```
struct S;
*Drop::drop(&mut T)
size
align
*Trait::f(&T, ...)
*Trait::g(&T, ...)
```

(static vtable)

Rust Language Cheat Sheet

20. November 2021

Contains clickable links to **The Book** BK, **Rust by Example** EX, **Std Docs** STD, **Nomicon** NOM, **Reference** REF. +

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

Data Structures

Data types and memory locations defined via keywords.

Example	Explanation
`struct S {}`	Define a **struct** BK EX STD REF with named fields.
`struct S { x: T }`	Define struct with named field `x` of type `T`.
`struct S (T);`	Define "tupled" struct with numbered field `.0` of type `T`.
`struct S;`	Define **zero sized** NOM unit struct. Occupies no space, optimized away.
`enum E {}`	Define an **enum**, BK EX REF *c.* algebraic data types, tagged unions.
`enum E { A, B (), C {} }`	Define variants of enum; can be unit- `A`, tuple- `B ()` and struct-like `C{}`.
`enum E { A = 1 }`	If variants are only unit-like, allow discriminant values, e.g., for FFI.
`union U {}`	Unsafe C-like **union** REF for FFI compatibility.
`static X: T = T();`	**Global variable** BK EX REF with `'static` lifetime, single memory location.
`const X: T = T();`	Defines **constant**, BK EX REF copied into a temporary when used.
`let x: T;`	Allocate `T` bytes on stack[1] bound as `x`. Assignable once, not mutable.
`let mut x: T;`	Like `let`, but allow for **mutability** BK EX and mutable borrow.[2]
`x = y;`	Moves `y` to `x`, invalidating `y` if `T` is not **`Copy`**, STD and copying `y` otherwise.

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

[2] Technically *mutable* and *immutable* are misnomer. Immutable binding or shared reference may still contain Cell STD, giving *interior mutability*.

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

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

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

* For now,RFC pending completion of tracking issue.

References & Pointers

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

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

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

Functions & Behavior

Define units of code and their abstractions.

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

Control Flow

Control execution within a function.

Example	Explanation

Example	Explanation
`while x {}`	**Loop**, [REF] run while expression `x` is true.
`loop {}`	**Loop indefinitely** [REF] until `break`. Can yield value with `break x`.
`for x in iter {}`	Syntactic sugar to loop over **iterators**. [BK] [STD] [REF]
`if x {} else {}`	**Conditional branch** [REF] if expression is true.
`'label: loop {}`	**Loop label**, [EX] [REF] useful for flow control in nested loops.
`break`	**Break expression** [REF] to exit a loop.
`break x`	Same, but make `x` value of the loop expression (only in actual `loop`).
`break 'label`	Exit not only this loop, but the enclosing one marked with `'label`.
`break 'label x`	Same, but make `x` the value of the enclosing loop marked with `'label`.
`continue`	**Continue expression** [REF] to the next loop iteration of this loop.
`continue 'label`	Same but instead of this loop, enclosing loop marked with 'label.
`x?`	If `x` is Err or None, **return and propagate**. [BK] [EX] [STD] [REF]
`x.await`	Only works inside `async`. Yield flow until **`Future`** [STD] or Stream `x` ready. [REF] ['18]
`return x`	Early return from function. More idiomatic way is to end with expression.
`f()`	Invoke callable `f` (e.g., a function, closure, function pointer, `Fn`, …).
`x.f()`	Call member function, requires `f` takes `self`, `&self`, … as first argument.
`X::f(x)`	Same as `x.f()`. Unless `impl Copy for X {}`, `f` can only be called once.
`X::f(&x)`	Same as `x.f()`.
`X::f(&mut x)`	Same as `x.f()`.
`S::f(&x)`	Same as `x.f()` if `X` derefs to `S`, i.e., `x.f()` finds methods of `S`.
`T::f(&x)`	Same as `x.f()` if `X impl T`, i.e., `x.f()` finds methods of `T` if in scope.
`X::f()`	Call associated function, e.g., `X::new()`.
`<X as T>::f()`	Call trait method `T::f()` implemented for `X`.

Organizing Code

Segment projects into smaller units and minimize dependencies.

Example	Explanation
`mod m {}`	Define a **module**, [BK] [EX] [REF] get definition from inside `{}`. ↓
`mod m;`	Define a module, get definition from `m.rs` or `m/mod.rs`. ↓
`a::b`	Namespace **path** [EX] [REF] to element `b` within `a` (`mod`, `enum`, …).
`::b`	Search `b` relative to crate root.
`crate::b`	Search `b` relative to crate root. ['18]
`self::b`	Search `b` relative to current module.
`super::b`	Search `b` relative to parent module.
`use a::b;`	**Use** [EX] [REF] `b` directly in this scope without requiring `a` anymore.
`use a::{b, c};`	Same, but bring `b` and `c` into scope.
`use a::b as x;`	Bring `b` into scope but name `x`, like `use std::error::Error as E`.
`use a::b as _;`	Bring `b` anonymously into scope, useful for traits with conflicting names.
`use a::*;`	Bring everything from `a` in, only recommended if `a` is some **prelude**.
`pub use a::b;`	Bring `a::b` into scope and reexport from here.

