Rust Language Cheat Sheet

05.03.2021

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

Data Structures

Data types and memory locations defined via keywords.

Example	Explanation
`struct S {}`	Define a **struct** BK EX STD REF with named fields.
`struct S { x: T }`	Define struct with named field `x` of type `T`.
`struct S (T);`	Define "tupled" struct with numbered field `.0` of type `T`.
`struct S;`	Define **zero sized** NOM unit struct. Occupies no space, optimized away.
`enum E {}`	Define an **enum** BK EX REF , *c.* algebraic data types, tagged unions.
`enum E { A, B(), C {} }`	Define variants of enum; can be unit- `A`, tuple- `B()` and struct-like `C{}`.
`enum E { A = 1 }`	If variants are only unit-like, allow discriminant values, e.g., for FFI.
`union U {}`	Unsafe C-like **union** REF for FFI compatibility.
`static X: T = T();`	**Global variable** BK EX REF with `'static` lifetime, single memory location.
`const X: T = T();`	Defines **constant** BK EX REF. Copied into a temporary when used.
`let x: T;`	Allocate `T` bytes on stack[1] bound as `x`. Assignable once, not mutable.
`let mut x: T;`	Like `let`, but allow for **mutability** 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 {}` code they become part 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.
`S { 0: x }`	Like `S(x)` below, but set field `.0` with struct syntax.
`S(x)`	Create `struct S(T)` or `use`'ed `enum E::S()` with field `.0` set to `x`.
`S`	If `S` is unit `struct S;` or `use`'ed `enum E::S` create value of `S`.
`E::C { x: y }`	Create enum variant `C`. Other methods above also work.
`()`	Empty tuple, both literal and type, aka **unit**. STD
`(x)`	Parenthesized expression.

Example	Explanation
`(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 of fixed length `n` holding elements of type `S`.
`[x; n]`	Array instance with `n` copies of `x`. REF
`[x, y]`	Array instance with given elements `x` and `y`.
`x[0]`	Collection indexing. Overloadable Index, IndexMut
`x[..]`	Collection slice-like indexing via RangeFull, *c.* slices.
`x[a..]`	Collection slice-like indexing via RangeFrom.
`x[..b]`	Collection slice-like indexing RangeTo.
`x[a..b]`	Collection slice-like indexing via Range.
`a..b`	Right-exclusive **range** REF creation, also seen as `..b`.
`a..=b`	Inclusive range creation, also seen as `..=b`.
`s.x`	Named **field access**, REF might try to Deref if `x` not part of type `S`.
`s.0`	Numbered field access, used for tuple types `S(T)`.

* For now, 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 (space for holding *any* `&s`).
`&[S]`	Special slice reference that contains (`address`, `length`).
`&str`	Special string slice reference that contains (`address`, `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`).
`*const S`	Immutable **raw pointer type** BK STD REF w/o memory safety.
`*mut S`	Mutable raw pointer type w/o memory safety.
`&s`	Shared **borrow** BK EX STD (e.g., address, len, vtable, … of *this* `s`, like `0x1234`).
`&mut s`	Exclusive borrow that allows **mutability**. EX
`ref s`	**Bind by reference**. EX 🗑
`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` that can also move out Box'ed content if it isn't `Copy`.
`'a`	A **lifetime parameter**, BK EX NOM REF, duration of a flow in static analysis.
`&'a S`	Only accepts an address holding an `s`; addr. existing `'a` or longer.
`&'a mut S`	Same, but allow content of address to be changed.
`struct S<'a> {}`	Signals `S` will contain address with lifetime `'a`. Creator of `S` decides `'a`.

Example	Explanation
`trait T<'a> {}`	Signals a `S` which `impl T for S` might contain address.
`fn f<'a>(t: &'a T)`	Same, for function. Caller decides `'a`.
`'static`	Special lifetime lasting the entire program execution.

Functions & Behavior

Define units of code and their abstractions.

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

Control Flow

Control execution within a function.

Example	Explanation
`while x {}`	**Loop** REF, run while expression `x` is true.
`loop {}`	**Loop infinitely** REF until `break`. Can yield value with `break x`.
`for x in iter {}`	Syntactic sugar to loop over **iterators**. BK STD REF
`if x {} else {}`	**Conditional branch** REF if expression is true.
`'label: loop {}`	**Loop label** EX REF, useful for flow control in nested loops.
`break`	**Break expression** REF to exit a loop.
`break x`	Same, but make `x` value of the loop expression (only in actual `loop`).
`break 'label`	Exit not only this loop, but the enclosing one marked with `'label`.

Example	Explanation
`break 'label x`	Same, but make `x` the value of the enclosing loop marked with `'label`.
`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`	Only works inside `async`. Yield flow until `Future` STD or Stream `x` ready. REF '18
`return x`	Early return from function. More idiomatic way is to end with expression.
`f()`	Invoke callable `f` (e.g., a function, closure, function pointer, `Fn`, ...).
`x.f()`	Call member function, requires `f` takes `self`, `&self`, ... as first argument.
`X::f(x)`	Same as `x.f()`. Unless `impl Copy for X {}`, `f` can only be called once.
`X::f(&x)`	Same as `x.f()`.
`X::f(&mut x)`	Same as `x.f()`.
`S::f(&x)`	Same as `x.f()` if `X` derefs to `S`, i.e., `x.f()` finds methods of `S`.
`T::f(&x)`	Same as `x.f()` if `X impl T`, i.e., `x.f()` finds methods of `T` if in scope.
`X::f()`	Call associated function, e.g., `X::new()`.
`<X as T>::f()`	Call trait method `T::f()` implemented for `X`.

Organizing Code

Segment projects into smaller units and minimize dependencies.

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

Type Aliases and Casts

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

Example	Explanation
`type T = S;`	Create a **type alias** BK REF, i.e., another name for `S`.
`Self`	Type alias for **implementing type** REF, e.g. `fn new() -> Self`.
`self`	Method subject in `fn f(self) {}`, same as `fn f(self: Self) {}`.
`&self`	Same, but refers to self as borrowed, same as `f(self: &Self)`
`&mut self`	Same, but mutably borrowed, same as `f(self: &mut Self)`
`self: Box<Self>`	Arbitrary self type, add methods to smart pointers (`my_box.f_of_self()`).
`S as T`	**Disambiguate** BK REF type `S` as trait `T`, e.g., `<S as T>::f()`.
`S as R`	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. NOM

Macros & Attributes

Code generation constructs expanded before the actual compilation happens.

Example	Explanation
`m!()`	**Macro** BK STD REF invocation, also `m!{}`, `m![]` (depending on macro).
`#[attr]`	Outer **attribute**. EX REF, annotating the following item.
`#![attr]`	Inner attribute, annotating the *upper*, surrounding item.

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

Within Macros	Explanation
`$x:ty`	Macro capture, with the `ty` part being:
`$x:item`	An item, like a function, struct, module, etc.
`$x:block`	A block `{}` of statements or expressions, e.g., `{ let x = 5; }`
`$x:stmt`	A statement, e.g., `let x = 1 + 1;`, `String::new();` or `vec![];`
`$x:expr`	An expression, e.g., `x`, `1 + 1`, `String::new()` or `vec![]`
`$x:pat`	A pattern, e.g., `Some(t)`, `(17, 'a')` or `_`.
`$x:ty`	A type, e.g., `String`, `usize` or `Vec<u8>`.
`$x:ident`	An identifier, for example in `let x = 0;` the identifier is `x`.
`$x:path`	A path (e.g. `foo`, `::std::mem::replace`, `transmute::<_, int>`).
`$x:literal`	A literal (e.g. `3`, `"foo"`, `b"bar"`, etc.).
`$x:lifetime`	A lifetime (e.g. `'a`, `'static`, etc.).
`$x:meta`	A meta item; the things that go inside `#[ ... ]` and `#![ ... ]` attributes.
`$x:vis`	A visibility modifier; `pub`, `pub(crate)`, etc.
`$x:tt`	A single token tree, see here for more details.
`$x`	Macro substitution, e.g., use the captured `$x:ty` from above.
`$(x),*`	Macro repetition "zero or more times" in macros by example.
`$(x),?`	Same, but "zero or one time".
`$(x),+`	Same, but "one or more times".
`$(x)<<+`	In fact separators other than `,` are also accepted. Here: `<<`.
`$crate`	Special hygiene variable, crate where macros is defined. ?

