



Rust Language Cheat Sheet

24. December 2022

Contains clickable links to [The Book](#) ^{BK}, [Rust by Example](#) ^{EX}, [Std Docs](#) ^{STD}, [Nomicon](#) ^{NOM}, [Reference](#) ^{REF}.



Data Structures

Data types and memory locations defined via keywords.

Example	Explanation
<code>struct S {}</code>	Define a struct ^{BK EX STD REF} with named fields.
<code>struct S { x: T }</code>	Define struct with named field <code>x</code> of type <code>T</code> .
<code>struct S (T);</code>	Define "tupled" struct with numbered field <code>.0</code> of type <code>T</code> .
<code>struct S;</code>	Define zero sized ^{NOM} unit struct. Occupies no space, optimized away.
<code>enum E {}</code>	Define an enum , ^{BK EX REF} c. algebraic data types, tagged unions .
<code>enum E { A, B (), C {} }</code>	Define variants of enum; can be unit- <code>A</code> , tuple- <code>B ()</code> and struct-like <code>C {}</code> .
<code>enum E { A = 1 }</code>	If variants are only unit-like, allow discriminant values , ^{REF} e.g., for FFI.
<code>enum E {}</code>	Enum w/o variants is uninhabited , ^{REF} can't be instantiated, c. 'never' ¹
<code>union U {}</code>	Unsafe C-like union ^{REF} for FFI compatibility.
<code>static X: T = T();</code>	Global variable ^{BK EX REF} with <code>'static</code> lifetime, single memory location.
<code>const X: T = T();</code>	Defines constant , ^{BK EX REF} copied into a temporary when used.
<code>let x: T;</code>	Allocate <code>T</code> bytes on stack ¹ bound as <code>x</code> . Assignable once, not mutable.
<code>let mut x: T;</code>	Like <code>let</code> , but allow for mutability ^{BK EX} and mutable borrow. ²
<code>x = y;</code>	Moves <code>y</code> to <code>x</code> , invalidating <code>y</code> if <code>T</code> is not copy , ^{STD} and copying <code>y</code> otherwise.

¹ **Bound variables** ^{BK EX REF} live on stack for synchronous code. In `async {}` they become part of `async`'s state machine, may reside on heap.

² Technically *mutable* and *immutable* are misnomer. Immutable binding or shared reference may still contain `Cell` ^{STD}, giving *interior mutability*.

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

Example	Explanation
<code>S { x: y }</code>	Create <code>struct S {}</code> or <code>use'ed enum E::S {}</code> with field <code>x</code> set to <code>y</code> .
<code>S { x }</code>	Same, but use local variable <code>x</code> for field <code>x</code> .
<code>S { ..s }</code>	Fill remaining fields from <code>s</code> , esp. useful with <code>Default::default()</code> , ^{STD}
<code>S { 0: x }</code>	Like <code>S (x)</code> below, but set field <code>.0</code> with struct syntax.
<code>S (x)</code>	Create <code>struct S (T)</code> or <code>use'ed enum E::S ()</code> with field <code>.0</code> set to <code>x</code> .
<code>S</code>	If <code>S</code> is unit <code>struct S;</code> or <code>use'ed enum E::S</code> create value of <code>S</code> .
<code>E::C { x: y }</code>	Create enum variant <code>C</code> . Other methods above also work.
<code>()</code>	Empty tuple, both literal and type, aka unit . ^{STD}

Example	Explanation
<code>(x)</code>	Parenthesized expression.
<code>(x,)</code>	Single-element tuple expression. EX STD REF
<code>(S,)</code>	Single-element tuple type.
<code>[S]</code>	Array type of unspecified length, i.e., slice . EX STD REF Can't live on stack. *
<code>[S; n]</code>	Array type EX STD REF of fixed length <code>n</code> holding elements of type <code>S</code> .
<code>[x; n]</code>	Array instance REF (expression) with <code>n</code> copies of <code>x</code> .
<code>[x, y]</code>	Array instance with given elements <code>x</code> and <code>y</code> .
<code>x[0]</code>	Collection indexing, here w. usize . Implementable with Index , IndexMut .
<code>x[...]</code>	Same, via range (here <i>full range</i>), also <code>x[a..b]</code> , <code>x[a..=b]</code> , ... <code>c</code> . below.
<code>a..b</code>	Right-exclusive range STD REF creation, e.g., <code>1..3</code> means <code>1, 2</code> .
<code>..b</code>	Right-exclusive range to STD without starting point.
<code>..=b</code>	Inclusive range to STD without starting point.
<code>a..=b</code>	Inclusive range , STD <code>1..=3</code> means <code>1, 2, 3</code> .
<code>a..</code>	Range from STD without ending point.
<code>..</code>	Full range , STD usually means <i>the whole collection</i> .
<code>s.x</code>	Named field access , REF might try to Deref if <code>x</code> not part of type <code>S</code> .
<code>s.0</code>	Numbered field access, used for tuple types <code>S (T)</code> .

* For now, [RFC](#) pending completion of [tracking issue](#).

References & Pointers

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

Example	Explanation
<code>&S</code>	Shared reference BK STD NOM REF (type; space for holding <i>any</i> <code>&s</code>).
<code>&[S]</code>	Special slice reference that contains (address, count).
<code>&str</code>	Special string slice reference that contains (address, <code>byte_length</code>).
<code>&mut S</code>	Exclusive reference to allow mutability (also <code>&mut [S]</code> , <code>&mut dyn S</code> , ...).
<code>&dyn T</code>	Special trait object BK reference that contains (address, <code>vtable</code>).
<code>&s</code>	Shared borrow BK EX STD (e.g., address, len, <code>vtable</code> , ... of <i>this</i> <code>s</code> , like <code>0x1234</code>).
<code>&mut s</code>	Exclusive borrow that allows mutability . EX
<code>*const S</code>	Immutable raw pointer type BK STD REF w/o memory safety.
<code>*mut S</code>	Mutable raw pointer type w/o memory safety.
<code>&raw const s</code>	Create raw pointer w/o going through reference; c. <code>ptr::addr_of!()</code> STD 🚧
<code>&raw mut s</code>	Same, but mutable. 🚧 Raw ptrs. are needed for unaligned, packed fields.
<code>ref s</code>	Bind by reference , EX makes binding reference type. 🗑️
<code>let ref r = s;</code>	Equivalent to <code>let r = &s</code> .
<code>let S { ref mut x } = s;</code>	Mutable ref binding (<code>let x = &mut s.x</code>), shorthand destructuring + version.
<code>*r</code>	Dereference BK STD NOM a reference <code>r</code> to access what it points to.
<code>*r = s;</code>	If <code>r</code> is a mutable reference, move or copy <code>s</code> to target memory.
<code>s = *r;</code>	Make <code>s</code> a copy of whatever <code>r</code> references, if that is <code>Copy</code> .
<code>s = *r;</code>	Won't work 🚫 if <code>*r</code> is not <code>Copy</code> , as that would move and leave empty place.
<code>s = *my_box;</code>	Special case 🗑️ for <code>Box</code> STD that can also move out b'ed content that isn't <code>Copy</code> .

Example	Explanation
<code>'a</code>	A lifetime parameter , ^{BK EX NOM REF} duration of a flow in static analysis.
<code>&'a S</code>	Only accepts address of some <code>S</code> ; address existing <code>'a</code> or longer.
<code>&'a mut S</code>	Same, but allow address content to be changed.
<code>struct S<'a> {}</code>	Signals this <code>S</code> will contain address with lifetime <code>'a</code> . Creator of <code>S</code> decides <code>'a</code> .
<code>trait T<'a> {}</code>	Signals any <code>S</code> , which <code>impl T for S</code> , might contain address.
<code>fn f<'a>(t: &'a T)</code>	Signals this function handles some address. Caller decides <code>'a</code> .
<code>'static</code>	Special lifetime lasting the entire program execution.

Functions & Behavior

Define units of code and their abstractions.

Example	Explanation
<code>trait T {}</code>	Define a trait , ^{BK EX REF} common behavior types can adhere to.
<code>trait T : R {}</code>	<code>T</code> is subtrait of supertrait ^{BK EX REF} <code>R</code> . Any <code>S</code> must <code>impl R</code> before it can <code>impl T</code> .
<code>impl S {}</code>	Implementation ^{REF} of functionality for a type <code>S</code> , e.g., methods.
<code>impl T for S {}</code>	Implement trait <code>T</code> for type <code>S</code> ; specifies <i>how exactly</i> <code>S</code> acts like <code>T</code> .
<code>impl !T for S {}</code>	Disable an automatically derived auto trait . ^{NOM REF} 🚫
<code>fn f() {}</code>	Definition of a function , ^{BK EX REF} or associated function if inside <code>impl</code> .
<code>fn f() -> S {}</code>	Same, returning a value of type <code>S</code> .
<code>fn f(&self) {}</code>	Define a method , ^{BK EX REF} e.g., within an <code>impl S {}</code> .
<code>struct S(T);</code>	More arcanelly, <i>also</i> defines <code>fn S(x: T) -> S</code> constructor function . ^{RFC}
<code>const fn f() {}</code>	Constant <code>fn</code> usable at compile time, e.g., <code>const X: u32 = f(Y)</code> . ^{'18}
<code>async fn f() {}</code>	Async ^{REF '18} function transformation, [!] makes <code>f</code> return an <code>impl Future</code> . ^{STD}
<code>async fn f() -> S {}</code>	Same, but make <code>f</code> return an <code>impl Future<Output=S></code> .
<code>async { x }</code>	Used within a function, make <code>{ x }</code> an <code>impl Future<Output=X></code> .
<code>fn() -> S</code>	Function pointers , ^{BK STD REF} memory holding address of a callable.
<code>Fn() -> S</code>	Callable Trait ^{BK STD} (also <code>FnMut</code> , <code>FnOnce</code>), implemented by closures, <code>fn</code> 's ...
<code> {}</code>	A closure ^{BK EX REF} that borrows its captures , [!] ^{REF} (e.g., a local variable).
<code> x {}</code>	Closure accepting one argument named <code>x</code> , body is block expression.
<code> x x + x</code>	Same, without block expression; may only consist of single expression.
<code>move x x + y</code>	Move closure ^{REF} taking ownership; i.e., <code>y</code> transferred into closure.
<code>return true</code>	Closures sometimes look like logical ORs (here: return a closure).
<code>unsafe</code>	If you enjoy debugging segfaults Friday night; unsafe code . [!] ^{BK EX NOM REF}
<code>unsafe fn f() {}</code>	Means " <i>calling can cause UB, [!] YOU must check requirements</i> ".
<code>unsafe trait T {}</code>	Means " <i>careless impl. of T can cause UB; implementor must check</i> ".
<code>unsafe { f(); }</code>	Guarantees to compiler " <i>I have checked requirements, trust me</i> ".
<code>unsafe impl T for S {}</code>	Guarantees <code>S</code> is <i>well-behaved w.r.t T</i> ; people may use <code>T</code> on <code>S</code> safely.

Control Flow

Control execution within a function.

