



# Rust Language Cheat Sheet

19. September 2021

Contains clickable links to [The Book](#) <sup>BK</sup>, [Rust by Example](#) <sup>EX</sup>, [Std Docs](#) <sup>STD</sup>, [Nomicon](#) <sup>NOM</sup>, [Reference](#) <sup>REF</sup>.



## Data Structures

Data types and memory locations defined via keywords.

Example	Explanation
<code>struct S {}</code>	Define a <b>struct</b> <sup>BK EX STD REF</sup> 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 <b>zero sized</b> <sup>NOM</sup> unit struct. Occupies no space, optimized away.
<code>enum E {}</code>	Define an <b>enum</b> , <sup>BK EX REF</sup> c. <a href="#">algebraic data types</a> , <a href="#">tagged unions</a> .
<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, e.g., for FFI.
<code>union U {}</code>	Unsafe C-like <b>union</b> <sup>REF</sup> for FFI compatibility.
<code>static X: T = T();</code>	<b>Global variable</b> <sup>BK EX REF</sup> with <code>'static</code> lifetime, single memory location.
<code>const X: T = T();</code>	Defines <b>constant</b> , <sup>BK EX REF</sup> copied into a temporary when used.
<code>let x: T;</code>	Allocate <code>T</code> bytes on stack <sup>1</sup> bound as <code>x</code> . Assignable once, not mutable.
<code>let mut x: T;</code>	Like <code>let</code> , but allow for <b>mutability</b> <sup>BK EX</sup> and mutable borrow. <sup>2</sup>
<code>x = y;</code>	Moves <code>y</code> to <code>x</code> , invalidating <code>y</code> if <code>T</code> is not <b>Copy</b> , <sup>STD</sup> and copying <code>y</code> otherwise.

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

<sup>2</sup> Technically *mutable* and *immutable* are misnomer. Immutable binding or shared reference may still contain `Cell` <sup>STD</sup>, 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</code> 'ed <code>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 <a href="#">Default</a> .
<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</code> 'ed <code>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</code> 'ed <code>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 <b>unit</b> . <sup>STD</sup>
<code>(x)</code>	Parenthesized expression.

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

## References & Pointers

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

Example	Explanation
<code>&amp;S</code>	Shared <b>reference</b> <sup>BK STD NOM REF</sup> (space for holding <i>any</i> <code>&amp;S</code> ).
<code>&amp;[S]</code>	Special slice reference that contains (address, length).
<code>&amp;str</code>	Special string slice reference that contains (address, length).
<code>&amp;mut S</code>	Exclusive reference to allow mutability (also <code>&amp;mut [S]</code> , <code>&amp;mut dyn S</code> , ...).
<code>&amp;dyn T</code>	Special <b>trait object</b> <sup>BK</sup> reference that contains (address, vtable).
<code>&amp;s</code>	Shared <b>borrow</b> <sup>BK EX STD</sup> (e.g., address, len, vtable, ... of <i>this</i> <code>s</code> , like <code>0x1234</code> ).
<code>&amp;mut s</code>	Exclusive borrow that allows <b>mutability</b> . <sup>EX</sup>
<code>*const S</code>	Immutable <b>raw pointer type</b> <sup>BK STD REF</sup> w/o memory safety.
<code>*mut S</code>	Mutable raw pointer type w/o memory safety.
<code>&amp;raw const s</code>	Create raw pointer w/o going through reference; c. <code>ptr::addr_of!()</code> <sup>STD 🚧</sup>
<code>&amp;raw mut s</code>	Same, but mutable. 🚧 Raw ptrs. are needed for unaligned, packed fields.
<code>ref s</code>	<b>Bind by reference</b> , <sup>EX</sup> makes binding reference type. 🗑
<code>let ref r = s;</code>	Equivalent to <code>let r = &amp;s</code> .
<code>let S { ref mut x } = s;</code>	Mutable ref binding ( <code>let x = &amp;mut s.x</code> ), shorthand destructuring <sup>1</sup> version.
<code>*r</code>	<b>Dereference</b> <sup>BK STD NOM</sup> 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 <b>Copy</b> .
<code>s = *r;</code>	Won't work 🚫 if <code>*r</code> is not <b>Copy</b> , as that would move and leave empty place.
<code>s = *my_box;</code>	Special case <sup>2</sup> for <b>Box</b> that can also move out Box'ed content if it isn't <b>Copy</b> .
<code>'a</code>	A <b>lifetime parameter</b> , <sup>BK EX NOM REF</sup> duration of a flow in static analysis.
<code>&amp;'a S</code>	Only accepts an address holding an <code>s</code> ; addr. existing <code>'a</code> or longer.

Example	Explanation
<code>&amp;'a mut S</code>	Same, but allow content of address to be changed.
<code>struct S&lt;'a&gt; {}</code>	Signals <code>S</code> will contain address with lifetime <code>'a</code> . Creator of <code>S</code> decides <code>'a</code> .
<code>trait T&lt;'a&gt; {}</code>	Signals a <code>S</code> which <code>impl T for S</code> might contain address.
<code>fn f&lt;'a&gt;(t: &amp;'a T)</code>	Same, for function. 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 <b>trait</b> ; <sup>BK EX REF</sup> common behavior others can implement.
<code>trait T : R {}</code>	<code>T</code> is subtrait of <b>supertrait</b> <sup>REF</sup> <code>R</code> . Any <code>S</code> must <code>impl R</code> before it can <code>impl T</code> .
<code>impl S {}</code>	<b>Implementation</b> <sup>REF</sup> 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> .
<code>impl !T for S {}</code>	Disable an automatically derived <b>auto trait</b> . <sup>NOM REF</sup> <del>??</del>
<code>fn f() {}</code>	Definition of a <b>function</b> ; <sup>BK EX REF</sup> 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(&amp;self) {}</code>	Define a <b>method</b> , <sup>BK EX</sup> e.g., within an <code>impl S {}</code> .
<code>const fn f() {}</code>	Constant <code>fn</code> usable at compile time, e.g., <code>const X: u32 = f(Y)</code> . <sup>'18</sup>
<code>async fn f() {}</code>	<b>Async</b> <sup>REF '18</sup> function transformation, <sup>↑</sup> makes <code>f</code> return an <code>impl Future</code> . <sup>STD</sup>
<code>async fn f() → S {}</code>	Same, but make <code>f</code> return an <code>impl Future&lt;Output=S&gt;</code> .
<code>async { x }</code>	Used within a function, make <code>{ x }</code> an <code>impl Future&lt;Output=X&gt;</code> .
<code>fn() → S</code>	<b>Function pointers</b> , <sup>BK STD REF</sup> memory holding address of a callable.
<code>Fn() → S</code>	<b>Callable Trait</b> <sup>BK STD</sup> (also <code>FnMut</code> , <code>FnOnce</code> ), implemented by closures, <code>fn</code> 's ...
<code>   {}</code>	A <b>closure</b> <sup>BK EX REF</sup> that borrows its <b>captures</b> , <sup>↑ REF</sup> (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>	Closure taking ownership of its captures; i.e., <code>y</code> transferred to 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; <b>unsafe code</b> . <sup>↑ BK EX NOM REF</sup>
<code>unsafe fn f() {}</code>	Means " <i>calling can cause UB, <sup>↑</sup> YOU must check requirements</i> ".
<code>unsafe trait T {}</code>	Means " <i>careless impl. of <code>T</code> 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 well-behaved w.r.t <code>T</code> ; people may use <code>T</code> on <code>S</code> safely.

## Control Flow



Control execution within a function.


Example	Explanation
<code>while x {}</code>	<b>Loop</b> , <sup>REF</sup> run while expression <code>x</code> is true.
<code>loop {}</code>	<b>Loop indefinitely</b> <sup>REF</sup> until <code>break</code> . Can yield value with <code>break x</code> .
<code>for x in iter {}</code>	Syntactic sugar to loop over <b>iterators</b> . <sup>BK STD REF</sup>
<code>if x {} else {}</code>	<b>Conditional branch</b> <sup>REF</sup> if expression is true.

Example	Explanation
<code>'label: loop {}</code>	<b>Loop label</b> , <a href="#">EX</a> <a href="#">REF</a> useful for flow control in nested loops.
<code>break</code>	<b>Break expression</b> <a href="#">REF</a> to exit a loop.
<code>break x</code>	Same, but make x value of the loop expression (only in actual <code>loop</code> ).
<code>break 'label</code>	Exit not only this loop, but the enclosing one marked with <code>'label</code> .
<code>break 'label x</code>	Same, but make x the value of the enclosing loop marked with <code>'label</code> .
<code>continue</code>	<b>Continue expression</b> <a href="#">REF</a> to the next loop iteration of this loop.
<code>continue 'label</code>	Same but instead of this loop, enclosing loop marked with 'label.
<code>x?</code>	If x is <code>Err</code> or <code>None</code> , <b>return and propagate</b> . <a href="#">BK</a> <a href="#">EX</a> <a href="#">STD</a> <a href="#">REF</a>
<code>x.await</code>	Only works inside <code>async</code> . Yield flow until <code>Future</code> <a href="#">STD</a> or Stream x ready. <a href="#">REF</a> <sup>18</sup>
<code>return x</code>	Early return from function. More idiomatic way is to end with expression.
<code>f()</code>	Invoke callable f (e.g., a function, closure, function pointer, <code>Fn</code> , ...).
<code>x.f()</code>	Call member function, requires f takes <code>self</code> , <code>&amp;self</code> , ... as first argument.
<code>X :: f(x)</code>	Same as <code>x.f()</code> . Unless <code>impl Copy for X {}</code> , f can only be called once.
<code>X :: f(&amp;x)</code>	Same as <code>x.f()</code> .
<code>X :: f(&amp;mut x)</code>	Same as <code>x.f()</code> .
<code>S :: f(&amp;x)</code>	Same as <code>x.f()</code> if X <a href="#">derefs</a> to S, i.e., <code>x.f()</code> finds methods of S.
<code>T :: f(&amp;x)</code>	Same as <code>x.f()</code> if X <code>impl T</code> , i.e., <code>x.f()</code> finds methods of T if in scope.
<code>X :: f()</code>	Call associated function, e.g., <code>X :: new()</code> .
<code>&lt;X as T&gt; :: f()</code>	Call trait method <code>T :: f()</code> implemented for X.

## Organizing Code

Segment projects into smaller units and minimize dependencies.

Example	Explanation
<code>mod m {}</code>	Define a <b>module</b> , <a href="#">BK</a> <a href="#">EX</a> <a href="#">REF</a> get definition from inside <code>{}</code> . <sup>1</sup>
<code>mod m;</code>	Define a module, get definition from <code>m.rs</code> or <code>m/mod.rs</code> . <sup>1</sup>
<code>a :: b</code>	Namespace <b>path</b> <a href="#">EX</a> <a href="#">REF</a> to element b within a ( <code>mod</code> , <code>enum</code> , ...).
<code>:: b</code>	Search b relative to crate root. 
<code>crate :: b</code>	Search b relative to crate root. <sup>18</sup>
<code>self :: b</code>	Search b relative to current module.
<code>super :: b</code>	Search b relative to parent module.
<code>use a :: b;</code>	<b>Use</b> <a href="#">EX</a> <a href="#">REF</a> b directly in this scope without requiring a anymore.
<code>use a :: {b, c};</code>	Same, but bring b and c into scope.
<code>use a :: b as x;</code>	Bring b into scope but name x, like <code>use std :: error :: Error as E</code> .
<code>use a :: b as _;</code>	Bring b anonymously into scope, useful for traits with conflicting names.
<code>use a :: *;</code>	Bring everything from a in, only recommended if a is some <b>prelude</b> . 
<code>pub use a :: b;</code>	Bring a :: b into scope and reexport from here.
<code>pub T</code>	"Public if parent path is public" <b>visibility</b> <a href="#">BK</a> <a href="#">REF</a> for T.
<code>pub(crate) T</code>	Visible at most <sup>1</sup> in current crate.
<code>pub(super) T</code>	Visible at most <sup>1</sup> in parent.
<code>pub(self) T</code>	Visible at most <sup>1</sup> in current module (default, same as no <code>pub</code> ).
<code>pub(in a :: b) T</code>	Visible at most <sup>1</sup> in ancestor a :: b.

Example	Explanation
<code>extern crate a;</code>	Declare dependency on external <b>crate</b> ; <sup>BK REF</sup>  just <code>use a :: b in '18</code> .
<code>extern "C" {}</code>	Declare external dependencies and ABI (e.g., "C") from <b>FFI</b> . <sup>BK EX NOM REF</sup>
<code>extern "C" fn f() {}</code>	Define function to be exported with ABI (e.g., "C") to FFI.

<sup>1</sup> 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 <b>type alias</b> , <sup>BK REF</sup> i.e., another name for <code>S</code> .
<code>Self</code>	Type alias for <b>implementing type</b> , <sup>REF</sup> e.g. <code>fn new() → Self</code> .
<code>self</code>	Method subject in <code>fn f(self) {}</code> , same as <code>fn f(self: Self) {}</code> .
<code>&amp;self</code>	Same, but refers to self as borrowed, same as <code>f(self: &amp;Self)</code>
<code>&amp;mut self</code>	Same, but mutably borrowed, same as <code>f(self: &amp;mut Self)</code>
<code>self: Box&lt;Self&gt;</code>	<b>Arbitrary self type</b> , add methods to smart pointers ( <code>my_box.f_of_self()</code> ).
<code>S as T</code>	<b>Disambiguate</b> <sup>BK REF</sup> type <code>S</code> as trait <code>T</code> , e.g., <code>&lt;S as T&gt; :: f()</code> .
<code>S as R</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 <b>cast</b> , <sup>EX REF</sup> may truncate and be a bit surprising. <sup>NOM</sup>

## Macros & Attributes

Code generation constructs expanded before the actual compilation happens.

Example	Explanation
<code>m!()</code>	<b>Macro</b> <sup>BK STD REF</sup> invocation, also <code>m!{}</code> , <code>m![]</code> (depending on macro).
<code>#[attr]</code>	Outer <b>attribute</b> , <sup>EX REF</sup> 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 (here a type); see <b>tooling directives</b> <sup>1</sup> for details.
<code>\$x</code>	Macro substitution, e.g., use the captured <code>\$x:ty</code> from above.
<code>\$(x),*</code>	Macro repetition "zero or more times" in macros by example.
<code>\$(x),?</code>	Same, but "zero or one time".
<code>\$(x),+</code>	Same, but "one or more times".
<code>\$(x)&lt;&lt;+</code>	In fact separators other than <code>,</code> are also accepted. Here: <code>&lt;&lt;</code> .