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 Some(x) = get();`	**Won't** work 🛑 if pattern can be **refuted** REF, use `if let` instead.
`if let Some(x) = get() {}`	Branch if pattern can be assigned (e.g., `enum` variant), syntactic sugar. *
`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`, wildcard any index.
`E::C { .. } => {}`	Match enum struct variant `C`, wildcard any field.
`S { x: 0, y: 1 } => {}`	Match struct with specific values (only accepts `s` with `s.x` of `0` and `s.y` of `1`).
`S { x: a, y: b } => {}`	Match struct with *any*(!) values and bind `s.x` to `a` and `s.y` to `b`.
`S { x, y } => {}`	Same, but shorthand with `s.x` and `s.y` bound as `x` and `y` respectively.
`S { .. } => {}`	Match struct with any values.
`D => {}`	Match enum variant `E::D` if `D` in `use`.
`D => {}`	Match anything, bind `D`; possibly false friend 🛑 of `E::D` if `D` not in `use`.
`_ => {}`	Proper wildcard that matches anything / "all the rest".
`(a, 0) => {}`	Match tuple with any value for `a` and `0` for second.
`[a, 0] => {}`	**Slice pattern**, REF 🔗 match array with any value for `a` and `0` for second.
`[1, ..] => {}`	Match array starting with `1`, any value for rest; **subslice pattern**. ?
`[1, .., 5] => {}`	Match array starting with `1`, ending with `5`.
`[1, x @ .., 5] => {}`	Same, but also bind `x` to slice representing middle (*c.* next entry).
`x @ 1..=5 => {}`	Bind matched to `x`; **pattern binding**, BK EX REF here `x` would be `1`, `2`, ... or `5`.
`0 | 1 => {}`	Pattern alternatives (or-patterns).
`E::A | E::Z`	Same, but on enum variants.
`E::C {x} | E::D {x}`	Same, but bind `x` if all variants have it.
`S { x } if x > 10 => {}`	Pattern **match guards**, BK EX REF condition must be true as well to match.

Generics & Constraints

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

Example	Explanation
`S<T>`	A **generic** BK EX type with a type parameter (`T` is placeholder name here).
`S<T: R>`	Type short hand **trait bound** BK EX specification (`R` *must* be actual trait).

Example	Explanation
`T: R, P: S`	**Independent trait bounds** (here one for `T` and one for `P`).
`T: R, S`	Compile error, 🛑 you probably want compound bound `R + S` below.
`T: R + S`	**Compound trait bound** BK EX, `T` must fulfill `R` and `S`.
`T: R + 'a`	Same, but w. lifetime. `T` must fulfill `R`, if `T` has lifetimes, must outlive `'a`.
`T: ?Sized`	Opt out of a pre-defined trait bound, here `Sized`. ?
`T: 'a`	Type **lifetime bound** EX; if T has references, they must outlive `'a`.
`T: 'static`	Same; does esp. *not* mean value `t` *will* 🛑 live `'static`, only that it could.
`'b: 'a`	Lifetime `'b` must live at least as long as (i.e., *outlive*) `'a` bound.
`S<const N: usize>`	**Generic const bound**; ? 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> where T: R`	Almost same as `S<T: R>` but more pleasant to read for longer bounds.
`S<T> where u8: R<T>`	Also allows you to make conditional statements involving *other* types.
`S<T = R>`	**Default type parameter** BK for associated type.
`S<'_>`	Inferred **anonymous lifetime**; asks compiler to *'figure it out'* if obvious.
`S<_>`	Inferred **anonymous type**, e.g., as `let x: Vec<_> = iter.collect()`
`S::<T>`	**Turbofish** STD call site type disambiguation, e.g. `f::<u32>()`.
`trait T<X> {}`	A trait generic over `X`. Can have multiple `impl T for S` (one per `X`).
`trait T { type X; }`	Defines **associated type** BK REF `X`. Only one `impl T for S` possible.
`type X = R;`	Set associated type within `impl T for S { type X = R; }`.
`impl<T> S<T> {}`	Implement functionality for any `T` in `S<T>`.
`impl S<T> {}`	Implement functionality for exactly `S<T>` (e.g., `S<u32>`).
`fn f() -> impl T`	**Existential types** BK, returns an unknown-to-caller `S` that `impl T`.
`fn f(x: &impl T)`	Trait bound,"**impl traits**" BK, somewhat similar to `fn f<S:T>(x: &S)`.
`fn f(x: &dyn T)`	Marker for **dynamic dispatch** BK REF, `f` will not be monomorphized.
`fn f() where Self: R;`	In `trait T {}`, make `f` accessible only on types known to also `impl R`.
`fn f() where Self: R {}`	Esp. useful w. default methods (non dflt. would need be impl'ed anyway).
`for<'a>`	**Higher-ranked trait bounds.** NOM REF
`trait T: for<'a> R<'a> {}`	Any `S` that `impl T` would also have to fulfill `R` for any lifetime.

Strings & Chars

Rust has several ways to create textual values.

Example	Explanation
`"..."`	**String literal**, REF UTF-8, will interpret `\n` as *line break* `0xA`, ...
`r"..."`	**Raw string literal**. REF UTF-8, won't interpret `\n`, ...
`r#"..."#`	Raw string literal, UTF-8, but can also contain `"`. Number of `#` can vary.
`b"..."`	**Byte string literal**; REF constructs ASCII `[u8]`, not a string.
`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

Documentation

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

Example	Explanation
//	Line comment, use these to document code flow or *internals*.
///	Outer line **doc comment**, BK EX REF use these on types.
//!	Inner line doc comment, mostly used at start of file to document module.
/* … */	Block comment.
/** … */	Outer block doc comment.
/*! … */	Inner block doc comment.

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.

Miscellaneous

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

Example	Explanation
!	Always empty **never type**. 🚧 BK EX STD REF
_	Unnamed variable binding, e.g., |x, _| {}.
let _ = x;	Unnamed assignment is no-op, does **not** 🛑 move out x or preserve scope!
_x	Variable binding explicitly marked as unused.
1_234_567	Numeric separator for visual clarity.
1_u8	Type specifier for **numeric literals** EX REF (also i8, u16, …).
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

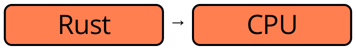
Rust supports most operators you would expect (+, *, %, =, ==, …), including **overloading**. STD Since they behave no differently in Rust we do not list them here.

Behind the Scenes

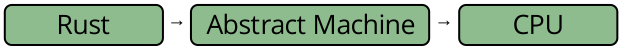
Arcane knowledge that may do terrible things to your mind, highly recommended.

The Abstract Machine

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



🔴 Less correctish.



More correctish.

The 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,
- forms a contract between programmer and machine,
- and **exploits all of the above for optimizations**.

Without AM	With AM
`0xffff_ffff` would make a valid `char`. 🔴	Memory more than just bits.
`0xff` and `0xff` are same pointer. 🔴	Pointers can come from different *domains*.
Any r/w pointer on `0xff` always fine. 🔴	Read and write reference may not exist same time.
Null reference is just `0x0` in some register. 🔴	Holding `0x0` in reference summons Cthulhu.

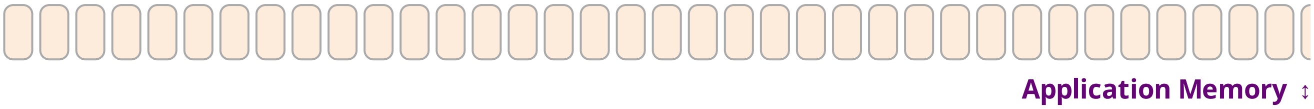
Practically this means:

- before assuming your **CPU** will do `A` when writing `B` you need positive proof **via documentation**(!),
- if you don't have that any physical behavior is *coincidental*,
- violate the abtract machine's contract and the optimizer makes your CPU do something **entirely else** — **undefined behavior**.↓

Memory & Lifetimes

Why moves, references and lifetimes are how they are.

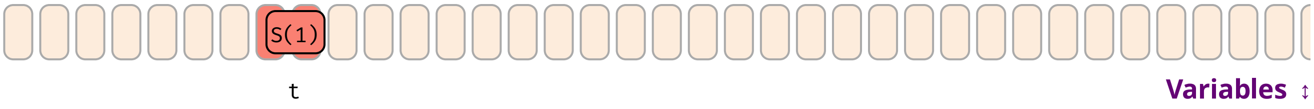
Types & Moves



- Application memory in itself is just array of bytes.
- It is segmented, 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).
- Programming languages such as Rust give developers tools to:

    - define what data goes into what segment,
    - express a desire for bitcode with specific properties to be produced,
    - protect themselves from errors while performing these operations.
- Most tricky part is tied to **how stack evolves**, which is **our focus**.

[1] While for each part of the heap someone (the allocator) needs to perform bookkeeping at runtime, the stack is trivially managable: *take a few bytes more while you need them, they will be discarded once you leave*. The (for performance reasons desired) simplicity of this appraoch, along with the fact that you can tell others about such *transient* locations (which in turn might want to access them long after you left), form the very essence of why *lifetimes* exist; and are the subject of the rest of this chapter.