Example	Explanation
<code>while x {}</code>	Loop , ^{REF} run while expression <code>x</code> is true.

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

Organizing Code

Segment projects into smaller units and minimize dependencies.

Example	Explanation
<code>mod m {}</code>	Define a module , BK EX REF get definition from inside <code>{}</code> . ⁴
<code>mod m;</code>	Define a module, get definition from <code>m.rs</code> or <code>m/mod.rs</code> . ⁴
<code>a::b</code>	Namespace path EX REF to element <code>b</code> within <code>a</code> (<code>mod</code> , <code>enum</code> , ...).
<code>::b</code>	Search <code>b</code> in crate root ¹⁵ REF or external prelude ; ¹⁸ REF global path . REF 🗑️
<code>crate::b</code>	Search <code>b</code> in crate root. ¹⁸
<code>self::b</code>	Search <code>b</code> in current module.
<code>super::b</code>	Search <code>b</code> in parent module.

Example	Explanation
<code>use a::b;</code>	Use EX REF <code>b</code> directly in this scope without requiring <code>a</code> anymore.
<code>use a::{b, c};</code>	Same, but bring <code>b</code> and <code>c</code> into scope.
<code>use a::b as x;</code>	Bring <code>b</code> into scope but name <code>x</code> , like <code>use std::error::Error as E</code> .
<code>use a::b as _;</code>	Bring <code>b</code> anonymously into scope, useful for traits with conflicting names.
<code>use a::*;</code>	Bring everything from <code>a</code> in, only recommended if <code>a</code> is some prelude . STD link
<code>pub use a::b;</code>	Bring <code>a::b</code> into scope and reexport from here.
<code>pub T</code>	"Public if parent path is public" visibility BK REF for <code>T</code> .
<code>pub(crate) T</code>	Visible at most ¹ in current crate.
<code>pub(super) T</code>	Visible at most ¹ in parent.
<code>pub(self) T</code>	Visible at most ¹ in current module (default, same as no <code>pub</code>).
<code>pub(in a::b) T</code>	Visible at most ¹ in ancestor <code>a::b</code> .
<code>extern crate a;</code>	Declare dependency on external crate ; BK REF link just <code>use a::b</code> in ¹⁸ .
<code>extern "C" {}</code>	Declare external dependencies and ABI (e.g., <code>"C"</code>) from FFI . BK EX NOM REF
<code>extern "C" fn f() {}</code>	Define function to be exported with ABI (e.g., <code>"C"</code>) to FFI.

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

Type Aliases and Casts

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

Example	Explanation
<code>type T = S;</code>	Create a type alias , BK REF i.e., another name for <code>S</code> .
<code>Self</code>	Type alias for implementing type , REF e.g., <code>fn new() -> Self</code> .
<code>self</code>	Method subject in <code>fn f(self) {}</code> , e.g., akin to <code>fn f(self: Self) {}</code> .
<code>&self</code>	Same, but refers to self as borrowed, would equal <code>f(self: &Self)</code>
<code>&mut self</code>	Same, but mutably borrowed, would equal <code>f(self: &mut Self)</code>
<code>self: Box<Self></code>	Arbitrary self type , add methods to smart pointers (<code>my_box.f_of_self()</code>).
<code><S as T></code>	Disambiguate BK REF type <code>S</code> as trait <code>T</code> , e.g., <code><S as T>::f()</code> .
<code>a::b as c</code>	In <code>use</code> of symbol, import <code>S</code> as <code>R</code> , e.g., <code>use a::S as R</code> .
<code>x as u32</code>	Primitive cast , EX REF may truncate and be a bit surprising. ¹ NOM

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

Macros & Attributes

Code generation constructs expanded before the actual compilation happens.

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

Inside Macros	Explanation
<code>\$x:ty</code>	Macro capture, the <code>!...</code> fragment REF declares what is allowed for <code>\$x</code> . ¹
<code>\$x</code>	Macro substitution, e.g., use the captured <code>\$x:ty</code> from above.
<code>\$(x),*</code>	Macro repetition REF <i>zero or more times</i> in macros by example.
<code>\$(x),?</code>	Same, but <i>zero or one time</i> .

Inside Macros	Explanation
<code>\$(x),+</code>	Same, but <i>one or more times</i> .
<code>\$(x)<<+</code>	In fact separators other than <code>,</code> are also accepted. Here: <code><<</code> .

¹ See [Tooling Directives](#) below for all captures.

Pattern Matching

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

Example	Explanation
<code>match m { }</code>	Initiate pattern matching , BK EX REF then use match arms, c. next table.
<code>let S(x) = get();</code>	Notably, <code>let</code> also destructures EX similar to the table below.
<code>let S { x } = s;</code>	Only <code>x</code> will be bound to value <code>s.x</code> .
<code>let (_, b, _) = abc;</code>	Only <code>b</code> will be bound to value <code>abc.1</code> .
<code>let (a, ..) = abc;</code>	Ignoring 'the rest' also works.
<code>let (.., a, b) = (1, 2);</code>	Specific bindings take precedence over 'the rest', here <code>a</code> is <code>1</code> , <code>b</code> is <code>2</code> .
<code>let s @ S { x } = get();</code>	Bind <code>s</code> to <code>S</code> while <code>x</code> is bound to <code>s.x</code> , pattern binding , BK EX REF c. below
<code>let w @ t @ f = get();</code>	Stores 3 copies of <code>get()</code> result in each <code>w</code> , <code>t</code> , <code>f</code> .
<code>let (x x) = get();</code>	Pathological or-pattern, ¹ not closure. ● Same as <code>let x = get();</code>
<code>let Some(x) = get();</code>	Won't work ● if pattern can be refuted , REF use <code>let else</code> or <code>if let</code> instead.
<code>let Some(x) = get() else { }</code>	Assign if possible, RFC if not <code>else { }</code> w. must <code>break</code> , <code>return</code> , <code>panic!</code> , ... ^{1.65+} 🔥
<code>if let Some(x) = get() { }</code>	Branch if pattern can be assigned (e.g., <code>enum</code> variant), syntactic sugar. [*]
<code>while let Some(x) = get() { }</code>	Equiv.; here keep calling <code>get()</code> , run <code>{ }</code> as long as pattern can be assigned.
<code>fn f(S { x }: S)</code>	Function parameters also work like <code>let</code> , here <code>x</code> bound to <code>s.x</code> of <code>f(s)</code> .

^{*} 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
<code>E::A => { }</code>	Match enum variant <code>A</code> , c. pattern matching . BK EX REF
<code>E::B (..) => { }</code>	Match enum tuple variant <code>B</code> , ignoring any index.
<code>E::C { .. } => { }</code>	Match enum struct variant <code>C</code> , ignoring any field.
<code>S { x: 0, y: 1 } => { }</code>	Match struct with specific values (only accepts <code>s</code> with <code>s.x</code> of <code>0</code> and <code>s.y</code> of <code>1</code>).
<code>S { x: a, y: b } => { }</code>	Match struct with <i>any</i> ● values and bind <code>s.x</code> to <code>a</code> and <code>s.y</code> to <code>b</code> .
<code>S { x, y } => { }</code>	Same, but shorthand with <code>s.x</code> and <code>s.y</code> bound as <code>x</code> and <code>y</code> respectively.
<code>S { .. } => { }</code>	Match struct with any values.
<code>D => { }</code>	Match enum variant <code>E::D</code> if <code>D</code> in <code>use</code> .
<code>D => { }</code>	Match anything, bind <code>D</code> ; possibly false friend ● of <code>E::D</code> if <code>D</code> not in <code>use</code> .
<code>_ => { }</code>	Proper wildcard that matches anything / "all the rest".
<code>0 1 => { }</code>	Pattern alternatives, or-patterns . RFC
<code>E::A E::Z => { }</code>	Same, but on enum variants.
<code>E::C {x} E::D {x} => { }</code>	Same, but bind <code>x</code> if all variants have it.
<code>Some(A B) => { }</code>	Same, can also match alternatives deeply nested.
<code> x x => { }</code>	Pathological or-pattern , ● leading <code> </code> ignored, is just <code>x x</code> , therefore <code>x</code> .
<code>(a, 0) => { }</code>	Match tuple with any value for <code>a</code> and <code>0</code> for second.
<code>[a, 0] => { }</code>	Slice pattern , REF 🔗 match array with any value for <code>a</code> and <code>0</code> for second.

Within Match Arm	Explanation
<code>[1, ..] => {}</code>	Match array starting with <code>1</code> , any value for rest; slice pattern . REF RFC
<code>[1, .., 5] => {}</code>	Match array starting with <code>1</code> , ending with <code>5</code> .
<code>[1, x @ .., 5] => {}</code>	Same, but also bind <code>x</code> to slice representing middle (c. pattern binding).
<code>[a, x @ .., b] => {}</code>	Same, but match any first, last, bound as <code>a</code> , <code>b</code> respectively.
<code>1 .. 3 => {}</code>	Range pattern , BK REF here matches <code>1</code> and <code>2</code> ; partially unstable. 🚧
<code>1 ..= 3 => {}</code>	Inclusive range pattern, matches <code>1</code> , <code>2</code> and <code>3</code> .
<code>1 .. => {}</code>	Open range pattern, matches <code>1</code> and any larger number.
<code>x @ 1..=5 => {}</code>	Bind matched to <code>x</code> ; pattern binding , BK EX REF here <code>x</code> would be <code>1</code> , <code>2</code> , ... or <code>5</code> .
<code>Err(x @ Error {..}) => {}</code>	Also works nested, here <code>x</code> binds to <code>Error</code> , esp. useful with <code>if</code> below.
<code>S { x } if x > 10 => {}</code>	Pattern match guards , BK EX REF condition must be true as well to match.