Example	Explanation
`pub T`	"Public if parent path is public" **visibility** BK REF for `T`.
`pub(crate) T`	Visible at most[1] in current crate.
`pub(super) T`	Visible at most[1] in parent.
`pub(self) T`	Visible at most[1] in current module (default, same as no `pub`).
`pub(in a::b) T`	Visible at most[1] in ancestor `a::b`.
`extern crate a;`	Declare dependency on external **crate**; BK REF just `use a::b` in '18.
`extern "C" {}`	*Declare* external dependencies and ABI (e.g., `"C"`) from **FFI**. BK EX NOM REF
`extern "C" fn f() {}`	*Define* function to be exported with ABI (e.g., `"C"`) to FFI.

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

Type Aliases and Casts

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

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

Macros & Attributes

Code generation constructs expanded before the actual compilation happens.

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

Inside Macros	Explanation
`$x:ty`	Macro capture (here a type); see **tooling directives** ↓ for details.
`$x`	Macro substitution, e.g., use the captured `$x:ty` from above.
`$(x),*`	Macro repetition "zero or more times" in macros by example.
`$(x),?`	Same, but "zero or one time".
`$(x),+`	Same, but "one or more times".
`$(x)<<+`	In fact separators other than `,` are also accepted. Here: `<<`.

Pattern Matching

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

Example	Explanation
`match m {}`	Initiate **pattern matching**, BK EX REF then use match arms, *c.* next table.
`let S(x) = get();`	Notably, `let` also **destructures** EX similar to the table below.
`let S { x } = s;`	Only `x` will be bound to value `s.x`.
`let (_, b, _) = abc;`	Only `b` will be bound to value `abc.1`.
`let (a, ..) = abc;`	Ignoring 'the rest' also works.
`let (.., a, b) = (1, 2);`	Specific bindings take precedence over 'the rest', here `a` is `1`, `b` is `2`.
`let s @ S { x } = get();`	Bind `s` to `S` while `x` is bound to `s.x`, **pattern binding**, BK EX REF *c.* below
`let w @ t @ f = get();`	Stores 3 copies of `get()` result in each `w`, `t`, `f`. Please don't do this.
`let Some(x) = get();`	**Won't** work if pattern can be **refuted**, REF use `if let` instead.
`if let Some(x) = get() {}`	Branch if pattern can be assigned (e.g., `enum` variant), syntactic sugar. *
`while let Some(x) = get() {}`	Equiv.; here keep calling `get()`, run `{}` as long as pattern can be assigned.
`fn f(S { x }: S)`	Function parameters also work like `let`, here `x` bound to `s.x` of `f(s)`.

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

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

Within Match Arm	Explanation
`E::A => {}`	Match enum variant `A`, *c.* **pattern matching**. BK EX REF
`E::B ( .. ) => {}`	Match enum tuple variant `B`, wildcard any index.
`E::C { .. } => {}`	Match enum struct variant `C`, wildcard any field.
`S { x: 0, y: 1 } => {}`	Match struct with specific values (only accepts `s` with `s.x` of `0` and `s.y` of `1`).
`S { x: a, y: b } => {}`	Match struct with *any*(!) values and bind `s.x` to `a` and `s.y` to `b`.
`S { x, y } => {}`	Same, but shorthand with `s.x` and `s.y` bound as `x` and `y` respectively.
`S { .. } => {}`	Match struct with any values.
`D => {}`	Match enum variant `E::D` if `D` in `use`.
`D => {}`	Match anything, bind `D`; possibly false friend of `E::D` if `D` not in `use`.
`_ => {}`	Proper wildcard that matches anything / "all the rest".
`0 | 1 => {}`	Pattern alternatives, **or-patterns**. RFC
`E::A | E::Z`	Same, but on enum variants.
`E::C {x} | E::D {x}`	Same, but bind `x` if all variants have it.
`Some(A | B)`	Same, can also match alternatives deeply nested.
`(a, 0) => {}`	Match tuple with any value for `a` and `0` for second.
`[a, 0] => {}`	**Slice pattern**, REF match array with any value for `a` and `0` for second.
`[1, ..] => {}`	Match array starting with `1`, any value for rest; **subslice pattern**. ?
`[1, .., 5] => {}`	Match array starting with `1`, ending with `5`.
`[1, x @ .., 5] => {}`	Same, but also bind `x` to slice representing middle (*c.* pattern binding).
`[a, x @ .., b] => {}`	Same, but match any first, last, bound as `a`, `b` respectively.
`1 .. 3 => {}`	**Range pattern**, BK REF here matches `1` and `2`; partially unstable.
`1 ..= 3 => {}`	Inclusive range pattern, matches `1`, `2` and `3`.
`1 .. => {}`	Open range pattern, matches `1` and any larger number.
`x @ 1..=5 => {}`	Bind matched to `x`; **pattern binding**, BK EX REF here `x` would be `1`, `2`, ... or `5`.

Within Match Arm	Explanation
`Err(x @ Error {..}) => {}`	Also works nested, here `x` binds to `Error`, esp. useful with `if` below.
`S { x } if x > 10 => {}`	Pattern **match guards**, [BK] [EX] [REF] condition must be true as well to match.