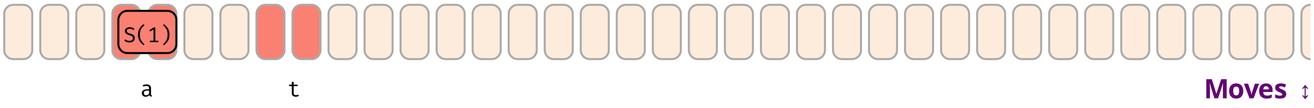
Variables ↕

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

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

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

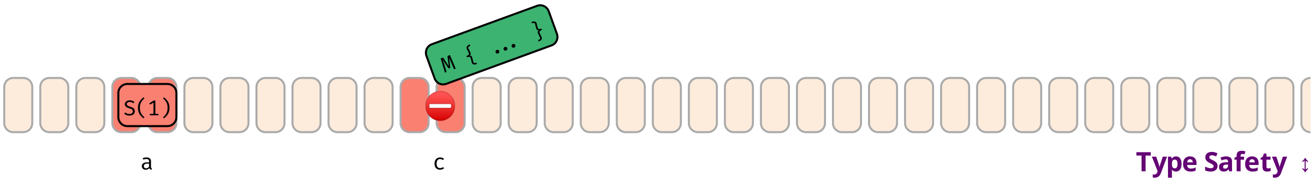
[2] It is the **author's opinion** that this ambiguity related to *variables* (and *lifetimes* and *scope* later) are some of the biggest contributors to the confusion around learning the basics of lifetimes. Whenever you hear one of these terms ask yourself "what *exactly* is meant here?"



Moves ↕

```
let a = t;
```

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

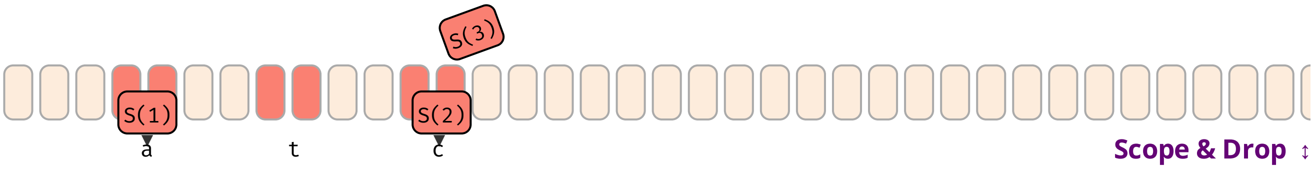


Type Safety ↕

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

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

> As an excercise to the reader, any time you see a value of type A being assignable to a location of some type not-exactly-A you should ask yourself: *through what mechanism is this possible*?

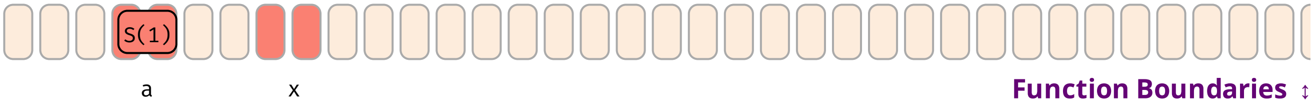


Scope & Drop ↕

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

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

Call Stack

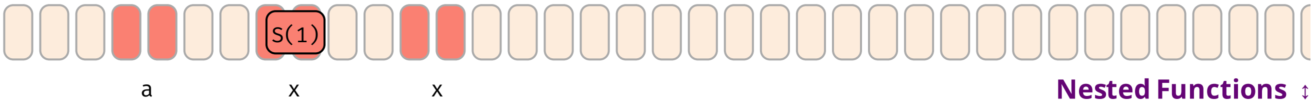


Function Boundaries ↕

```
fn f(x: S) { … }

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

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

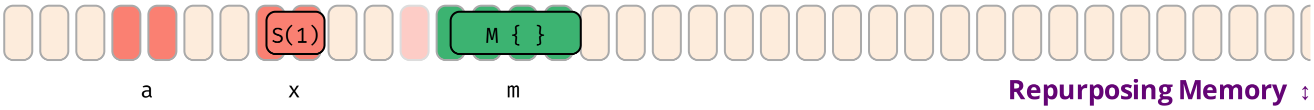
[1] Actual location depends on calling convention, might practically not end up on stack at all, but that doesn't change mental model.



Nested Functions ↕

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

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

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



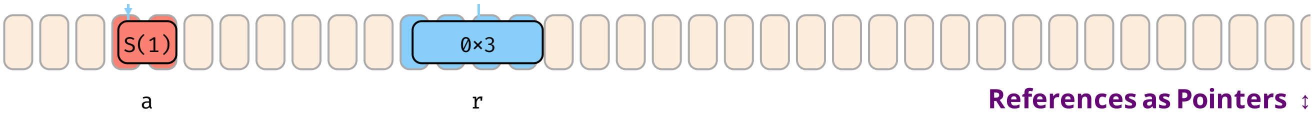
Repurposing Memory ↕

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

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

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

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

References & Pointers

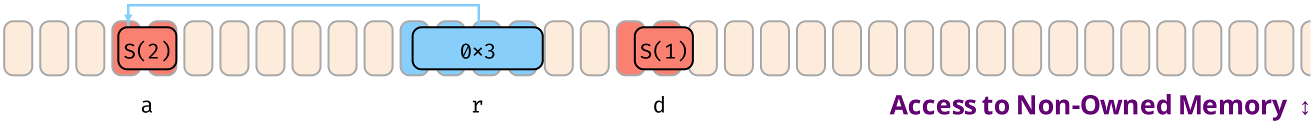


References as Pointers ↕

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

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

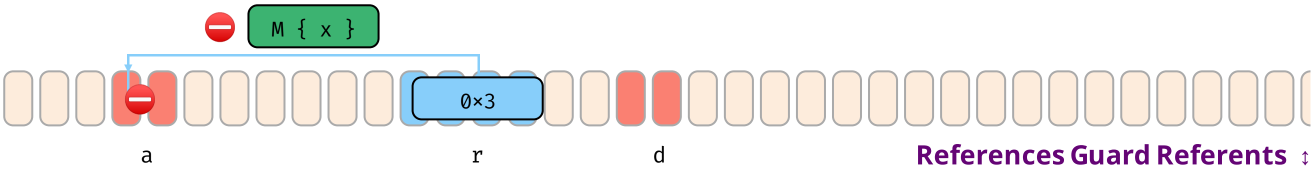
  ```
  let r: &S = &a;
  let r = &a;
  ```



Access to Non-Owned Memory ↕

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

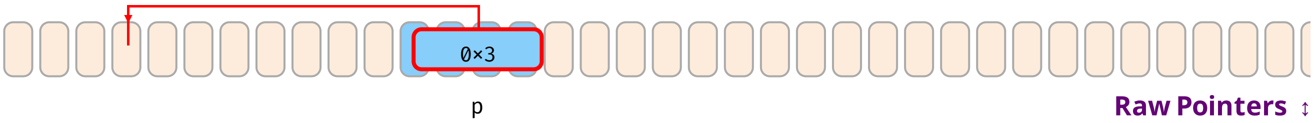
- References can **read from** (`&S`) and also **write to** (`&mut S`) location they point to.
- The *dereference* `*r` means to neither use the *location of* or *value within* `r`, but the **location `r` points to**.
- In example above, clone `d` is created from `*r`, and `S(2)` written to `*r`.
  - Method `Clone::clone(&T)` expects a reference itself, which is why we can use `r`, not `*r`.
  - On assignment `*r = ...` old value in location also dropped (not shown above).



References Guard Referents ↕

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

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

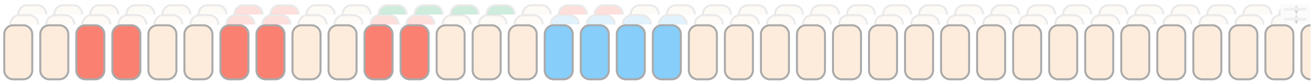


Raw Pointers ↕

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

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

Lifetime Basics

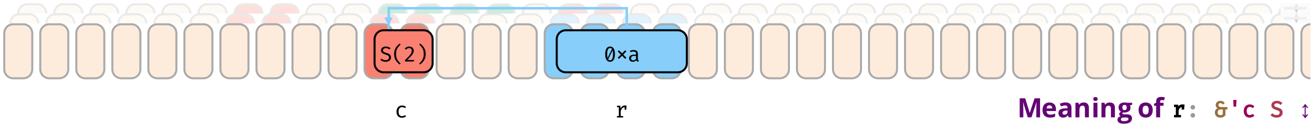


"Lifetime" of Things ↕

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

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

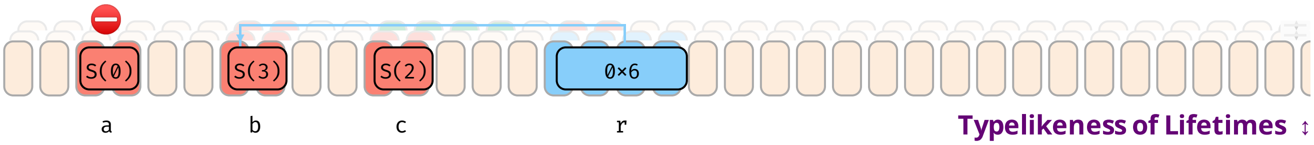
[2] *Live lines* might have been a more appropriate term ...



Meaning of `r: &'c S` ↕

- Assume you got a `r: &'c S` from somewhere it means:

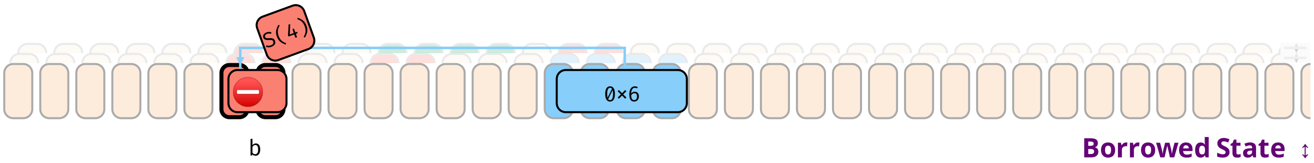
  - `r` holds an address of some `S`,
  - any address `r` points to must and will exist for at least `'c`,
  - the variable `r` itself cannot live longer than `'c`.



Typelikeness of Lifetimes ↕

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

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

- Assume you got a `mut r: &mut 'c S` from somewhere.
  - That is, a mutable location that can hold a mutable reference.
- As mentioned, that reference must guard the targeted memory.
- However, the **`'c` part**, like a type, also **guards what is allowed into `r`**.
- Here assiging `&b` (`0x6`) to `r` is valid, but `&a` (`0x3`) would not, as only `&b` lives equal or longer than `&c`.

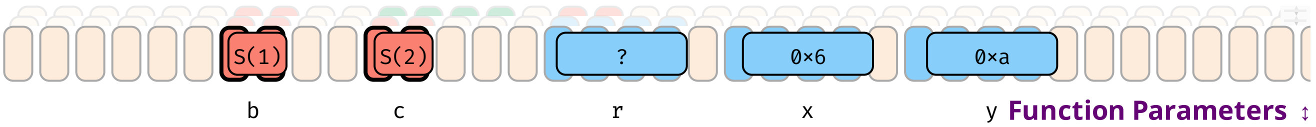


Borrowed State ↕

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

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

print_byte(r);
```