Generics & Constraints

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

Example	Explanation
<code>struct S<T> ...</code>	A generic BK EX type with a type parameter (<code>T</code> is placeholder name here).
<code>S<T> where T: R</code>	Trait bound , BK EX REF limits allowed <code>T</code> , guarantees <code>T</code> has <code>R</code> ; <code>R</code> must be trait.
<code>where T: R, P: S</code>	Independent trait bounds , here one for <code>T</code> and one for (not shown) <code>P</code> .
<code>where T: R, S</code>	Compile error, 🚫 you probably want compound bound <code>R + S</code> below.
<code>where T: R + S</code>	Compound trait bound , BK EX T must fulfill <code>R</code> and <code>S</code> .
<code>where T: R + 'a</code>	Same, but w. lifetime. <code>T</code> must fulfill <code>R</code> , if <code>T</code> has lifetimes, must outlive <code>'a</code> .
<code>where T: ?Sized</code>	Opt out of a pre-defined trait bound, here <code>Sized</code> . ?
<code>where T: 'a</code>	Type lifetime bound ; EX if <code>T</code> has references, they must outlive <code>'a</code> .
<code>where T: 'static</code>	Same; does esp. <i>not</i> mean value <code>t</code> <i>will</i> 🚫 live <code>'static</code> , only that it could.
<code>where 'b: 'a</code>	Lifetime <code>'b</code> must live at least as long as (i.e., <i>outlive</i>) <code>'a</code> bound.
<code>where u8: R<T></code>	Also allows you to make conditional statements involving <i>other</i> types.
<code>S<T: R></code>	Short hand bound, almost same as above, shorter to write.
<code>S<const N: usize></code>	Generic const bound ; REF user of type <code>S</code> can provide constant value <code>N</code> .
<code>S<10></code>	Where used, const bounds can be provided as primitive values.
<code>S<{5+5}></code>	Expressions must be put in curly brackets.
<code>S<T = R></code>	Default parameters ; BK makes <code>S</code> a bit easier to use, but keeps it flexible.
<code>S<const N: u8 = 0></code>	Default parameter for constants; e.g., in <code>f(x: S) {}</code> param <code>N</code> is <code>0</code> .
<code>S<T = u8></code>	Default parameter for types, e.g., in <code>f(x: S) {}</code> param <code>T</code> is <code>u8</code> .
<code>S<'_></code>	Inferred anonymous lifetime ; asks compiler to <i>'figure it out'</i> if obvious.
<code>S<_></code>	Inferred anonymous type , e.g., as <code>let x: Vec<_> = iter.collect()</code>
<code>S::<T></code>	Turbofish STD call site type disambiguation, e.g., <code>f::<u32>()</code> .
<code>trait T<X> {}</code>	A trait generic over <code>X</code> . Can have multiple <code>impl T for S</code> (one per <code>X</code>).
<code>trait T { type X; }</code>	Defines associated type BK REF RFC X . Only one <code>impl T for S</code> possible.
<code>trait T { type X<G>; }</code>	Defines generic associated type (GAT), RFC e.g., <code>X</code> can be generic <code>Vec<></code> . ^{1.65+}
<code>trait T { type X<'a>; }</code>	Defines a GAT generic over a lifetime.
<code>type X = R;</code>	Set associated type within <code>impl T for S { type X = R; }</code> .
<code>type X<G> = R<G>;</code>	Same for GAT, e.g., <code>impl T for S { type X<G> = Vec<G>; }</code> .

Example	Explanation
<code>impl<T> S<T> {}</code>	Implement <code>fn</code> 's for any <code>T</code> in <code>S<T></code> generically , REF here <code>T</code> type parameter.
<code>impl S<T> {}</code>	Implement <code>fn</code> 's for exactly <code>S<T></code> inherently , REF here <code>T</code> specific type, e.g., <code>u8</code> .
<code>fn f() -> impl T</code>	Existential types , BK returns an unknown-to-caller <code>S</code> that <code>impl T</code> .
<code>fn f(x: &impl T)</code>	Trait bound via " impl traits ", BK somewhat like <code>fn f<S: T>(x: &S)</code> below.
<code>fn f(x: &dyn T)</code>	Invoke <code>f</code> via dynamic dispatch , BK REF <code>f</code> will not be instantiated for <code>x</code> .
<code>fn f<X: T>(x: X)</code>	Function generic over <code>X</code> , <code>f</code> will be instantiated (' monomorphized ') per <code>X</code> .
<code>fn f() where Self: R;</code>	In <code>trait T {}</code> , make <code>f</code> accessible only on types known to also <code>impl R</code> .
<code>fn f() where Self: Sized;</code>	Using <code>Sized</code> can opt <code>f</code> out of <code>dyn T</code> trait object vtable, enabling trait obj.
<code>fn f() where Self: R {}</code>	Other <code>R</code> useful w. dflt. methods (non dflt. would need be impl'ed anyway).

Higher-Ranked Items

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

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

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

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

Strings & Chars

Rust has several ways to create textual values.

Example	Explanation
<code>"..."</code>	String literal , REF . ¹ UTF-8, will interpret the following escapes, ...
<code>"\n\r\t\0\\"</code>	Common escapes REF , e.g., <code>"\n"</code> becomes <i>new line</i> .
<code>"\x36"</code>	ASCII e. REF up to 7f, e.g., <code>"\x36"</code> would become <code>6</code> .
<code>"\u{7fff}"</code>	Unicode e. REF up to 6 digits, e.g., <code>"\u{7fff}"</code> becomes <code> </code> .
<code>r"..."</code>	Raw string literal . REF . ¹ UTF-8, but won't interpret any escape above.
<code>r#"..."#</code>	Raw string literal, UTF-8, but can also contain <code>"</code> . Number of <code>#</code> can vary.
<code>b"..."</code>	Byte string literal ; REF . ¹ constructs ASCII <code>[u8]</code> , not a string.
<code>br"...", br#"..."#</code>	Raw byte string literal, ASCII <code>[u8]</code> , combination of the above.
<code>'🍌'</code>	Character literal , REF fixed 4 byte unicode <code>'char'</code> . STD

Example	Explanation
<code>b'x'</code>	ASCII byte literal , REF a single <code>u8</code> byte.

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

Documentation

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

Example	Explanation
<code>///</code>	Outer line doc comment , ¹ BK EX REF use these on types, traits, functions, ...
<code>/*!</code>	Inner line doc comment, mostly used at start of file to document module.
<code>//</code>	Line comment, use these to document code flow or <i>internals</i> .
<code>/* ... */</code>	Block comment. ² 🗑️
<code>/** ... */</code>	Outer block doc comment. ² 🗑️
<code>/*! ... */</code>	Inner block doc comment. ² 🗑️

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

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

Miscellaneous

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

Example	Explanation
<code>!</code>	Always empty never type . BK EX STD REF
<code>fn f() -> ! {}</code>	Function that never returns; compat. with any type e.g., <code>let x: u8 = f();</code>
<code>fn f() -> Result<, !> {}</code>	Function that must return <code>Result</code> but signals it can never <code>Err</code> . 🦋
<code>fn f(x: !) {}</code>	Function that exists, but can never be called. Not very useful. 🦋
<code>_</code>	Unnamed wildcard REF variable binding, e.g., <code> x, _ {}</code> .
<code>let _ = x;</code>	Unnamed assignment is no-op, does not 🟡 move out <code>x</code> or preserve scope!
<code>_ = x;</code>	You can assign <i>anything</i> to <code>_</code> without <code>let</code> , i.e., <code>_ = ignore_error();</code> 1.59+ 🔥
<code>_x</code>	Variable binding explicitly marked as unused.
<code>1_234_567</code>	Numeric separator for visual clarity.
<code>1_u8</code>	Type specifier for numeric literals EX REF (also <code>i8</code> , <code>u16</code> , ...).
<code>0xBEEF, 0o777, 0b1001</code>	Hexadecimal (<code>0x</code>), octal (<code>0o</code>) and binary (<code>0b</code>) integer literals.
<code>r#foo</code>	A raw identifier BK EX for edition compatibility.
<code>x;</code>	Statement REF terminator, c. expressions EX REF

Common Operators

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

Behind the Scenes

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

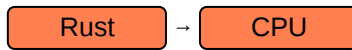


The Abstract Machine

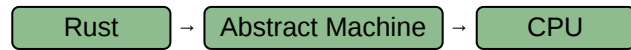


Like `C` and `C++`, Rust is based on an *abstract machine*.

Overview



● Misleading.



Correct.

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

This *abstract machine*

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

Misconceptions

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

Without AM	With AM
<code>0xffff_ffff</code> would make a valid <code>char</code> . ●	AM may exploit ' <i>invalid</i> ' bit patterns to pack unrelated data.
<code>0xff</code> and <code>0xff</code> are same pointer. ●	AM pointers can have ' <i>domain</i> ' attached for optimization.
Any r/w pointer on <code>0xff</code> always fine. ●	AM may issue cache-friendly ops trusting ' <i>no read can happen</i> '.
Reading un-init just gives random value. ●	AM ' <i>knows</i> ' read impossible, may remove all related bitcode.
Data race just gives random value. ●	AM may split R/W, produce <i>impossible</i> value, see above.
Null reference is just <code>0x0</code> in some register. ●	Holding <code>0x0</code> in reference summons Cthulhu.

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

Language Sugar

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

Name	Description
Coercions <small>NOM</small>	Weakens types to match signature, e.g., <code>&mut T</code> to <code>&T</code> ; c. <i>type conversions</i> . ¹
Deref <small>NOM</small> ↗	Derefs <code>x: T</code> until <code>*x</code> , <code>**x</code> , ... compatible with some target <code>S</code> .
Prelude <small>STD</small>	Automatic import of basic items, e.g., <code>Option</code> , <code>drop()</code> , ...
Reborrow	Since <code>x: &mut T</code> can't be copied; moves new <code>&mut *x</code> instead.
Lifetime Elision <small>BK NOM REF</small>	Allows you to write <code>f(x: &T)</code> , instead of <code>f<'a>(x: &'a T)</code> , for brevity.
Method Resolution <small>REF</small>	Derefs or borrow <code>x</code> until <code>x.f()</code> works.
Match Ergonomics <small>RFC</small>	Repeatedly dereferences <code>scrutinee</code> and adds <code>ref</code> and <code>ref mut</code> to bindings.
Rvalue Static Promotion <small>RFC</small>	Makes references to constants <code>'static</code> , e.g., <code>&42</code> , <code>&None</code> , <code>&mut []</code> .
Dual Definitions <small>RFC</small>	Defining one thing (e.g., <code>struct S(u8)</code>) implicitly def. another (e.g., <code>fn S</code>).

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

Memory & Lifetimes

An illustrated guide to moves, references and lifetimes.

Types & Moves



Application Memory [↓](#)

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

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



t

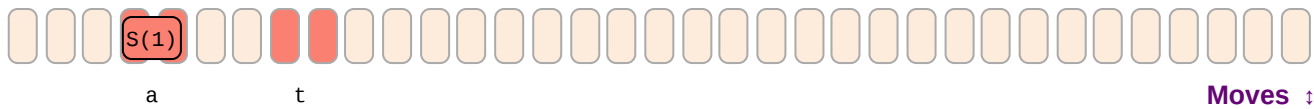
Variables [↓](#)

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

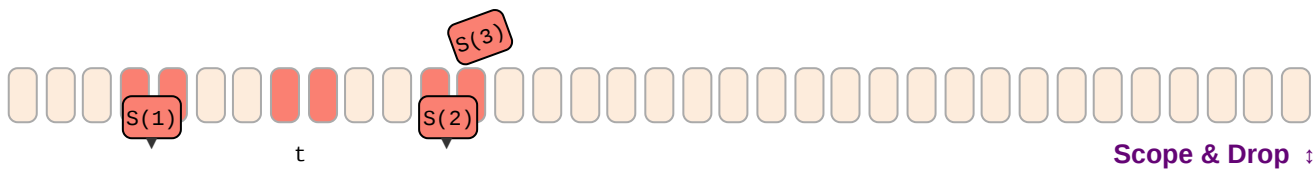
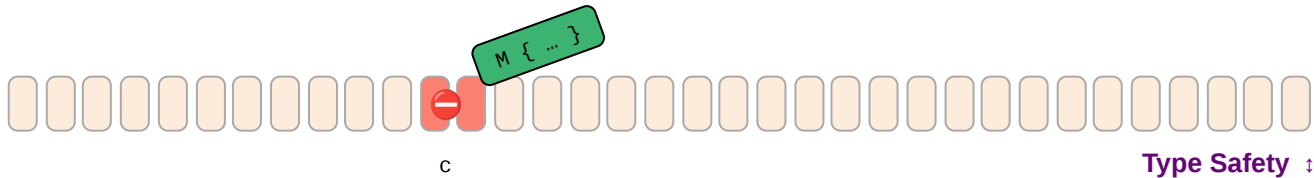
- Reserves memory location with name `t` of type `S` and the value `S(1)` stored inside.