## Pattern Matching

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

Example	Explanation
<code>match m {}</code>	Initiate <b>pattern matching</b> , <sup>BK EX REF</sup> then use match arms, c. next table.
<code>let S(x) = get();</code>	Notably, <code>let</code> also <b>destructures</b> <sup>EX</sup> 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.

Example	Explanation
<code>let ( .. , a, b) = (1, 2);</code>	Specific bindings take precedence over 'the rest', here a is 1, b is 2.
<code>let Some(x) = get();</code>	<b>Won't work</b> ● if pattern can be <b>refuted</b> , <sup>REF</sup> use <code>if let</code> instead.
<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 x bound to s.x of f(s).

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

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

Within Match Arm	Explanation
<code>E :: A =&gt; {}</code>	Match enum variant A, c. <b>pattern matching</b> . <sup>BK EX REF</sup>
<code>E :: B ( .. ) =&gt; {}</code>	Match enum tuple variant B, wildcard any index.
<code>E :: C { .. } =&gt; {}</code>	Match enum struct variant C, wildcard any field.
<code>S { x: 0, y: 1 } =&gt; {}</code>	Match struct with specific values (only accepts s with s.x of 0 and s.y of 1).
<code>S { x: a, y: b } =&gt; {}</code>	Match struct with <i>any</i> (!) values and bind s.x to a and s.y to b.
<code>S { x, y } =&gt; {}</code>	Same, but shorthand with s.x and s.y bound as x and y respectively.
<code>S { .. } =&gt; {}</code>	Match struct with any values.
<code>D =&gt; {}</code>	Match enum variant E :: D if D in use.
<code>D =&gt; {}</code>	Match anything, bind D; possibly false friend ● of E :: D if D not in use.
<code>_ =&gt; {}</code>	Proper wildcard that matches anything / "all the rest".
<code>0   1 =&gt; {}</code>	Pattern alternatives, <b>or-patterns</b> . <sup>RFC</sup>
<code>E :: A   E :: Z</code>	Same, but on enum variants.
<code>E :: C {x}   E :: D {x}</code>	Same, but bind x if all variants have it.
<code>Some(A   B)</code>	Same, can also match alternatives deeply nested.
<code>(a, 0) =&gt; {}</code>	Match tuple with any value for a and 0 for second.
<code>[a, 0] =&gt; {}</code>	<b>Slice pattern</b> , <sup>REF</sup> ♂ match array with any value for a and 0 for second.
<code>[1, ..] =&gt; {}</code>	Match array starting with 1, any value for rest; <b>slice pattern</b> . ?
<code>[1, .., 5] =&gt; {}</code>	Match array starting with 1, ending with 5.
<code>[1, x @ .., 5] =&gt; {}</code>	Same, but also bind x to slice representing middle (c. pattern binding).
<code>[a, x @ .., b] =&gt; {}</code>	Same, but match any first, last, bound as a, b respectively.
<code>1 .. 3 =&gt; {}</code>	<b>Range pattern</b> , <sup>BK REF</sup> here matches 1 and 2; partially unstable. 🚧
<code>1 ..= 3 =&gt; {}</code>	Inclusive range pattern, matches 1, 2 and 3.
<code>1 .. =&gt; {}</code>	Open range pattern, matches 1 and any larger number.
<code>x @ 1..=5 =&gt; {}</code>	Bind matched to x; <b>pattern binding</b> , <sup>BK EX REF</sup> here x would be 1, 2, ... or 5.
<code>Err(x @ Error { .. }) =&gt; {}</code>	Also works nested, here x binds to <code>Error</code> , esp. useful with <code>if</code> below.
<code>S { x } if x &gt; 10 =&gt; {}</code>	Pattern <b>match guards</b> , <sup>BK EX REF</sup> 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>S&lt;T&gt;</code>	A <b>generic</b> <sup>BK EX</sup> type with a type parameter (T is placeholder name here).
<code>S&lt;T: R&gt;</code>	Type short hand <b>trait bound</b> <sup>BK EX</sup> specification (R <i>must</i> be actual trait).

Example	Explanation
<code>T: R, P: S</code>	<b>Independent trait bounds</b> (here one for <code>T</code> and one for <code>P</code> ).
<code>T: R, S</code>	Compile error, <span style="color:red">●</span> you probably want compound bound <code>R + S</code> below.
<code>T: R + S</code>	<b>Compound trait bound</b> , <sup>BK EX</sup> <code>T</code> must fulfill <code>R</code> and <code>S</code> .
<code>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>T: ?Sized</code>	Opt out of a pre-defined trait bound, here <code>Sized</code> . <sup>?</sup>
<code>T: 'a</code>	Type <b>lifetime bound</b> ; <sup>EX</sup> if <code>T</code> has references, they must outlive <code>'a</code> .
<code>T: 'static</code>	Same; does esp. <i>not</i> mean value <code>t</code> <i>will</i> <span style="color:red">●</span> live <code>'static</code> , only that it could.
<code>'b: 'a</code>	Lifetime <code>'b</code> must live at least as long as (i.e., <i>outlive</i> ) <code>'a</code> bound.
<code>S&lt;const N: usize&gt;</code>	<b>Generic const bound</b> ; <sup>?</sup> user of type <code>S</code> can provide constant value <code>N</code> .
<code>S&lt;10&gt;</code>	Where used, const bounds can be provided as primitive values.
<code>S&lt;{5+5}&gt;</code>	Expressions must be put in curly brackets.
<code>S&lt;T&gt; where T: R</code>	Almost same as <code>S&lt;T: R&gt;</code> but more pleasant to read for longer bounds.
<code>S&lt;T&gt; where u8: R&lt;T&gt;</code>	Also allows you to make conditional statements involving <i>other</i> types.
<code>S&lt;T = R&gt;</code>	<b>Default type parameter</b> <sup>BK</sup> for associated type.
<code>S&lt;'_&gt;</code>	Inferred <b>anonymous lifetime</b> ; asks compiler to 'figure it out' if obvious.
<code>S&lt;_&gt;</code>	Inferred <b>anonymous type</b> , e.g., as <code>let x: Vec&lt;_&gt; = iter.collect()</code>
<code>S::&lt;T&gt;</code>	<b>Turbofish</b> <sup>STD</sup> call site type disambiguation, e.g. <code>f::&lt;u32&gt;()</code> .
<code>trait T&lt;X&gt; {}</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 <b>associated type</b> <sup>BK REF RFC</sup> <code>X</code> . Only one <code>impl T for S</code> possible.
<code>type X = R;</code>	Set associated type within <code>impl T for S { type X = R; }</code> .
<code>impl&lt;T&gt; S&lt;T&gt; {}</code>	Implement functionality for any <code>T</code> in <code>S&lt;T&gt;</code> , here <code>T</code> type parameter.
<code>impl S&lt;T&gt; {}</code>	Implement functionality for exactly <code>S&lt;T&gt;</code> , here <code>T</code> specific type (e.g., <code>S&lt;u32&gt;</code> ).
<code>fn f() → impl T</code>	<b>Existential types</b> , <sup>BK</sup> returns an unknown-to-caller <code>S</code> that <code>impl T</code> .
<code>fn f(x: &amp;impl T)</code>	Trait bound, " <b>impl traits</b> ", <sup>BK</sup> somewhat similar to <code>fn f&lt;S:T&gt;(x: &amp;S)</code> .
<code>fn f(x: &amp;dyn T)</code>	Marker for <b>dynamic dispatch</b> , <sup>BK REF</sup> <code>f</code> will not be monomorphized.
<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&lt;'a&gt;</code>	Marker for <b>higher-ranked bounds</b> . <sup>NOM REF</sup>
<code>trait T: for&lt;'a&gt; R&lt;'a&gt; {}</code>	Any <code>S</code> that <code>impl T</code> would also have to fulfill <code>R</code> for any lifetime.
<code>fn(&amp;'a u8)</code>	<i>Fn. ptr.</i> type holding fn callable with <b>specific</b> lifetime <code>'a</code> .
<code>for&lt;'a&gt; fn(&amp;'a u8)</code>	<b>Higher-ranked type</b> <sup>1</sup> <sup>♂</sup> holding fn callable with <b>any</b> <i>lt.</i> ; subtype of above.
<code>fn(&amp;'_ u8)</code>	Same; automatically expanded to type <code>for&lt;'a&gt; fn(&amp;'a u8)</code> .
<code>fn(&amp;u8)</code>	Same; automatically expanded to type <code>for&lt;'a&gt; fn(&amp;'a u8)</code> .
<code>dyn for&lt;'a&gt; Fn(&amp;'a u8)</code>	Higher-ranked (trait-object) type, works like <code>fn</code> above.
<code>dyn Fn(&amp;'_ u8)</code>	Same; automatically expanded to type <code>dyn for&lt;'a&gt; Fn(&amp;'a u8)</code> .
<code>dyn Fn(&amp;u8)</code>	Same; automatically expanded to type <code>dyn for&lt;'a&gt; Fn(&amp;'a u8)</code> .

<sup>1</sup> 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&lt;'a&gt; T for fn(&amp;'a u8) {}</code>	For fn. pointer, where call accepts <b>specific</b> <i>lt.</i> <code>'a</code> , impl trait <code>T</code> .
<code>impl T for for&lt;'a&gt; fn(&amp;'a u8) {}</code>	For fn. pointer, where call accepts <b>any</b> <i>lt.</i> , impl trait <code>T</code> .
<code>impl T for fn(&amp;u8) {}</code>	Same, short version.

Strings & Chars

Rust has several ways to create textual values.

Example	Explanation
<code>" ... "</code>	<b>String literal</b> , <a href="#">REF</a> , <sup>1</sup> UTF-8, will interpret <code>\n</code> as <i>line break</i> <code>0xA</code> , ...
<code>r" ... "</code>	<b>Raw string literal</b> , <a href="#">REF</a> , <sup>1</sup> UTF-8, won't interpret <code>\n</code> , ...
<code>r#" ... "#</code>	Raw string literal, UTF-8, but can also contain <code>"</code> . Number of <code>#</code> can vary.
<code>b" ... "</code>	<b>Byte string literal</b> , <a href="#">REF</a> , <sup>1</sup> 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>	<b>Character literal</b> , <a href="#">REF</a> fixed 4 byte unicode <code>'char'</code> . <a href="#">STD</a>
<code>b'x'</code>	ASCII <b>byte literal</b> . <a href="#">REF</a>

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

Documentation

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

Example	Explanation
<code>///</code>	Outer line <b>doc comment</b> , <a href="#">BK</a> <a href="#">EX</a> <a href="#">REF</a> 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.
<code>/** ... */</code>	Outer block doc comment.
<code>/*! ... */</code>	Inner block doc comment.

Tooling directives<sup>1</sup> outlines what you can do inside doc comments.

Miscellaneous

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

Example	Explanation
<code>!</code>	Always empty <b>never type</b> . <a href="#">🔥</a> <a href="#">BK</a> <a href="#">EX</a> <a href="#">STD</a> <a href="#">REF</a>
<code>_</code>	Unnamed variable binding, e.g., <code> x, _  {}</code> .
<code>let _ = x;</code>	Unnamed assignment is no-op, does <b>not</b> 🟡 move out <code>x</code> or preserve scope!
<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 <b>numeric literals</b> <a href="#">EX</a> <a href="#">REF</a> (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 <b>raw identifier</b> <a href="#">BK</a> <a href="#">EX</a> for edition compatibility.
<code>x;</code>	<b>Statement</b> <a href="#">REF</a> terminator, c. <b>expressions</b> <a href="#">EX</a> <a href="#">REF</a>



## Common Operators

Rust supports most operators you would expect (+, \*, %, =, ==, ...), including **overloading**.<sup>STD</sup> 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

Rust

→

CPU

● Less correctish.

Rust

→

Abstract Machine

→

CPU

More correctish.

The abstract machine

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

Misconceptions


Things people may incorrectly assume they *should get away with* if Rust targeted CPU directly, and *more correct* counterparts:


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

## 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
------	-------------


Name	Description
<b>Coercions</b> <small>NOM</small>	<i>Weakens</i> types to match signature, e.g., <code>&amp;mut T</code> to <code>&amp;T</code> ; c. <i>type conversions</i> . <sup>1</sup>
<b>Deref</b> <small>NOM</small> 	<i>Derefs</i> <code>x: T</code> until <code>*x</code> , <code>**x</code> , ... compatible with some target <code>S</code> .
<b>Prelude</b> <small>STD</small>	Automatic import of basic items, e.g., <code>Option</code> , <code>drop</code> , ...
<b>Reborrow</b>	Since <code>x: &amp;mut T</code> can't be copied; moves new <code>&amp;mut *x</code> instead.
<b>Lifetime Elision</b> <small>BK NOM REF</small>	Automatically annotates <code>f(x: &amp;T)</code> to <code>f&lt;'a&gt;(x: &amp;'a T)</code> .
<b>Method Resolution</b> <small>REF</small>	Derefs or borrow <code>x</code> until <code>x.f()</code> works.
<b>Match Ergonomics</b> <small>RFC</small>	Repeatedly dereferences <i>scrutinee</i> and adds <code>ref</code> and <code>ref mut</code> to bindings.
<b>Rvalue Static Promotion</b> <small>RFC</small>	Makes references to constants <code>'static</code> , e.g., <code>&amp;42</code> , <code>&amp;None</code> , <code>&amp;mut []</code> .

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

## Memory & Lifetimes

Why moves, references and lifetimes are how they are.

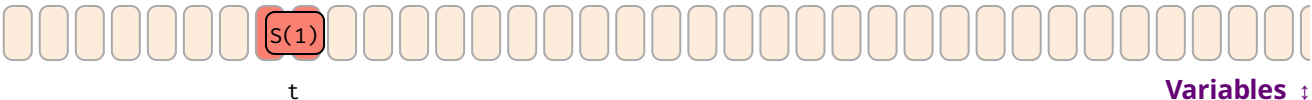
Types & Moves



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,<sup>1</sup> 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**.