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

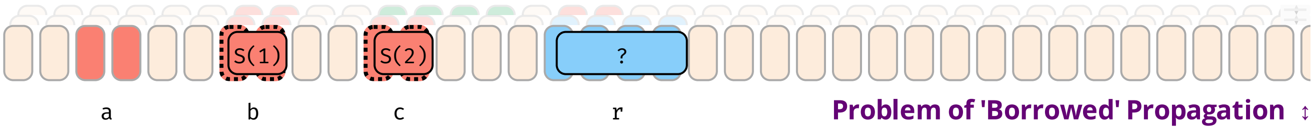
Lifetimes in Functions



Function Parameters ↕

```
fn f(x: &S, y:&S) -> &u8 { … }

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

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

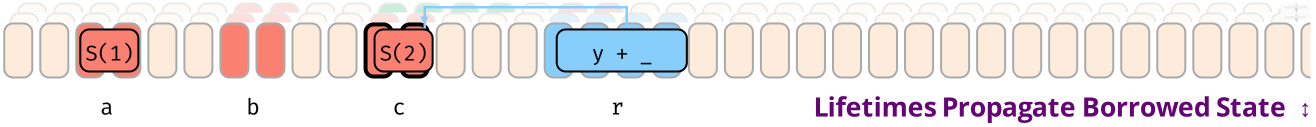
- When calling functions that take and return references two interesting things happen:
    - The used local variables are placed in a borrowed state,
    - But it is during compilation unknown which address will be returned.



Problem of 'Borrowed' Propagation ↕

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

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

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

print_byte(r);
```

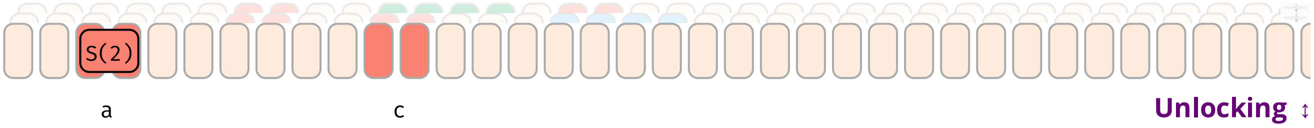
- Since `f` can return only one address, not in all cases `b` and `c` need to stay locked.
- In many cases we can get quality-of-life improvements.
    - Notably, when we know one parameter *couldn't* have been used in return value anymore.



Lifetimes Propagate Borrowed State ↕

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

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

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

let a = b;

print_byte(r);
```

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



Unlocking ↕

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

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

print_byte(s);

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

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

↕ Examples expand by clicking.

Language Sugar

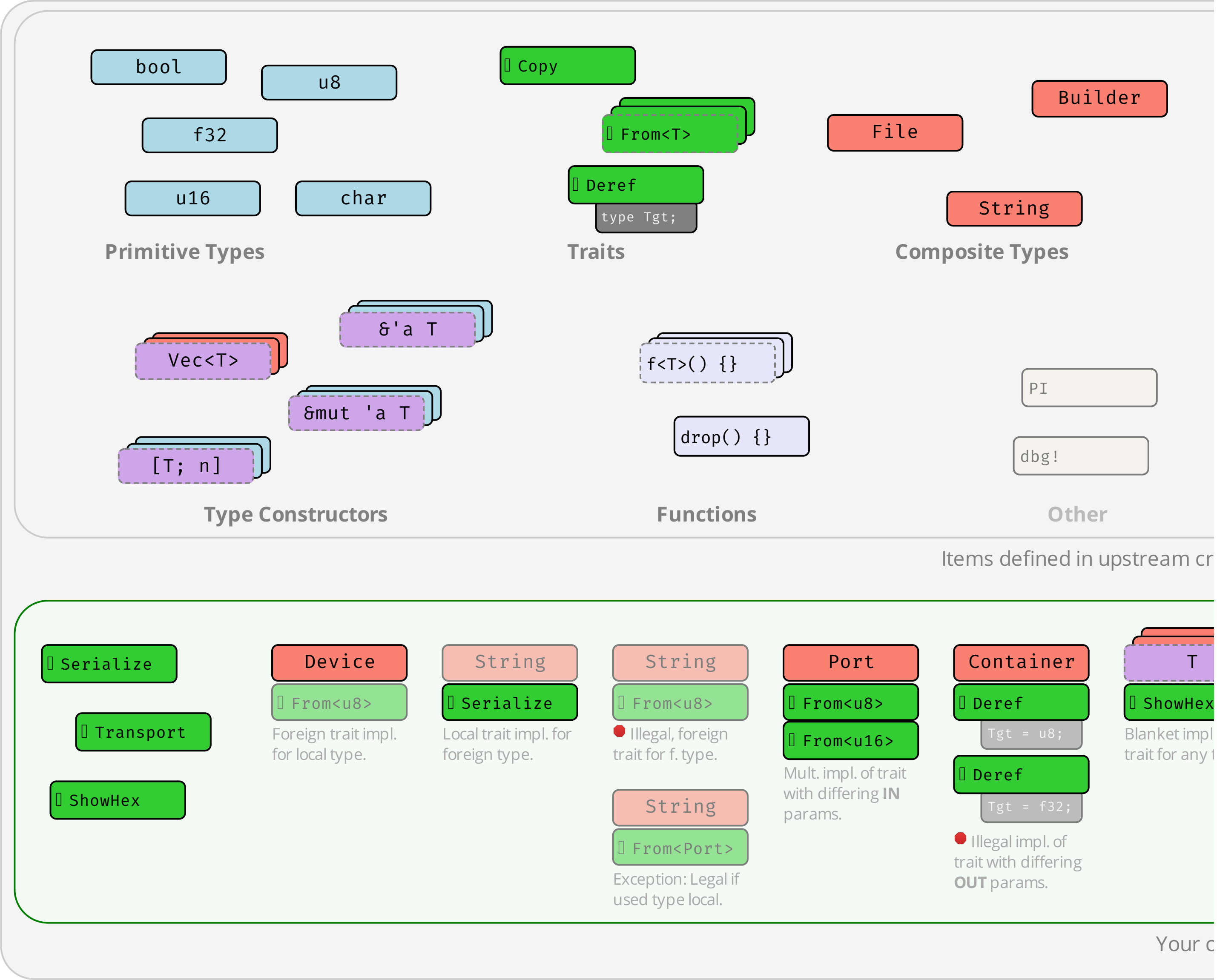
If something works that "shouldn't work now that you think about it", it might be due to one of these.

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

Author's Opinion 💬 — The features above will make your life easier, but might hinder your understanding. If any (type-related) operation ever feels *inconsistent* it might be worth revisiting this list.

Types, Traits, Generics

The building blocks of compile-time safety.



A walk through the jungle of types, traits, and implementations that (might possibly) exist in your application.

Type Paraphernalia

Types & Traits

Types

`u8` `String` `Device`

- Set of values with given semantics, layout, …

Type	Values

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

Sample types and sample values.

Type Equivalence and Conversions

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

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

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

Type	Values
`u8`	{ 0_{u8}, 1_{u8}, … , 255_{u8} }
`u16`	{ 0_{u16}, 1_{u16}, … , 65_535_{u16} }
`&u8`	{ $0xffaa_{\&u8}$, $0xffbb_{\&u8}$, … }
`&mut u8`	{ $0xffaa_{\&mut\ u8}$, $0xffbb_{\&mut\ u8}$, … }

How values differ between types.

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

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

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

Implementations — `impl S { }`

`u8`	`String`	`Port`
`impl { … }`	`impl { … }`	`impl { … }`

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

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

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

Traits — `trait T { }`

□ Copy □ Clone □ Sized □ ShowHex

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

Copy Trait
Self
u8
u16
...

Clone Trait
Self
u8
String
...

Sized Trait
Self
char
Port
...

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

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

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

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

□ Copy