1

↓

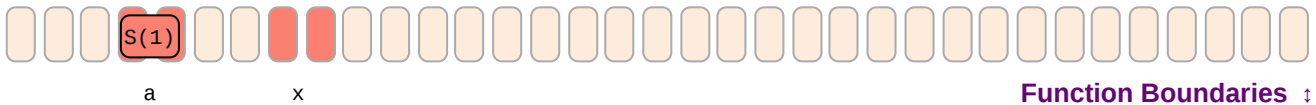


↓



mem :

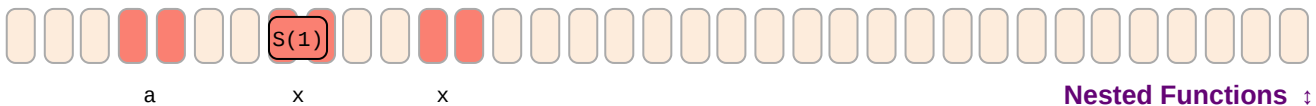
Call Stack



```
fn f(x: S) { ... }  
  
let a = S(1); // <- We are here  
f(a);
```

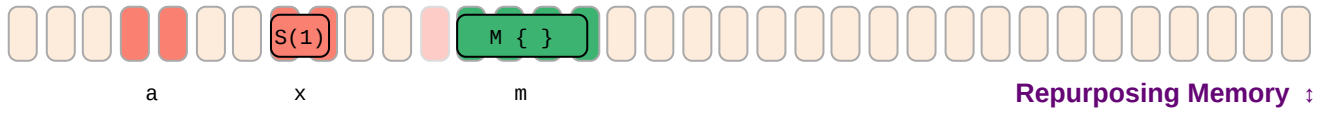
- When a **function is called**, memory for parameters (and return values) are reserved on stack.¹
- Here before `f` is invoked value in `a` is moved to 'agreed upon' location on stack, and during `f` works like 'local variable' `x`.

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



```
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.

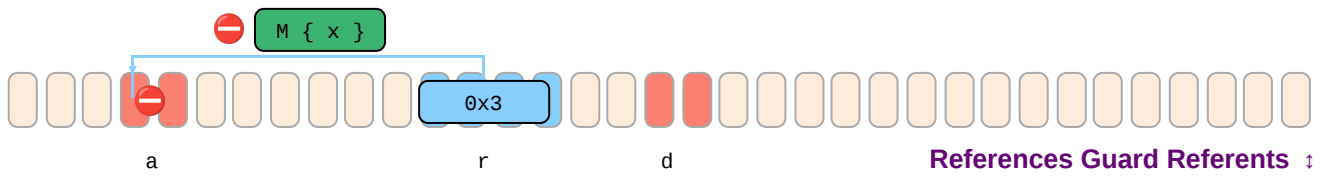
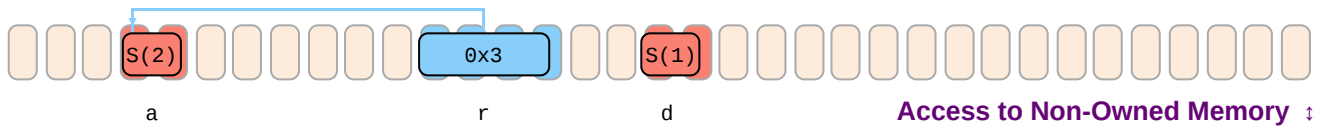
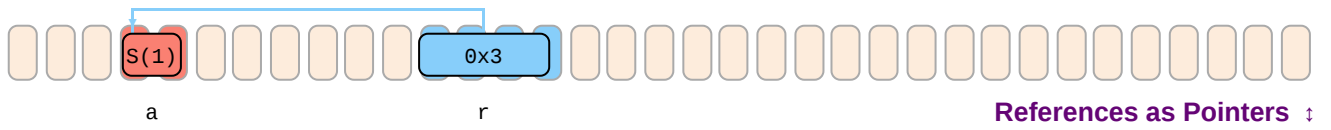


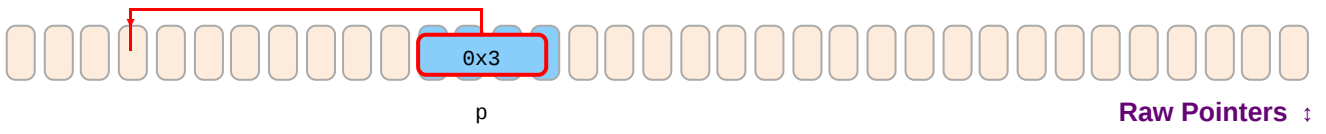
References & Pointers

Lifetime Basics

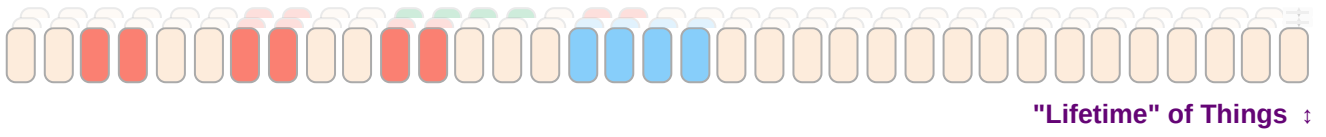
Lifetimes in Functions

Advanced





↓

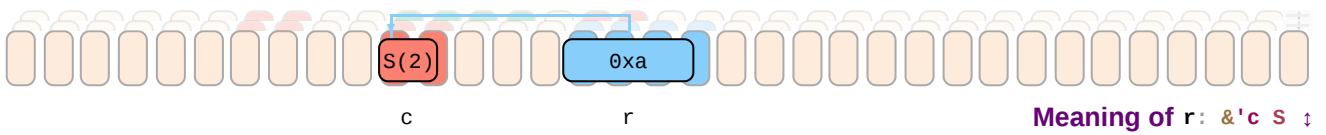


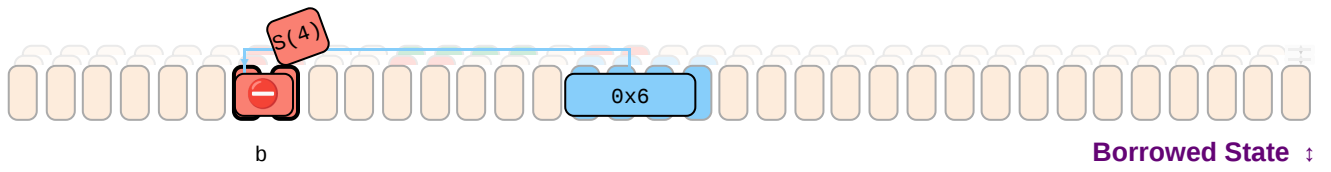
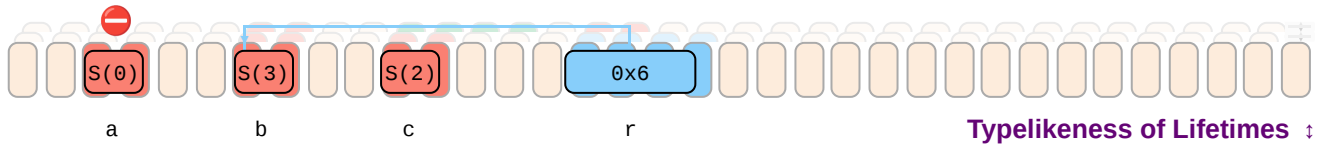
1

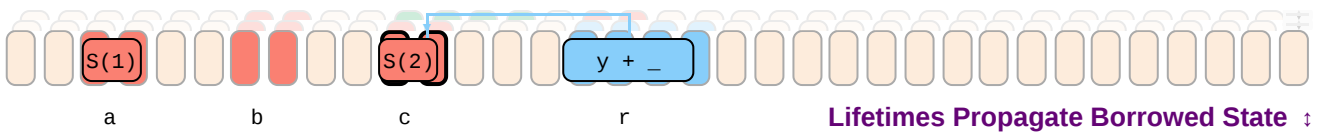
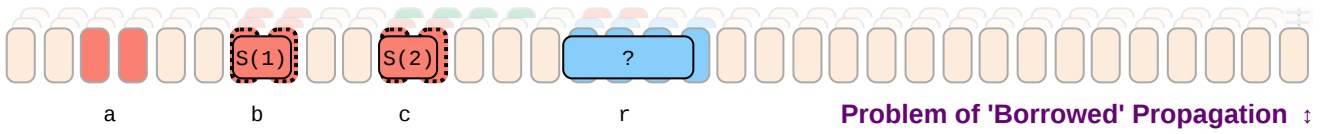
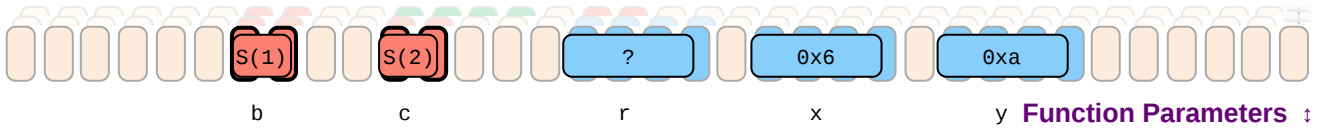
2

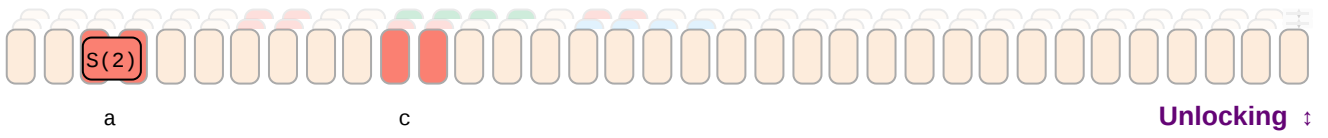
1

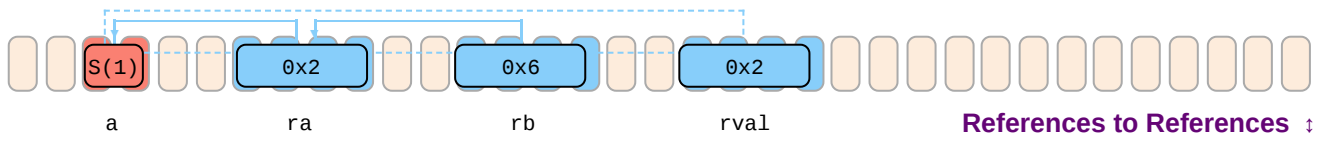
2

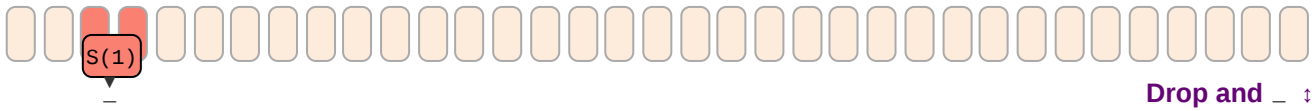












↕ Examples expand by clicking.

Memory Layout

Byte representations of common types.

Basic Types

Essential types built into the core of the language.

Numeric Types ^{REF}

u8, i8



u16, i16



u32, i32



u64, i64



u128, i128



f32



f64



usize, isize



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:



Explanation:

2 E-127
2 -126

2 E-1023
2 -1022

Casting Pitfalls ●

Cast ¹	Gives	Note
<code>3.9_f32 as u8</code>	3	Truncates, consider <code>x.round()</code> first.
<code>314_f32 as u8</code>	255	Takes closest available number.
<code>f32::INFINITY as u8</code>	255	Same, treats <code>INFINITY</code> as <i>really</i> large number.
<code>f32::NAN as u8</code>	0	-
<code>_314 as u8</code>	58	Truncates excess bits.
<code>_200 as i8</code>	56	-
<code>_257 as i8</code>	-1	-

Arithmetic Pitfalls ●

Operation ¹	Gives	Note
<code>200_u8 / 0_u8</code>	Compile error.	-
<code>200_u8 / _0^{d,r}</code>	Panic.	Regular math may panic; here: division by zero.
<code>200_u8 + 200_u8</code>	Compile error.	-
<code>200_u8 + _200^d</code>	Panic.	Consider <code>checked_</code> , <code>wrapping_</code> , ... instead. STD
<code>200_u8 + _200^r</code>	144	In release mode this will overflow.
<code>1_u8 / 2_u8</code>	0	Other integer division truncates.
<code>0.8_f32 + 0.1_f32</code>	0.90000004	-

¹ Expression `_100` means anything that might contain the value `100`, e.g., `100_132`, but is opaque to compiler.

^d Debug build.

^r Release build.

Textual Types ^{REF}

char



Any Unicode scalar.

str



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

Basics

Type

`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**.

Description

Usage

Chars

