Rust Language Cheat Sheet

02.12.2018

Contains clickable links to **The Book** BK, **Rust by Example** EX, **Std Docs** STD, **Nomicon** NOM, **Reference** REF. Furthermore, entries are marked as largely **deprecated** 🗑, have a **minimum edition** '18. or are **bad** ⚡.

The latest version of this document can be found at cheats.rs.

Data Structures

Define data types and memory locations, and use them.

Example	Explanation
`struct S {}`	Define a **struct**, BK EX STD REF with named fields.
`struct S();`	Define "tupled" struct with numbered fields `.0`, `.1`, ...
`struct S;`	Define zero sized unit struct.
`enum E {}`	Define an **enum** BK EX REF, *c.* algebraic data types, tagged unions.
`enum E { A, 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 discriminants values, e.g., for FFI.
`union U {}`	Unsafe C-like **union** REF for FFI compatibility.
`static X: T = x;`	**Global variable** BK EX REF with `'static` lifetime, single memory location.
`const X: T = x;`	Define inlineable **constant**, BK EX REF. Inlined values are mutable!!!
`let x;`	**Variable binding** ? that can't be changed or `&mut`'ed.
`let mut x;`	Same, but allow for change or mutable borrow.

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(x)`	Create `struct S(T)` or `use`'ed `enum E::S()` with field `.0` set to `x`.
`S`	If `S` is unit `struct S;` or `use`'ed `enum E::S` create value of `S`.
`E::C { x: y }`	Create enum variant `C`. Other methods above also work.
`()`	Empty tuple, both literal and type, aka **unit** STD
`(x)`	Parenthesized expression.
`(x,)`	Single-element **tuple** expression. EX STD REF
`(T,)`	Single-element tuple type.
`[T; n]`	**Array type** EX STD with `n` elements of type `T`.
`[x; n]`	Array with `n` copies of `x`. REF
`[x, y]`	Array with given elements.
`x[0]`	Collection indexing. Overloadable Index, IndexMut
`x[..]`	Collection slice-like indexing via RangeFull, *c.* **slices** STD EX REF
`x[a..]`	Collection slice-like indexing via RangeFrom.
`x[..b]`	Collection slice-like indexing RangeTo.
`x[a..b]`	Collection slice-like indexing via Range.

Example	Explanation
`a..b`	Right-exclusive **range** REF creation, also seen as `..`, `a..`, `..b`.
`a..=b`	Inclusive range creation, also seen as `..=b`.
`x.i`	Member **access**. REF
`x.0`	Tuple access

References & Pointers

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

Example	Explanation
`&t`	Immutable **borrow** BK EX STD (i.e., an actual "pointer to t", like `0x1234`).
`&T`	Immutable **reference** BK STD NOM REF (i.e., safe pointer *type* holding *any* `&t`).
`&mut t`	Borrow that allows **mutability**. EX
`&mut T`	Reference that allows mutability.
`*const T`	Immutable **raw pointer type** BK STD REF.
`*mut T`	Mutable raw pointer type.
`ref t`	**Bind by reference**. BK EX
`*x`	**Dereference**. BK STD NOM
`'static`	Lifetime lasting the entire program execution.
`'a`	Often seen as `&'a T`, a **lifetime parameter**. BK EX NOM REF

Functions & Behavior

Define units of code and their abstractions.

Sigil	Explanation
`trait T {}`	Define a **trait**. BK EX REF
`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`.
`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() -> T {}`	Same, returning a type T.
`fn f(&self) {}`	Define a method as part of an `impl`.
`fn() -> T`	**Function pointers**, BK STD REF don't confuse with traits `Fn`, `FnOnce`, `FnMut`.
`|| {}`	A **closure** BK EX REF that borrows its captures.
`|x| {}`	Closure with a bound parameter `x`.
`|x| x + x`	Closure without block expression.
`move |x| x + y`	Closure taking ownership of its captures.
`return || true`	Closures may sometimes look like logical ORs (here: return a closure).
`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 `T::f()` implemented for `X`.
`unsafe {}`	Marker for **unsafe code**. BK EX NOM REF that will probably segfa#%$@.

