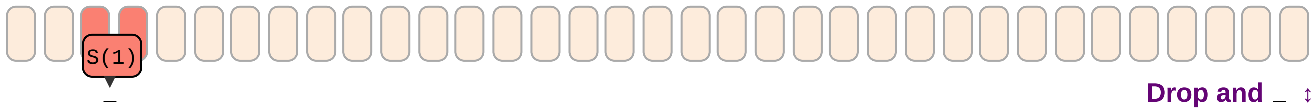


**Unlocking** ↕

Advanced



**References to References** ↕



↕ Examples expand by clicking.

---

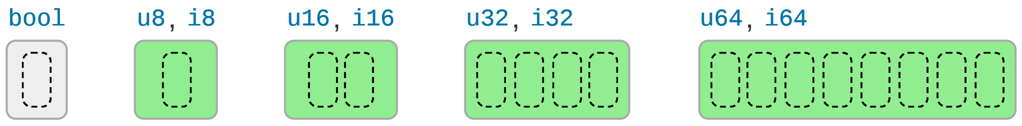
## Memory Layout

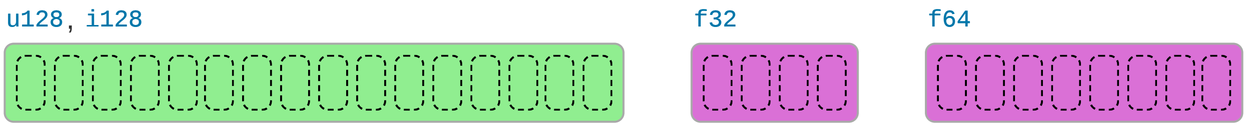
Byte representations of common types.

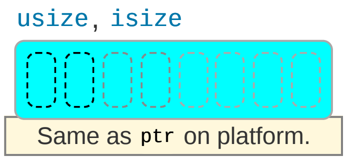
### Basic Types

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

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 ●

**1**

d, r

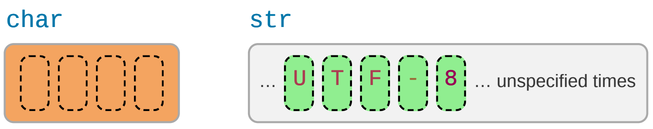
d STD

r

STD

[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

**Encoding**

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

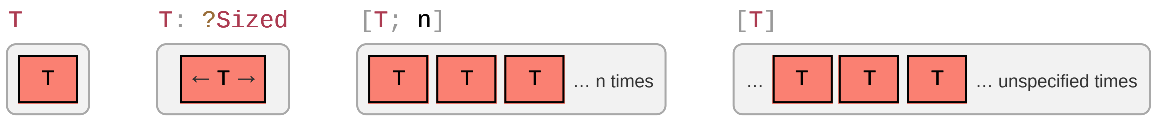
| Variant | Memory Representation[2] |
|---|---|
| `s.as_bytes()` | 49 20 **e2 9d a4** 20 52 75 73 74 [3] |
| `s.chars()`[1] | 49 00 00 00 20 00 00 00 **64 27 00 00** 20 00 00 00 52 00 00 00 75 00 00 00 73 00 … |
| `t.as_bytes()` | 49 20 **e2 9d a4** **ef b8 8f** 20 52 75 73 74 [4] |
| `t.chars()`[1]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

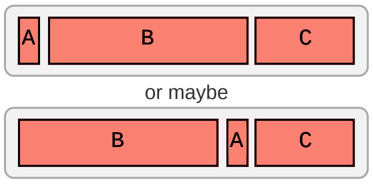


Sized ↓ Maybe DST ↓ Fixed array of `n` elements. **Slice type** of unknown-many elements. Neither `Sized` (nor carries `len` information), and most often lives behind reference as `&[T]`. ↓

`struct S;`

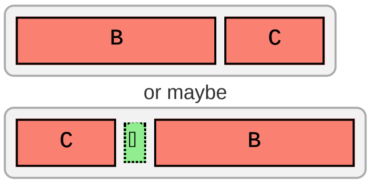
Zero-Sized ↓

`(A, B, C)`



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

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

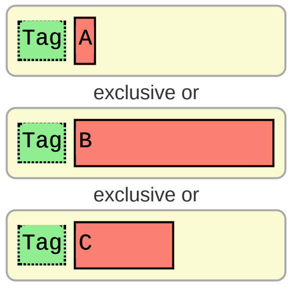


Compiler may also add padding.

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.

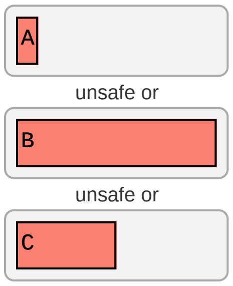
These **sum types** hold a value of one of their sub types:

`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 { … }`

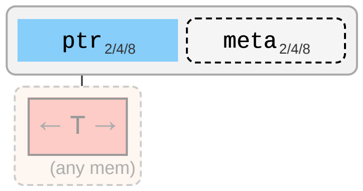


Can unsafely reinterpret memory. Result might be undefined.