```
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

```
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



Custom Types

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

T	T: ?Sized	[T; n]	[T]
Sized ¹	Maybe DST ¹	Fixed array of n elements.	Slice type of unknown-many elements. Neither Sized (nor carries len information), and most often lives behind reference as &[T] . ¹

struct S;

Zero-Sized ¹

(A, B, C)

or maybe

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

struct S { b: B, c: C }

or maybe

Compiler may also add padding.

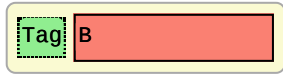
Also note, two types **A(X, Y)** and **B(X, Y)** with exactly the same fields can still have differing layout; never **transmute()** ^{STD} without representation guarantees.

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

enum E { A, B, C }



exclusive or

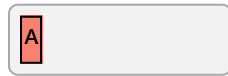


exclusive or

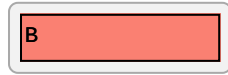


Safely holds A or B or C, also called 'tagged union', though compiler may squeeze tag into 'unused' bits.

union { ... }



unsafe or



unsafe or

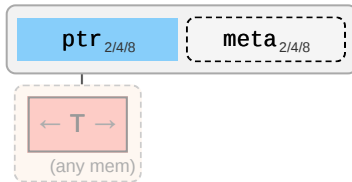


Can unsafely reinterpret memory. Result might be undefined.

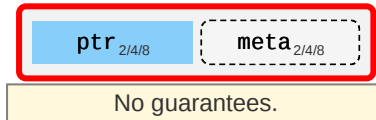
References & Pointers

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

`&'a T`



`*const T`

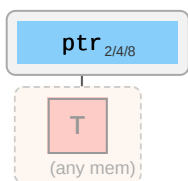


Must target some valid `t` of `T`, and any such target must exist for at least `'a`.

Pointer Meta

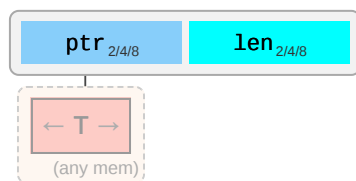
Many reference and pointer types can carry an extra field, **pointer metadata**.^{STD} It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

`&'a T`



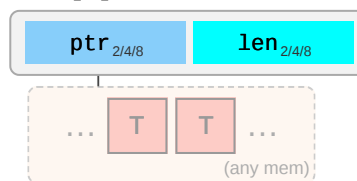
No meta for sized target. (pointer is thin).

`&'a T`



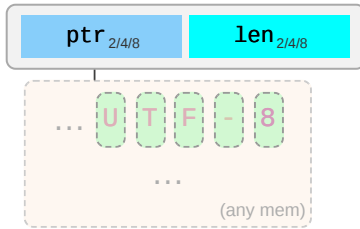
If `T` is a DST `struct` such as `S { x: [u8] }` meta field `len` is count of dyn. sized content.

`&'a [T]`



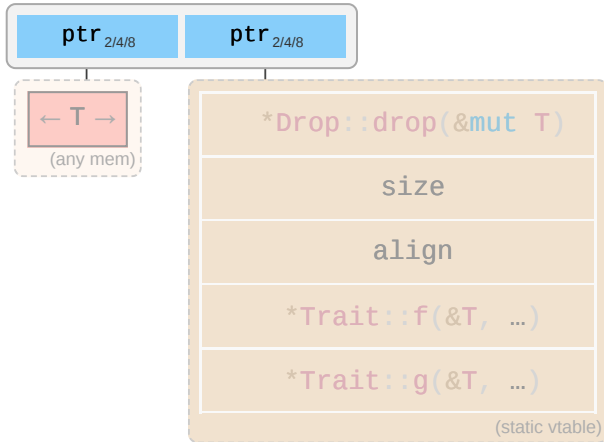
Regular **slice reference** (i.e., the reference type of a slice type `[T]`)¹ often seen as `&[T]` if `'a` elided.

&'a str



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

&'a dyn Trait



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

Closures

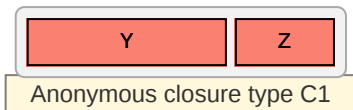
Ad-hoc functions with an automatically managed data block **capturing** ^{REF.1} environment where closure was defined. For example, if you had:

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

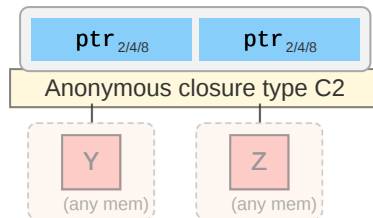
with_closure(move |x| x + y.f() + z); // y and z are moved into closure instance (of type C1)
with_closure(|x| x + y.f() + z); // y and z are pointed at from closure instance (of type C2)
```

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

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



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



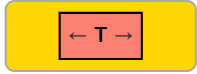
Also produces anonymous `fn` such as `fC1(C1, X)` or `fC2(&C2, X)`. Details depend on which `FnOnce`, `FnMut`, `Fn` ... is supported, based on properties of captured types.

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

Standard Library Types

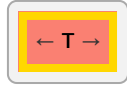
Rust's standard library combines the above primitive types into useful types with special semantics, e.g.:

UnsafeCell<T>



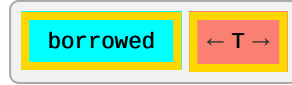
Magic type allowing aliased mutability.

Cell<T>



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

RefCell<T>



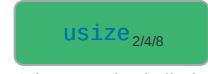
Also support dynamic borrowing of `T`. Like `cell` this is `Send`, but not `Sync`.

ManuallyDrop<T>



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

AtomicUsize



Other atomic similarly.

Option<T>



or

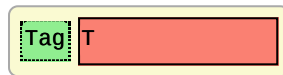


Tag may be omitted for certain `T`, e.g., `NonNull`.

Result<T, E>



or

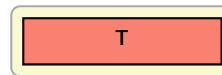


Either some error `E` or value of `T`.

MaybeUninit<T>^{STD}



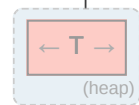
unsafe or



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

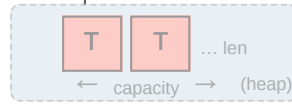
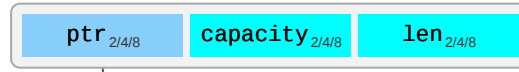
Order-Preserving Collections

Box<T>



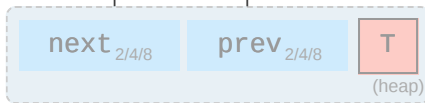
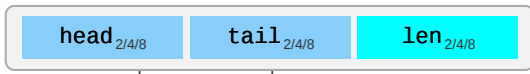
For some `T` stack proxy may carry meta' (e.g., `Box<[T]>`).

Vec<T>



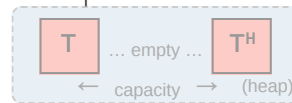
Regular *growable array* vector of single type.

LinkedList<T>



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

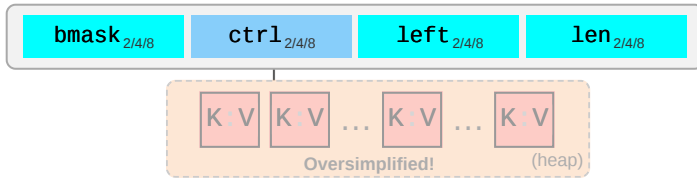
VecDeque<T>



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

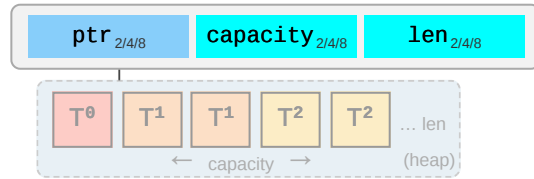
Other Collections

HashMap<K, V>



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

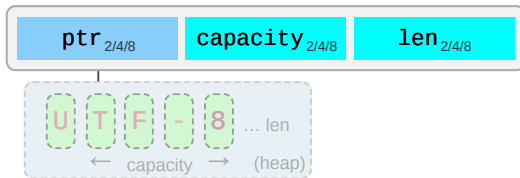
BinaryHeap<T>



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

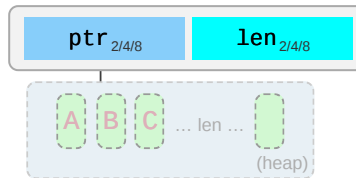
Owned Strings

String



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

CString



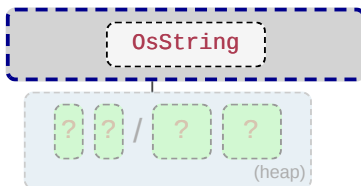
NUL-terminated but w/o NUL in middle.

