Rust Cheatsheet
by Blair Labatt III, page 1 of 2

1 Literals

Decimal	98_2222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte	b'A '
&str	"hello"
Array	[1,2,3]

2 Syntax

Variable Assignment

```
let x = 5; // immutable binding
let mut x = 5; // mutable binding
let tuple = ("hello", 5, 'c'); // tuple
let (x,y,z) = tuple; // deconstruction
const PI = 3.1415; // immutable literal
// type parameterization:
let v: Vec<i32> = Vec::new();
```

Functions & Parameters

```
fn main()  // simple declaration
fn f(x: i32)  // parameter type declaration
fn f() -> i32  // return type declaration
|p| { ... } // anonymous function
// type annotations, like following are inferred:
|p: i32| -> String { ... } // usually unnecessary
```

Structs & Enums

```
struct MyStruct { name: String, flag: bool, }
enum MyEnum { Variant1, Variant2, Variant3, }
impl MyStruct { fn myMeth(){ ... }; } // method
```

References & Pointers

```
v = 5; &r = v; println!(r); // 5
// Rust uses automatic dereferencing:
p1.mthd(&p2) == (&p1).mthd(&p2); // true
// "Smart" Pointers:
sp1 = Box::new(5); println!(sp1,*sp1); // 5 5
// "Raw" Pointers:
*rp1 = 5; println!(*rp1); // 5 ... but unsafe!
```

Comments

```
// This is a comment
/* This is a multi-line comment */
//! Inner line doc comment
/// Outer line doc comment
```

3 Ownership

Ownership passes when new binding occurs. This is called a "move". To avoid owning, use "borrowed" references instead. If copy trait is implemented, moves are obviated.

Borrowing

```
let var1 = &x; // immutable borrow
let var2 = &mut x; // mutable borrow
let slice = &s[6..11]; // slice of string s
// pass by [borrowed] reference:
let var3 = my_func( &concrete );
// Note: spawned threads can't generally
// borrow because lifetime in parent is
// not guaranteed.
```

Moving

```
// Force ownership into closure with "move":
let c = move |z| z == x; // (aka a "capture")
```

Dropping

```
// "drop" forces relinquishment of ownership:
drop(var); // drop is in  std::mem::drop
```

Lifetimes

```
&'a i32 // basic annotation
fn f<'a>(x: &str, y: &'a str) -> &'a str { // in sig.
struct s<'a> { // on structs
'static // static
```

Lifetime Elision Rules

Given sets P, L, OL for parameters, lifetimes, output lifetime, respectively:

1. $\forall P \exists L$ "each param. has a lifetime"
2. $|L| = 1 \implies OL = L(0)$
   "output lifetime is same as a solitary input lifetime"
3. $\exists$ &self $\in P \implies OL = L(self)$
   "&self's lifetime taken by default as output lifetime"

4 Control

Conditional

```
if (x < 5) {} else if(y == true) {} else {}
let x = if (y = true) { 5 } else { 6 };
```

Looping

```
loop{ ... ; break ; } // explicit break
while(num !=0 ){ ... } // break implied
for a in b.iter() { ... }
```

Matching

```
// single, non-exhaustive pattern match
if let Some(3) = <var> { println!(); }
// exhaustive match
match coin {
    Coin::Nickel => 5, // a match "arm"
    Coin::Dime => 10, // second match arm
    Coin::Quarter(state) => // arm's "pattern"
        // block for arm's "expression"
        { println!("Is from {:?}!", state); 25 },
    _, // catch-all, ensuring exhaustivity
}
// in loops:
while let Some(top) = stack.pop() { ... }
for (index, value) in v.iter().enumerate() { ... }
// elsewhere, unnoticeably:
let x = 5; // stmt is an "irrefutable" match
let (x, y, _) = (1, 2, 3); // ... more visibly so
fn f(x: i32) { // x is pattern, eg could use a tuple
```

5 Types

Numeric

	Signed	Unsigned
Integers:	i8	u8
	i16	u16
	i32	u32
	i64	u64
	isize	usize
Floats:	f32	f64

Scalar

```
// Numerics are all Scalar & implement Copy
bool        impl Copy
char        impl Copy
        .is_[digit|whitespace|...]()
        .is_ascii_[digit|uppercase|...]()
        .[len|encode]_utf[8|16]()
```

Compound

```
tuple               impl Copy
        .push()     .push()     .push()
array[T; N]
        .push()     .push()     .push()
```

Collection

```
        .collect()  .get()      .len()
std::vec::Vec<T>
        .push()     .pop()
std::string::String
        .push()     .chars()    .bytes()
std::collections::HashMap<K,V>
        .insert()   .entry()    .or_insert()
```

Monadic

```
std::option::Option<T>
        .push()     .push()     .push()
std::result::Result<T,E>
        .unwrap() .expect()
```