## References & Pointers

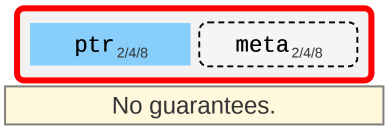
References give safe access to 3rd party memory, raw pointers `unsafe` access. The corresponding `mut` types have an identical data layout to their immutable counterparts.

`&'a T`



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

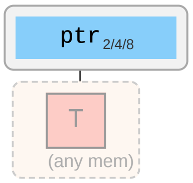
`*const T`



### Pointer Meta

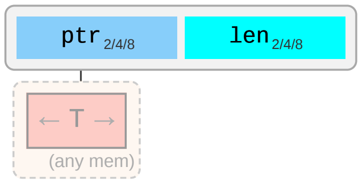
Many reference and pointer types can carry an extra field, **pointer metadata**. STD It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

`&'a T`



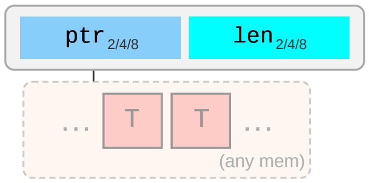
No meta for sized target. (pointer is thin).

`&'a T`



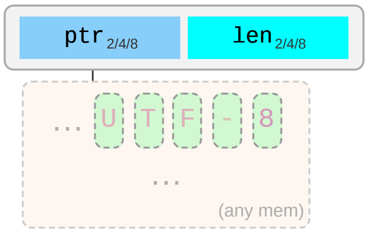
If `T` is a DST `struct` such as `S { x: [u8] }` meta field `len` is count of dyn. sized content.

`&'a [T]`



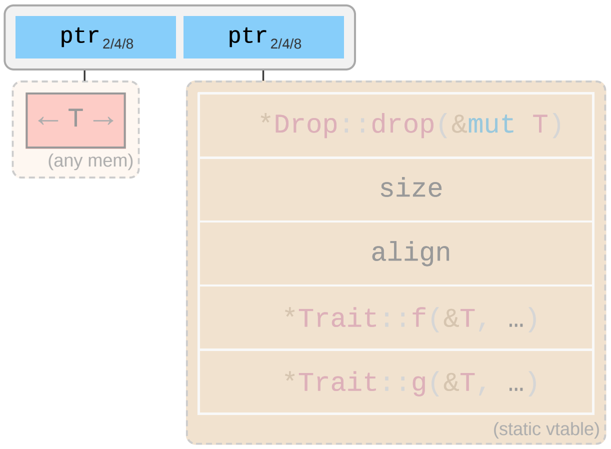
Regular **slice reference** (i.e., the reference type of a slice type `[T]`) ↑ often seen as `&[T]` if `'a` elided.

`&'a str`



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

`&'a dyn Trait`



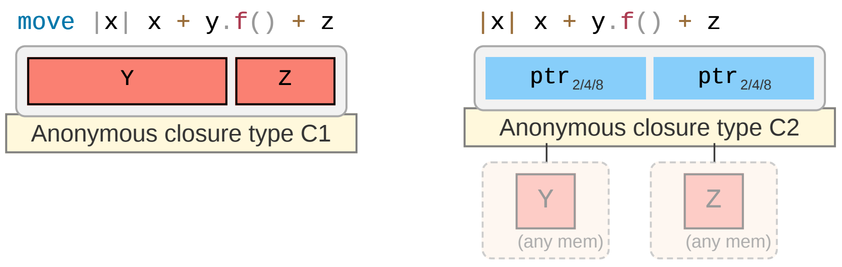
Meta points to vtable, where `*Drop::drop()`, `*Trait::f()`, … are pointers to their respective `impl` for `T`.

## 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:



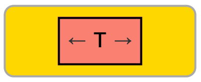
Also produces anonymous `fn` such as `f_c1(C1, X)` or `f_c2(&C2, X)`. Details depend on which `FnOnce`, `FnMut`, `Fn` ... is supported, based on properties of captured types.

[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

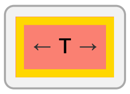
Rust's standard library combines the above primitive types into useful types with special semantics, e.g.:

`UnsafeCell<T>`



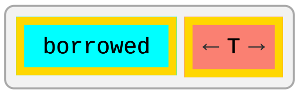
Magic type allowing aliased mutability.

`Cell<T>`



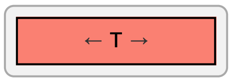
Allows `T`'s to move in and out.

`RefCell<T>`



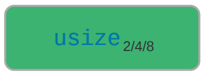
Also support dynamic borrowing of `T`. Like `Cell` this is `Send`, but not `Sync`.