```
trait Copy { }
```

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

□ Sized

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

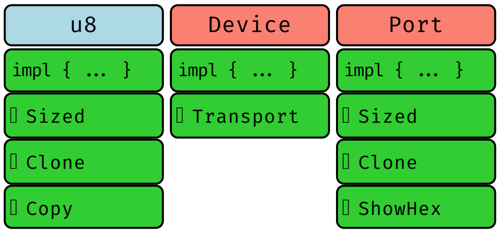
Implementing Traits for Types — `impl T for S { }`

```
impl ShowHex for Port { … }
```

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

ShowHex Trait
Self
Port

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



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

Traits vs. Interfaces

👩‍🦰 □ Eat

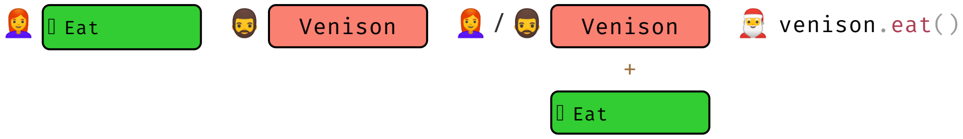
🧔 Venison
□ Eat

🎅 `venison.eat()`

Interfaces

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

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

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



👩‍🦰 □ Eat

🧔 Venison

👩‍🦰 / 🧔 Venison + □ Eat

🎅 `venison.eat()`

Traits

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

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

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

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

Generics

Type Constructors — `Vec<>`

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

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

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

Types vs type constructors.

`Vec<>`

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

Generic Parameters — `<T>`

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

- Parameter for `Vec<>` often named `T` therefore `Vec<T>`.
- `T` "variable name for type" for user to plug in something 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.

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

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

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

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

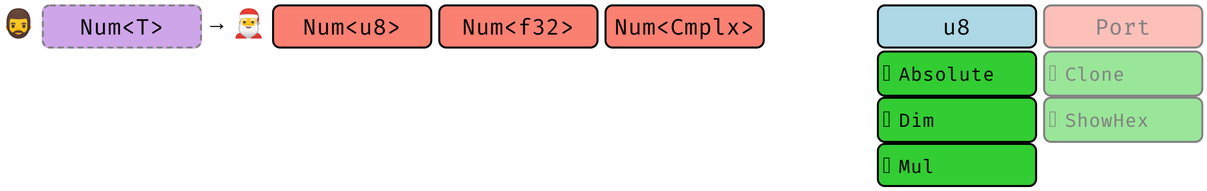
- Some type constructors not only accept specific type, but also **specific constant**.
- `[T; n]` constructs array type holding `T` type `n` times.
- For custom types declared as `MyArray<T, const N: usize>`.

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

Type constructors based on constant.

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

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

Bounds (Simple) — `where T: X`



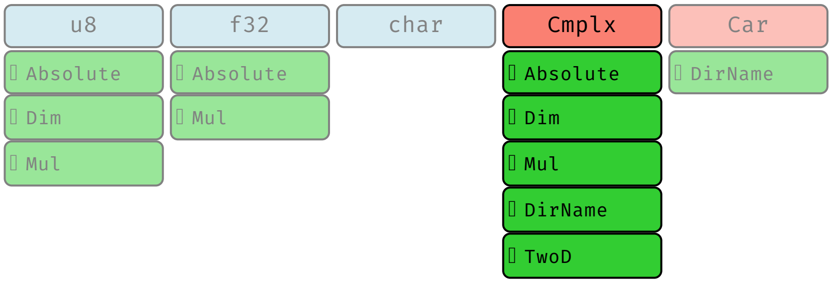
- If `T` can be any type, how can we *reason* about (write code) for such a `Num<T>`?
- Parameter **bounds**:
  - limit what types (**trait bound**) or values (**const bound** [?]) allowed,
  - we now can make use of these limits!
- Trait bounds act as "membership check":

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

Absolute Trait
Self
u8
u16
...

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

Bounds (Compound) — `where T: X + Y`



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

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

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

Implementing Families — `impl<>`

When we write:

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

It can be read as:

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

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

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

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

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

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

These are called **blanket implementations**.

ToHex
`Self`
`Port`
`Device`
...

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

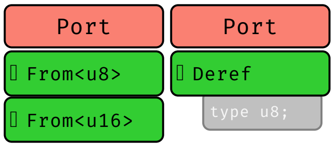
Serialize Trait
`Self`
`u8`
`Port`
...

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

Advanced Concepts

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

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



Why is that?

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

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

From	
Self	I
u16	u8
u32	u16
…	

Deref	
Self	O
Port	u8
String	str
…	

Input and output parameters.

Now here's the twist,

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

A more complex example:

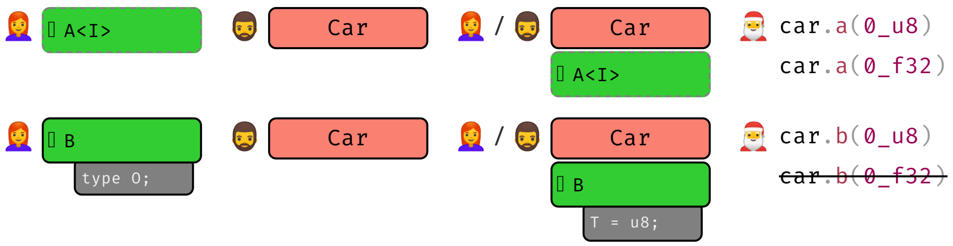
```
trait Complex<I1, I2> {
    type O1;
    type O2;
}
```

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

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

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

Trait Authoring Considerations (Abstract)



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

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

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

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

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

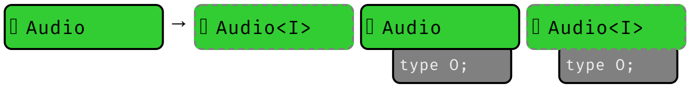
A	
Self	I
X	u32
Y	…

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

B	
Self	O
Player	String
X	u32

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

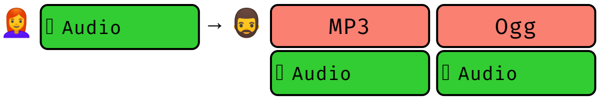
Trait Authoring Considerations (Example)



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

No Additional Parameters

```
trait Audio {
    fn play(&self, volume: f32);
}

impl Audio for MP3 { … }
impl Audio for Ogg { … }

mp3.play(0_f32);
```

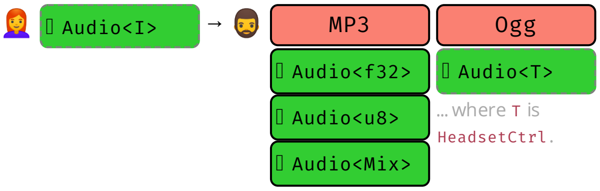


Trait author assumes:

- neither implementor nor user need to customize API.

Input Parameters

```
trait Audio<I> {
    fn play(&self, volume: I);
}

impl Audio<f32> for MP3 { … }
impl Audio<u8> for MP3 { … }
impl Audio<Mixer> for MP3 { … }
impl<T> Audio<T> for Ogg where T: HeadsetControl { … }

mp3.play(0_f32);
mp3.play(mixer);
```

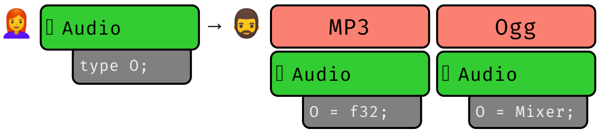


Trait author assumes:

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

Output Parameters