OsString



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

PathBuf

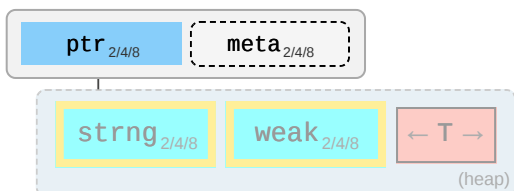


Encapsulates how operating system represents paths.

Shared Ownership

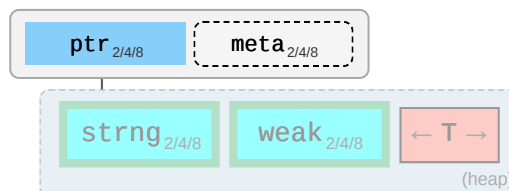
If the type does not contain a `Cell` for `T`, these are often combined with one of the `Cell` types above to allow shared de-facto mutability.

Rc<T>



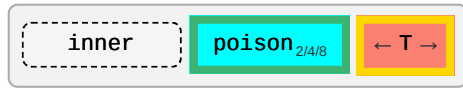
Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell` to allow mutation. Is neither `Send` nor `Sync`.

Arc<T>



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

Mutex<T> / RwLock<T>



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

Standard Library

One-Liners

Snippets that are common, but still easy to forget. See [Rust Cookbook](#) for more.

Strings

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

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

² Requires `regex` crate.

I/O

Intent	Snippet
Create a new file	<code>File::create(PATH)?</code>
Same, via <code>OpenOptions</code>	<code>OpenOptions::new().create(true).write(true).truncate(true).open(PATH)?</code>

Macros

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

Esoterics

Thread Safety

Examples	Send*	!Send
Sync*	Most types ... <code>Arc<T></code> ^{1,2} , <code>Mutex<T></code> ²	<code>MutexGuard<T></code> ¹ , <code>RwLockReadGuard<T></code> ¹
!Sync	<code>Cell<T></code> ² , <code>RefCell<T></code> ²	<code>Rc<T></code> , <code>&dyn Trait</code> , <code>*const T</code> ³ , <code>*mut T</code> ³

* 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`.

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

Iterators

Usage

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`.

Iterator Compatibility

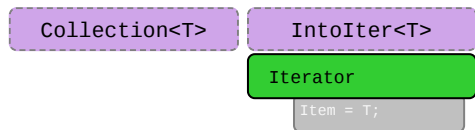
- `let c = other_iter.collect::<C<_>>()` — Collect foreign(!) iterable into your `C`.

Creating Iterators

Essentials

Let's assume you have a `struct Collection<T> {}`.

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



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

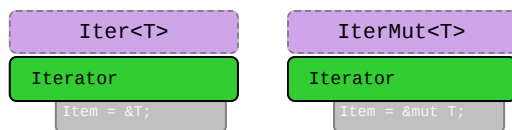
Shared & Mutable Iterators

These are essential if you want your collection to be useful when borrowed.

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

In addition, you might want to add convenience methods:

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

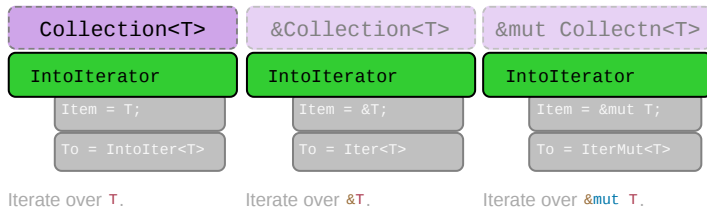


Providing Iterators

Native Loop Support

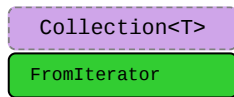
Many users would expect your collection to *just work* in `for` loops:

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



STD

rd



Number Conversions

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

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

¹ If type true subset `from()` works directly, e.g., `u32::from(my_u8)`.

² Truncating (`11.9_f32 as u8` gives `11`) and saturating (`1024_f32 as u8` gives `255`); c. below.

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

Also see [Casting-](#) and [Arithmetic Pitfalls](#) ¹ for more things that can go wrong working with numbers.

String Conversions

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

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

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 ⁱ	OsString::from(x) ⁱ

PathBuf

If you have x of type ...

Use this ...

String	PathBuf::from(x) ⁱ
CString	PathBuf::from(x.to_str())?
OsString	PathBuf::from(x) ⁱ
PathBuf	x
Vec<u8> ¹	?
&str	PathBuf::from(x) ⁱ
&CStr	PathBuf::from(x.to_str())?
&OsStr	PathBuf::from(x) ⁱ
&Path	PathBuf::from(x) ⁱ
&[u8] ¹	?

Vec<u8>

If you have x of type ...

Use this ...

String	x.into_bytes()
CString	x.into_bytes()
OsString	?
PathBuf	?
Vec<u8> ¹	x
&str	Vec::from(x.as_bytes())
&CStr	Vec::from(x.to_bytes_with_nul())
&OsStr	?
&Path	?
&[u8] ¹	x.to_vec()

&str

If you have x of type ...

Use this ...

String	x.as_str()
CString	x.to_str()?

1

std::

1

std::

&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()

&Path

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

&[u8]

If you have x of type ...	Use this ...
String	x.as_bytes()
CString	x.as_bytes()
OsString	?
PathBuf	?
Vec<u8> ¹	&x
&str	x.as_bytes()
&CStr	x.to_bytes_with_nul()
&OsStr	x.as_bytes() ²
&Path	?
&[u8] ¹	x

Other

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

¹ Short form `x.into()` possible if type can be inferred.

[†] 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
<code>format!(fmt)</code>	<code>String</code>	Bread-and-butter "to <code>String</code> " converter.
<code>print!(fmt)</code>	Console	Writes to standard output.
<code>println!(fmt)</code>	Console	Writes to standard output.
<code>eprint!(fmt)</code>	Console	Writes to standard error.
<code>eprintln!(fmt)</code>	Console	Writes to standard error.
<code>write!(dst, fmt)</code>	Buffer	Don't forget to also <code>use std::io::Write;</code>
<code>writeln!(dst, fmt)</code>	Buffer	Don't forget to also <code>use std::io::Write;</code>

Method	Notes
<code>x.to_string()</code> ^{STD}	Produces <code>String</code> , implemented for any <code>Display</code> 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
<code>String</code>	<code>Debug</code> , <code>Display</code>
<code>CString</code>	<code>Debug</code>

Formatting

Each argument designator in format macro is either empty {}, {argument}, or follows a basic [syntax](#):

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

Element	Meaning
argument	Number (0, 1, ...), variable ²¹ or name, ¹⁸ e.g., <code>print!("{x}")</code> .
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 <code>Debug^{STD}</code> formatter <code>?</code> or prefix hex with <code>0x</code> .
width	Minimum width (≥ 0), padding with <code>fill</code> (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	<code>Debug^{STD}</code> (?) formatting, hex (x), binary (b), octal (o), pointer (p), exp (e) ... see more .

Format Example	Explanation
{}	Print the next argument using <code>Display^{STD}</code>

'21

STD

rd

STD

rd

STD

'15 

'21








STD

'21

Tooling

Project Anatomy

Basic project layout, and common files and folders, as used by cargo. ¹

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 <code>my_crate::f()</code> starts.
 src/bin/	Place for additional binaries, even in library projects.
extra.rs	Additional binary, run with cargo run --bin extra .
 tests/	Integration tests go here, invoked via cargo test . Unit tests often stay in <code>src/</code> file.
.rustfmt.toml	In case you want to customize how cargo fmt works.
.clippy.toml	Special configuration for certain clippy lints , utilized via cargo clippy
build.rs	Pre-build script , 🔗 useful when compiling C / FFI, ...
Cargo.toml	Main project manifest , 🔗 Defines dependencies, artifacts ...
Cargo.lock	Dependency details for reproducible builds; add to <code>git</code> for apps, not for libs.
rust-toolchain.toml	Define toolchain override 🔗 (channel, components, targets) for this project.

¹ On stable consider [Criterion](#).

Minimal examples for various entry points might look like:

Applications

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

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

Libraries

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

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

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

Unit Tests

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

fn f() -> u32 { 0 }

#[cfg(test)]
mod test {
    use super::f; // Need to import items from parent module. Has
                  // access to non-public members.

    #[test]
    fn ff() {
        assert_eq!(f(), 0);
    }
}
```

Integration Tests

Benchmarks^{beta}

Build Scripts

my_crate::

test

test::

my_crate::

env::

*

Proc Macros