`ManuallyDrop<T>`



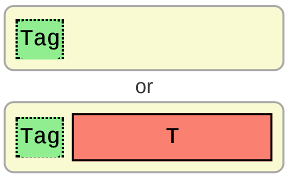
Prevents `T::drop()` from being called.

`AtomicUsize`



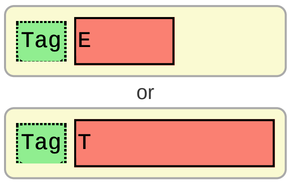
Other atomic similarly.

`Option<T>`



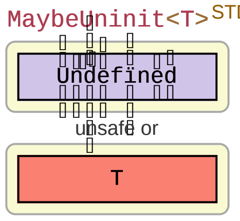
Tag may be omitted for certain T, e.g., `NonNull`.

`Result<T, E>`



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

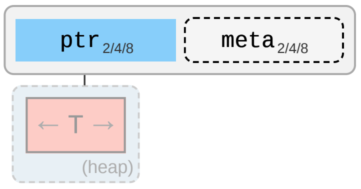
`MaybeUninit<T>`STD



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

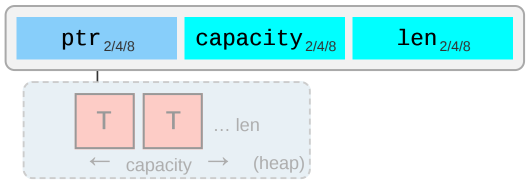
### Order-Preserving Collections

`Box<T>`



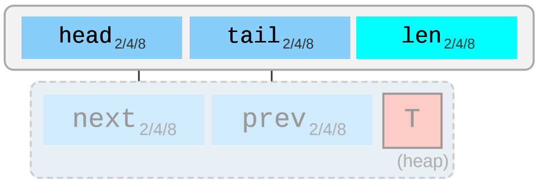
For some `T` stack proxy may carry meta† (e.g., `Box<[T]>`).

`Vec<T>`



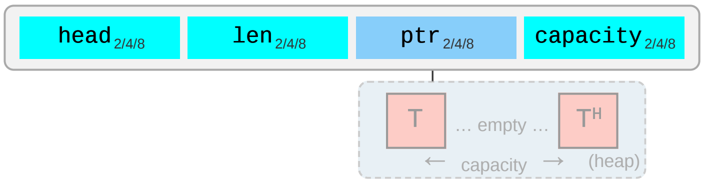
Regular *growable array* vector of single type.

`LinkedList<T>` 🛑



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. 🛑

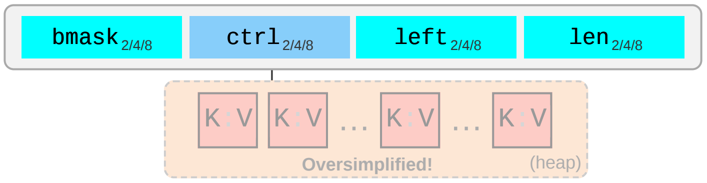
`VecDeque<T>`



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

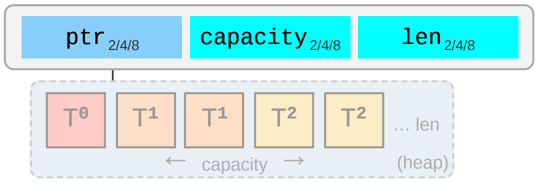
### Other Collections

`HashMap<K, V>`



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

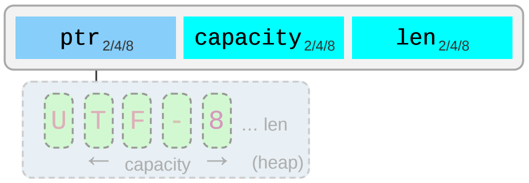
`BinaryHeap<T>`



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.

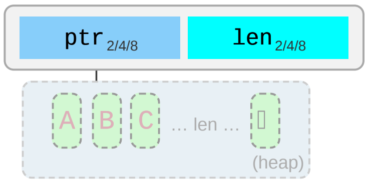
### Owned Strings

`String`



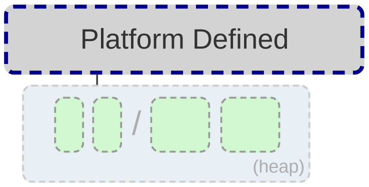
Observe how `String` differs from `&str` and `&[char]`.

`CString`



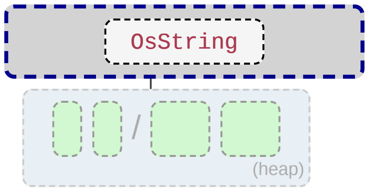
NUL-terminated but w/o NUL in middle.

`OsString`



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

`PathBuf`

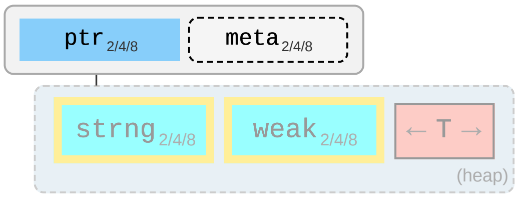


Encapsulates how operating system represents paths.