Smart Pointer

```
std::boxed::Box<T> // smart pointer
        .push()             .push()
std::rc::Rc<T> // reference counter
        .weak_count()  .strong_count()
std::cell::RefCell<T> // internal mutability
        .borrow_mut() .push()
std::thread::JoinHandle // thread reference
        .push()             .push()
std::sync::Mutex<T> // mutually-exclusive
        .push()             .push()
std::sync::Arc<T> // atomic reference counter
        .push()             .push()
```

Advanced

```
// "never" types require no type matching
fn f(a: usize) -> ! { // return is "never" type
// creating synonyms for long type annotations:
type Thunk = Box<dyn Fn() + Send + 'static>;
// Generics cannot use dynmically sized types
fn generic<T>(t: T) { ... } // <T: Sized> inferred
```

6 Traits

Implementations

```
impl my_struct {... // simple method
impl my_trait for my_struct { // impl. a trait
impl<T> Point<T> {... // generic impl.
// interfaces & defaults
pub trait Summary {
    // unimplemented interface:
    fn summarize_auth(&self) -> String;
    // default implementation:
    fn summarize(&self) -> String
        { println!("..."); }
}
```

Generics

```
fn g1<T>(p: T) { ... //as param
fn g2<T>(p: T) -> T { ... // gen. return
struct Point<T, U> { // on structs
enum Result<T, E> { // on enums
```

Trait Bounds

```
fn f<T: my_trait>(p: T) { // long-form
fn f(p: impl my_trait) { // sugar'ed bounds
fn f<T: my_trait1 + my_trait2>(p: T) { // mult.
fn f<T: Fn(u32) -> u32> { ... // closure bounds
// complex bounds using "where"
fn f<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug { ... }
// type-conditional implementations:
impl<T> Pair<T> { ... } // ... and later ...
impl<T: Display + PartialOrd> Pair<T> { ... }
impl<T: Display> ToString for { // "blanket"
```

Common Traits

Copy	Clone	Hash
Display	Debug	
From	Into	
Deref[Mut]	Borrow[Mut]	Drop
Read	Write	ReadBuf
[Partial]Ord	[Partial]Eq	
Iterator	IntoIterator	
Fn	FnMut	FnOnce
Future	Generator	
Error		
BitAnd	BitOr	Not
Send	Sync	

Trait Objects

TOs relax generics' restriction (as Rust implements them) whereby only a single concrete type will be substituted at compile-time per: monomorphism : static dispatch : generics :: polymorphism : dynamic dispatch : trait objects
`Box<dyn Draw>` // type must implement Draw

Advanced

```
// Associated Types
// Default Generic Types
// Supertraits
// Newtype pattern
```

7 Idioms

Statement-Orientation

```
// bind to statement eval
let y = { let x = 4; x + 1 }
```

Collections

```
// Vector of differing “types”
enum a {Int(i32),
        Float(f64),
        Text(String),}
let b = vec!(a::Int(3),
        a::Float(3.1),
        a::Text(String::from("3.2")));
```

Error Handling

```
// Error propagation:
let mut f = File::open("hello.txt");
f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
}; // ... is equivalent to ...
let mut f = File::open("hello.txt")?;
```

Concurrency

```
// threads & channels:
let handle = std::thread::spawn(|| { ... } );
let (tx, rc) = std::sync::mpsc::channel();
// spawn returned a Result<JoinHandle,_>
handle.join().unwrap();
```

Unsafe

```
// 5 unsafe “superpowers”
let mut num = 5;
let r1 = &num as *const i32; // raw pointer
let r2 = &mut num as *mut i32; // note “as”
unsafe { // unsafe actions only allowed here
    println!("r1 is: ", *r1); // (1) derefencing
    println!("r2 is: ", *r2); // raw pointer
}
unsafe fn dangerous() {...} // (2) unsafe func.
// (3) employing unsafe “FFI” call (here, C):
 extern "C"{ fn abs(x: i32) -> i32; }
 static mut CTR: u32 = 0; // (4) mut global
unsafe trait Foo { ... } // (5) unsafe trait
```

8 Macros

Metaprogramming constructs used to generate verbose or arbitrary rust code. Unlike functions, they can be used to implement traits on given types, as expansion occurs at compile-type.