Generics & Constraints

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

Example	Explanation
`S<T>`	A **generic** [BK] [EX] type with a type parameter (`T` is placeholder name here).
`S<T: R>`	Type short hand **trait bound** [BK] [EX] specification (`R` *must* be actual trait).
`T: R, P: S`	**Independent trait bounds** (here one for `T` and one for `P`).
`T: R, S`	Compile error, you probably want compound bound `R + S` below.
`T: R + S`	**Compound trait bound**, [BK] [EX] `T` must fulfill `R` and `S`.
`T: R + 'a`	Same, but w. lifetime. `T` must fulfill `R`, if `T` has lifetimes, must outlive `'a`.
`T: ?Sized`	Opt out of a pre-defined trait bound, here `Sized`. [?]
`T: 'a`	Type **lifetime bound**; [EX] if T has references, they must outlive `'a`.
`T: 'static`	Same; does esp. *not* mean value `t` *will* live `'static`, only that it could.
`'b: 'a`	Lifetime `'b` must live at least as long as (i.e., *outlive*) `'a` bound.
`S<const N: usize>`	**Generic const bound**; [?] user of type `S` can provide constant value `N`.
`S<10>`	Where used, const bounds can be provided as primitive values.
`S<{5+5}>`	Expressions must be put in curly brackets.
`S<T> where T: R`	Almost same as `S<T: R>` but more pleasant to read for longer bounds.
`S<T> where u8: R<T>`	Also allows you to make conditional statements involving *other* types.
`S<T = R>`	**Default type parameter** [BK] for associated type.
`S<'_>`	Inferred **anonymous lifetime**; asks compiler to *'figure it out'* if obvious.
`S<_>`	Inferred **anonymous type**, e.g., as `let x: Vec<_> = iter.collect()`
`S::<T>`	**Turbofish** [STD] call site type disambiguation, e.g. `f::<u32>()`.
`trait T<X> {}`	A trait generic over `X`. Can have multiple `impl T for S` (one per `X`).
`trait T { type X; }`	Defines **associated type** [BK] [REF] [RFC] `X`. Only one `impl T for S` possible.
`type X = R;`	Set associated type within `impl T for S { type X = R; }`.
`impl<T> S<T> {}`	Implement functionality for any `T` in `S<T>`, here `T` type parameter.
`impl S<T> {}`	Implement functionality for exactly `S<T>`, here `T` specific type (e.g., `S<u32>`).
`fn f() -> impl T`	**Existential types**, [BK] returns an unknown-to-caller `S` that `impl T`.
`fn f(x: &impl T)`	Trait bound,"**impl traits**", [BK] somewhat similar to `fn f<S:T>(x: &S)`.
`fn f(x: &dyn T)`	Marker for **dynamic dispatch**, [BK] [REF] `f` will not be monomorphized.
`fn f() where Self: R;`	In `trait T {}`, make `f` accessible only on types known to also `impl R`.
`fn f() where Self: Sized;`	Using `Sized` can opt `f` out of `dyn T` trait object vtable, enabling trait obj.
`fn f() where Self: R {}`	Other `R` useful w. dflt. methods (non dflt. would need be impl'ed anyway).

Higher-Ranked Items

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

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

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

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

Strings & Chars

Rust has several ways to create textual values.

Example	Explanation
`"..."`	**String literal**, [REF, 1] UTF-8, will interpret `\n` as *line break* `0xA`, …
`r"..."`	**Raw string literal**. [REF, 1] UTF-8, won't interpret `\n`, …
`r#"..."#`	Raw string literal, UTF-8, but can also contain `"`. Number of `#` can vary.
`b"..."`	**Byte string literal**; [REF, 1] constructs ASCII `[u8]`, not a string.
`br"..."`, `br#"..."#`	Raw byte string literal, ASCII `[u8]`, combination of the above.
`'🦀'`	**Character literal**, [REF] fixed 4 byte unicode '**char**'. [STD]
`b'x'`	ASCII **byte literal**. [REF]

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

Documentation

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

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

Tooling directives ↓ 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
`!`	Always empty **never type**. [BK] [EX] [STD] [REF]
`_`	Unnamed variable binding, e.g., `|x, _| {}`.
`let _ = x;`	Unnamed assignment is no-op, does **not** move out `x` or preserve scope!
`_x`	Variable binding explicitly marked as unused.
`1_234_567`	Numeric separator for visual clarity.
`1_u8`	Type specifier for **numeric literals** [EX] [REF] (also `i8`, `u16`, ...).
`0xBEEF`, `0o777`, `0b1001`	Hexadecimal (`0x`), octal (`0o`) and binary (`0b`) integer literals.
`r#foo`	A **raw identifier** [BK] [EX] for edition compatibility.
`x;`	**Statement** [REF] terminator, *c.* **expressions** [EX] [REF]

Common Operators

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

Behind the Scenes

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

The Abstract Machine

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

Overview | **Misconceptions**

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**.

