Contents

Cheat Sheet

… You may use this cheat sheet
for your own good.

Basic Types & Variables

Types
bool — Boolean

Unsigned integers
u8, u16, u32, u64, u128

Signed integers
i8, i16, i32, i64, i128

Floating point
f32, f64

Platform-specific
usize: Unsinged integer, Same number of bits as the platform's pointer type.

isize: signed Integer. Same number of bits as the platform's pointer type.

char — Unicode Scalar Value
&str — String slice
String — Owned string

Tuple

```
let coordinates = (82, 64);
let score = ("Team A", 12);
```

Array & Slice

```
let array = [1, 2, 3, 4, 5];
let array2 = [0; 3]; // [0, 0, 0]
let slice = &array[1..3];
```

HashMap

```
use std::collections::HashMap;

let mut newHash = HashMap::new();
newHash.insert(String::from("RustMY"), 100000);
newHash.entry("".to_owned())
    .or_insert(3);
```

Mutability

```
let mut x = 5;
x = 6
```

Struct

```
//Definition
struct User {
  username: String,
  active: bool,
}

//Instantiation
let user1 = User {
  username: String::from("bogdan"),
  active: true,
};

//Tuple Struct
struct Color(i32, i32, i32);
let black = Color(0, 0, 0);
```

Enum

```
//Definition
enum Command {
    Quit,
    Move { x: i32, y: i32 },
    Speak(String),
    ChangeBGColor(i32, i32, i32),
}
//Instatniation
let msg1 = Command::Quit;
let msg2 = Command::Move{ x: 1, y: 2 };
let msg3 = Command::Speak("Rust".to_owned());
let msg4 = Command::ChangeBGColor(0, 0, 0);
```

Constant

```
const MAX_POINTS: u32 = 100_000;
```

Static Variable

```
// Unlike constants static variables are
// stored in a dedicated memory location
// and can be mutated.
static MAJOR_VERSION: u32 = 1;
static mut COUNTER: u32 = 0;
```

Type alias

```
// `NanoSecond` is a new name for `u64`.
type NanoSecond = u64;
```

Control Flow

if & if let

```
let num = Some(22);
if num.is_some() {
println!("number is: {}", num.unwrap());
}
// match pattern and assign variable
if let Some(i) = num {
println!("number is: {}", i);
}
```

loop

```
let mut count = 0;
loop {
count += 1;
if count == 5 {
    break; // Exit loop
  }
}
```

Nested loops & labels

```
outer: loop {
  inner: loop {
    // This breaks the inner loop
    break;
  }
  // This breaks the outer loop
  break 'outer;
}
```

Returning from loops

```
let mut count = 0;
let result = loop {
  count += 1;
  if count == 10 {
    break count;
  }
};
```

while & while let

```
while n < 101 {
  n += 1;
}
let mut optional = Some(0);
while let Some(i) = optional
{
  print!("{}", i);
}
```

for loop

```
for n in 1..101 {
  println!("{}", n);
}
let nms = vec!["Ivn","Han","Nik","Nix","Jef"];
for name in nms.iter() {
  println!("{}", name);
}
```

match

```
let optional = Some(0);
match optional {
    Some(i) => println!("{}", i),
    None => println!("No value.")
}
```

References, Ownership & Borrowing

Ownership rules

1 - Every value in Rust is owned by a single variable, which acts as its “owner.”
2- Ownership is exclusive—only one variable can own a value at a time.
3 - When the owner variable goes out of scope (is no longer used), the value is automatically removed from memory.
4- This system ensures efficient memory management without requiring manual cleanup.

Borrowing rules

1 - At any moment, you can either have a single mutable reference or multiple immutable references to a value.
2 - References must remain valid and never point to invalid or deallocated memory.