<sup>1</sup> 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

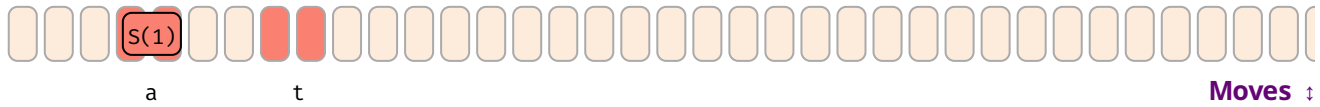
Variables ↓

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

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

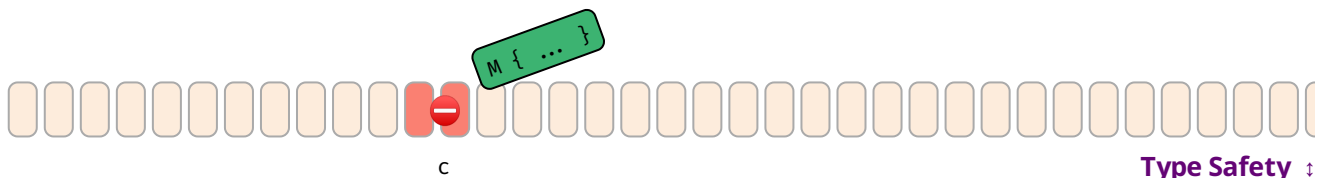
- Specifically towards the compiler `t` can mean **location of** `t`, here `0x7`, and **value within** `t`, here `S(1)`.

<sup>1</sup> Compare above, <sup>1</sup> true for fully synchronous code, but `async` stack frame might place it on heap via runtime.



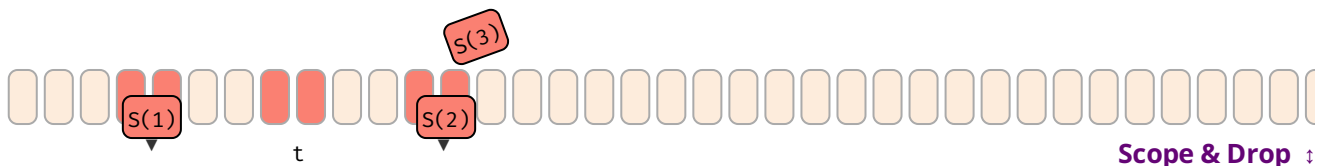
```
let a = t;
```

- This will **move** value within `t` to location of `a`, or copy it, if `S` is `Copy`.
- After move location `t` is **invalid** and cannot be read anymore.
  - Technically the bits at that location are not really *empty*, but *undefined*.
  - If you still had access to `t` (via `unsafe`) they might still *look* like valid `S`, but any attempt to use them as valid `S` is undefined behavior.<sup>1</sup>
- We do not cover `Copy` types explicitly here. They change the rules a bit, but not much:
  - They won't be dropped.
  - They never leave behind an 'empty' variable location.



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

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

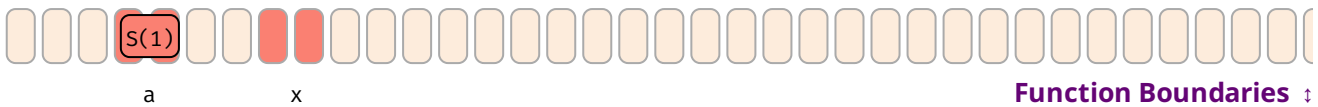


```
{
  let mut c = S(2);
  c = S(3); // ← Drop called on `c` before assignment.
  let t = S(1);
  let a = t;
} // ← Scope of `a`, `t`, `c` ends here, drop called on `a`, `c`.
```

- Once the 'name' of a non-vacated variable goes out of (drop-)scope, the contained value is **dropped**.
  - Rule of thumb: execution reaches point where name of variable leaves `{ }`-block it was defined in

- In detail more tricky, esp. temporaries, ...
- Drop also invoked when new value assigned to existing variable location.
- In that case **Drop** :: **drop()** is called on the location of that value.
  - In the example above **drop()** is called on **a**, twice on **c**, but not on **t**.
- Most non-**Copy** values get dropped most of the time; exceptions include `mem :: forget()`, `Rc` cycles, `abort()`.

#### Call Stack

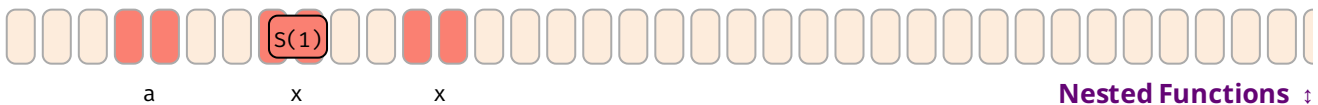


```
fn f(x: S) { ... }

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

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

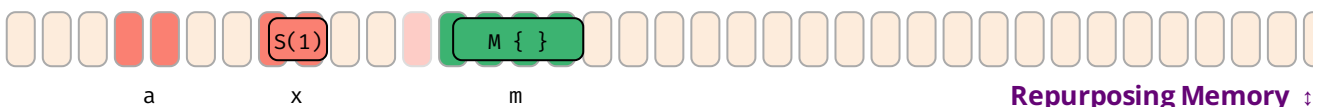
<sup>1</sup> Actual location depends on calling convention, might practically not end up on stack at all, but that doesn't change mental model.



```
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.



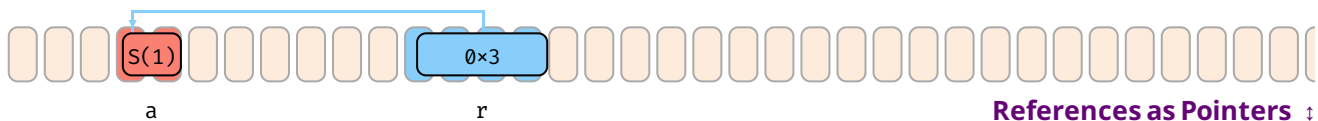
```
fn f(x: S) {
  if once() { f(x) }
  let m = M::new() // ← We are here (after recursion)
}

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

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

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

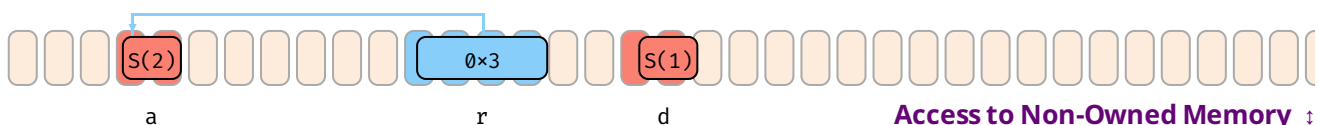
## References & Pointers



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

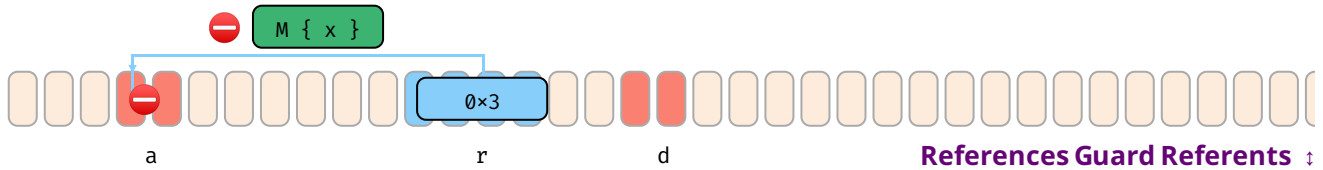
- A **reference type** such as `&S` or `&mut S` can hold the **location of** some `s`.
- Here type `&S`, bound as name `r`, holds *location of* variable `a` (`0x3`), that must be type `S`, obtained via `&a`.
- If you think of variable `c` as *specific location*, reference `r` **is a switchboard for locations**.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

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



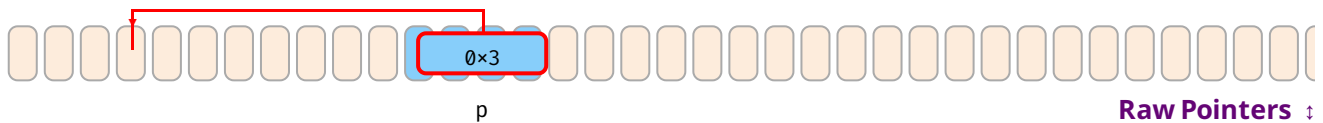
```
let mut a = S(1);
let r = &mut a;
let d = r.clone(); // Valid to clone (or copy) from r-target.
*r = S(2);         // Valid to set new S value to r-target.
```

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



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

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



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

- In contrast to references, pointers come with almost no guarantees.
- They may point to invalid or non-existent data.
- Dereferencing them is *unsafe*, and treating an invalid `*p` as if it were valid is undefined behavior. <sup>1</sup>

## Lifetime Basics

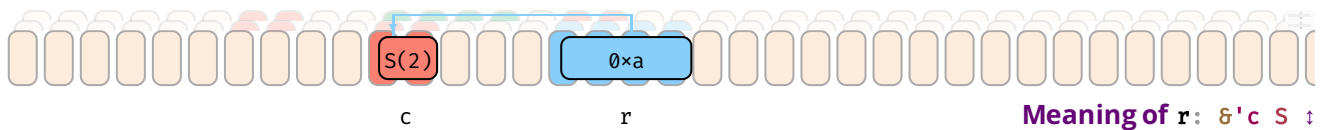


- Every entity in a program has some (temporal / spatial) room where it is relevant, i.e., *alive*.
- Loosely speaking, this *alive time* can be<sup>1</sup>
  1. the **LOC** (lines of code) where an **item is available** (e.g., a module name).
  2. the **LOC** between when a *location* is **initialized** with a value, and when the location is **abandoned**.
  3. the **LOC** between when a location is first **used in a certain way**, and when that **usage stops**.
  4. the **LOC (or actual time)** between when a *value* is created, and when that value is dropped.
- Within the rest of this section, we will refer to the items above as the:
  1. **scope** of that item, irrelevant here.
  2. **scope** of that variable or location.
  3. **lifetime**<sup>2</sup> of that usage.
  4. **lifetime** of that value, might be useful when discussing open file descriptors, but also irrelevant here.
- Likewise, lifetime parameters in code, e.g., `r: &'a S`, are

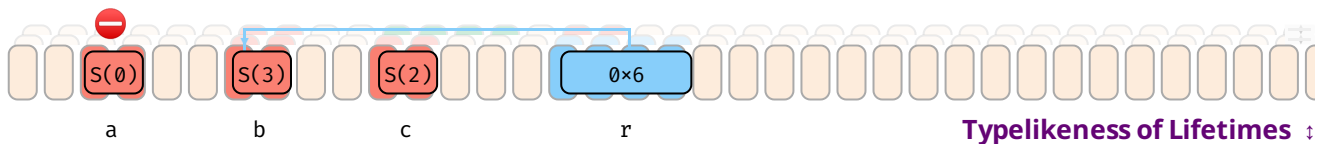
- concerned with LOC any **location  $r$  points to** needs to be accessible or locked;
- unrelated to the 'existence time' (as LOC) of  $r$  itself (well, it needs to exist shorter, that's it).
- $\&'static\ S$  means address must be *valid during all lines of code*.

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

<sup>2</sup> *Live lines* might have been a more appropriate term ...



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

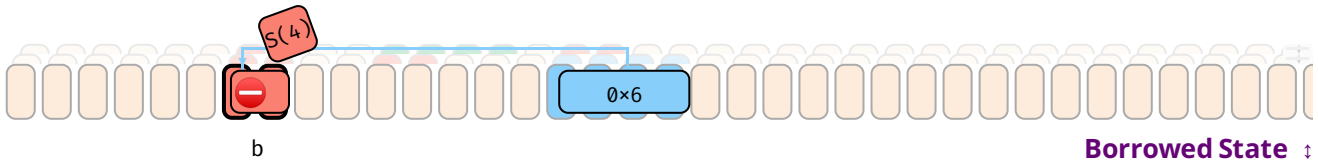


```
{
  let b = S(3);
  {
    let c = S(2);
    let r:  $\&'c\ S$  =  $\&c$ ;    // Does not quite work since we can't name lifetimes of local
    {                      // variables in a function body, but very same principle applies
      let a = S(0);        // to functions next page.

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

not ok.

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



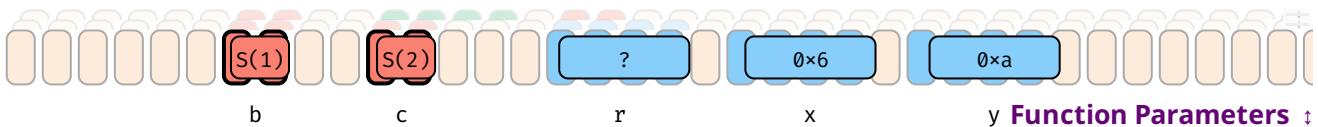
```
let mut b = S(0);
let r = &mut b;

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

print_byte(r);
```

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

#### Lifetimes in Functions

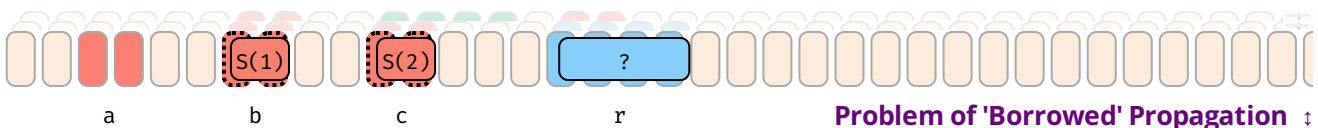


```
fn f(x: &S, y: &S) → &u8 { ... }

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

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

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



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

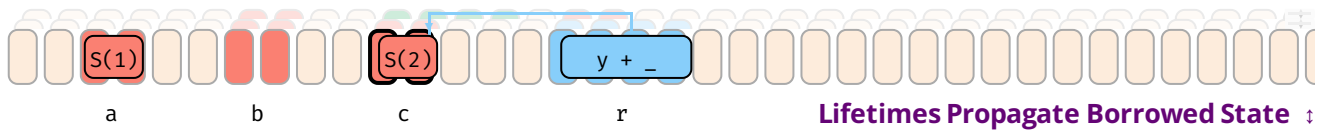
let r = f(&b, &c);

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

print_byte(r);
```



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



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

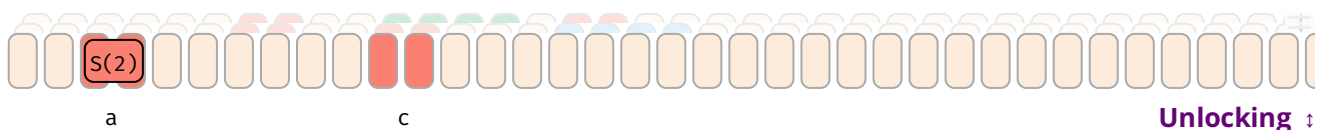
let b = S(1);
let c = S(2);

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

let a = b;

print_byte(r);
```

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



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

let r = f(&c);
let s = r;

// ← Not here, `s` prolongs locking of `c`.

print_byte(s);

let a = c;

// ← But here, no more use of `r` or `s`.
```

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

† Examples expand by clicking.