```
trait Audio {
    type O;
    fn play(&self, volume: Self::O);
}

impl Audio for MP3 { type O = f32; }
impl Audio for Ogg { type O = Mixer; }

mp3.play(0_f32);
ogg.play(mixer);
```



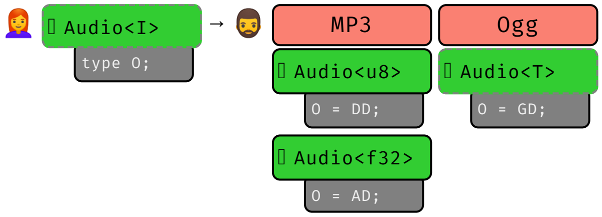
Trait author assumes:

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

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

Multiple In- and Output Parameters

```
trait Audio<I> {
    type O;
    fn play(&self, volume: I) -> Self::O;
}

impl Audio<u8> for MP3 { type O = DigitalDevice; }
impl Audio<f32> for MP3 { type O = AnalogDevice; }
impl<T> Audio<T> for Ogg { type O = GenericDevice; }

mp3.play(0_u8).flip_bits();
mp3.play(0_f32).rewind_tape();
```



Like examples above, in particular trait author assumes:

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

?Sized

S<T> → S<u8> S<char> S<str>

```
struct S<T> { … }
```

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

S<T> → S<u8> S<char> S<str>

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

Generics and Lifetimes — <'a>

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

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

S<'a> → S<'auto> S<'static>

- `'static` is only nameable instance of the *typespace* lifetimes.

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

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

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

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

> Note to self and TODO: that analogy seems somewhat flawed, as if `S<'a>` is to `S<'static>` like `S<T>` is to `S<u32>`, then `'static` would be a *type*; but then what's the value of that type?

Examples expand by clicking.

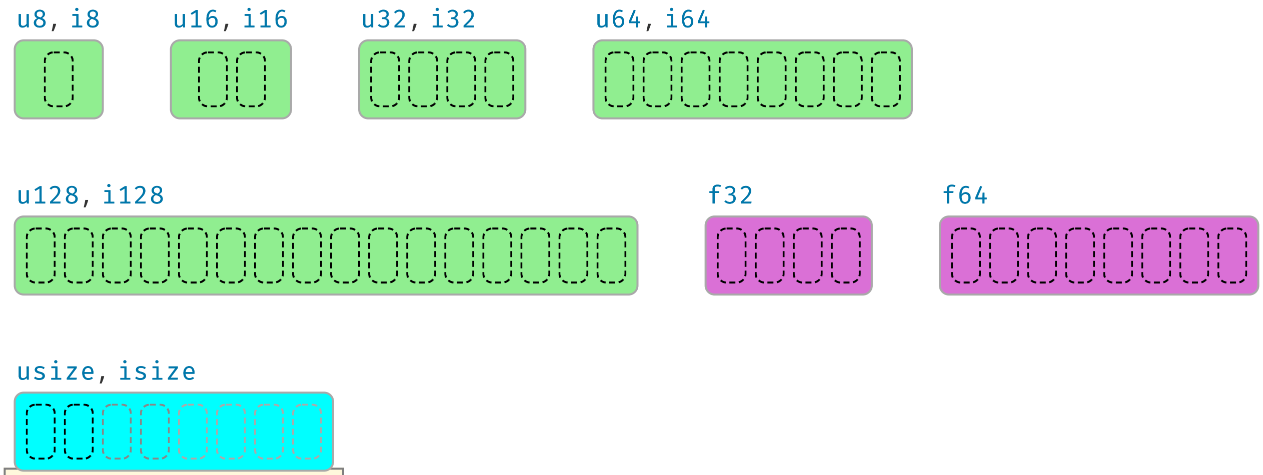
Data Layout

Memory representations of common data types.

Basic Types

Essential types built into the core of the language.

Numeric Types REF



Same as `ptr` on platform.

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	±∞

f32	S (1)	E (8)	F (23)	Value
NaN	±	255	non-zero	NaN

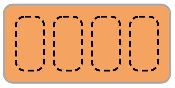
Similarly, for `f64` types this would look like:

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.

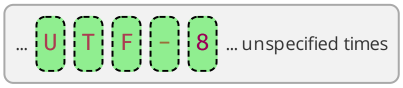
Textual Types REF

`char`



Any UTF-8 scalar.

`str`



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

Basics

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

Usage

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

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

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

Encoding

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

Variant	Memory Representation[2]
`s.as_bytes()`	`49 20 e2 9d a4 20 52 75 73 74` [3]

`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`.

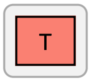
[4] Also observe how the emoji Red Heart ❤️, is a combination of ❤ and the U+FE0F Variation Selector, thus `t` has a higher char count than `s`.

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

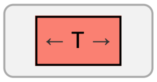
Basic types definable by users. Actual **layout** REF is subject to **representation**; REF padding can be present. [1]

`T`



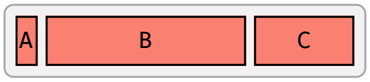
Sized ↓

`T: ?Sized`

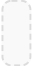


Maybe DST ↓

`(A, B, C)`

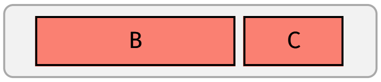


`struct S;`

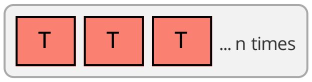


Zero-Sized ↓

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

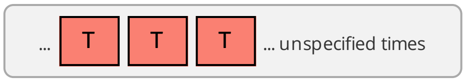


`[T; n]`



Fixed array of `n` elements.

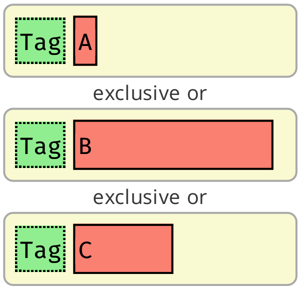
`[T]`



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

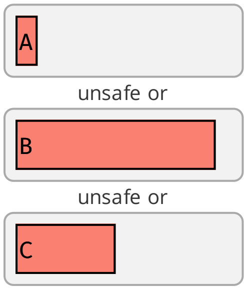
These **sum types** hold a value of one of their sub types:

`enum E { A, B, C }`



Safely holds A or B or C, also called 'tagged union', though compiler may omit tag.

`union { ... }`



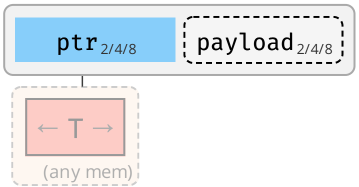
Can unsafely reinterpret memory. Result might be undefined.

[1] To be clear, the depiction of types here merely illustrates a *random* representation. Unless a certain one is forced (e.g., via `#[repr(C)]`, Rust will, for example, be free to layout `A(u8, u16)` as `u8 u16` and `B(u8, u16)` as `u16 u8`, even inside the same application!

References & Pointers

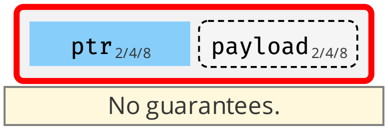
References give safe access to other memory, raw pointers `unsafe` access. For some referents additional `payload` may be present, see below. The respective `mut` types are identical.

`&'a T`



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

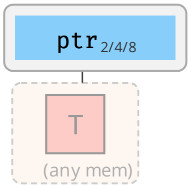
`*const T`



Pointer Payload

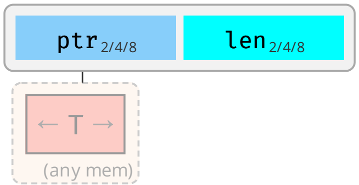
Many reference and pointer types can carry an extra field. This **`payload`**, if it exists, is either element- or byte-length of the target, or a pointer to a *vtable*.

`&'a T`



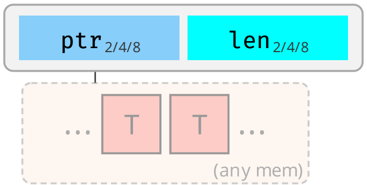
No payload for *normal*, sized referents.

`&'a T`



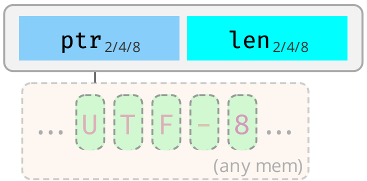
If `T` is a DST `struct` such as `S { x: [u8] }` field `len` is length of dyn. sized content.

`&'a [T]`



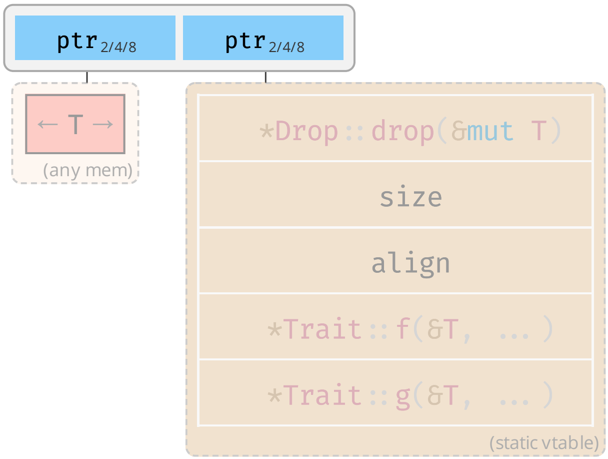
Regular **slice reference** (i.e., the reference type of a slice type `[T]`) ↑ often seen as `&[T]` if `'a` elided.

`&'a str`



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

`&'a dyn Trait`

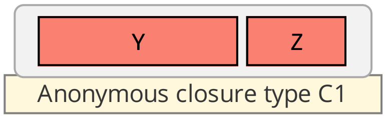


Where `*Drop::drop()`, `*Trait::f()`, ... are pointers to their respective `impl` for `T`.

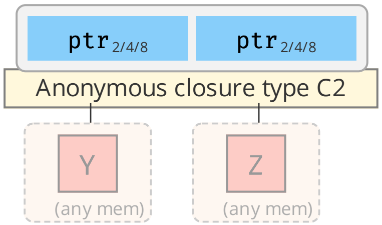
Closures

Ad-hoc functions with an automatically managed data block **capturing** REF environment where closure was defined. For example:

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



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

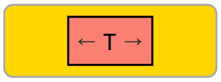


Also produces anonymous `fn` such as f_{c1}`(C1, X)` or f_{c2}`(&C2, X)`. Details depend which `FnOnce`, `FnMut`, `Fn` ... is supported, based on properties of captured types.

Standard Library Types

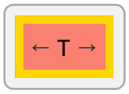
Rust's standard library combines the above primitive types into useful types with special semantics, e.g.:

`UnsafeCell<T>`



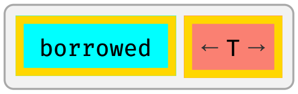
Magic type allowing aliased mutability.

`Cell<T>`



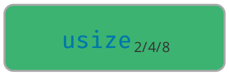
Allows `T`'s to move in and out.

`RefCell<T>`



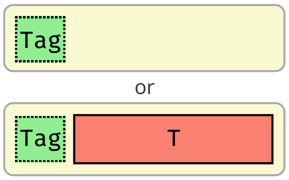
Also support dynamic borrowing of `T`. Like `Cell` this is `Send`, but not `Sync`.

`AtomicUsize`



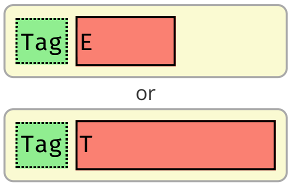
Other atomic similarly.

`Option<T>`



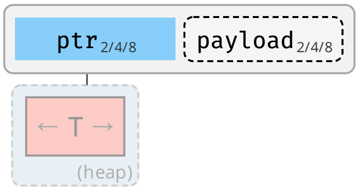
Tag may be omitted for certain T, e.g., `NonNull`.

`Result<T, E>`



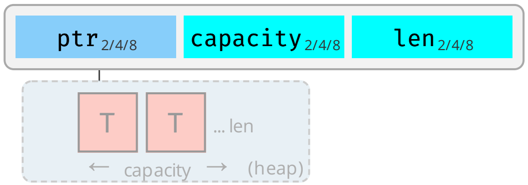
General Purpose Heap Storage

`Box<T>`