Creating references

```
let s1 = String::from("hello world!");
let s1_ref = &s1; // immutable reference
let mut s2 = String::from("hello");
let s2_ref = &mut s2; // mutable reference
s2_ref.push_str(" world!");
```

Copy, Move & Clone

```
// Simple values which implement the Copy trait
are copied by value
let x = 5;
let y = x;
println!("{}", x); // x is still valid
// The string is moved to s2 and s1 is
invalidated
let s1 = String::from("Welcometo Workshop");
let s2 = s1; // Shallow copy a.k.a move
println!("{}", s1); // Error: s1 is invalid
let s1 = String::from("Welcometo Workshop");
let s2 = s1.clone(); // Deep copy
// Valid because s1 isn't moved
println!("{}", s1);
```

Ownership & Functions

```
fn main() {
    let x = 5;
    takes_copy(x); // x is copied by value
    let s = String::from("Welcome to Workshop");
    // s is moved into the function
    takes_ownership(s);
    // return value is moved into s1
    let s1 = gives_ownership();
    let s2 = String::from("RustMY");
    let s3 = takes_and_gives_back(s2);
  }

  fn takes_copy(some_integer: i32) {
    println!("{}", some_integer);
  }

  fn takes_ownership(some_string: String) {
    println!("{}", some_string);
  } // some_string goes out of scope and drop is
called. The backing memory is freed.

  fn gives_ownership() -> String {
    let some_string = String::from("WorkShop");
    some_string
  }

  fn takes_and_gives_back(some_string: String)->
String {
    some_string
  }
```

Pattern Matching

Basics

```
let x = 5;
match x {
// matching literals
    1 => println!("one"),
    // matching multiple patterns
    2 | 3 => println!("two or three"),
    // matching ranges
    4..=9 => println!("within range"),
    // matching named variables
    x => println!("{}", x),
    // default case (ignores value)
    _ => println!("default Case")
}
```

Destructing

```
  struct Point {
    x: i32,
    y: i32,
  }
  let p = Point { x: 0, y: 7 };
  match p {
  Point { x, y: 0 } => {
      println!("{}" , x);
    },
    Point { x, y } => {
      println!("{} {}" , x, y);
    },
  }

  enum Shape {
    Rectangle { width: i32, height: i32 },
    Circle(i32),
  }
  let shape = Shape::Circle(10);
  match shape {
    Shape::Rectangle { x, y } => //...
    Shape::Circle(radius) => //...
  }
```

Ignoring Values

```
struct SemVer(i32, i32, i32);
 let version = SemVer(1, 32, 2);
 match version {
   SemVer(major, _, _) => {
     println!("{}", major);
   }
 }
 let numbers = (2, 4, 8, 16, 32);
 match numbers {
   (first, .., last) => {
     println!("{}, {}", first, last);
   }
 }
```

Match guards

```
let num = Some(4);
match num {
  Some(x) if x < 5 => println!("less than five:
{}", x),
  Some(x) => println!("{}", x),
  None => (),
}
```

Bindings

```
struct User {
    id: i32
  }
let user = User { id: 5 };
match user {
User {
  id: id_variable @ 3..=7,
    } => println!("id: {}", id_variable),
  User { id: 10..=12 } => {
    println!("within range");
  },
  User { id } => println!("id: {}", id),
}
```

Iterators

Usage

```
// Methods that consume iterators
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
let total: i32 = v1_iter.sum();
// Methods that produce new iterators
let v1: Vec<i32> = vec![1, 2, 3];
let iter = v1.iter().map(|x| x + 1);
// Turning iterators into a collection
let v1: Vec<i32> = vec![1, 2, 3];
let v2: Vec<_> = v1.iter().map(|x| x +
1).collect();
```

Implementing the Iterator trait

```
struct Counter {
  count: u32,
}
impl Counter {
  fn new() -> Counter {
    Counter { count: 0 }
    }
}
impl Iterator for Counter {
  type Item = u32;
}
fn next(&mut self) -> Option<Self::Item> {
  if self.count < 5 {
    self.count += 1;
    Some(self.count)
  } else {
    None
  }
}
```