Control Flow

Control execution within a function.

Sigil	Explanation
`while x {}`	**Loop** REF, run while expression `x` is true.
`loop {}`	**Loop infinitely** REF until `break`. Can yield value with `break x`.
`for x in iter {}`	Syntactic sugar to loop over **iterators**. BK STD REF
`if x {} else {}`	**Conditional branch** REF if expression is true.
`'label: loop {}`	**Loop label** EX REF, useful for flow control in nested loops.
`break`	**Break expression** REF to exit a loop.
`break x`	Same, but make `x` value of the loop expression (only in actual `loop`).
`break 'label`	Exit not only this loop, but the enclosing one marked with `'label`.
`continue`	**Continue expression** REF to the next loop iteration of this loop.
`continue 'label`	Same, but instead of enclosing loop marked with `'label`.
`return x`	Early return from function. More idiomatic way is to end with expression.
`x?`	If `x` is `Result::Err` or `Option::None`, **return and propagate**. BK EX STD REF

Organizing Code

Segment projects into smaller units and minimize dependencies.

Sigil	Explanation
`mod m {}`	Define a **module**. BK EX REF
`a::b`	Namespace **path** BK EX REF to element `b` within `a` (`mod`, `enum`, ...).
`::x`	Search `x` relative to crate root.
`crate::x`	Search `x` relative to crate root. '18
`self::x`	Search `x` relative to current module.
`super::x`	Search `x` 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::*;`	Bring everything from `a` into scope and reexport.
`pub use a::b;`	Bring `a::b` into scope and reexport from here.
`pub T`	"Public if parent path public" **visibility** BK EX REF for `T`.
`pub(crate) T`	Visible at most in current crate.
`pub(self) T`	Visible at most in current module.
`pub(super) T`	Visible at most in parent.
`pub(in a::b) T`	Visible at most in `a::b`.
`extern crate x;`	Declare dependency on external **crate** BK EX REF ; just `use x::f` in '18.
`extern "C" fn`	External dependency for **FFI**. BK EX NOM REF

Type Aliases and Casts

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

Sigil	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`.

Sigil	Explanation
`x as u32`	Primitive **cast** EX REF, may truncate and be a bit surprising. NOM

Code Generation

Constructs expanded before the actual compilation happens.

Example	Explanation
`m!()`	**Macro** BK STD REF invocation, also `m!{}`, `m![]` (depending on macro).
`$x:ty`	Macro capture, also `$x:expr`, `$x:ty`, `$x:path`, ... REF
`$x`	Macro substitution in **macros by example**. BK EX REF
`$(x),*`	Macro repetition "zero or more times" in macros by example.
`$(x),+`	Same, but "one or more times".
`$(x)<<+`	In fact separators other than `,` are also accepted. Here: `<<`.
`$crate`	Special hygiene variable, crate where macros is defined. ?
`#[attr]`	Outer **attribute**. EX REF, annotating the following item.
`#![attr]`	Inner attribute, annotating the surrounding item.

Pattern Matching

These constructs are found in `match` or `let` expressions.

Example	Explanation
`match m {}`	Initiate **pattern matching**. BK EX REF
`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 params.
`S { x, y } => {}`	Match struct with any values for fields `x` and `y`.
`S { .. } => {}`	Match struct with any values.
`D => {}`	Match enum variant `E::D` if `D` in `use`.
`D => {}`	Match anything, bind `D`; ⚡ possibly false friend of `E::D` if `D` not in `use`.
`_ => {}`	Proper wildcard that matches anything / "all the rest".
`[a, 0] => {}`	Match array with any value for `a` and `0` for second.
`(a, 0) => {}`	Match tuple with any value for `a` and `0` for second.
`x @ 1 .. 5 => {}`	Bind matched to `x`; **pattern binding** BK EX.
`0 | 1 => {}`	Pattern alternatives (or-patterns).
`S { x } if x > 10`	Pattern match **guards**. BK EX