Language Sugar

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

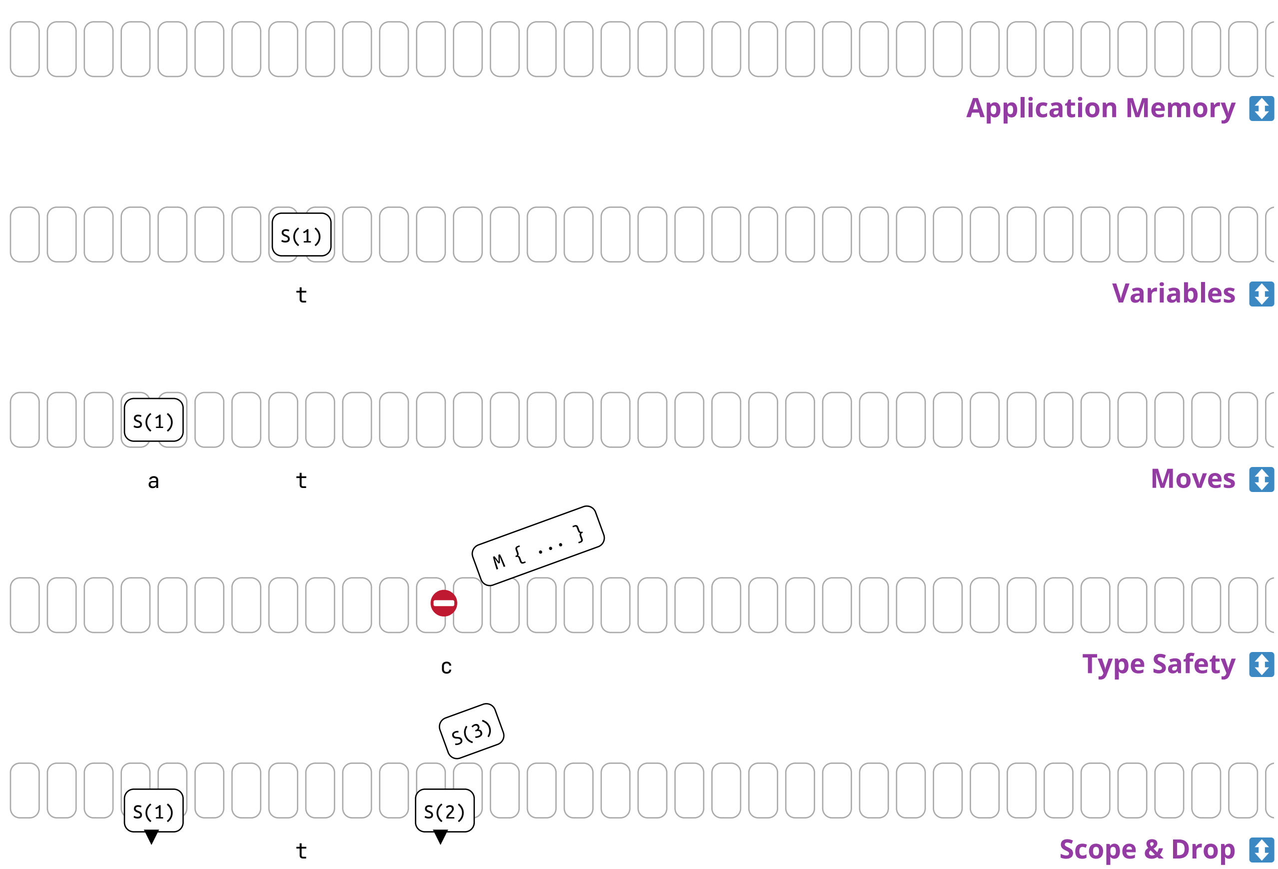
Name	Description
Coercions [NOM]	*Weakens* types to match signature, e.g., `&mut T` to `&T`; c. *type conversions*. ↓
Deref [NOM] 🔗	Derefs `x: T` until `*x`, `**x`, … compatible with some target `S`.
Prelude [STD]	Automatic import of basic items, e.g., `Option`, `drop`, …
Reborrow	Since `x: &mut T` can't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

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 | **Call Stack** | **References & Pointers** | **Lifetime Basics** | **Lifetimes in Functions**



Examples expand by clicking.

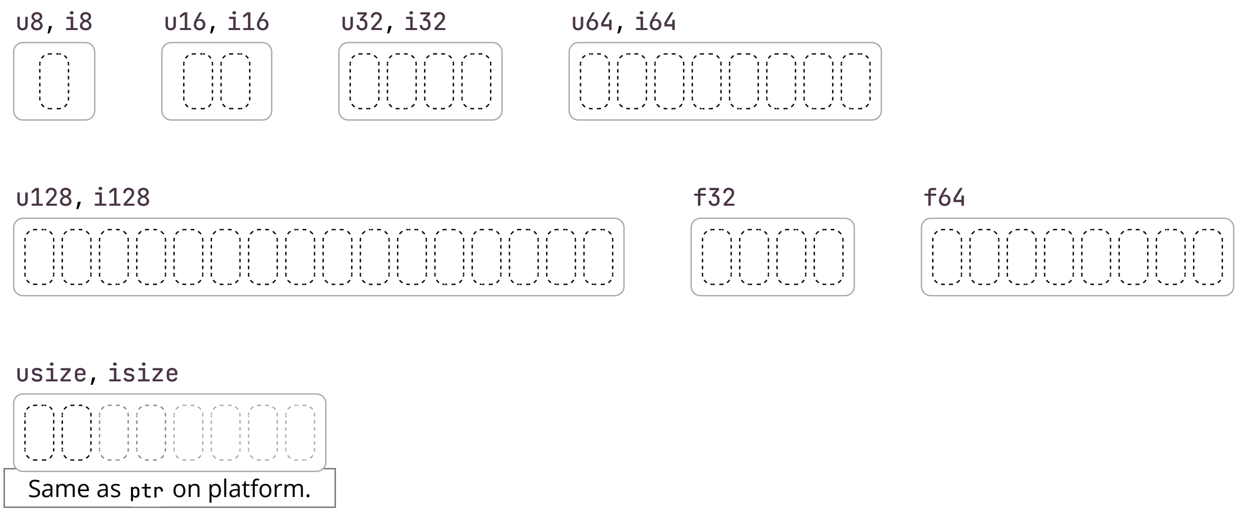
Data Layout

Memory representations of common data types.