### Shared Ownership

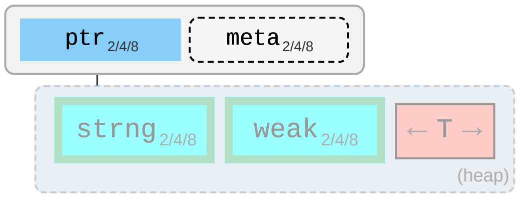
If the type does not contain a `Cell` for `T`, these are often combined with one of the `Cell` types above to allow shared de-facto mutability.

## `Rc<T>`



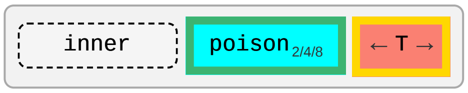
Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell` to allow mutation. Is neither `Send` nor `Sync`.

## `Arc<T>`



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

## `Mutex<T>` / `RwLock<T>`



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

## 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). [1] '21 | `format!("{x}{y}")` |
| Append string (any `Display` to any `Write`). '21 | `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. | `s.split_whitespace()` |
| Split by newlines. | `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 | `File::create(PATH)?` |
| Same, via OpenOptions | `OpenOptions::new().create(true).write(true).truncate(true).open(PATH)?` |

**Macros**

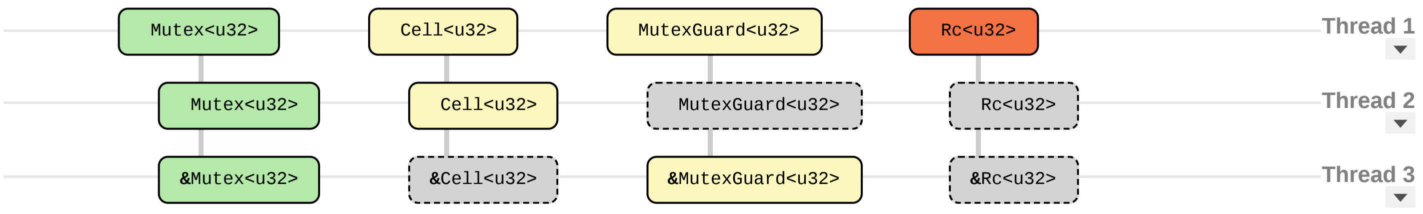
| Intent | Snippet |
|---|---|
| Macro w. variable arguments | `macro_rules! var_args { ($($args:expr),*) => {{ }} }` |
| Using `args`, e.g., calling `f` multiple times. | `$( f($args); )*` |

**Transforms** 🔥

**Esoterics**

## 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], `*mut 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.

## 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`. *Standard* 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.

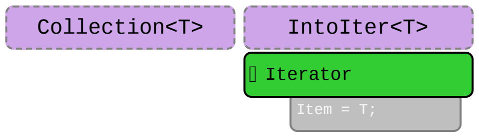
### For Loops

- **`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



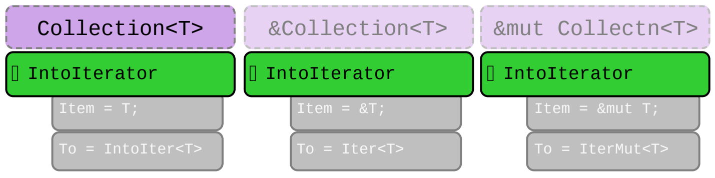
STD

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`.

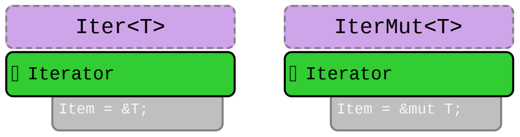
As you can see, the **IntoIterator** STD trait is what actually connects your collection with the **IntoIter** trait you created in the previous tab.

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



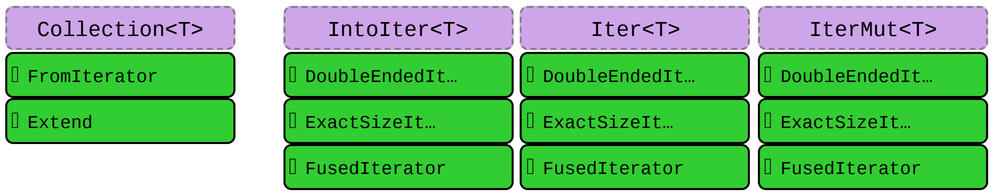
Interoperability

### 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 iterators:



Writing collections can be work. The good news is, if you followed all steps in this section your collection will feel like a *first class citizen*.

## 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` 2 | `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` 2 | `CString::new(x.to_os_string().into_string()?)?` |
| `&Path` | `CString::new(x.to_str()?)?` |
| `&[u8]` 1 | `CString::new(Vec::from(x))?` |
| `*mut c_char` 3 | `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` |

1 ?

i

i

1 ?