# Data Layout

Memory representations of common data types.

## Basic Types

Essential types built into the core of the language.

### Numeric Types <sup>REF</sup>

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

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 follow [IEEE 754-2008](#) and depend on platform endianness.

## Casting Pitfalls

Cast <sup>1</sup>	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	-

## Arithmetical Pitfalls

Operation <sup>1</sup>	Gives	Note
<code>200_u8 / 0_u8</code>	Compile error.	-
<code>200_u8 / _0<sup>d</sup></code>	Panic.	Regular math may panic; here: division by zero.
<code>200_u8 / _0<sup>r</sup></code>	Panic.	Same.
<code>200_u8 + 200_u8</code>	Compile error.	-
<code>200_u8 + _200<sup>d</sup></code>	Panic.	Consider <code>checked_</code> , <code>wrapping_</code> , ... instead. <sup>STD</sup>
<code>200_u8 + _200<sup>r</sup></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	-
<code>1.0_f32 / 0.0_f32</code>	<code>f32::INFINITY</code>	-
<code>0.0_f32 / 0.0_f32</code>	<code>f32::NaN</code>	-

<sup>1</sup> Expression `_100` means anything that might contain the value 100, e.g., `100_i32`, but is opaque to compiler.

<sup>d</sup> Debug build.

<sup>r</sup> Release build.

## Textual Types <sup>REF</sup>

### char



Any UTF-8 scalar.

### str



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

## Basics

## Type

## Description

<code>char</code>	Always 4 bytes and only holds a single Unicode <b>scalar value</b> <sup>🔗</sup> .
<code>str</code>	An <code>u8</code> -array of unknown length guaranteed to hold <b>UTF-8 encoded code points</b> .

## Usage

### Chars

### Description

<code>let c = 'a';</code>	Often a <code>char</code> (unicode scalar) can coincide with your intuition of <i>character</i> .
<code>let c = '❤️';</code>	It can also hold many Unicode symbols.
<code>let c = '❤️';</code>	But not always. Given emoji is <b>two</b> <code>char</code> (see Encoding) and <b>can't</b> ❤️ be held by <code>c</code> . <sup>1</sup>
<code>c = 0xffff_ffff;</code>	Also, chars are <b>not allowed</b> ❤️ to hold arbitrary bit patterns.

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

### Strings

### Description

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

## Encoding

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

### Variant

### Memory Representation<sup>2</sup>

<code>s.as_bytes()</code>	49 20 e2 9d a4 20 52 75 73 74 <sup>3</sup>
<code>s.chars()</code> <sup>1</sup>	49 00 00 00 20 00 00 00 64 27 00 00 20 00 00 00 52 00 00 00 75 00 00 00 73 00 ...
<code>t.as_bytes()</code>	49 20 e2 9d a4 ef b8 8f 20 52 75 73 74 <sup>4</sup>
<code>t.chars()</code> <sup>1</sup>	49 00 00 00 20 00 00 00 64 27 00 00 0f fe 01 00 20 00 00 00 52 00 00 00 75 00 ...

<sup>1</sup> Result then collected into array and transmuted to bytes.

<sup>2</sup> Values given in hex, on x86.

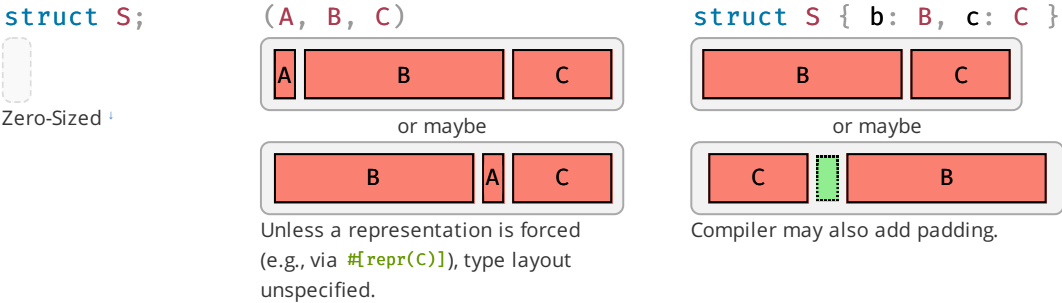
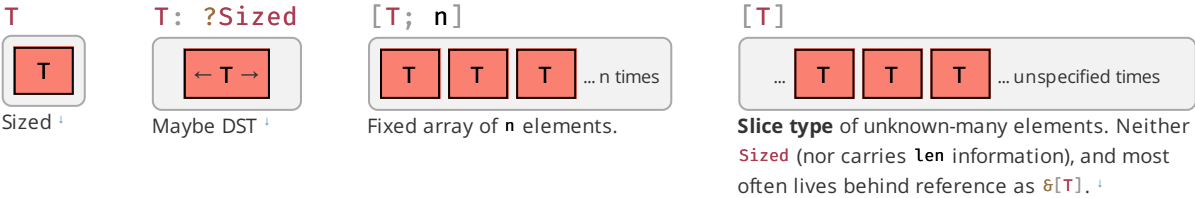
<sup>3</sup> Notice how ❤️, having [Unicode Code Point \(U+2764\)](#), is represented as **64 27 00 00** inside the `char`, but got [UTF-8 encoded](#) to **e2 9d a4** in the `str`.

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

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

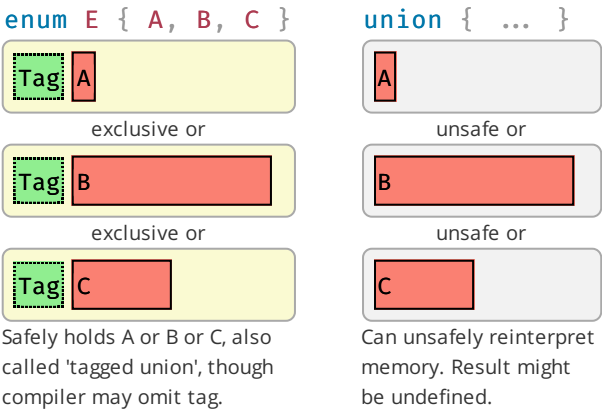
# Custom Types

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



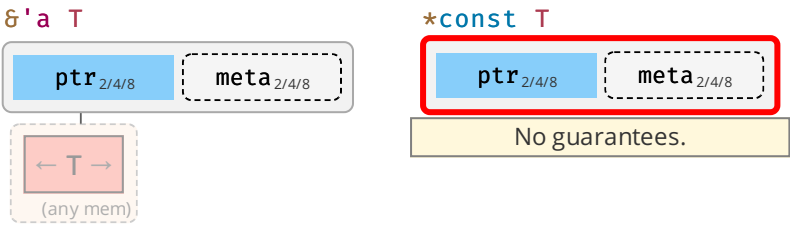
Also note, two types **A(X, Y)** and **B(X, Y)** with exactly the same fields can still have differing layout; never **transmute()** without representation guarantees.

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



## References & Pointers

References give safe access to other memory, raw pointers **unsafe** access. The respective **mut** types are identical.

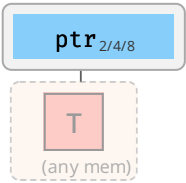


Must target some valid **t** of **T**, and any such target must exist for at least **'a**.

## Pointer Meta

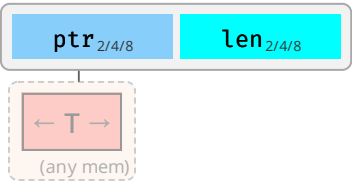
Many reference and pointer types can carry an extra field, **pointer metadata**.<sup>STD</sup> It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

`&'a T`



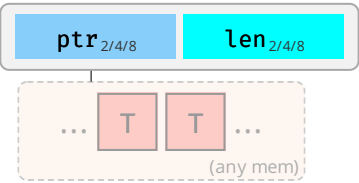
No meta for sized target. (pointer is thin).

`&'a T`



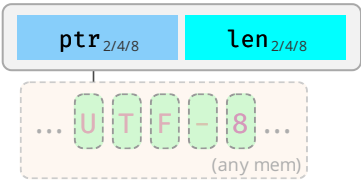
If `T` is a DST **struct** such as `S { x: [u8] }` meta field `len` is length of dyn. sized content.

`&'a [T]`



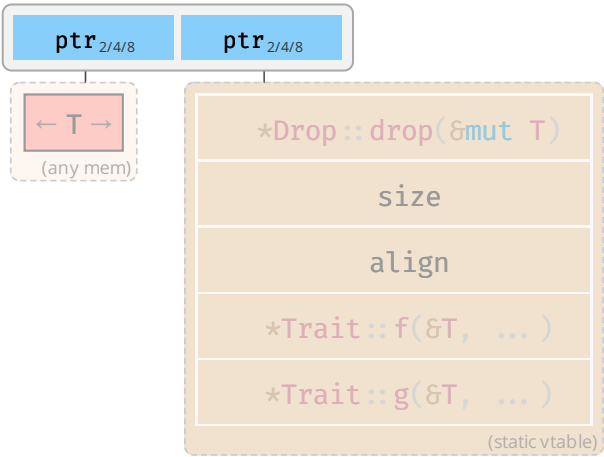
Regular **slice reference** (i.e., the reference type of a slice type `[T]`)<sup>1</sup> often seen as `&[T]` if `'a` elided.

`&'a str`



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

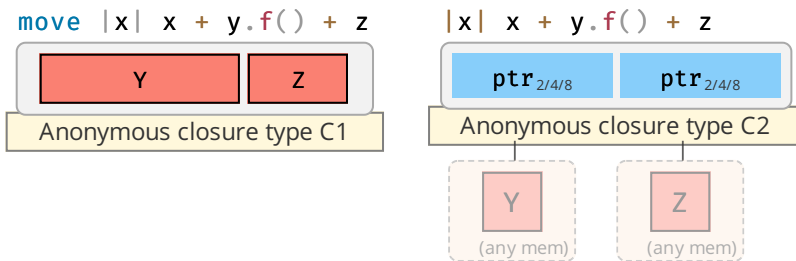
`&'a dyn Trait`



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**<sup>REF</sup> environment where closure was defined. For example:

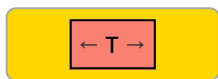


Also produces anonymous `fn` such as `fC1(C1, X)` or `fC2(C2, X)`. Details depend which `FnOnce`, `FnMut`, `Fn` ... is supported, based on properties of captured types.

## Standard Library Types

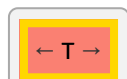
Rust's standard library combines the above primitive types into useful types with special semantics, e.g.:

### UnsafeCell<T>



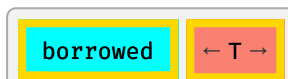
Magic type allowing aliased mutability.

### Cell<T>



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

### RefCell<T>



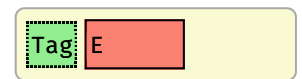
Also support dynamic borrowing of `T`. Like `Cell` this is `Send`, but not `Sync`.

### AtomicUsize

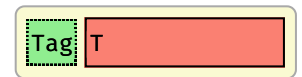


Other atomic similarly.

### Result<T, E>



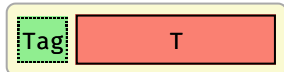
or



### Option<T>



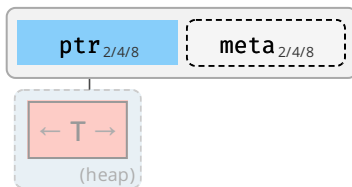
or



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

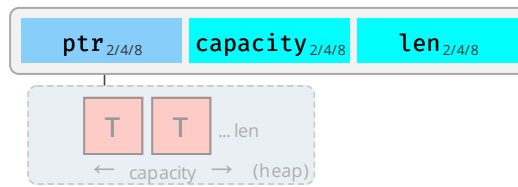
## General Purpose Heap Storage

### Box<T>



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

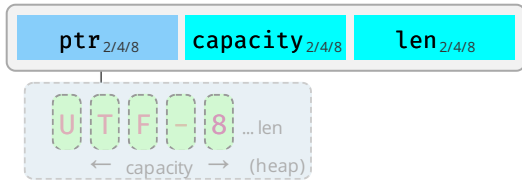
### Vec<T>



## Owned Strings

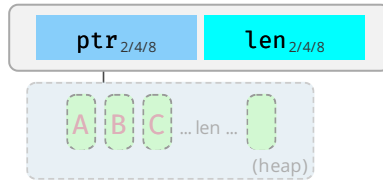


## String



Observe how `String` differs from `ustr` and `u8[char]`.

## CString



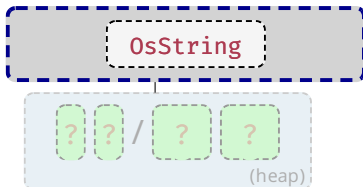
NUL-terminated but w/o NUL in middle.

## OsString ?



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

## PathBuf ?



Encapsulates how operating system represents paths.

## Shared Ownership

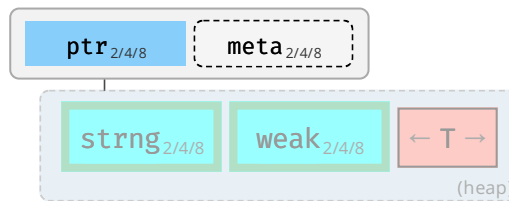
If the type does not contain a `Cell` for `T`, these are often combined with one of the `Cell` types above to allow shared de-facto mutability.