Basic Types

Essential types built into the core of the language.

Numeric Types REF



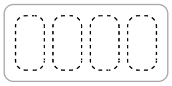
Unsigned Types | **Signed Types** | **Float Types** | **Casting Pitfalls** | **Arithmetical Pitfalls**

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`.

[1] Expression `_100` means anything that might contain the value `100`, e.g., `100_i32`, but is opaque to compiler.
[d] Debug build.
[r] Release build.

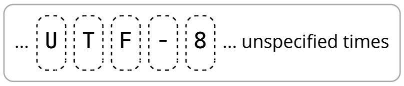
Textual Types REF

`char`



Any UTF-8 scalar.

`str`



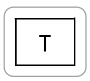
Rarely seen alone, but as `&str` instead.

Basics | **Usage** | **Encoding**

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

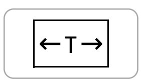
Custom Types

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



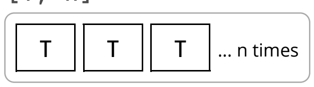
Sized ↓

`T: ?Sized`



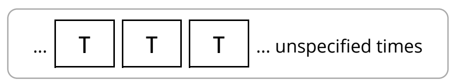
Maybe DST ↓

`[T; n]`



Fixed array of `n` elements.

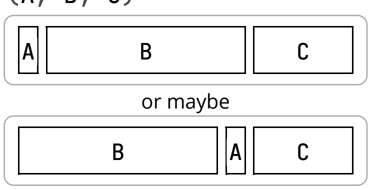
`[T]`



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

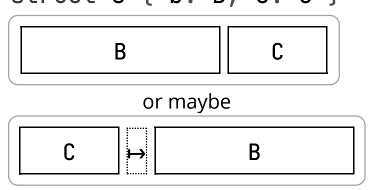
Zero-Sized ↓

`(A, B, C)`



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

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

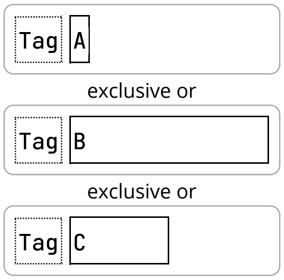


Compiler may also add padding.

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

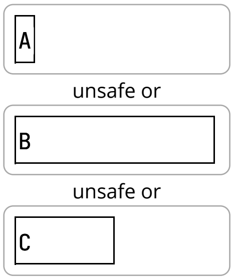
These **sum types** hold a value of one of their sub types:

`enum E { A, B, C }`



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

`union { ... }`

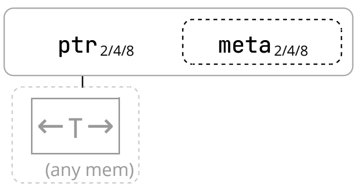


Can unsafely reinterpret memory. Result might be undefined.

References & Pointers

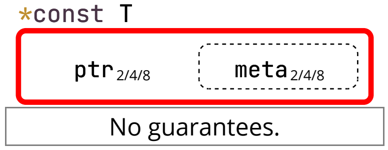
References give safe access to other memory, raw pointers `unsafe` access. The respective `mut` types are identical.

`&'a T`



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

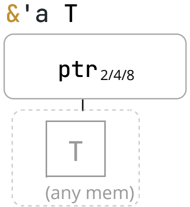
`*const T`



Pointer Meta

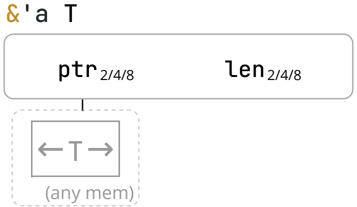
Many reference and pointer types can carry an extra field, **pointer metadata**. STD It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

`&'a T`



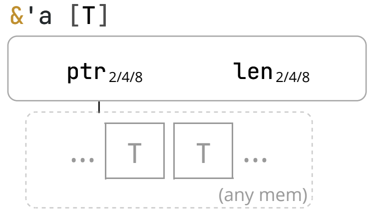
No meta for sized target. (pointer is thin).

`&'a T`



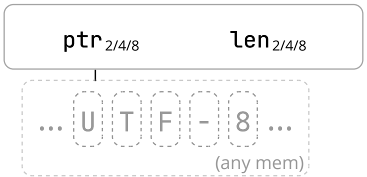
If `T` is a DST `struct` such as `S { x: [u8] }` meta field `len` is length of dyn. sized content.

`&'a [T]`



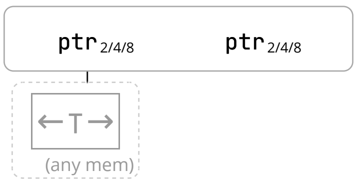
Regular **slice reference** (i.e., the reference type of a slice type `[T]`) ↑ often seen as `&[T]` if `'a` elided.

`&'a str`



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

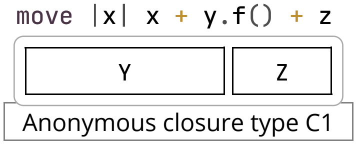
`&'a dyn Trait`



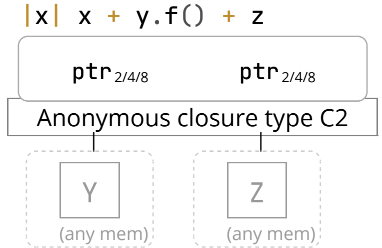
Closures

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

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



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

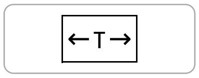


Also produces anonymous `fn` such as `fc1(C1, X)` or `fc2(&C2, X)`. Details depend which `FnOnce`, `FnMut`, `Fn` ... is supported, based on properties of captured types.