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 | ? |
| `&str` | `PathBuf::from(x)` i |
| `&CStr` | `PathBuf::from(x.to_str()?)` |
| `&OsStr` | `PathBuf::from(x)` i |
| `&Path` | `PathBuf::from(x)` i |
| `&[u8]` 1 | ? |

Vec<u8>

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

&str

| If you have `x` of type ... | Use this ... |
|---|---|
| `String` | `x.as_str()` |
| `CString` | `x.to_str()?` |
| `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` [2] | `x.to_str()?` |
| `PathBuf` | ?,4 |
| `Vec<u8>` [1,5] | `CStr::from_bytes_with_nul(&x)?` |
| `&str` | ?,4 |
| `&CStr` | `x` |
| `&OsStr` [2] | ? |
| `&Path` | ? |
| `&[u8]` [1,5] | `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()` |

1 ?

?

1 ?

&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 | ? |
| `&str` | `Path::new(x)` r |
| `&CStr` | `Path::new(x.to_str()?)` |
| `&OsStr` | `Path::new(x)` r |
| `&Path` | `x` |
| `&[u8]` 1 | ? |

&[u8]

| If you have `x` of type ... | Use this ... |
|---|---|
| `String` | `x.as_bytes()` |
| `CString` | `x.as_bytes()` |
| `OsString` | ? |
| `PathBuf` | ? |
| `Vec<u8>` 1 | `&x` |
| `&str` | `x.as_bytes()` |
| `&CStr` | `x.to_bytes_with_nul()` |
| `&OsStr` | `x.as_bytes()` 2 |
| `&Path` | ? |
| `&[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 should, or must if call is `unsafe`, ensure raw data comes with a valid representation for the string type (e.g., UTF-8 data for a `String`).

[2] Only on some platforms `std::os::<your_os>::ffi::OsStrExt` exists with helper methods to get a raw `&[u8]` representation of the underlying `OsStr`. Use the rest of the table to go from there, e.g.:

```
use std::os::unix::ffi::OsStrExt;
let bytes: &[u8] = my_os_str.as_bytes();
CString::new(bytes)?
```

[3] The `c_char` **must** have come from a previous `CString`. If it comes from FFI see `&CStr` instead.

[4] No known shorthand as `x` will lack terminating `0x0`. Best way to probably go via `CString`.

[5] Must ensure vector 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. |

STD

STD

'21

STD

rd STD

rd

STD '15 🗑️

'21

STD '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. |
| `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, … |

| Entry | Code |
|---|---|
| `Cargo.toml` | Main **project manifest**, 🔗 Defines dependencies, artifacts … |
| `Cargo.lock` | Dependency details for reproducible builds; add to `git` for apps, 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**

```
// 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);
    }
}
```

**Integration Tests**

```
// 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.
```

**Benchmarks🚧**

```
// 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 Scripts**

```
// 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**

proc_macro

proc_macro::

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 {}` | | |

← 1 →

← 2 →

1

2

`my_mod::`

## 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 --open` | Locally generate documentation for your code and dependencies. |
| `cargo run` | Run your project, if a binary is produced (main.rs). |
| `cargo run --bin b` | Run binary `b`. Unifies features with other dependents (can be confusing). |
| `cargo run -p w` | Run main of sub-workspace `w`. Treats features more as you would expect. |
| `cargo … --timings` | Show what crates caused your build to take so long. 🔥 |
| `cargo tree` | Show dependency graph. |
| `cargo +{nightly, stable} …` | Use given toolchain for command, e.g., for 'nightly only' tools. |
| `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.

**Macros**

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

| Within Macros | Explanation |
|---|---|
| `$x:ty` | Macro capture (here a type). |
| `$x:item` | An item, like a function, struct, module, etc. |

std::mem::

?

**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 |

STD REF

REF

🚧

REF ▯

? REF ▯

REF ▯

REF ▯

REF ▯

REF

🔗 🚧

STD 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 |

| Codegen | On | Explanation |
|---|---|---|
| #[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 disappointed if it still inlines the function. REF |
| #[cold] | F | Hint that function probably isn't going to be called. REF |
| #[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(X)][1] | T | Use another representation instead of the default rust REF one: |
| #[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 |

REF

REF

REF

1

REF

REF

REF

REF

REF

REF

REF

#[quality]

Attributes used by Rust tools to improve code quality:

| Code Patterns | On | Explanation |
|---|---|---|
| `#[allow(X)]` | * | Instruct `rustc` / `clippy` to … ignore class `X` of possible issues. 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. 🔗 |

`#[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 modules; or import from `extern crate`. REF |

| Proc Macros | On | Explanation |
|---|---|---|
| `#[proc_macro]` | `F` | Mark `fn` as **function-like** procedural macro 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** which can understand 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 |

REF

REF

REF

REF

REF

REF

REF

REF

REF

?