### Rc<T>



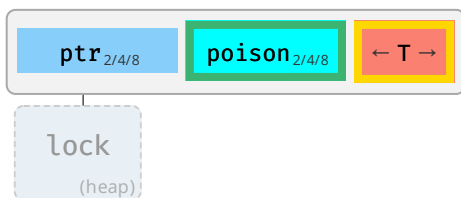
Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell` to allow mutation. Is neither `Send` nor `Sync`.

### Arc<T>



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

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



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

## Standard Library

### One-Liners

Snippets that are common, but still easy to forget. See **Rust Cookbook** <sup>o</sup> for more.

## Strings

Intent	Snippet
Concatenate strings (any <code>Display</code> <sup>1</sup> that is). <sup>1</sup>	<code>format!("{}", x, y)</code>
Split by separator pattern. <sup>STD</sup> <sup>o</sup>	<code>s.split(pattern)</code>
... with <code>&amp;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. <sup>2</sup>	<code>Regex::new(r"\s")?.split("one two three")</code>

<sup>1</sup> Allocates; might not be fastest solution if `x` is `String` already.

<sup>2</sup> Requires `regex` crate.

## I/O

Intent	Snippet
Create a new file	<code>File::create(PATH)?</code>
Same, via OpenOptions	<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),*) =&gt; {{ }} }</code>
Using args, e.g., calling <code>f</code> multiple times.	<code>\$( f(\$args); )*</code>

## Esoterics

Intent	Snippet
Cleaner closure captures	<code>wants_closure({ let c = outer.clone(); move    use_clone(c) })</code>
Fix inference in ' <code>try</code> ' closures	<code>iter.try_for_each( x  { Ok::&lt;(), Error&gt;(() ) })?;</code>
Iterate <i>and</i> edit <code>&amp;mut [T]</code> if <code>T</code> Copy.	<code>Cell::from_mut(mut_slice).as_slice_of_cells()</code>
Get subslice with length.	<code>&amp;original_slice[offset..][..length]</code>

Intent	Snippet
Canary to ensure trait <code>T</code> is object safe.	<code>const _: Option&lt;&amp;dyn T&gt; = None;</code>

## Thread Safety

Examples	Send*	!Send
<code>Sync*</code>	<i>Most types ...</i> <code>Mutex&lt;T&gt;</code> , <code>Arc&lt;T&gt;</code> <sup>1,2</sup>	<code>MutexGuard&lt;T&gt;</code> <sup>1</sup> , <code>RwLockReadGuard&lt;T&gt;</code> <sup>1</sup>
<code>!Sync</code>	<code>Cell&lt;T&gt;</code> <sup>2</sup> , <code>RefCell&lt;T&gt;</code> <sup>2</sup>	<code>Rc&lt;T&gt;</code> , <code>&amp;dyn Trait</code> , <code>*const T</code> <sup>3</sup> , <code>*mut T</code> <sup>3</sup>

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

<sup>1</sup> If `T` is `Sync`.

<sup>2</sup> If `T` is `Send`.

<sup>3</sup> 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

### Obtaining Iterators

#### Basics

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

- `c.into_iter()` — Turns collection `c` into an `Iterator` <sup>STD</sup> `i` and **consumes**\* `c`. Requires `IntoIterator` <sup>STD</sup> for `c` to be implemented. Type of item depends on what `c` was. 'Standardized' way to get Iterators.
- `c.iter()` — Courtesy method **some** collections provide, returns **borrowing** Iterator, doesn't consume `c`.
- `c.iter_mut()` — Same, but **mutably borrowing** Iterator that allow collection to be changed.

#### The Iterator

Once you have an `i`:

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

#### For Loops

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

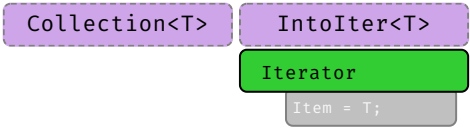
\* If it looks as if it doesn't consume `c` that's because type was `Copy`. For example, if you call `(&c).into_iter()` it will invoke `.into_iter()` on `&c` (which will consume the reference and turn it into an Iterator), but `c` remains untouched.

### Implementing Iterators

#### Basics

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

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

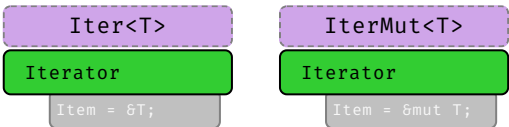


Shared & Mutable Iterators

- `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.

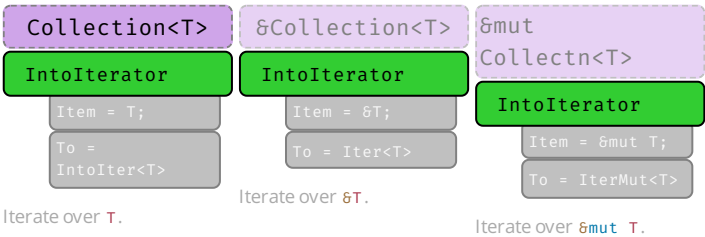
In addition, you might want to add convenience methods:

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



Making Loops Work

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



Number Conversions

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

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

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

# String Conversions

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

## String

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

## CString

If you have x of type ...	Use this ...
String	CString::new(x)?
CString	x
OsString <sup>2</sup>	CString::new(x.to_str())?
PathBuf	CString::new(x.to_str())?
Vec<u8> <sup>1</sup>	CString::new(x)?
&str	CString::new(x)?
&CStr	x.to_owned() <sup>i</sup>
&OsStr <sup>2</sup>	CString::new(x.to_os_string().into_string())?
&Path	CString::new(x.to_str())?
&[u8] <sup>1</sup>	CString::new(Vec::from(x))?
*mut c_char <sup>3</sup>	unsafe { CString::from_raw(x) }

## OsString

If you have x of type ...	Use this ...
String	OsString::from(x) <sup>i</sup>
CString	OsString::from(x.to_str())

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

#### PathBuf

If you have x of type ...	Use this ...
<code>String</code>	<code>PathBuf::from(x)</code> <sup>i</sup>
<code>CString</code>	<code>PathBuf::from(x.to_str())</code>
<code>OsString</code>	<code>PathBuf::from(x)</code> <sup>i</sup>
<code>PathBuf</code>	<code>x</code>
<code>Vec&lt;u8&gt;</code> <sup>1</sup>	<code>?</code>
<code>&amp;str</code>	<code>PathBuf::from(x)</code> <sup>i</sup>
<code>&amp;CStr</code>	<code>PathBuf::from(x.to_str())</code>
<code>&amp;OsStr</code>	<code>PathBuf::from(x)</code> <sup>i</sup>
<code>&amp;Path</code>	<code>PathBuf::from(x)</code> <sup>i</sup>
<code>&amp;[u8]</code> <sup>1</sup>	<code>?</code>

#### Vec<u8>

If you have x of type ...	Use this ...
<code>String</code>	<code>x.into_bytes()</code>
<code>CString</code>	<code>x.into_bytes()</code>
<code>OsString</code>	<code>?</code>
<code>PathBuf</code>	<code>?</code>
<code>Vec&lt;u8&gt;</code> <sup>1</sup>	<code>x</code>
<code>&amp;str</code>	<code>Vec::from(x.as_bytes())</code>
<code>&amp;CStr</code>	<code>Vec::from(x.to_bytes_with_nul())</code>
<code>&amp;OsStr</code>	<code>?</code>
<code>&amp;Path</code>	<code>?</code>
<code>&amp;[u8]</code> <sup>1</sup>	<code>x.to_vec()</code>

## &str

If you have x of type ...	Use this ...
String	<code>x.as_str()</code>
CString	<code>x.to_str()</code> ?
OsString	<code>x.to_str()</code> ?
PathBuf	<code>x.to_str()</code> ?
<code>Vec&lt;u8&gt;</code> <sup>1</sup>	<code>std::str::from_utf8(&amp;x)?</code>
&str	<code>x</code>
&CStr	<code>x.to_str()</code> ?
&OsStr	<code>x.to_str()</code> ?
&Path	<code>x.to_str()</code> ?
<code>&amp;[u8]</code> <sup>1</sup>	<code>std::str::from_utf8(x)?</code>

## &CStr

If you have x of type ...	Use this ...
String	<code>CString::new(x)?.as_c_str()</code>
CString	<code>x.as_c_str()</code>
OsString <sup>2</sup>	<code>x.to_str()</code> ?
PathBuf	? ,4
<code>Vec&lt;u8&gt;</code> <sup>1,5</sup>	<code>CStr::from_bytes_with_nul(&amp;x)?</code>
&str	? ,4
&CStr	<code>x</code>
&OsStr <sup>2</sup>	?
&Path	?
<code>&amp;[u8]</code> <sup>1,5</sup>	<code>CStr::from_bytes_with_nul(x)?</code>
<code>*const c_char</code> <sup>1</sup>	<code>unsafe { CStr::from_ptr(x) }</code>

## &OsStr

If you have x of type ...	Use this ...
String	<code>OsStr::new(&amp;x)</code>
CString	?
OsString	<code>x.as_os_str()</code>
PathBuf	<code>x.as_os_str()</code>

If you have x of type ...	Use this ...
<code>Vec&lt;u8&gt;</code> <sup>1</sup>	?
<code>&amp;str</code>	<code>OsStr::new(x)</code>
<code>&amp;CStr</code>	?
<code>&amp;OsStr</code>	<code>x</code>
<code>&amp;Path</code>	<code>x.as_os_str()</code>
<code>&amp;[u8]</code> <sup>1</sup>	?

`&Path`

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

`&[u8]`

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



You want	And have x	Use this ...
<code>*const c_char</code>	<code>CString</code>	<code>x.as_ptr()</code>

<sup>1</sup> Short form `x.into()` possible if type can be inferred.

<sup>2</sup> Short form `x.as_ref()` possible if type can be inferred.

<sup>3</sup> 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`).<sup>4</sup>

<sup>5</sup> Only on some platforms `std::os::<your_os>::ffi::OsStrExt` exists with helper methods to get a raw `8[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: 8[u8] = my_os_str.as_bytes();
CString::new(bytes)?
```

<sup>3</sup> The `c_char` **must** have come from a previous `CString`. If it comes from FFI see `8CStr` instead.

<sup>4</sup> No known shorthand as `x` will lack terminating `0x0`. Best way to probably go via `CString`.

<sup>5</sup> Must ensure vector actually ends with `0x0`.

## String Output

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

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> <sup>STD</sup>	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.

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>
<code>OsString</code>	<code>Debug</code>
<code>PathBuf</code>	<code>Debug</code>
<code>Vec&lt;u8&gt;</code>	<code>Debug</code>
<code>&amp;str</code>	<code>Debug</code> , <code>Display</code>
<code>&amp;CStr</code>	<code>Debug</code>
<code>&amp;OsStr</code>	<code>Debug</code>
<code>&amp;Path</code>	<code>Debug</code>
<code>&amp;[u8]</code>	<code>Debug</code>
<code>bool</code>	<code>Debug</code> , <code>Display</code>
<code>char</code>	<code>Debug</code> , <code>Display</code>
<code>u8 ... i128</code>	<code>Debug</code> , <code>Display</code>
<code>f32</code> , <code>f64</code>	<code>Debug</code> , <code>Display</code>
<code>!</code>	<code>Debug</code> , <code>Display</code>
<code>()</code>	<code>Debug</code>

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

## Formatting

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

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

Element	Meaning
<code>argument</code>	Number ( <code>0</code> , <code>1</code> , ...) or argument name, e.g., <code>print!("{}", x = 3)</code> .
<code>fill</code>	The character to fill empty spaces with (e.g., <code>0</code> ), if <code>width</code> is specified.
<code>align</code>	Left ( <code>&lt;</code> ), center ( <code>^</code> ), or right ( <code>&gt;</code> ), if <code>width</code> is specified.
<code>sign</code>	Can be <code>+</code> for sign to always be printed.
<code>#</code>	<a href="#">Alternate formatting</a> , e.g. prettify <code>Debug</code> <sup>STD</sup> formatter <code>?</code> or prefix hex with <code>0x</code> .
<code>width</code>	Minimum width ( $\geq 0$ ), padding with <code>fill</code> (default to space). If starts with <code>0</code> , zero-padded.
<code>precision</code>	Decimal digits ( $\geq 0$ ) for numerics, or max width for non-numerics.
<code>\$</code>	Interpret <code>width</code> or <code>precision</code> as argument identifier instead to allow for dynamic formatting.

Element	Meaning
<code>type</code>	<code>Debug<sup>STD</sup> (?)</code> formatting, hex (x), binary (b), octal (o), pointer (p), exp (e) ... <a href="#">see more</a> .

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







  

Full Example	Explanation
<code>println!("{}", x)</code>	Print x using <code>Display<sup>STD</sup></code> on std. out and append new line.
<code>format!("{a:.3} {b:?}", a = PI, b = 2)</code>	Convert <code>PI</code> with 3 digits, add space, b with <code>Debug<sup>STD</sup></code> , return <code>String</code> .

## Tooling

### Project Anatomy

Basic project layout, and common files and folders, as used by `cargo`.<sup>1</sup>

Entry	Code
 <code>.cargo/</code>	<b>Project-local cargo configuration</b> , may contain <code>config.toml</code> . <sup>2</sup>
 <code>benches/</code>	Benchmarks for your crate, run via <code>cargo bench</code> , requires nightly by default. * 
 <code>examples/</code>	Examples how to use your crate, they see your crate like external user would.
<code>my_example.rs</code>	Individual examples are run like <code>cargo run --example my_example</code> .
 <code>src/</code>	Actual source code for your project.
<code>main.rs</code>	Default entry point for applications, this is what <code>cargo run</code> uses.
<code>lib.rs</code>	Default entry point for libraries. This is where lookup for <code>my_crate::f()</code> starts.
 <code>tests/</code>	Integration tests go here, invoked via <code>cargo test</code> . Unit tests often stay in <code>src/</code> file.
<code>.rustfmt.toml</code>	In case you want to <a href="#">customize</a> how <code>cargo fmt</code> works.
<code>.clippy.toml</code>	Special configuration for certain <a href="#">clippy lints</a> , utilized via <code>cargo clippy</code>
<code>build.rs</code>	<b>Pre-build script</b> , <sup>3</sup> useful when compiling C / FFI, ...
<code>Cargo.toml</code>	Main <b>project manifest</b> , <sup>4</sup> Defines dependencies, artifacts ...
<code>Cargo.lock</code>	Dependency details for reproducible builds, recommended to <code>git</code> for apps, not for libs.

\* 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

## Build Scripts

## Proc Macros

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

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

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

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

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

use test::{black_box, Bencher};

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

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

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

\*[See here for list](#) of environment variables set.

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

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

use proc_macro::TokenStream;

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