Example	Explanation
`let Some(x) = Some(5)`	Notably, `let` also pattern matches similar to the table above.
`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 Some(x) = get()`	⚡ Will **not** work if pattern can be 'refuted', use `if let` instead.
`if let Some(x) = get()`	Branch if pattern can actually be assigned REF (e.g., `enum` variant).
`fn f(S { x }: S)`	Function parameters also work like `let`, here `x` bound to `s.x` of `f(s)`.

Generics & Constraints

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

Example	Explanation

Example	Explanation
`S<T>`	A **generic** BK EX type with a type parameter (`T` is placeholder name).
`S<T: R>`	Type short hand **trait bound** BK EX specification (`R` *must* be trait).
`T: R + S`	**Compound type bound** BK EX, also seen as `T: R + 'a`
`T: ?Sized`	Opt out of a pre-defined trait bound Sized. ?
`T: 'a`	Type **lifetime bound** EX, all references in T must outlive `'a`.
`'b: 'a`	Lifetime `'b` must live at least as long as (i.e., *outlives*) `'a` bound.
`S<T> where T: R`	Same as `S<T: R>` but easier for longer bounds.
`S<T = R>`	**Default type parameter** BK for associated type.
`S<'_>`	Inferred **anonymous lifetime**. BK
`S<_>`	Inferred **anonymous type**. ?
`S::<T>`	**Turbofish** STD call site type disambiguation, e.g. `f::<u32>()`.
`trait T { type X; }`	Defines an **associated type** BK REF `X` for trait `T`.
`type X = R;`	Set associated type within `impl T for S { type X = R; }`.
`impl<T> S<T> {}`	Implement functionality for any `T` in `S<T>`.
`impl S<T> {}`	Implement functionality for exactly `S<T>` (e.g., `S<u32>`).
`fn f() -> impl T`	**Existential types** BK, returns an unknown-to-caller `S` that `impl T`.
`fn f(x: &impl T)`	Trait bound,"**impl traits**" BK, somewhat similar to `fn f<S:T>(x: &S)`.
`fn f(x: &dyn T)`	Marker for **dynamic dispatch** BK REF, `f` will not be monomorphized.
`for<'a>`	**Higher-rank trait bounds.** NOM REF

Strings & Chars

Rust has several ways to create string or char literals, depending on your needs.

Example	Explanation
`"..."`	**String literal** REF, will escape `\n`, ...
`r"..."`,	**Raw string literal**. REF, won't escape `\n`, ...
`r#"..."#`, etc.	Raw string literal, but can also contain `"`.
`b"..."`	**Byte string literal** REF; constructs ASCII `[u8]`, not a string.
`br"..."`, `br#"..."#`, etc.	Raw byte string literal, combination of the above.
`'🦀'`	**Character literal** REF, can contain unicode.
`b'x'`	ASCII **byte literal**. REF

Comments

No comment.

Example	Explanation
`//`	Line comment.
`//!`	Inner line **doc comment**. BK EX REF
`///`	Outer line doc comment.
`/*...*/`	Block comment.
`/*!...*/`	Inner block doc comment.
`/**...*/`	Outer block doc comment.

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, _| {}`.

Example	Explanation
`_x`	Variable binding explicitly marked as unused.
`1_234_567`	Numeric separator for visual clarity.
`1u8`	Type specifier for **numeric literals** EX REF (also `i8`, `u16`, ...).
`r#foo`	A **raw identifier** BK EX for edition compatibility.
`x;`	**Statement** REF terminator, *c.* **expressions** EX REF

Common Operators

Rust supports all common operators you would expect to find in a language (`+`, `*`, `%`, `=`, `==`...). Since they behave no differently in Rust we do not list them here. For some of them Rust also support **operator overloading**. STD

Invisible Sugar

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

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