REF

🔥 REF

🔥 REF

**build.rs**

Environment variables and outputs related to the pre-build script.

| Input Environment | Explanation 🔗 |
|---|---|
| `CARGO_FEATURE_X` | Environment variable set for each feature `x` activated. |
| `CARGO_FEATURE_SERDE` | If feature `serde` were enabled. |
| `CARGO_FEATURE_SOME_FEATURE` | If feature `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.

?

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`

`u8` `u16`

`u8` `u8` `u8`

`u16` `u16` `u16`

`&u8` `&u8`

`&mut u8` `&mut u8`

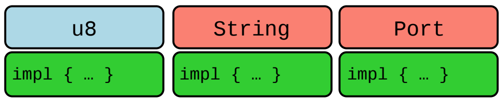
1

↑ 2

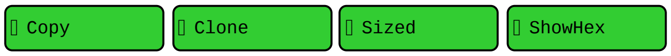
1

🔗

2

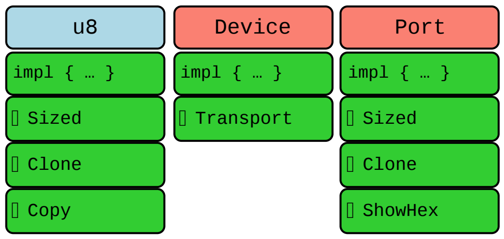


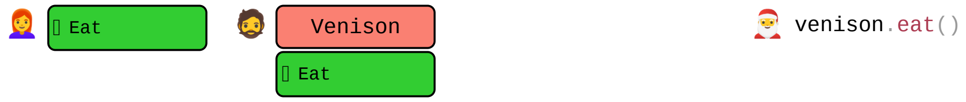
REF

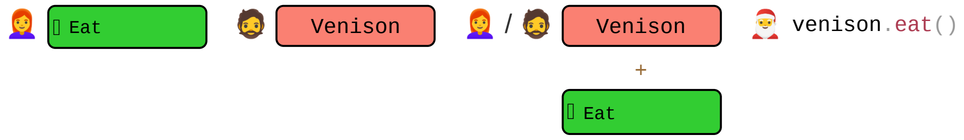


Copy

Sized


u8
Device
Port
impl { … }
impl { … }
impl { … }
Sized
Transport
Sized
Clone
Clone
Copy
ShowHex


Eat
Venison
Eat
venison.eat()


Eat
Venison
/
Venison
+
Eat
venison.eat()

*

```
food::
tasks::
```

*

?

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 specfic, `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`, … |

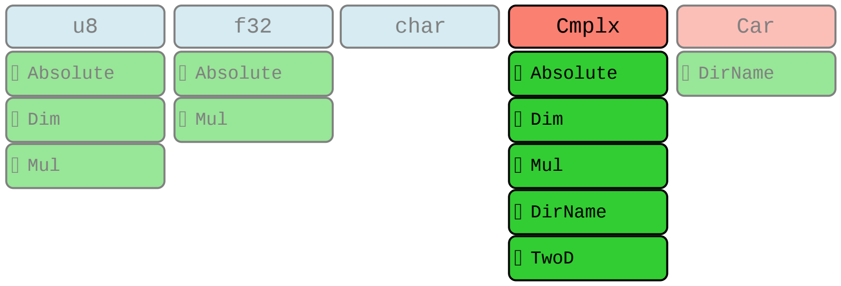
Type vs type constructors.

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

🧔 `Num<T>` → 🎅 `Num<u8>` `Num<f32>` `Num<Cmplx>`

| u8 | Port |
|---|---|
| Absolute | Clone |
| Dim | ShowHex |
| Mul | |

?


u8
Absolute
Dim
Mul
f32
Absolute
Mul
char
Cmplx
Absolute
Dim
Mul
DirName
TwoD
Car
DirName

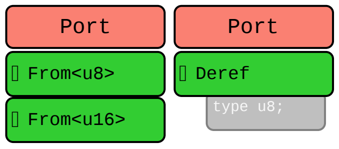
REF

rd

Advanced Concepts

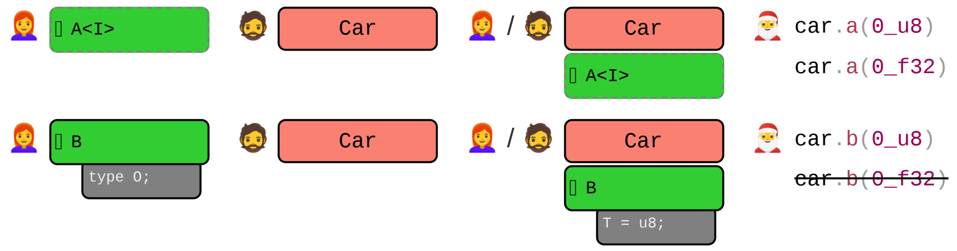
### Trait Parameters — `Trait<In> { type Out; }`

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

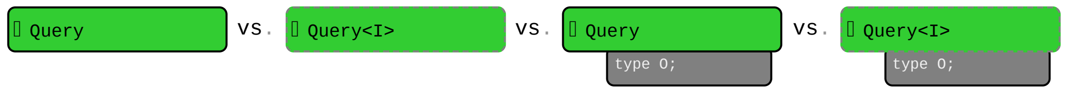


Why is that?