```
// Cargo.toml


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

[lib]
proc-macro = true
```


Module trees and imports:

#### Module Trees

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

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

Actual **module definitions** work as follows:

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

#### Namespaces

Rust has three kinds of **namespaces**:

Namespace <i>Types</i>	Namespace <i>Functions</i>	Namespace <i>Macros</i>
<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>		
<code>union X {}</code>		
<code>struct X {}</code>		
	<code>struct X;<sup>1</sup></code>	
	<code>struct X();<sup>1</sup></code>	

<sup>1</sup> Counts in *Types* and in *Functions*.

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

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

## Cargo

Commands and tools that are good to know.

Command	Description
<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 ( <code>main.rs</code> ).
<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 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 <code>...</code> with command below)
<code>build -Z timings</code>	Show what crates caused your build to take so long, highly useful. 🕒🔥
<code>rustc -- -Zunpretty=expanded</code>	Show expanded macros. 🕒
<code>rustup doc</code>	Open offline Rust documentation (incl. the books), good on a plane!

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

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

Tool	Description
<code>cargo clippy</code>	Additional (links) catching common API misuses and unidiomatic 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** <sup>BK</sup> **macro by example** <sup>BK EX REF</sup> **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.
<code>\$x:block</code>	A block <code>{ }</code> of statements or expressions, e.g., <code>{ let x = 5; }</code>
<code>\$x:stmt</code>	A statement, e.g., <code>let x = 1 + 1;</code> , <code>String::new();</code> or <code>vec![]</code> ;
<code>\$x:expr</code>	An expression, e.g., <code>x</code> , <code>1 + 1</code> , <code>String::new()</code> or <code>vec![]</code>
<code>\$x:pat</code>	A pattern, e.g., <code>Some(t)</code> , <code>(17, 'a')</code> or <code>_</code> .
<code>\$x:ty</code>	A type, e.g., <code>String</code> , <code>usize</code> or <code>Vec&lt;u8&gt;</code> .
<code>\$x:ident</code>	An identifier, for example in <code>let x = 0;</code> the identifier is <code>x</code> .
<code>\$x:path</code>	A path (e.g. <code>foo</code> , <code>::std::mem::replace</code> , <code>transmute::&lt;_, int&gt;</code> ).
<code>\$x:literal</code>	A literal (e.g. <code>3</code> , <code>"foo"</code> , <code>b"bar"</code> , etc.).
<code>\$x:lifetime</code>	A lifetime (e.g. <code>'a</code> , <code>'static</code> , etc.).
<code>\$x:meta</code>	A meta item; the things that go inside <code>#[ ... ]</code> and <code>#![ ... ]</code> attributes.
<code>\$x:vis</code>	A visibility modifier; <code>pub</code> , <code>pub(crate)</code> , etc.
<code>\$x:tt</code>	A single token tree, <a href="#">see here</a> for more details.
<code>\$crate</code>	Special hygiene variable, crate where macros is defined. <sup>?</sup>



Inside a **doc comment** <sup>BK EX REF</sup> these work:

Within Doc Comments	Explanation
<code>``` ... ```</code>	Include a <b>doc test</b> (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. <b>Default behavior.</b>
<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.

#### #[globals]

Attributes affecting the whole crate or app:

Opt-Out's	On	Explanation
<code>#[no_std]</code>	C	Don't (automatically) import <code>std</code> <sup>STD</sup> ; use <code>core</code> <sup>STD</sup> instead. <sup>REF</sup>
<code>#[no_implicit_prelude]</code>	CM	Don't add <code>prelude</code> <sup>STD</sup> , need to manually import <code>None</code> , <code>Vec</code> , ... <sup>REF</sup>
<code>#[no_main]</code>	C	Don't emit <code>main()</code> in apps if you do that yourself. <sup>REF</sup>
Opt-In's	On	Explanation
<code>#[feature(a, b, c)]</code>	C	Rely on features that may never get stabilized, c. <a href="#">Unstable Book</a> . <sup>🔥</sup>
Builds	On	Explanation
<code>#[windows_subsystem = "x"]</code>	C	On Windows, make a console or windows app. <sup>REF</sup>
<code>#[crate_name = "x"]</code>	C	Specify current crate name, e.g., when not using <code>cargo</code> . <sup>? REF</sup>
<code>#[crate_type = "bin"]</code>	C	Specify current crate type (bin, lib, dylib, cdylib, ...). <sup>REF</sup>
<code>#[recursion_limit = "123"]</code>	C	Set <i>compile-time</i> recursion limit for deref, macros, ... <sup>REF</sup>
<code>#[type_length_limit = "456"]</code>	C	Limits maximum number of type substitutions. <sup>REF</sup>
Handlers	On	Explanation
<code>#[panic_handler]</code>	F	Make some <code>fn f(&amp;PanicInfo) → !</code> app's <b>panic handler</b> . <sup>REF</sup>

Handlers	On	Explanation
<code>#[global_allocator]</code>	S	Make static item impl. <code>GlobalAlloc</code> <sup>STD</sup> <b>global allocator</b> . <sup>REF</sup>

`#[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. <sup>REF</sup>
<code>#[path = "x.rs"]</code>	M	Get module from non-standard file. <sup>REF</sup>

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

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

Linking	On	Explanation
<code>#[no_mangle]</code>	*	Use item name directly as symbol name, instead of mangling. <sup>REF</sup>
<code>#[no_link]</code>	X	Don't link <code>extern crate</code> when only wanting macros. <sup>REF</sup>
<code>#[link(name="x", kind="y")]</code>	X	Native lib to link against when looking up symbol. <sup>REF</sup>
<code>#[link_name = "foo"]</code>	F	Name of symbol to search for resolving <code>extern fn</code> . <sup>REF</sup>
<code>#[link_section = ".sample"]</code>	FS	Section name of object file where item should be placed. <sup>REF</sup>
<code>#[export_name = "foo"]</code>	FS	Export a <code>fn</code> or <code>static</code> under a different name. <sup>REF</sup>

Linking	On	Explanation
<code>#[used]</code>	S REF	Don't optimize away <code>static</code> variable despite it looking unused.

`#[quality]`

Attributes used by Rust tools to improve code quality:

Code Patterns	On	Explanation
<code>#[allow(X)]</code>	*	Instruct <code>rustc</code> / <code>clippy</code> to ... ignore class <code>X</code> of possible issues. REF
<code>#[warn(X)]</code> <sup>1</sup>	*	... emit a warning, mixes well with <code>clippy</code> lints. 🔥 REF
<code>#[deny(X)]</code> <sup>1</sup>	*	... fail compilation. REF
<code>#[forbid(X)]</code> <sup>1</sup>	*	... fail compilation and prevent subsequent <code>allow</code> 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

<sup>1</sup> 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 <code>cargo test</code> . 🔥 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 <code>cargo bench</code> . 🚧 REF

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

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

## #[macros]


Attributes related to the creation and use of macros:

Macros By Example	On	Explanation
<code>#[macro_export]</code>	<code>!</code>	Export <code>macro_rules!</code> as <code>pub</code> on crate level <a href="#">REF</a>
<code>#[macro_use]</code>	<code>MX</code>	Let macros persist past modules; or import from <code>extern crate</code> . <a href="#">REF</a>
Proc Macros	On	Explanation
<code>#[proc_macro]</code>	<code>F</code>	Mark <code>fn</code> as <b>function-like</b> procedural macro callable as <code>m!()</code> . <a href="#">REF</a>
<code>#[proc_macro_derive(Foo)]</code>	<code>F</code>	Mark <code>fn</code> as <b>derive macro</b> which can <code>#[derive(Foo)]</code> . <a href="#">REF</a>
<code>#[proc_macro_attribute]</code>	<code>F</code>	Mark <code>fn</code> as <b>attribute macro</b> which can understand new <code>#[x]</code> . <a href="#">REF</a>
Derives	On	Explanation
<code>#[derive(X)]</code>	<code>T</code>	Let some proc macro provide a goodish <code>impl</code> of <code>trait X</code> . <a href="#">REF</a>

## #[cfg]

Attributes governing conditional compilation:

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


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

Known Options	On	Explanation
<code>#[cfg(target_arch = "x86_64")]</code>	<code>*</code>	The CPU architecture crate is compiled for. <a href="#">REF</a>
<code>#[cfg(target_feature = "avx")]</code>	<code>*</code>	Whether a particular class of instructions is available. <a href="#">REF</a>
<code>#[cfg(target_os = "macos")]</code>	<code>*</code>	Operating system your code will run on. <a href="#">REF</a>
<code>#[cfg(target_family = "unix")]</code>	<code>*</code>	Family operating system belongs to. <a href="#">REF</a>
<code>#[cfg(target_env = "msvc")]</code>	<code>*</code>	How DLLs and functions are interfaced with on OS. <a href="#">REF</a>
<code>#[cfg(target_endian = "little")]</code>	<code>*</code>	Main reason your cool new zero-cost protocol fails. <a href="#">REF</a>
<code>#[cfg(target_pointer_width = "64")]</code>	<code>*</code>	How many bits pointers, <code>usize</code> and CPU words have. <a href="#">REF</a>


Known Options	On	Explanation
<code>#[cfg(target_vendor = "apple")]</code>	*	Manufacturer of target. <a href="#">REF</a>
<code>#[cfg(debug_assertions)]</code>	*	Whether <code>debug_assert!()</code> and friends would panic. <a href="#">REF</a>
<code>#[cfg(proc_macro)]</code>	*	Whether crate compiled as proc macro. <a href="#">REF</a>
<code>#[cfg(test)]</code>	*	Whether compiled with cargo test. 🔥 <a href="#">REF</a>
<code>#[cfg(feature = "serde")]</code>	*	When your crate was compiled with feature <code>serde</code> . 🔥 <a href="#">REF</a>

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

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

For the *On* column in attributes:

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

**M** means on modules.

**F** means on functions.

**S** means on static.

T means on types.  
X means something special.  
! means on macros.  
\* means on almost any item.

# Working with Types

## Types, Traits, Generics

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

### Types & Traits

#### Types

u8

String

Device

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

Type	Values
u8	{ 0 <sub>u8</sub> , 1 <sub>u8</sub> , ... , 255 <sub>u8</sub> }
char	{ 'a', 'b', ... '🦀' }
struct S(u8, char)	{ (0 <sub>u8</sub> , 'a'), ... (255 <sub>u8</sub> , '🦀') }

Sample types and sample values.

#### Type Equivalence and Conversions

u8

⌘u8

⌘mut u8

[u8; 1]

String

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

Yes, 0 ≠ 0 (in a mathematical sense) when it comes to types! In a language sense, the operation `=(0_u8, 0_u16)` just isn't defined to prevent happy little accidents.

Type	Values
u8	{ 0 <sub>u8</sub> , 1 <sub>u8</sub> , ... , 255 <sub>u8</sub> }
u16	{ 0 <sub>u16</sub> , 1 <sub>u16</sub> , ... , 65_535 <sub>u16</sub> }
⌘u8	{ 0xffaa <sub>⌘u8</sub> , 0xffbb <sub>⌘u8</sub> , ... }
⌘mut u8	{ 0xffaa <sub>⌘mut u8</sub> , 0xffbb <sub>⌘mut u8</sub> , ... }

How values differ between types.

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

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

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

## Implementations — `impl S { }`

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

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

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

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

## Traits — `trait T { }`

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

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

Copy Trait	Clone Trait	Sized Trait
Self	Self	Self
u8	u8	char
u16	String	Port
...	...	...

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

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

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

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

Copy

```
trait Copy { }
```

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

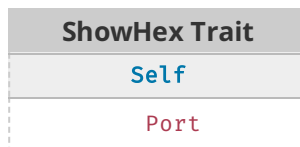
Sized

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

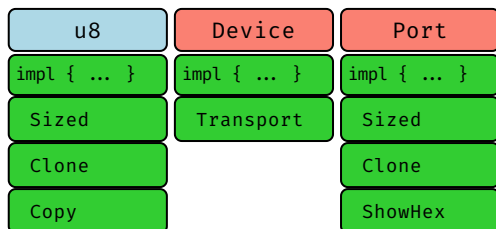
## Implementing Traits for Types — `impl T for S { }`