Prelude

```
println!("x = {}", x);
eprintln!() // ... to stderr
env!("CARGO_PKG_VERSION")
let v = vec![1,2,3];
format!();
panic!
```

Declarative

Macros declared with `macro_rules!`

Derive

Procedural macros invoked like:

```
#[derive(HelloMacro)]
// only for structs & enums, and defined like:
use proc_macro::TokenStream;
#[proc_macro_attribute]
pub fn m(a: TokenStream) -> TokenStream {
// where parameter “a” usually parsed into
// AST using external crate syn::parse, then
// acted on and reconstituted as a TokenStream
// using external crate quote::quote::into
```

Attribute-like

A procedural macro defined similarly to “derives”, but more generally applicable to functions, etc. Invoked like:

```
#[route(GET, "/ ")]
fn f() { ... }
```

Function-like

A procedural macro defined similarly to derives, but invoked identically to macro_rules!:

```
sql!(SELECT * FROM my_table); // sql checker
```

9 Tools

common flags:

- –help
- –version
- –verbose

rustup

```
component add <comp_name>
update stable
```

cargo

- –release // default is –debug
- –list
- –locked
- –frozen // don't touch network

```
build
bench // performance testing
check // verify compile-ability
clean // remove old target dir's artifacts
clippy // linter, must first install
doc // invoke rustdoc
fetch // update cache of Cargo.lock depend's
fix // apply accept rustc warning suggestions
fmt // invoke rustfmt formatter
init [–lib|–bin] // new package, existing dir
[un]install // a rust binary
login // locally cache crates.io API token
new [–lib|–bin] // new package & dir.
package // create distributable tarball
publish // ... to crates.io
run
search // in crates.io
test [[name] – [–test-threads=1][–nocapture]]
update // create Cargo.lock from .toml & fetch
vendor // vendor depend's to specified path
yank // ... release from crates.io
```

rustdoc

See separate Code Documentation cheatsheet.

rustfmt

Idiomatic, cosmetic Δs to source code as written or displayed. Configurable on cmd line or in `rustfmt.toml` *file:*

```
wrap_comments = true   # wrapping
comment_width = 80     # config ↑ this
```

10 Testing

General Pattern

```
// Inline unit tests alongside source code:
#[cfg(test)]
mod tests {
    #[test]
    fn test_name1()  { assert!(a == b); }
}
// Whereas int. tests go in own “test” dir.
```

Assertions

```
assert!
assert_[ne|eq]!
```

Control

```
#[cfg(test)] // selectively compile, per arg.
#[test]
#[derive(PartialEq, Debug)]  // derive traits
#[should_panic] // optional arg. (expect = "...")
#[ignore]
```

11 Documentation

rustdoc

- –output // target location
- –crate-name
- –library-path
- –cfg
- –extern // explicit depend. loc.
- –passes // run these passes
- –no-defaults // no default passes
- –test // run examples as unit tests
- –test-args
- –target
- –markdown-css
- –html-in-header
- –markdown-playground-url

Inline

```
/// ```<executable_code> ```
/// ```# <hidden_code> ```
/// ```should_panic <code> ```
/// ```no_run <code> ```
/// ```compile_fail <code> ```
```

Control

```
// Crate-level attributes:
#![doc(html_playground_url = "a.com")]
#![doc(html_root_url = "a.com")]
#![doc(html_no_source)]
#![doc(test(attr(deny(warnings))))]
#![doc(no_inline)]
// Item level attributes:
#[doc(hidden)]
#[doc = "comment"] # /// is sugar
```

12 Dependencies

Update Cargo.toml [dependencies] section for inclusion of ancillary crates:

```
[dependencies]
rand = "0.5.5 " // crates.io dependency
// uses “semver”; can operate with ~^* >=
my_mod = { path = "../rel/path"} // filesyst.
// Or fetch, build, & include from source:
git clone *.com/rust-lang-nursery/rand.git
cd rand; cargo build
// above re-issues cargo.lock, so that refs to
// specific SHAs as follows are unnecessary:
rand = {
      git = "*.com/rust-lang-nursery/rand.git ",
      rev = "9f35b8e" // can also use branch = ...
}
// dependency graph low-level manipulation:
[patch.crates-io] // or [patch."abc.com"]
uuid = { git = "*.com/rust-lang-nursery/uuid",
     branch = "2.0.0"}
// development & build dependencies:
[dev-dependencies]
tempdir = "0.3" // used on test, example, bench
[build-dependencies]
cc = "1.0.3" // script used only in build
// use non-crates.io repository:
[registries]
my-registry = {
     index = "https://my-intranet:8080 "}
[dependencies]
other-crate = {      version = "1.0 ",
     registry = "my-registry "}
```

Modules

```
mod fn my_function1(){}; // private
pub mod fn my_function2(){}; // public
// Module in another file, still need to “use”:
mod file_name;
```

Paths

```
use absolute::path::to::module;
use relative_path::to::{mod1, mod2};
use path::to::{sub_path::mod1, mod2};
use std::io::Result as IoResult; // aliasing
pub use rand::Rng; // “re-exporting”
```

Crates

```
<crate> = <lib_crate> | <bin_crate>
```

Packages

Packages contain (∋) crates, according to:

- ∀ packages ∋ <crate>+
- ∀ packages ∋ <lib_crate>?
- ∀ packages ∋ <bin_crate>*