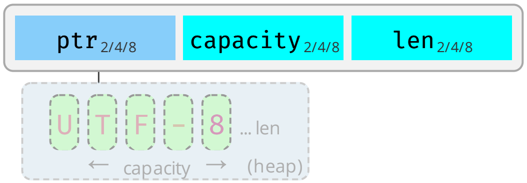
For some `T` stack proxy may carry payload↑ (e.g., `Box<[T]>`).

`Vec<T>`



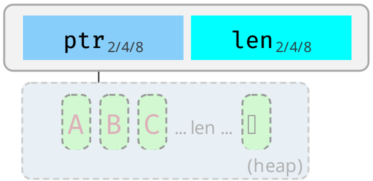
Owned Strings

String



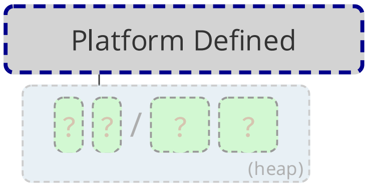
Observe how `String` differs from `&str` and `&[char]`.

CString



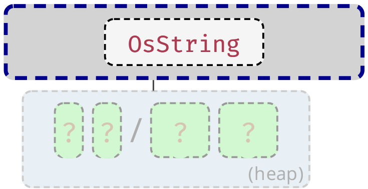
Nul-terminated but w/o nul in middle.

OsString ?



Encapsulates how operating system represents strings (e.g., UTF-16 on Windows).

PathBuf ?

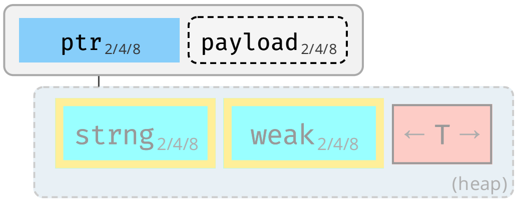


Encapsulates how operating system represents paths.

Shared Ownership

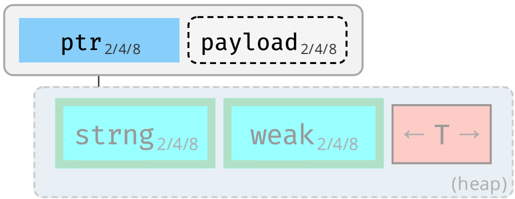
If the type does not contain a `Cell` for `T`, these are often combined with one of the `Cell` types above to allow shared de-facto mutability.

Rc<T>



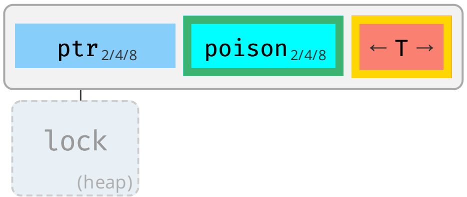
Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell`to allow mutation. Is neither `Send` nor `Sync`.

Arc<T>



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

Mutex<T> / RwLock<T>



Needs to be held in `Arc` to be shared between threads, always `Send` and `Sync`. Consider using parking_lot instead (faster, no heap usage).

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]	`format!("{}{}", 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; might not be fastest solution if `x` is `String` already.

[2] Requires regex crate.

I/O

Intent	Snippet
Create a new file	`File::create(PATH)?`
Same, via OpenOptions*	`OpenOptions::new().create(t).write(t).truncate(t).open(PATH)?`

* We're a bit short on space here, `t` means true.

Macros

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

Esoterics

Intent	Snippet
Cleaner closure captures	`wants_closure({ let c = outer.clone(); move || use_clone(c) })`
Fix inference in '`try`' closures	`iter.try_for_each(|x| { Ok::<(), Error>(()) })?;`

Intent	Snippet
Iterate *and* edit `&mut [T]` if `T` Copy.	`Cell::from_mut(mut_slice).as_slice_of_cells()`

Thread Safety

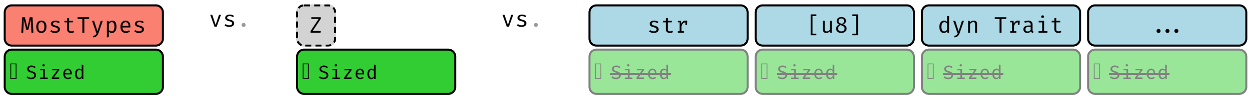
Examples	`Send`*	`!Send`
`Sync`*	*Most types* ... `Mutex<T>`, `Arc<T>`[1,2]	`MutexGuard<T>`[1], `RwLockReadGuard<T>`[1]
`!Sync`	`Cell<T>`[2], `RefCell<T>`[2]	`Rc<T>`, `Formatter`, `&dyn Trait`

* An instance `t` where `T: Send` can be moved to another thread, a `T: Sync` means `&t` can be moved to another thread.

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

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

(Dynamically / Zero) Sized Types



Normal types.

Zero sized.

Dynamically sized.

Overview

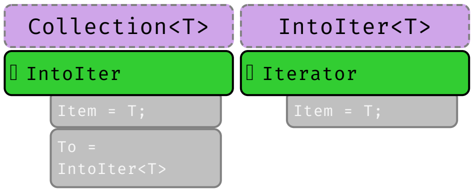
- A type `T` is **`Sized`** STD if at compile time it is known how many bytes it occupies, `u8` and `&[u8]` are, `[u8]` isn't.
- Being `Sized` means `impl Sized for T {}` holds. Happens automatically and cannot be user impl'ed.
- Types not `Sized` are called **dynamically sized types** BK NOM REF (DSTs),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

Sized in Bounds

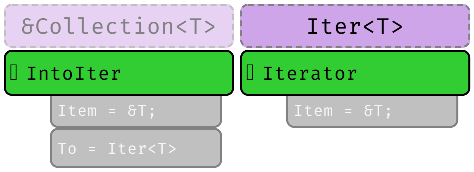
Example	Explanation
`struct A { x: u8 }`	Type `A` is sized, i.e., `impl Sized for A` holds, this is a 'regular' type.
`struct B { x: [u8] }`	Since `[u8]` is a DST, `B` in turn becomes DST, i.e., does not `impl Sized`.
`struct C<T> { x: T }`	Type params **have** implicit `T: Sized` bound, e.g., `C<A>`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

Example	Explanation
`trait F: Sized {}`	Traits can however opt into `Sized` via supertraits.[↑]
`trait G { fn g(self); }`	For `Self`-like params DST `impl` may still fail as params can't go on stack.

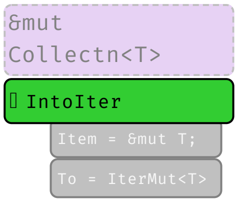
Iterators



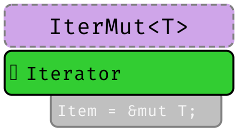
Iterate over `T`.



Iterate over `&T`.



Iterate over `&mut T`.



Using Iterators

Basics

Assume you have a collection `c` of type `C`:

- `c.into_iter()` — Turns collection `c` into an `Iterator` STD `i` and **consumes*** `c`. Requires `IntoIterator` STD for `C` to be implemented. Type of item depends on what `C` was. 'Standardized' way to get Iterators.
- `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`.

* 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 the reference and turn it into an Iterator), but `c` remains untouched.

Implementing Iterators

Basics

Let's assume you have a `struct C {}` that is your collection.

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

In addition, you might want to add a convenience `C::iter(&self) → IntoIter`.

Mutable Iterators

- `struct IterMut {}` — To provide mutable iterators create another struct that can hold `C` as `&mut`.
- `impl Iterator for IterMut {}` — In that case `Iterator::Item` is probably a `&mut item`

Similarly, providing a `C::iter_mut(&mut self) → IterMut` might be a good idea.

Making Loops Work

- `impl IntoIterator for C {}` — Now `for` loops work as `for x in c {}`.
- `impl IntoIterator for &C {}` — For conveninece you might want to add these as well.
- `impl IntoIterator for &mut C {}` — Same ...

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`).

[3] Might misrepresent number (`u64::MAX as f32`) or produce `Inf` (`u128::MAX as f32`).

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()`

If you have x of type ...	Use this ...
`&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`
`PathBuf`	`x.into_os_string()`
`Vec<u8>` [1]	?
`&str`	`OsString::from(x)` [i]
`&CStr`	`OsString::from(x.to_str()?)`
`&OsStr`	`OsString::from(x)` [i]
`&Path`	`x.as_os_str().to_owned()`
`&[u8]` [1]	?

PathBuf

If you have x of type ...	Use this ...

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()?

If you have x of type ...	Use this ...
&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()
Vec<u8> [1]	?
&str	OsStr::new(x)
&CStr	?
&OsStr	x
&Path	x.as_os_str()
&[u8] [1]	?

&Path

If you have x of type ...	Use this ...
String	Path::new(x) [r]

If you have x of type ...	Use this ...
`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`

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

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

Full Example	Explanation
`println!("{}", x)`	Print `x` using `Display`$^{\text{STD}}$ on std. out and append new line.

Full Example	Explanation
`format!("{a:.3} {b:?}", a = PI, b = 2)`	Convert `PI` with 3 digits, add space, b with `Debug`[STD], return `String`.

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.
📁 `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** 🔗, e.g., when compiling C / FFI.
`Cargo.toml`	Main project configuration. Defines dependencies, artifacts ...
`Cargo.lock`	Dependency details for reproducible builds, recommended to `git` for apps, not for libs.

[*] On stable consider Criterion.

Minimal examples for various entry points might look like:

Applications

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

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

Libraries

Proc Macros