```
impl ShowHex for Port { ... }
```

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



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



## Traits vs. Interfaces



Eat



Venison

Eat



venison.eat()

## Interfaces

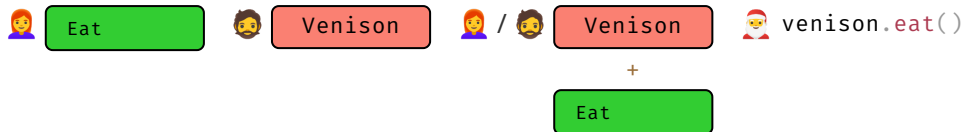
- In **Java**, Alice creates interface `Eat`.



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

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

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



## Traits

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

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

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

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

## 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&lt;u8&gt;</code>	{ [], [1], [1, 2, 3], ... }
<code>Vec&lt;char&gt;</code>	{ [], ['a'], ['x', 'y', 'z'], ... }
<code>Vec&lt;&gt;</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 3<sup>rd</sup> party to construct concrete type via parameter,
  - only then would this Vec<UserType> become real type itself.

## Generic Parameters — <T>

Vec<T>

[T; 128]

&T

&mut T

S<T>

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

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

Type vs type constructors.

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

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

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

[T; n]

S<const N>

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

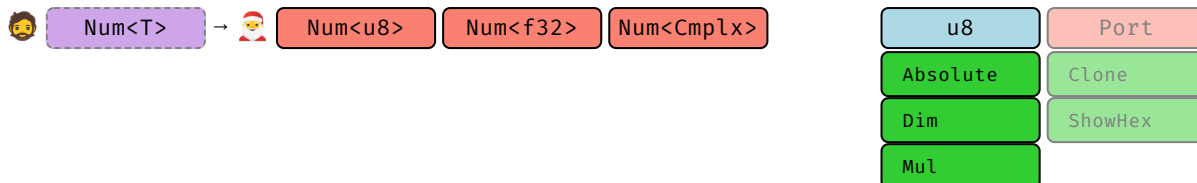
Type Constructor	Produces Family
[u8; N]	[u8; 0], [u8; 1], [u8; 2], ...
struct S<const N: usize> {}	S<1>, S<6>, S<123>, ...

Type constructors based on constant.

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

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

## Bounds (Simple) — where T: X



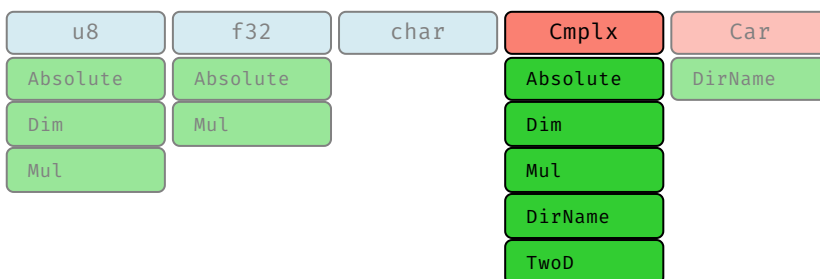
- If `T` can be any type, how can we *reason* about (write code) for such a `Num<T>`?
- Parameter **bounds**:
  - limit what types (**trait bound**) or values (**const bound**?) allowed,
  - we now can make use of these limits!
- Trait bounds act as "membership check":

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

Absolute Trait
Self
u8
u16
...

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

## Bounds (Compound) — where T: X + Y



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

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

## Implementing Families — impl<>

When we write:

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

It can be read as:

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

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

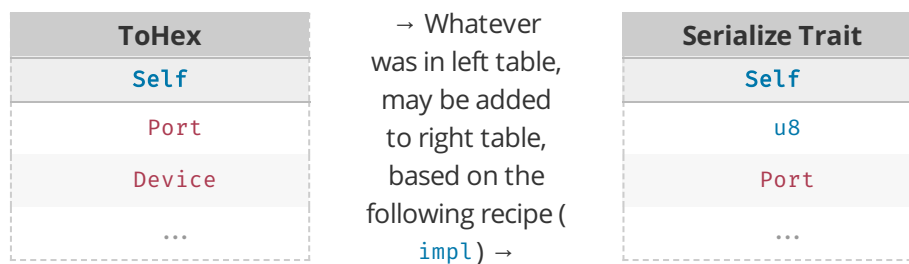
```
// If compiler encounters this, it will  
// - check `0` and `x` fulfill the membership requirements of `T`  
// - create two new version of `f`, one for `char`, another one for `u32`.  
// - based on "family implementation" provided  
s.f(0_u32);  
s.f('x');
```

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

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

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

These are called **blanket implementations**.

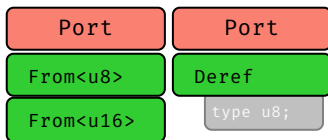


They can be neat way to give foreign types functionality in a modular way if they just implement another interface.

## Advanced Concepts

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

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



Why is that?

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

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

From	
Self	I
u16	u8
u32	u16
...	

Deref	
Self	O
Port	u8
String	str
...	

Input and output parameters.

Now here's the twist,

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

A more complex example:

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

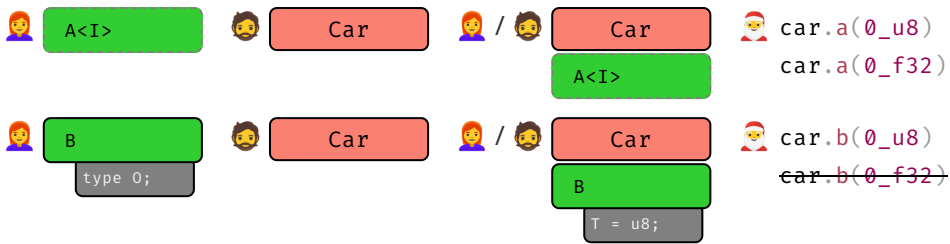
- this creates a relation relation of types named `Complex`,
- with 3 inputs (`Self` is always one) and 2 outputs, and it holds  $(\text{Self}, I1, I2) \Rightarrow (O1, O2)$

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

Complex				
Self [I]	I1	I2	O1	O2
NiceMonster	u16	String	u8	u8
NiceMonster <sup>•</sup>	u16	String	u8	u16

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

### Trait Authoring Considerations (Abstract)



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

```

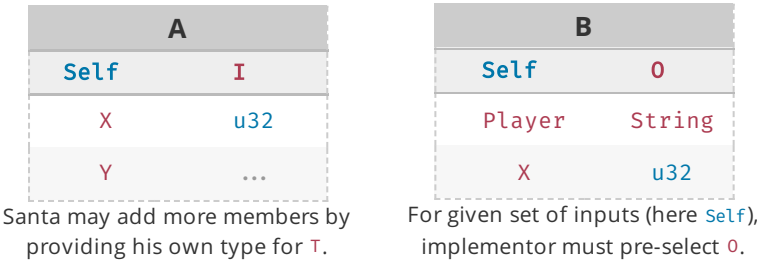
trait A<I> { }
trait B { type O; }

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

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

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

```



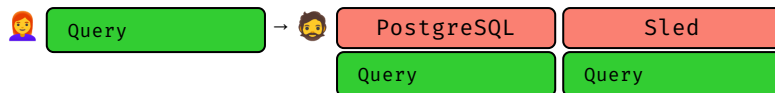
### Trait Authoring Considerations (Example)



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

## No Additional Parameters

```
trait Query {  
    fn search(&self, needle: &str);  
}  
  
impl Query for PostgreSQL { ... }  
impl Query for Sled { ... }  
  
postgres.search("SELECT ... ");
```

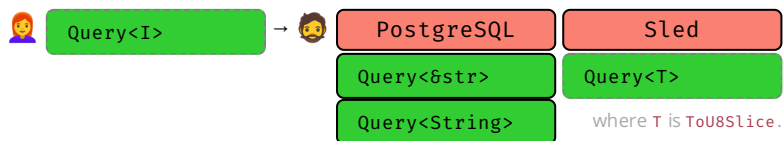


Trait author assumes:

- neither implementor nor user need to customize API.

## Input Parameters

```
trait Query<I> {  
    fn search(&self, needle: I);  
}  
  
impl Query<&str> for PostgreSQL { ... }  
impl Query<String> for PostgreSQL { ... }  
impl<T> Query<T> for Sled where T: ToU8Slice { ... }  
  
postgres.search("SELECT ... ");  
postgres.search(input.to_string());  
sled.search(file);
```



Trait author assumes:

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

## Output Parameters

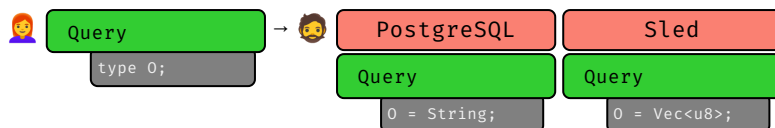
```

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

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

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

```



Trait author assumes:

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

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

## Multiple In- and Output Parameters

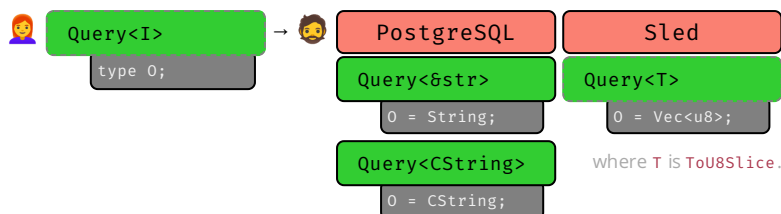
```

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

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

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

```

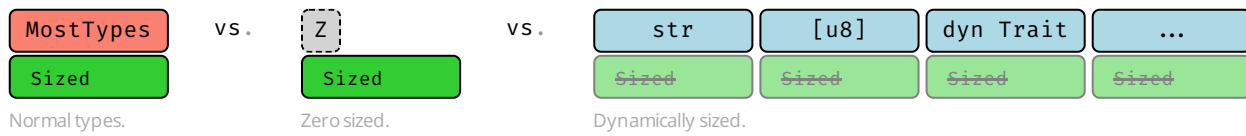


Like examples above, in particular trait author assumes:

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



## Dynamic / Zero Sized Types



- A type **T** is **Sized**<sup>STD</sup> if at compile time it is known how many bytes it occupies, **u8** and **8[u8]** are, **[u8]** isn't.
- Being **Sized** means **impl Sized for T {}** holds. Happens automatically and cannot be user impl'ed.
- Types not **Sized** are called **dynamically sized types**<sup>BK NOM REF</sup> (DSTs), sometimes **unsized**.
- Types without data are called **zero sized types**<sup>NOM</sup> (ZSTs), do not occupy space.

Example	Explanation
<pre>struct A { x: u8 }</pre>	Type <b>A</b> is sized, i.e., <b>impl Sized for A</b> holds, this is a 'regular' type.
<pre>struct B { x: [u8] }</pre>	Since <b>[u8]</b> is a DST, <b>B</b> in turn becomes DST, i.e., does not <b>impl Sized</b> .
<pre>struct C&lt;T&gt; { x: T }</pre>	Type params <b>have</b> implicit <b>T: Sized</b> bound, e.g., <b>C&lt;A&gt;</b> is valid, <b>C&lt;B&gt;</b> is not.
<pre>struct D&lt;T: ?Sized&gt; { x: T }</pre>	Using <b>?Sized</b> <sup>REF</sup> allows opt-out of that bound, i.e., <b>D&lt;B&gt;</b> is also valid.
<pre>struct E;</pre>	Type <b>E</b> is zero-sized (and also sized) and will not consume memory.
<pre>trait F { fn f(&amp;self); }</pre>	Traits <b>do not have</b> an implicit <b>Sized</b> bound, i.e., <b>impl F for B {}</b> is valid.
<pre>trait F: Sized {}</pre>	Traits can however opt into <b>Sized</b> via supertraits. <sup>1</sup>
<pre>trait G { fn g(self); }</pre>	For <b>Self</b> -like params DST <b>impl</b> may still fail as params can't go on stack.

### ?Sized



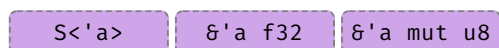
```
struct S<T> { ... }
```

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



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

## Generics and Lifetimes — <'a>



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

`S<'a>` → `S<'auto>` `S<'static>`

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

```
// `a is free parameter here (user can pass any specific lifetime)
struct S<'a> {
    x: 8'a u32
}

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

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

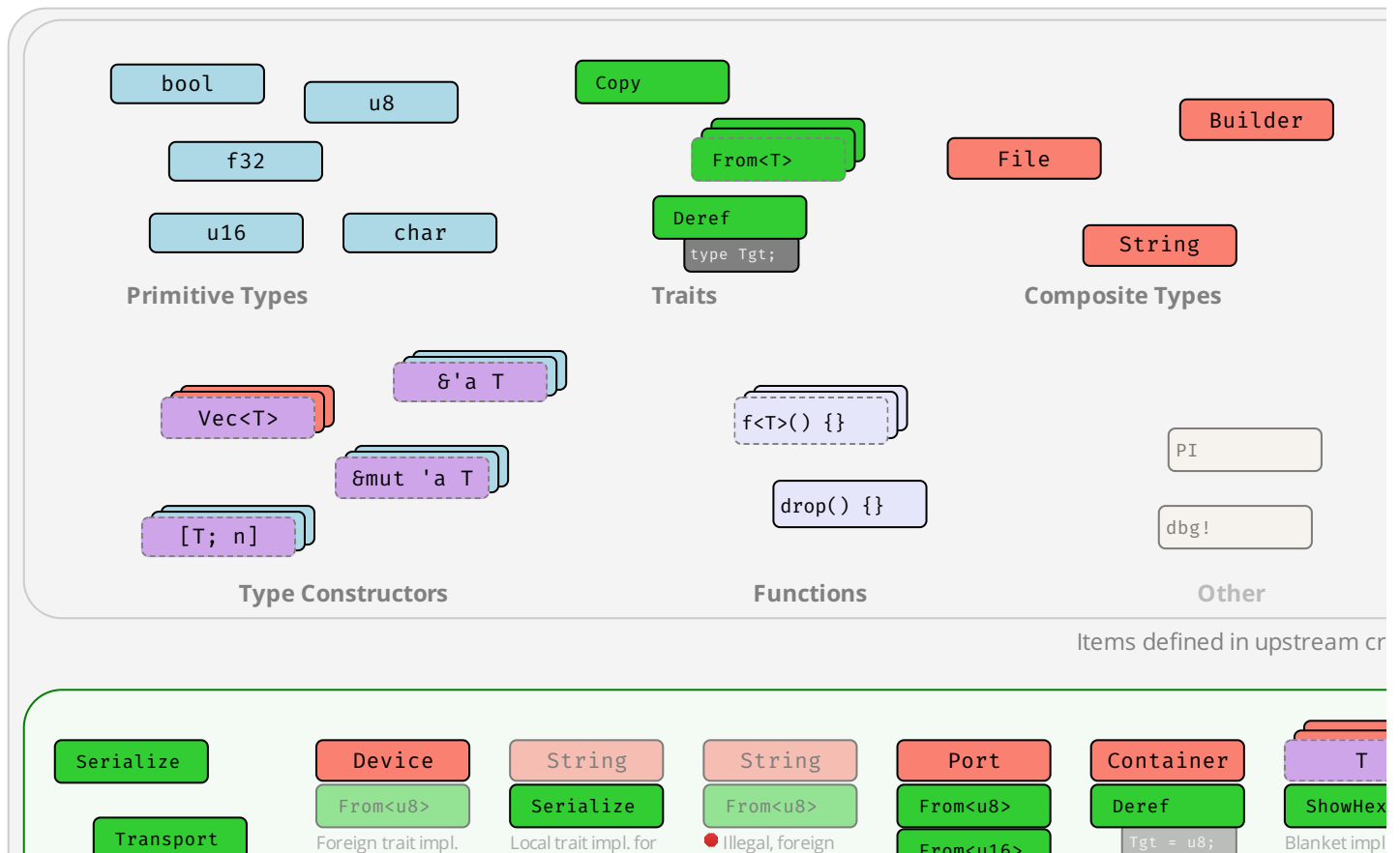
\* There are subtle differences, for example you can create an explicit instance `0` of a type `u32`, but with the exception of `'static` you can't really create a lifetime, e.g., "lines 80 - 100", the compiler will do that for you. [↗](#)

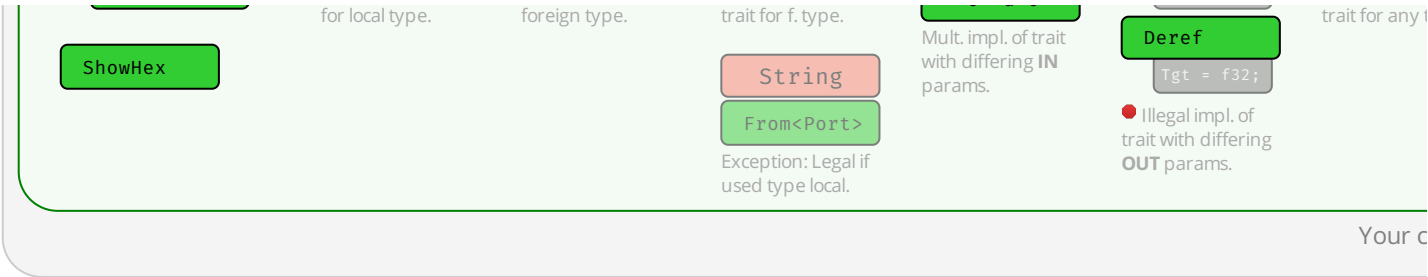
Note to self and TODO: that analogy seems somewhat flawed, as if `S<'a>` is to `S<'static>` like `S<T>` is to `S<u32>`, then `'static` would be a *type*; but then what's the value of that type?