```
proc_macro
```

```
proc_macro::
```

Module trees and imports:

Module Trees

Modules [BK](#) [EX](#) [REF](#) and **source files** work as follows:

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

Actual **module definitions** work as follows:

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

Namespaces

Rust has three kinds of **namespaces**:

Namespace Types	Namespace Functions	Namespace Macros
<code>mod X {}</code>	<code>fn X() {}</code>	<code>macro_rules! X { ... }</code>
<code>X (crate)</code>	<code>const X: u8 = 1;</code>	
<code>trait X {}</code>	<code>static X: u8 = 1;</code>	
<code>enum X {}</code>		

← 1 →
 ← 2 →

1
 2

my_mod :

Cargo

Commands and tools that are good to know.

Command	Description
<code>cargo init</code>	Create a new project for the latest edition.
<code>cargo build</code>	Build the project in debug mode (<code>--release</code> for all optimization).
<code>cargo check</code>	Check if project would compile (much faster).
<code>cargo test</code>	Run tests for the project.
<code>cargo doc --open</code>	Locally generate documentation for your code and dependencies.
<code>cargo run</code>	Run your project, if a binary is produced (main.rs).
<code>cargo run --bin b</code>	Run binary <code>b</code> . Unifies features with other dependents (can be confusing).
<code>cargo run -p w</code>	Run main of sub-workspace <code>w</code> . Treats features more as you would expect.
<code>cargo ... --timings</code>	Show what crates caused your build to take so long. 🔥
<code>cargo tree</code>	Show dependency graph.
<code>cargo +{nightly, stable} ...</code>	Use given toolchain for command, e.g., for 'nightly only' tools.
<code>cargo +nightly ...</code>	Some nightly-only commands (substitute ... with command below)
<code>rustc -- -Zunpretty=expanded</code>	Show expanded macros. 🦄
<code>rustup doc</code>	Open offline Rust documentation (incl. the books), good on a plane!

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

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

Tool	Description
<code>cargo clippy</code>	Additional (lints) catching common API misuses and idiomatic code. 🔗
<code>cargo fmt</code>	Automatic code formatter (<code>rustup component add rustfmt</code>). 🔗

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

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

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

- Compile with `cargo build --target=X`

Tooling Directives

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

Macros

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

Within Macros	Explanation
<code>\$x:ty</code>	Macro capture (here a type).
<code>\$x:item</code>	An item, like a function, struct, module, etc.

std::mem::

?

Documentation

Inside a **doc comment** [BK](#) [EX](#) [REF](#) these work:

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

`#![global]`

Attributes affecting the whole crate or app:

Opt-Out's	On	Explanation
<code>#![no_std]</code>	<code>C</code>	Don't (automatically) import <code>std</code> ^{STD} ; use <code>core</code> ^{STD} instead. REF

STD

REF

REF



REF

? REF

REF

REF

REF

REF

STD

REF

#[code]

Attributes primarily governing emitted code:

Developer UX	On	Explanation
<code>#[non_exhaustive]</code>	T	Future-proof <code>struct</code> or <code>enum</code> ; hint it may grow in future. REF
<code>#[path = "x.rs"]</code>	M	Get module from non-standard file. REF

Codegen	On	Explanation
<code>#[inline]</code>	F	Nicely suggest compiler should inline function at call sites. REF
<code>#[inline(always)]</code>	F	Emphatically threaten compiler to inline call, or else. REF
<code>#[inline(never)]</code>	F	Instruct compiler to feel disappointed if it still inlines the function. REF
<code>#[cold]</code>	F	Hint that function probably isn't going to be called. REF
<code>#[target_feature(enable="x")]</code>	F	Enable CPU feature (e.g., <code>avx2</code>) for code of <code>unsafe fn</code> . REF
<code>#[track_caller]</code>	F	Allows <code>fn</code> to find <code>caller</code> ^{STD} for better panic messages. REF
<code>#[repr(X)]</code> ¹	T	Use another representation instead of the default <code>rust</code> REF one:
<code>#[repr(C)]</code>	T	Use a C-compatible (f. FFI), predictable (f. <code>transmute</code>) layout. REF
<code>#[repr(C, u8)]</code>	<code>enum</code>	Give <code>enum</code> discriminant the specified type. REF
<code>#[repr(transparent)]</code>	T	Give single-element type same layout as contained field. REF

REF

REF

1

REF

REF

REF

REF

REF

REF

REF

#[quality]

Attributes used by Rust tools to improve code quality:

Code Patterns	On	Explanation
<code>#[allow(x)]</code>	*	Instruct rustc / clippy to ... ignore class <code>x</code> of possible issues. REF
<code>#[warn(x)]</code> ¹	*	... emit a warning, mixes well with clippy lints. 🔥 REF
<code>#[deny(x)]</code> ¹	*	... fail compilation. REF
<code>#[forbid(x)]</code> ¹	*	... fail compilation and prevent subsequent allow overrides. REF
<code>#[deprecated = "msg"]</code>	*	Let your users know you made a design mistake. REF
<code>#[must_use = "msg"]</code>	FTX	Makes compiler check return value is <i>processed</i> by caller. 🔥 REF

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

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

Formatting	On	Explanation
<code>#[rustfmt::skip]</code>	*	Prevent cargo fmt from cleaning up item. 🔗
<code>#![rustfmt::skip::macros(x)]</code>	CM	... from cleaning up macro <code>x</code> . 🔗

#[macros]

Attributes related to the creation and use of macros:

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

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

Derives	On	Explanation
<code>#[derive(X)]</code>	T	Let some proc macro provide a goodish <code>impl</code> of <code>trait X</code> . REF

#[cfg]

Attributes governing conditional compilation:

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

REF

REF

REF

REF

REF

REF

REF

REF

REF

?

REF

🔥 REF

🔥 REF

build.rs

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

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

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

Output String	Explanation 🔗
<code>cargo:rerun-if-changed=PATH</code>	(Only) run this <code>build.rs</code> again if <code>PATH</code> changed.



?

For the *On* column in attributes:

C means on crate level (usually given as `#![my_attr]` in the top level file).

M means on modules.

F means on functions.

S means on static.

T means on types.

X means something special.

! means on macros.

***** means on almost any item.

Working with Types

Types, Traits, Generics

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

Types & Traits

Types

u8

String

Device

- Set of values with given semantics, layout, ...

Type	Values
u8	{ 0 _{u8} , 1 _{u8} , ..., 255 _{u8} }
char	{ 'a', 'b', ... '🐛' }
struct S(u8, char)	{ (0 _{u8} , 'a'), ... (255 _{u8} , '🐛') }

Sample types and sample values.

Type Equivalence and Conversions

u8

&u8

&mut u8

[u8; 1]

String

- It may be obvious but `u8`, `&u8`, `&mut u8`, are entirely different from each other

u8 u16

u8 u8 u8
u16 u16 u16
&u8 &u8
&mut u8 &mut u8

1

f

2

1



2

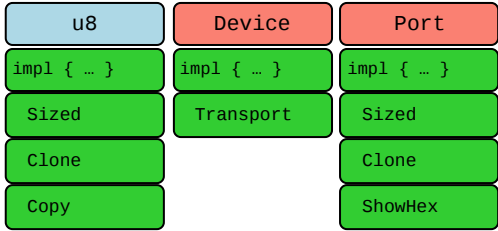
u8	String	Port
impl { ... }	impl { ... }	impl { ... }

REF

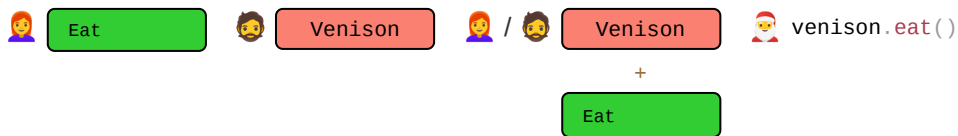
Copy	Clone	Sized	ShowHex
------	-------	-------	---------

Copy

Sized



🎅 venison.eat()



*

food::
tasks::

?

Generics

Type Constructors — `Vec<>`

`Vec<u8>`

`Vec<char>`

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

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

Types vs type constructors.

`Vec<>`

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

Generic Parameters — `<T>`

`Vec<T>`

`[T; 128]`

`&T`

`&mut T`

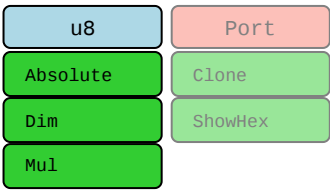
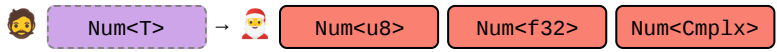
`S<T>`

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

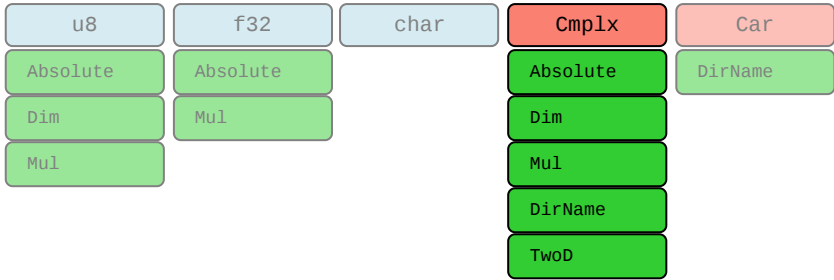
Type Constructor	Produces Family
<code>struct Vec<T> {}</code>	<code>Vec<u8></code> , <code>Vec<f32></code> , <code>Vec<Vec<u8>></code> , ...
<code>[T; 128]</code>	<code>[u8; 128]</code> , <code>[char; 128]</code> , <code>[Port; 128]</code> ...
<code>&T</code>	<code>&u8</code> , <code>&u16</code> , 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

Type vs type constructors.

[T; n] S<const N>

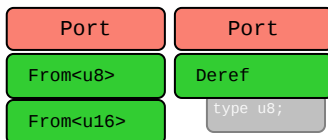


?



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

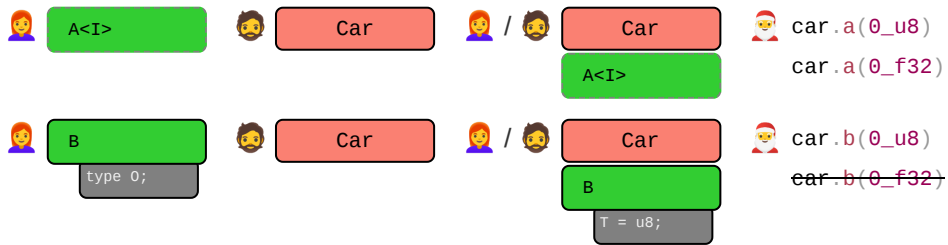
Notice how some traits can be "attached" multiple times, but others just once?

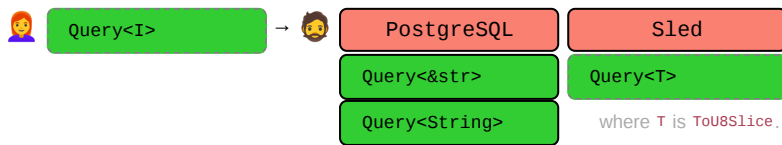
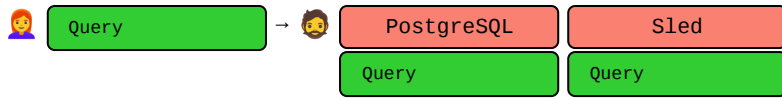


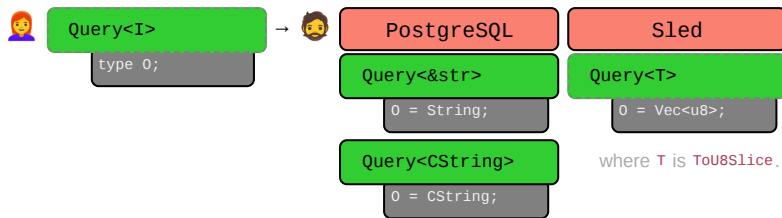
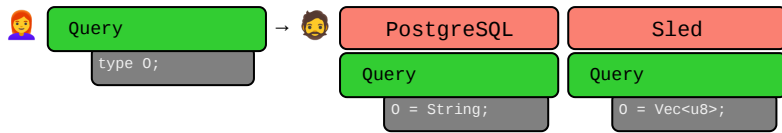
Why is that?

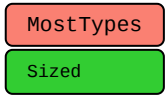
- Traits themselves can be generic over two **kinds of parameters**:
 - `trait From<I> {}`
 - `trait Deref { type O; }`
- Remember we said traits are "membership lists" for types and called the list `Self`?
- Turns out, parameters `I` (for **input**) and `O` (for **output**) are just more *columns* to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





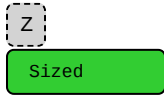






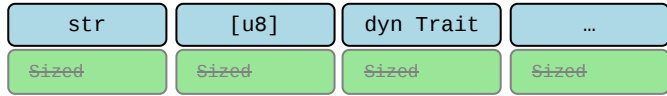
Normal types.

vs.



Zero sized.

vs.



Dynamically sized.

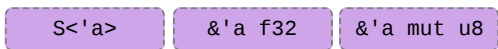
STD

BK NOM REF

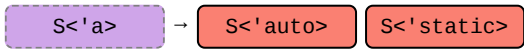
NOM

REF

↑



*



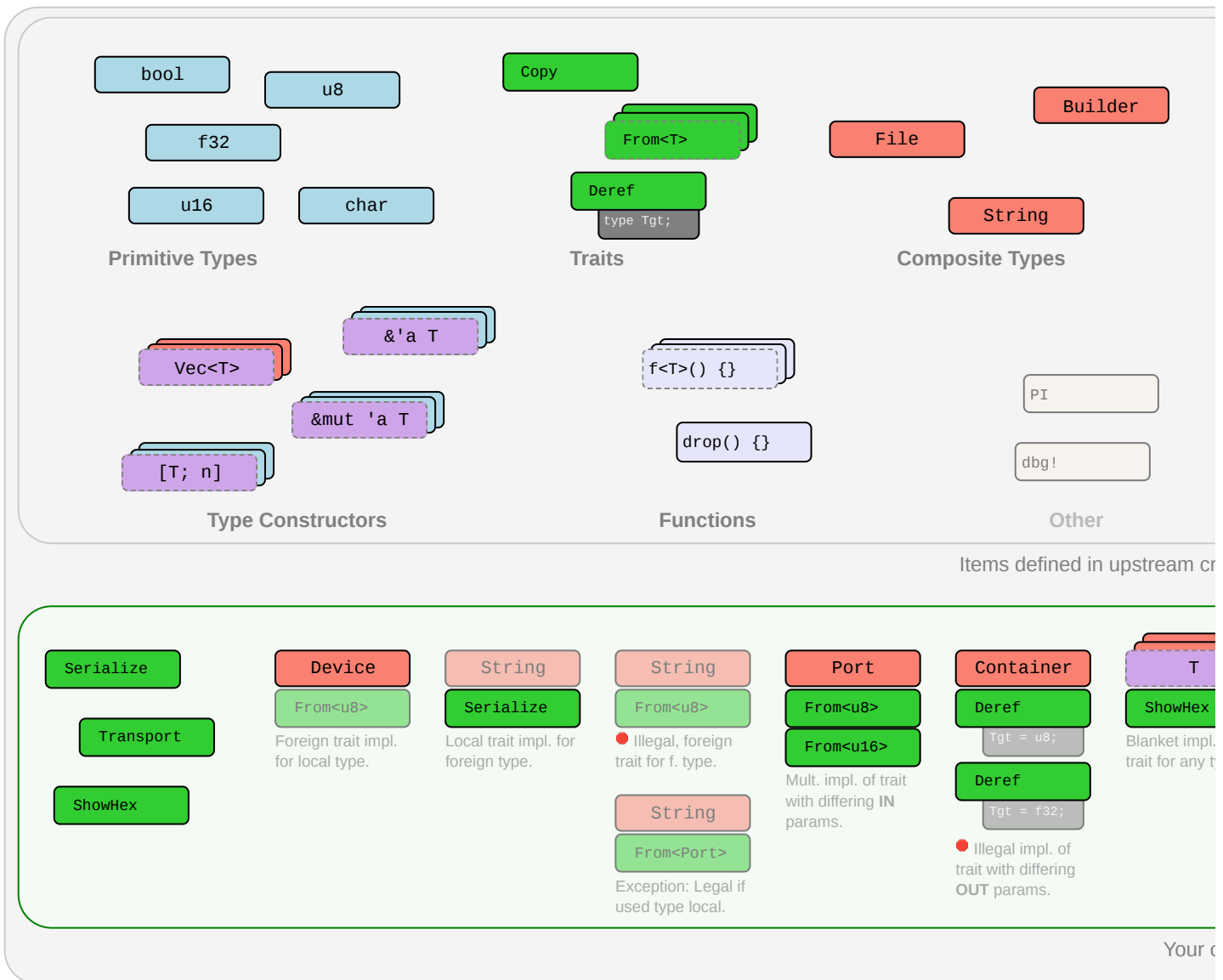
*



Examples expand by clicking.

Foreign Types and Traits

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



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

Type Conversions