Standard Library Types

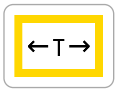
Rust's standard library combines the above primitive types into useful types with special semantics, e.g.:

`UnsafeCell<T>`



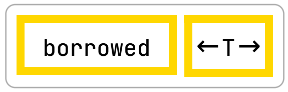
Magic type allowing aliased mutability.

`Cell<T>`



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

`RefCell<T>`

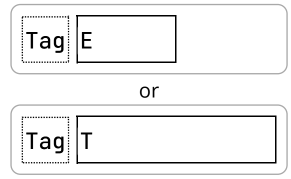


Also support dynamic borrowing of `T`. Like `Cell` this is `Send`, but not `Sync`.

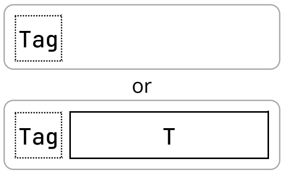
`AtomicUsize`

Other atomic similarly.

`Result<T, E>`



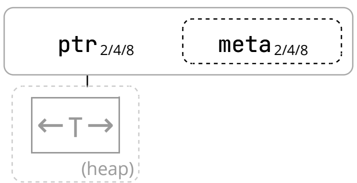
`Option<T>`



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

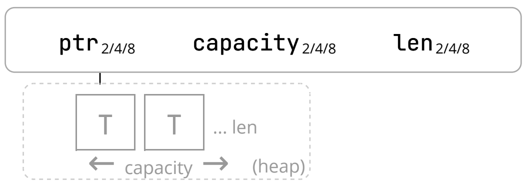
General Purpose Heap Storage

`Box<T>`



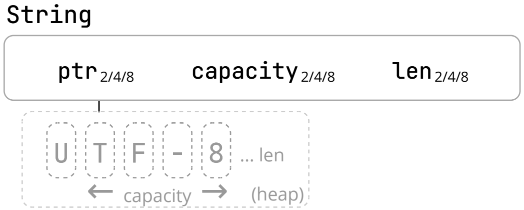
For some `T` stack proxy may carry meta↑ (e.g., `Box<[T]>`).

`Vec<T>`



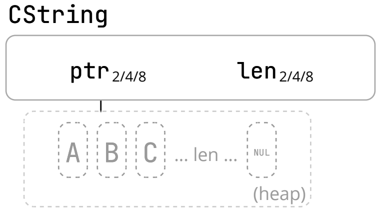
Owned Strings

`String`



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

`CString`

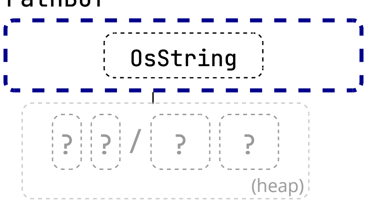


NUL-terminated but w/o NUL in middle.

`OsString` ?

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

`PathBuf` ?

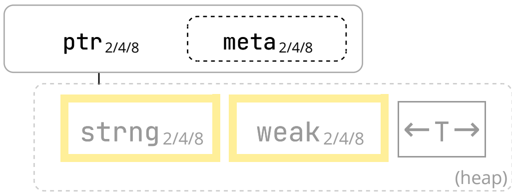


Encapsulates how operating system represents paths.

Shared Ownership

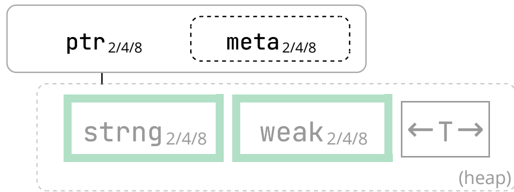
If the type does not contain a `Cell` for `T`, these are often combined with one of the `Cell` types above to allow shared de-facto mutability.

`Rc<T>`



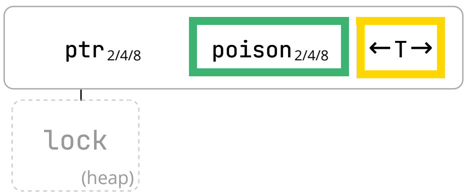
Share ownership of `T` in same thread. Needs nested `Cell` or `RefCell` to allow mutation. Is neither `Send` nor `Sync`.

`Arc<T>`



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

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



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

Standard Library

One-Liners

Snippets that are common, but still easy to forget. See **Rust Cookbook** for more.

Strings | **I/O** | **Macros** | **Esoterics**

Intent	Snippet
Concatenate strings (any `Display` that is). [1]	`format!("{}{}", x, y)`
Split by separator pattern. STD	`s.split(pattern)`
... with `&str`	`s.split("abc")`
... with `char`	`s.split('/')`
... with closure	`s.split(char::is_numeric)`
Split by whitespace.	`s.split_whitespace()`
Split by newlines.	`s.lines()`
Split by regular expression. [2]	`Regex::new(r"\s")?.split("one two three")`

[1] Allocates; might not be fastest solution if `x` is `String` already.

[2] Requires regex crate.