Examples expand by clicking.

## Type Zoo

A visual overview of types and traits in crates.





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

## 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>B is exactly A</b> .
Computation	Create and manipulate instance of <b>B</b> by <b>writing code</b> transforming data.
Casts	<b>On-demand</b> conversion between types where caution is advised.
Coercions	<b>Automatic</b> conversion within ' <i>weakening ruleset</i> '. <sup>1</sup>
Subtyping	<b>Automatic</b> conversion within ' <i>same-layout-different-lifetimes ruleset</i> '. <sup>1</sup>

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

Computation (Traits)

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

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

Trait	Example	Trait implies ...
<code>impl From&lt;A&gt; for B {}</code>	<code>a.into()</code>	<i>Obvious</i> , always-valid relation.
<code>impl TryFrom&lt;A&gt; for B {}</code>	<code>a.try_into()?</code>	<i>Obvious</i> , sometimes-valid relation.
<code>impl Deref for A {}</code>	<code>*a</code>	<b>A</b> is smart pointer carrying <b>B</b> ; also enables coercions.
<code>impl AsRef&lt;B&gt; for A {}</code>	<code>a.as_ref()</code>	<b>A</b> can be <i>viewed</i> as <b>B</b> .

Trait	Example	Trait implies ...
<code>impl AsMut&lt;B&gt; for A {}</code>	<code>a.as_mut()</code>	A can be mutably viewed as B.
<code>impl Borrow&lt;B&gt; for A {}</code>	<code>a.borrow()</code>	A has borrowed <i>analog</i> B (behaving same under Eq, ...).
<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**.<sup>NOM</sup>

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. <sup>†</sup>
<code>enum</code> w/o fields	<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>	
<code>&amp;[T; N]</code>	<code>*const T</code>	<code>my_ref as *const u8</code>	
<code>fn( ... )</code>	<code>Ptr</code>	<code>f as *const u8</code>	If <code>Ptr</code> is <code>Sized</code> .
<code>fn( ... )</code>	<code>Integer</code>	<code>f as usize</code>	

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

- `Ptr` any `*const T` or `*mut T`;
- `Integer` any countable `u8 ... i128`;
- `Number` any `Integer`, `f32`, `f64`.

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

## Coercions

```
fn f(x: A) → B {
  x
}
```

Automatically **weaken** type **A** to **B**; types can be *substantially*<sup>1</sup> different. <sup>NOM</sup>

A	B	Explanation
<code>&amp;mut T</code>	<code>&amp;T</code>	<b>Pointer weakening.</b>
<code>&amp;mut T</code>	<code>*mut T</code>	-
<code>&amp;T</code>	<code>*const T</code>	-
<code>*mut T</code>	<code>*const T</code>	-
<code>&amp;T</code>	<code>&amp;U</code>	<b>Deref</b> , if <code>impl Deref&lt;Target=U&gt; for T</code> .
<code>T</code>	<code>U</code>	<b>Unsizing</b> , if <code>impl CoerceUnsize&lt;U&gt; for T</code> . <sup>2</sup> ⚡
<code>T</code>	<code>V</code>	<b>Transitivity</b> , 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>	<b>Non-capturing closure</b> , to equivalent <code>fn</code> pointer.

<sup>1</sup> *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**.

<sup>2</sup> 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** <sup>NOM</sup> - subtyping **examples**:

A(subtype)	B(supertype)	Explanation
<code>&amp;'static u8</code>	<code>&amp;'a u8</code>	Valid, <i>forever</i> -pointer is also <i>transient</i> -pointer.
<code>&amp;'a u8</code>	<code>&amp;'static u8</code>	● Invalid, transient should not be forever.
<code>&amp;'a &amp;'b u8</code>	<code>&amp;'a &amp;'b u8</code>	Valid, same thing. <b>But now things get interesting. Read on.</b>
<code>&amp;'a &amp;'static u8</code>	<code>&amp;'a &amp;'b u8</code>	Valid, <code>&amp;'static u8</code> is also <code>&amp;'b u8</code> ; <b>covariant</b> inside <code>&amp;</code> .
<code>&amp;'a mut &amp;'static u8</code>	<code>&amp;'a mut &amp;'b u8</code>	● Invalid and surprising; <b>invariant</b> inside <code>&amp;mut</code> .
<code>Box&lt;&amp;'a static&gt;</code>	<code>Box&lt;&amp;'a u8&gt;</code>	Valid, box with forever is also box with transient; covariant.
<code>Box&lt;&amp;'a u8&gt;</code>	<code>Box&lt;&amp;'static u8&gt;</code>	● Invalid, box with transient may not be with forever.
<code>Box&lt;&amp;'a mut u8&gt;</code>	<code>Box&lt;&amp;'a u8&gt;</code>	● ⚡ Invalid, see table below, <code>&amp;mut u8</code> never <i>was</i> a <code>&amp;u8</code> .

A(subtype)	B(supertype)	Explanation
<code>Cell&lt;&amp;'a static&gt;</code>	<code>Cell&lt;&amp;'a u8&gt;</code>	● Invalid, cells are <b>never</b> something else; invariant.
<code>fn(&amp;'static u8)</code>	<code>fn(&amp;'u8 u8)</code>	● If <code>fn</code> needs forever it may choke on transients; <b>contravar.</b>
<code>fn(&amp;'a u8)</code>	<code>fn(&amp;'static u8)</code>	But sth. that eats transients <b>can be(!)</b> sth. that eats forevers.
<code>for&lt;'r&gt; fn(&amp;'r u8)</code>	<code>fn(&amp;'a u8)</code>	Higher-ranked type <code>for&lt;'r&gt; fn(&amp;'r u8)</code> is also <code>fn(&amp;'a u8)</code> .

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

A	B	Explanation
<code>u16</code>	<code>u8</code>	● <b>Obviously invalid</b> ; <code>u16</code> should never automatically be <code>u8</code> .
<code>u8</code>	<code>u16</code>	● Invalid <b>by design</b> ; types w. different data still never subtype even if they <i>could</i> .
<code>&amp;'a mut u8</code>	<code>&amp;'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** <sup>NOM</sup> - subtyping **variance rules**:

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

Construct <sup>1</sup>	'a	T	U
<code>&amp;'a T</code>	covariant	covariant	
<code>&amp;'a mut T</code>	covariant	invariant	
<code>Box&lt;T&gt;</code>		covariant	
<code>Cell&lt;T&gt;</code>		invariant	
<code>fn(T) → U</code>		<b>contravariant</b>	covariant
<code>*const T</code>		covariant	
<code>*mut T</code>		invariant	

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

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

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

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

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

## Coding Guides

### Idiomatic Rust

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

Idiom	Code
Think in Expressions	<pre>x = if x { a } else { b };  x = 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>x = 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>
Provide Builders	<pre>Car::new("Model T").hp(20).build();</pre>
Split Implementations	<p>Generic types <code>S&lt;T&gt;</code> can have a separate <code>impl</code> per <code>T</code>.</p> <p>Rust doesn't have OO, but with separate <code>impl</code> you can get specialization.</p>
Unsafe	<p>Avoid <code>unsafe {}</code>, often safer, faster solution without it. Exception: FFI.</p>
Implement Traits	<pre>#[derive(Debug, Copy, ...)]</pre> and custom <code>impl</code> where needed.
Tooling	<p>With <code>clippy</code> you can improve your code quality.</p> <p>Formatting with <code>rustfmt</code> helps others to read your code.</p> <p>Add <b>unit tests</b> <sup>BK</sup> (<code>#[test]</code>) to ensure your code works.</p> <p>Add <b>doc tests</b> <sup>BK</sup> (<code>``` my_api::f() ```</code>) to ensure docs match code.</p>
Documentation	<p>Annotate your APIs with doc comments that can show up on <a href="#">docs.rs</a>.</p> <p>Don't forget to include a <b>summary sentence</b> and the <b>Examples</b> heading.</p> <p>If applicable: <b>Panics, Errors, Safety, Abort</b> and <b>Undefined Behavior</b>.</p>

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

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&lt;Output=_.&gt;</code> . <sup>STD</sup>
<code>async fn f() {}</code>	Function <code>f</code> returns an <code>impl Future&lt;Output=()&gt;</code> .
<code>async fn f() → S {}</code>	Function <code>f</code> returns an <code>impl Future&lt;Output=S&gt;</code> .
<code>async { x }</code>	Transforms <code>{ x }</code> into an <code>impl Future&lt;Output=X&gt;</code> .
<code>let sm = f();</code>	Calling <code>f()</code> that is <code>async</code> will <b>not</b> execute <code>f</code> , but produce state machine <code>sm</code> . <sup>1 2</sup>
<code>sm = async { g() };</code>	Likewise, does <b>not</b> 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> . <sup>3 4</sup>
<code>sm.await</code>	Inside an <code>async {}</code> , run <code>sm</code> until complete. Yield to runtime if <code>sm</code> not ready.

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

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

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

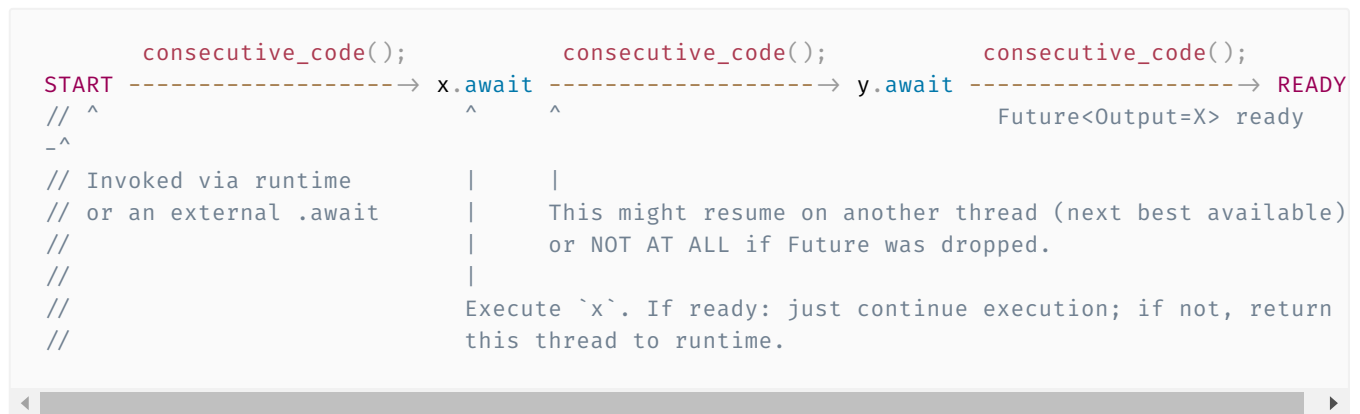
<sup>4</sup> Rust doesn't come with runtime, need external crate instead, e.g., [async-std](#) or [tokio 0.2+](#). Also, more helpers in [futures crate](#).

## 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 :



## Caveats

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



Constructs <sup>1</sup>	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 <b>not return</b> if <code>Future</code> dropped while waiting. <sup>2</sup>
<code>Rc::new(); x.await; rc();</code>	Non-Send types prevent <code>impl Future</code> from being <code>Send</code> ; less compatible.

<sup>1</sup> 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.

<sup>2</sup> 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` <sup>STD</sup> also implements `FnMut` and `FnOnce`. Likewise a closure that implements `FnMut` <sup>STD</sup> also implements `FnOnce`. <sup>STD</sup>

From a call site perspective that means:

Signature	Function <code>g</code> can call ...	Function <code>g</code> accepts ...
<code>g&lt;F: FnOnce()&gt;(f: F)</code>	... <code>f()</code> once.	<code>Fn</code> , <code>FnMut</code> , <code>FnOnce</code>
<code>g&lt;F: FnMut()&gt;(mut f: F)</code>	... <code>f()</code> multiple times.	<code>Fn</code> , <code>FnMut</code>
<code>g&lt;F: Fn()&gt;(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>   { &amp;mut s; }</code>	<code>FnOnce</code> , <code>FnMut</code>	Allows <code>g()</code> to change caller's local state <code>s</code> .
<code>   { &amp;s; }</code>	<code>FnOnce</code> , <code>FnMut</code> , <code>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.

## Unsafe Code

- Code marked `unsafe` has special permissions, e.g., to deref raw pointers, or invoke other `unsafe` functions.
- Along come special **promises the author *must* uphold to the compiler**, and the compiler *will* trust you.
- By itself `unsafe` code is not bad, but dangerous, and needed for FFI or exotic data structures.

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

## Undefined Behavior

### Undefined Behavior (UB)

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

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

## Unsound Code

### Unsound Code

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

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

## Responsible use of Unsafe

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

## API Stability

When updating an API, these changes can break client code.<sup>RFC</sup> 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]` <sup>REF</sup>
- 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>`).

### Signatures in Type Definitions

- 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.*