```rust
// 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.
```

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

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

use proc_macro::TokenStream;

#[proc_macro_attribute]   // Can now be used as `#[my_attribute]`
pub fn my_attribute(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

```toml
// Cargo.toml

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

[lib]
proc-macro = true
```

Unit Tests

```rust
// 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.

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`).
- A `mod m {}` defines module in-file, while `mod m;` will read `m.rs` or `m/mod.rs`.
  - Path of `.rs` based on nesting, e.g., `mod a { mod b { mod c; }}}` is either `a/b/c.rs` or `a/b/c/mod.rs`.
  - Files not pathed from module tree root via some `mod m;` won't be touched by compiler! 🛑

Namespaces

Rust has three kinds of **namespaces**:

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

[1] Counts in *Types* and in *Functions*.

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

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

Cargo

Commands and tools that are good to know.

Command	Description
`cargo init`	Create a new project for the latest edition.
`cargo build`	Build the project in debug mode (`--release` for all optimization).
`cargo check`	Check if project would compile (much faster).
`cargo test`	Run tests for the project.
`cargo 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).

Command	Description
`cargo run -p w`	Run main of sub-workspace `w`. Treats features more as you would expect.
`cargo tree`	Show dependency graph.
`cargo doc --open`	Locally generate documentation for your code and dependencies.
`cargo +{nightly, stable} ...`	Use given toolchain for command, e.g., for 'nightly only' tools.
`cargo +nightly ...`	Some nightly-only commands (substitute `...` with command below)
`build -Z timings`	Show what crates caused your build to take so long, highly useful. 🚧 🔥
`rustc -- -Zunpretty=expanded`	Show expanded macros. 🚧
`rustup doc`	Open offline Rust documentation (incl. the books), good on a plane!

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

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

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

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

Cross Compilation

🔵 Check target is supported.

🔵 Install target via **`rustup target install X`**.

🔵 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 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=X`**

Coding Guides

Idiomatic Rust

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

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

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

Async-Await 101

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

Basics

Construct	Explanation
`async`	Anything declared `async` always returns an `impl Future<Output=_>`. [STD]
`async fn f() {}`	Function `f` returns an `impl Future<Output=()>`.

Construct	Explanation
`async fn f() → S {}`	Function `f` returns an `impl Future<Output=S>`.
`async { x }`	Transforms `{ x }` into an `impl Future<Output=X>`.
`let sm = f();`	Calling `f()` that is `async` will **not** execute `f`, but produce state machine `sm`.[1] [2]
`sm = async { g() };`	Likewise, does **not** execute the `{ g() }` block; produces state machine.
`runtime.block_on(sm);`	Outside an `async {}`, schedules `sm` to actually run. Would execute `g()`.[3] [4]
`sm.await`	Inside an `async {}`, run `sm` until complete. Yield to runtime if `sm` not ready.

[1] Technically `async` transforms following code into anonymous, compiler-generated state machine type; `f()` instantiates that machine.
[2] The state machine always `impl Future`, possibly `Send` & co, depending on types used inside `async`.
[3] State machine driven by worker thread invoking `Future::poll()` via runtime directly, or parent `.await` indirectly.
[4] Rust doesn't come with runtime, need external crate instead, e.g., [async-std](https://github.com/async-rs/async-std) or [tokio 0.2+](https://crates.io/crates/tokio). Also, more helpers in [futures crate](https://github.com/rust-lang-nursery/futures-rs).

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]

Constructs [1]	Explanation
`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](https://doc.rust-lang.org/std/ops/trait.Drop.html) 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()` 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.

Unsafe Code

Unsafe Code

- Code marked `unsafe` has special permissions, e.g., to deref raw pointers, or invoke other `unsafe` functions.

- Along come special **promises the author *must* uphold to the compiler**, and the compiler *will* trust you.
- By itself `unsafe` code is not bad, but dangerous, and needed for FFI or exotic data structures.

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

Undefined Behavior

Undefined Behavior (UB)

- As mentioned, `unsafe` code implies special promises to the compiler (it wouldn't need be `unsafe` otherwise).
- Failure to uphold any promise makes compiler produce fallacious code, execution of which leads to UB.
- After triggering undefined behavior *anything* can happen. Insidiously, the effects may be 1) subtle, 2) manifest far away from the site of violation or 3) be visible only under certain conditions.
- A seemingly *working* program (incl. any number of unit tests) is no proof UB code might not fail on a whim.
- Code with UB is objectively dangerous, invalid and should never exist.

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

Unsound Code

Unsound Code

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

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

Responsible use of Unsafe

- Do not use `unsafe` unless you absolutely have to.

- Follow the Nomicon, Unsafe Guidelines, **always** uphold **all** safety invariants, 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.

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.
🔴 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

Signatures in Functions

🔴 Adding / removing arguments.

🟡 Introducing a new type parameter.

🟡 Generalizing to generics.

Behavioral Changes

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

Ralf Biedert, 2021 – cheats.rs