Thread Safety

Examples	Send*	!Send
Sync*	*Most types* ... `Mutex<T>`, `Arc<T>` [1,2]	`MutexGuard<T>` [1], `RwLockReadGuard<T>` [1]
!Sync	`Cell<T>` [2], `RefCell<T>` [2]	`Rc<T>`, `&dyn Trait`, `*const T` [3], `*mut T` [3]

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

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

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

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

Iterators

Obtaining Iterators | **Implementing Iterators**

Basics

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

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

The Iterator

Once you have an `i`:

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

For Loops

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

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

Number Conversions

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

↓ Have / Want →	`u8 ... i128`	`f32` / `f64`	String
`u8 ... i128`	`u8::try_from(x)?` [1]	`x as f32` [3]	`x.to_string()`

↓ Have / Want →	u8 … i128	f32 / f64	String
f32 / f64	x as u8 [2]	x as f32	x.to_string()
String	x.parse::<u8>()?	x.parse::<f32>()?	x

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

String Conversions

If you **want** a string of type …

String | CString | OsString | PathBuf | Vec<u8> | &str | &CStr | &OsStr | &Path | &[u8] | **Other**

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

[i] Short form x.into() possible if type can be inferred.
[r] Short form x.as_ref() possible if type can be inferred.

[1] You should, or must if call is unsafe, ensure raw data comes with a valid representation for the string type (e.g., UTF-8 data for a String).

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

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

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

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

[5] Must ensure vector actually ends with 0x0.

String Output

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

APIs | Printable Types | Formatting

Rust has, among others, these APIs to convert types to stringified output, collectively called *format* macros:

Macro	Output	Notes
`format!(fmt)`	`String`	Bread-and-butter "to `String`" converter.
`print!(fmt)`	Console	Writes to standard output.
`println!(fmt)`	Console	Writes to standard output.
`eprint!(fmt)`	Console	Writes to standard error.
`eprintln!(fmt)`	Console	Writes to standard error.
`write!(dst, fmt)`	Buffer	Don't forget to also `use std::io::Write;`
`writeln!(dst, fmt)`	Buffer	Don't forget to also `use std::io::Write;`

Method	Notes
`x.to_string()` STD	Produces `String`, implemented for any `Display` type.

Here `fmt` is string literal such as `"hello {}"`, that specifies output (compare "Formatting" tab) and additional parameters.

Tooling

Project Anatomy

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

Entry	Code
`.cargo/`	**Project-local cargo configuration**, may contain `config.toml`.
`benches/`	Benchmarks for your crate, run via **`cargo bench`**, requires nightly by default. *
`examples/`	Examples how to use your crate, they see your crate like external user would.
`my_example.rs`	Individual examples are run like **`cargo run --example my_example`**.
`src/`	Actual source code for your project.
`main.rs`	Default entry point for applications, this is what **`cargo run`** uses.
`lib.rs`	Default entry point for libraries. This is where lookup for `my_crate::f()` starts.
`tests/`	Integration tests go here, invoked via **`cargo test`**. Unit tests often stay in `src/` file.
`.rustfmt.toml`	In case you want to **customize** how **`cargo fmt`** works.
`.clippy.toml`	Special configuration for certain **clippy lints**, utilized via **`cargo clippy`**
`build.rs`	**Pre-build script**, useful when compiling C / FFI, ...

Entry	Code
`Cargo.toml`	Main **project manifest**, 🔗 Defines dependencies, artifacts ...
`Cargo.lock`	Dependency details for reproducible builds, recommended to `git` for apps, not for libs.

* On stable consider Criterion.

Minimal examples for various entry points might look like:

Applications | Libraries | Unit Tests | Integration Tests | Benchmarks | Build Scripts | Proc Macros

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

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

Module trees and imports:

Module Trees | Namespaces

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! 🛑

Cargo

Commands and tools that are good to know.

Command	Description
`cargo init`	Create a new project for the latest edition.
`cargo build`	Build the project in debug mode (`--release` for all optimization).
`cargo check`	Check if project would compile (much faster).
`cargo test`	Run tests for the project.
`cargo doc --open`	Locally generate documentation for your code and dependencies.
`cargo run`	Run your project, if a binary is produced (main.rs).
`cargo run --bin b`	Run binary `b`. Unifies features with other dependents (can be confusing).

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

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

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

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

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

Cross Compilation

1. Check target is supported.

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

3. 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`).