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

Intro

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

Method	Explanation
Identity	Trivial case, B is exactly A .
Computation	Create and manipulate instance of B by writing code transforming data.
Casts	On-demand conversion between types where caution is advised.

Computation (Traits)

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

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

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

Casts

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

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

A	B	Example	Explanation
<code>Ptr</code>	<code>Ptr</code>	<code>device_ptr as *const u8</code>	If <code>*A</code> , <code>*B</code> are <code>Sized</code> .
<code>Ptr</code>	<code>Integer</code>	<code>device_ptr as usize</code>	
<code>Integer</code>	<code>Ptr</code>	<code>my_usize as *const Device</code>	
<code>Number</code>	<code>Number</code>	<code>my_u8 as u16</code>	Often surprising behavior. [†]
<code>enum w/o fields</code>	<code>Integer</code>	<code>E::A as u8</code>	
<code>bool</code>	<code>Integer</code>	<code>true as u8</code>	
<code>char</code>	<code>Integer</code>	<code>'A' as u8</code>	



Coercions

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

Automatically **weaken** type **A** to **B**; types can be *substantially*¹ different. ^{NOM}

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

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

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

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

Subtyping

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

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

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

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

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

Variance

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

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

- A longer lifetime 'a that outlives a shorter 'b is a subtype of 'b.
- Implies 'static is subtype of all other lifetimes 'a.

Coding Guides

Idiomatic Rust

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

Idiom	Code
Think in Expressions	<pre>y = if x { a } else { b }; y = loop { break 5 }; fn f() -> u32 { 0 }</pre>
Think in Iterators	<pre>(1..10).map(f).collect() names.iter().filter(x x.starts_with("A"))</pre>
Handle Absence with ?	<pre>y = try_something()?; get_option()?.run()?</pre>
Use Strong Types	<pre>enum E { Invalid, Valid { ... } } over ERROR_INVALID = -1 enum E { Visible, Hidden } over visible: bool struct Charge(f32) over f32</pre>
Illegal State: Impossible	<pre>my_lock.write()?.guaranteed_at_compile_time_to_be_locked = 10; thread::scope(s { /* Threads can't exist longer than scope() */ });</pre>
Provide Builders	<pre>Car::new("Model T").hp(20).build();</pre>

Idiom	Code
Don't Panic	<p>Panics are <i>not</i> exceptions, they suggest immediate process abortion!</p> <p>Only panic on programming error; use <code>Option<T>STD</code> or <code>Result<T,E>STD</code> otherwise.</p> <p>If clearly user requested, e.g., calling <code>obtain()</code> vs. <code>try_obtain()</code>, panic ok too.</p>
Generics in Moderation	<p>A simple <code><T: Bound></code> (e.g., <code>AsRef<Path></code>) can make your APIs nicer to use.</p> <p>Complex bounds make it impossible to follow. If in doubt don't be creative with <code>g</code>.</p>
Split Implementations	<p>Generics like <code>Point<T></code> can have separate <code>impl</code> per <code>T</code> for some specialization.</p> <pre>impl<T> Point<T> { /* Add common methods here */ } impl Point<f32> { /* Add methods only relevant for Point<f32> */ }</pre>
Unsafe	Avoid <code>unsafe {}</code> , ¹ often safer, faster solution without it.
Implement Traits	<code>#[derive(Debug, Copy, ...)]</code> and custom <code>impl</code> where needed.
Tooling	<p>Run <code>clippy</code> regularly to significantly improve your code quality. 🔥</p> <p>Format your code with <code>rustfmt</code> for consistency. 🔥</p> <p>Add unit tests ^{BK} (<code>#[test]</code>) to ensure your code works.</p> <p>Add doc tests ^{BK} (<code>`` my_api::f() ``</code>) to ensure docs match code.</p>
Documentation	<p>Annotate your APIs with doc comments that can show up on docs.rs.</p> <p>Don't forget to include a summary sentence and the Examples heading.</p> <p>If applicable: Panics, Errors, Safety, Abort and Undefined Behavior.</p>

🔥 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
<code>async</code>	Anything declared <code>async</code> always returns an <code>impl Future<Output=_,>.STD</code>
<code>async fn f() {}</code>	Function <code>f</code> returns an <code>impl Future<Output=()></code> .
<code>async fn f() -> S {}</code>	Function <code>f</code> returns an <code>impl Future<Output=S></code> .
<code>async { x }</code>	Transforms <code>{ x }</code> into an <code>impl Future<Output=X></code> .
<code>let sm = f();</code>	Calling <code>f()</code> that is <code>async</code> will not execute <code>f</code> , but produce state machine <code>sm</code> . ^{1 2}
<code>sm = async { g() };</code>	Likewise, does not execute the <code>{ g() }</code> block; produces state machine.
<code>runtime.block_on(sm);</code>	Outside an <code>async {}</code> , schedules <code>sm</code> to actually run. Would execute <code>g()</code> . ^{3 4}
<code>sm.await</code>	Inside an <code>async {}</code> , run <code>sm</code> until complete. Yield to runtime if <code>sm</code> not ready.

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

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

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

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

Execution Flow

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

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

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

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

Caveats

With the execution flow in mind, some considerations when writing code inside an `async` construct:

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

¹ Here we assume `s` is any non-local that could temporarily be put into an invalid state; `TL` is any thread local storage, and that the `async {}` containing the code is written without assuming executor specifics.

² Since `Drop` is run in any case when `Future` is dropped, consider using drop guard that cleans up / fixes application state if it has to be left in bad condition across `.await` points.

Closures in APIs

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

From a call site perspective that means:

Signature	Function <code>g</code> can call ...	Function <code>g</code> accepts ...
<code>g<F: FnOnce()>(f: F)</code>	... <code>f()</code> once.	<code>Fn, FnMut, FnOnce</code>

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

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
<code> { moved_s; }</code>	<code>FnOnce</code>	Caller must give up ownership of <code>moved_s</code> .
<code> { &mut s; }</code>	<code>FnOnce, FnMut</code>	Allows <code>g()</code> to change caller's local state <code>s</code> .
<code> { &s; }</code>	<code>FnOnce, FnMut, Fn</code>	May not mutate state; but can share and reuse <code>s</code> .

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

Unsafe, Unsound, Undefined

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

Safe Code

Safe Code

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

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

Unsafe Code

Unsafe Code

- Code marked `unsafe` has special permissions, e.g., to deref raw pointers, or invoke other `unsafe` functions.
- Along come special **promises the author *must* uphold to the compiler**, and the compiler *will* trust you.

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.
```

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

Adversarial Code

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

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

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

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

Implications

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

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

API Stability

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

Crates

- Making a crate that previously compiled for *stable* require *nightly*.
- Altering use of Cargo features (e.g., adding or removing features).

Modules

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

Structs

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

Enums

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

Traits

- Adding a non-defaulted item, breaks all existing `impl T for S {}`.
- Any non-trivial change to item signatures, will affect either consumers or implementors.
- Adding a defaulted item; might cause dispatch ambiguity with other existing trait.
- Adding a defaulted type parameter.

Traits

- Implementing any "fundamental" trait, as *not* implementing a fundamental trait already was a promise.
- Implementing any non-fundamental trait; might also cause dispatch ambiguity.

Inherent Implementations

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

Signatures in Type Definitions

- Tightening bounds (e.g., `<T>` to `<T: Clone>`).
- Loosening bounds.
- Adding defaulted type parameters.
- Generalizing to generics.

Signatures in Functions

- Adding / removing arguments.
- Introducing a new type parameter.
- Generalizing to generics.

Behavioral Changes

- / ● Changing semantics might not cause compiler errors, but might make clients do wrong thing.