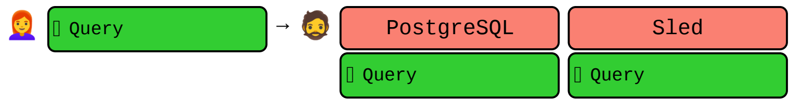
- Traits themselves can be generic over two **kinds of parameters**:
  - `trait From<I> {}`
  - `trait Deref { type O; }`
- Remember we said traits are "membership lists" for types and called the list `Self`?
- Turns out, parameters `I` (for **input**) and `O` (for **output**) are just more *columns*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

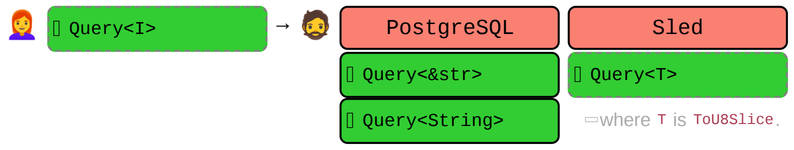


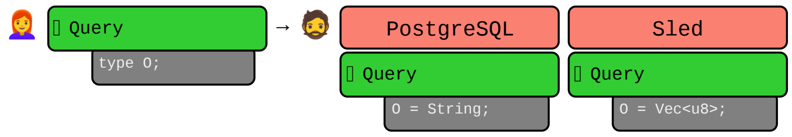
A<I>
Car
/
Car
A<I>
car.a(0_u8)
car.a(0_f32)
B
type O;
Car
/
Car
B
T = u8;
car.b(0_u8)
~~car.b(0_f32)~~

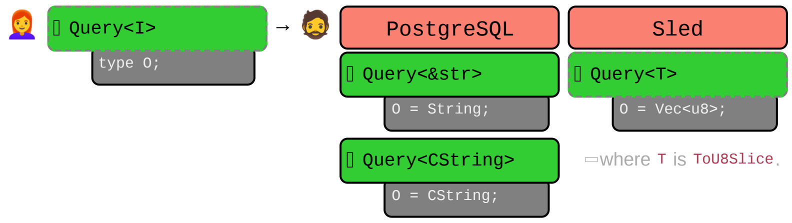


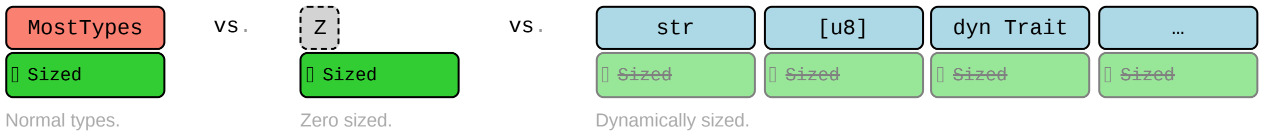
Query
vs.
Query<I>
vs.
Query
type O;
vs.
Query<I>
type O;


Query
→
PostgreSQL
Sled
Query
Query


Query<I>
→
PostgreSQL
Sled
Query<&str>
Query<String>
Query<T>
where T is ToU8Slice.


Query
type O;
→
PostgreSQL
Query
O = String;
Sled
Query
O = Vec<u8>;


Query<I>
type O;
→
PostgreSQL
Query<&str>
O = String;
Query<CString>
O = CString;
Sled
Query<T>
O = Vec<u8>;
where T is ToU8Slice.