4. 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"
```

5. 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 | **Documentation** | `#![globals]` | `#[code]` | `#[quality]` | `#[macros]` | `#[cfg]` | `build.rs`

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

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

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 | Generics | Advanced Concepts

Types

Type Equivalence and Conversions

Implementations — `impl S { }`

Traits — `trait T { }`

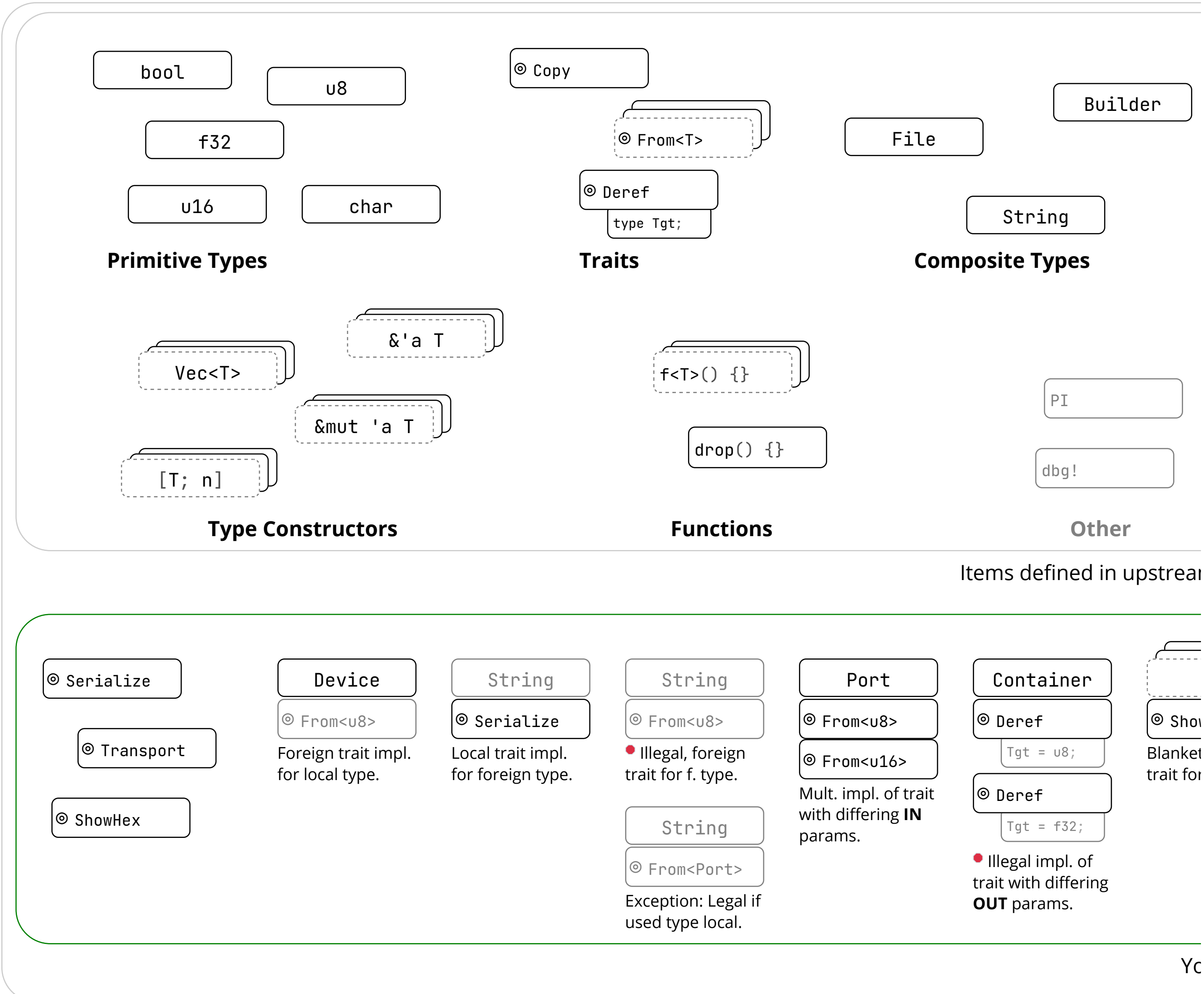
Implementing Traits for Types — `impl T for S { }`

Traits vs. Interfaces

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 | **Computation (Traits)** | **Casts** | **Coercions** | **Subtyping** | **Variance**

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

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

Method	Explanation
Coercions	**Automatic** conversion within *'weakening ruleset'*.[1]
Subtyping	**Automatic** conversion within *'same-layout-different-lifetimes ruleset'*.[1]

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

Coding Guides

Idiomatic Rust

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

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

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

Async-Await 101

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

Basics | **Execution Flow** | **Caveats**

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

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

Closures in APIs

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

From a call site perspective that means:

Signature	Function `g` can call …	Function `g` accepts …
`g<F: FnOnce()>(f: F)`	… `f()` once.	`Fn`, `FnMut`, `FnOnce`
`g<F: FnMut()>(mut f: F)`	… `f()` multiple times.	`Fn`, `FnMut`
`g<F: Fn()>(f: F)`	… `f()` multiple times.	`Fn`

Notice how **asking** for a `Fn` closure as a function is most restrictive for the caller; but **having** a `Fn` closure as a caller is most compatible with any function.

From the perspective of someone defining a closure:

Closure	Implements*	Comment
`|| { moved_s; }`	`FnOnce`	Caller must give up ownership of `moved_s`.
`|| { &mut s; }`	`FnOnce`, `FnMut`	Allows `g()` to change caller's local state `s`.
`|| { &s; }`	`FnOnce`, `FnMut`, `Fn`	May not mutate state; but can share and reuse `s`.

* Rust prefers capturing by reference (resulting in the most "compatible" `Fn` closures from a caller perspective), but can be forced to

capture its environment by copy or move via the `move || {}` syntax.

That gives the following advantages and disadvantages:

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

Unsafe, Unsound, Undefined

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

Unsafe Code | **Undefined Behavior** | **Unsound Code**

Unsafe Code

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

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

Responsible use of Unsafe

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

API Stability

When updating an API, these changes can break client code.[RFC] Major changes (🔴) are **definitely breaking**, while minor changes (🟡) **might be breaking**:

Crates

Crates

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

Modules

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

Structs

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

Enums

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

Traits

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

Traits

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

Inherent Implementations

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

Signatures in Type Definitions

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

Signatures in Functions

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

Behavioral Changes

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

Ralf Biedert, 2021 – cheats.rs