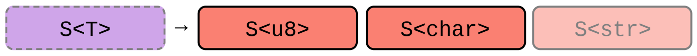
STD

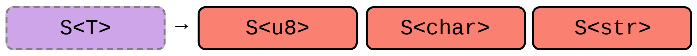
BK NOM REF

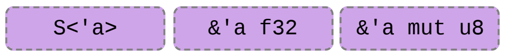
NOM

REF

↑







*

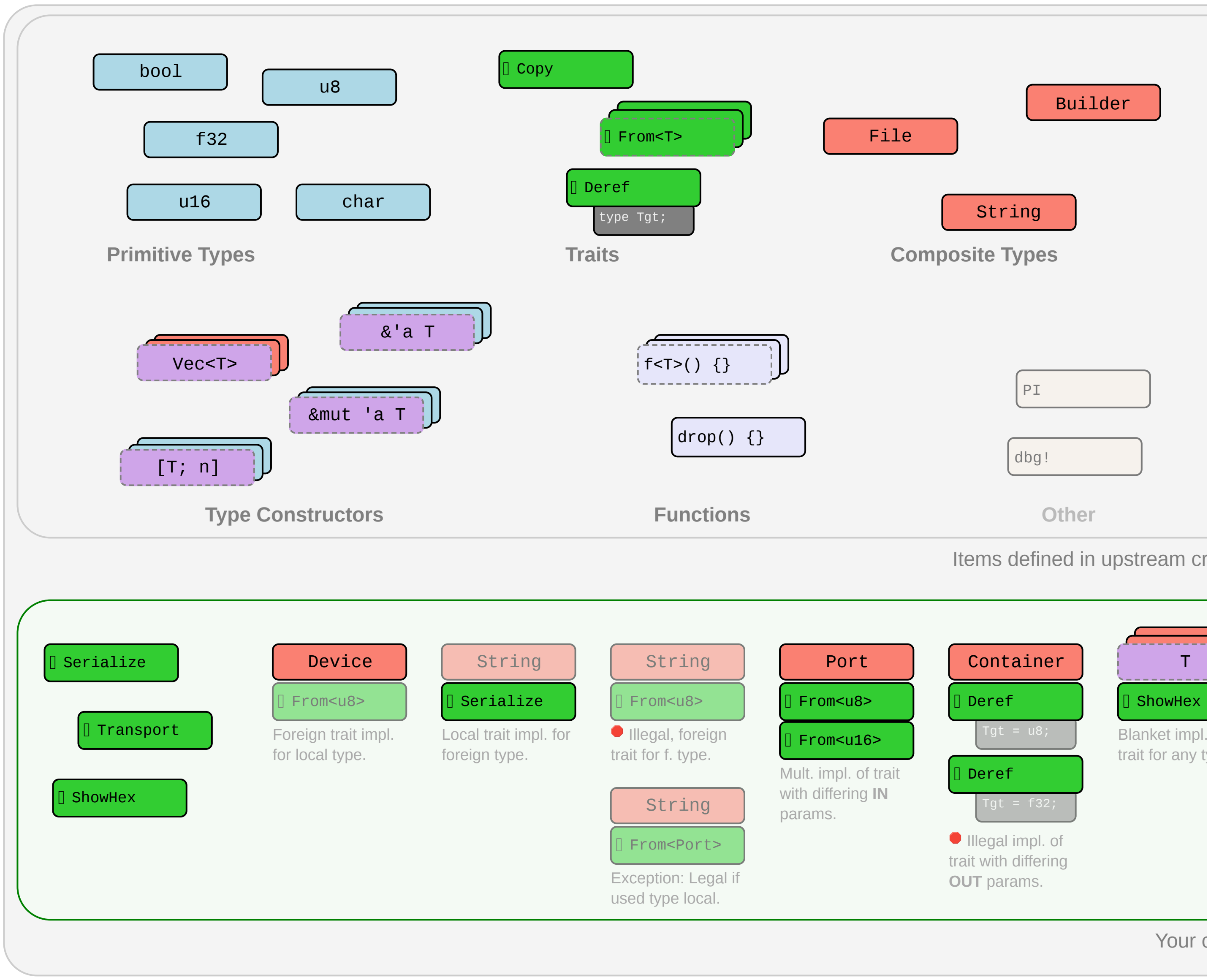
`S<'a>` → `S<'auto>` `S<'static>`

*

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. |

1

1

1

**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` | |

💬

## Coercions

```
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** $^{\text{NOM}}$ - subtyping **examples**:

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

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

Automatically converts `A` to `B` for types **only differing in lifetimes** $^{\text{NOM}}$ - 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`.

**1**

1

---

# 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"))` |
| **Handle 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() */ });` |
| **Provide Builders** | `Car::new("Model T").hp(20).build();` |

| Idiom | Code |
|---|---|
| **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. |
| **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 {}`,[1] often safer, faster solution without it. |
| **Implement Traits** | `#[derive(Debug, Copy, …)]` and custom `impl` where needed. |
| **Tooling** | Run **clippy** regularly to significantly improve your code quality. 🔥 |
| | Format your code with **rustfmt** for consistency. 🔥 |
| | Add **unit tests** BK (`#[test]`) to ensure your code works. |
| | Add **doc tests** BK (` ``` my_api::f() ``` `) to ensure docs match code. |
| **Documentation** | Annotate your APIs with doc comments that can show up on **docs.rs**. |
| | Don't forget to include a **summary sentence** and the **Examples** heading. |
| | If applicable: **Panics**, **Errors**, **Safety**, **Abort** and **Undefined Behavior**. |

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

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

## Async-Await 101

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

**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
4

**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 *intrinsically*.

```
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.

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. |
| `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*. |
| 🟡 Altering use of Cargo features (e.g., adding or removing 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. |
| 🟡 Adding a defaulted item; might cause dispatch ambiguity with other existing trait. |
| 🟡 Adding a defaulted type parameter. |

| Traits |
|---|
| 🔴 Implementing any "fundamental" trait, as *not* implementing a fundamental trait already was a promise. |
| 🟡 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.* |