Error Handling

Throw unrecoverable error

```
panic!("Critical error! Exiting!");
```

Option enum

```
fn get_user_id(name: &str) -> Option<u32> {
  if database.user_exists(name) {
    return Some(database.get_id(name))
  }
  None
}
```

Result enum

```
fn get_user(id: u32) -> Result<User, Error> {
  if is_logged_in_as(id) {
   return Ok(get_user_object(id))
  }
  Err(Error { msg: "not logged in" })
}
```

? Operator

```
fn get_salary(db: Database, id: i32) ->
Option<u32> {
  Some(db.get_user(id)?.get_job()?.salary)
}

fn connect(db: Database) -> Result<Connection,
Error> {
  // ? works if the return type is Result
  let conn =
db.get_active_instance()?.connect()?;
  Ok(conn)
}
```

Combinators

.map

```
let some_string = Some("RustMy".to_owned());
let some_len = some_string.map(|s| s.len());

struct Error { msg: String }
struct User { name: String }

let string_result: Result<String, Error> =
Ok("Bogdan".to_owned());

let user_result: Result<User, Error> =
string_result.map(|name| {
  User { name }
});
```

.and_then

```
let vec = Some(vec![1, 2, 3]);
let first_element = vec.and_then(
  |vec| vec.into_iter().next()
);
let string_result: Result<&'static str, _>
= Ok("5");
let number_result =
  string_result
  and_then(|s| s.parse::<u32>());
```

Multiple Error Types

Define custom error type

```
type Result<T> = std::result::Result<T,
CustomError>;

#[derive(Debug, Clone)]
struct CustomError;

impl fmt::Display for CustomError {
  fn fmt(&self, f: &mut fmt::Formatter) ->
fmt::Result {
    write!(f, "custom error message")
  }
}
```

Boxing errors

```
use std::error;
type Result<T> =
  std::result::Result<T, Box<dyn
error::Error>>;
```

Iterating Over Errors

Ignore failed items with filter_map()

```
let strings = vec!["RustMY", "22", "7"];
let numbers: Vec<_> = strings
  .into_iter()
  .filter_map(|s| s.parse::<i32>().ok())
  .collect();
```

Failed the entire operation with collect()

```
let strings = vec!["RustMY", "22", "7"];
let numbers: Result<Vec<_>, _> = strings
  .into_iter()
  .map(|s| s.parse::<i32>())
  .collect();
```

Collect all valid values & failures with partition()

```
let strings = vec!["RustMY", "22", "7"];
let (numbers, errors): (Vec<_>, Vec<_>) =
strings
  .into_iter()
  .map(|s| s.parse::<i32>())
  .partition(Result::is_ok);

let numbers: Vec<_> = numbers
  .into_iter()
  .map(Result::unwrap)
  .collect();

let errors: Vec<_> = errors
  .into_iter()
  .map(Result::unwrap_err)
  .collect();
```

Using generics

```
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
  fn mixup<V, W>(self, other: Point<V, W>) ->
Point<T, W> {
      Point {
          x: self.x,
          y: other.y,
      }
  }
}
```

Defining traits

```
trait Animal {
    fn new(name: &'static str) -> Self;
    fn noise(&self) -> &'static str {
        ""
    }
}

struct Dog {
    name: &'static str,
}

impl Dog {
    fn fetch() {
        // ... implementation
    }
}

impl Animal for Dog {
    fn new(name: &'static str) -> Dog {
        Dog { name }
    }
    fn noise(&self) -> &'static str {
        "woof!"
    }
}
```

Default Implementations with Derive

```
// A tuple struct that can be printed
#[derive(Debug)]
struct Inches(i32);
```

Trait Bounds


```
fn largest<T: PartialOrd + Copy>(list: &[T]) ->
T {
    let mut largest = list[0];
    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

impl trait

```
fn m_function(y: i32) -> impl Fn(i32) -> i32 {
  let closure = move |x: i32| { x + y };
  closure
}
```

Trait Objects

```
pub struct Screen {
  pub components: Vec<Box<dyn Draw>>,
}
```

Opertor Overloading

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}
impl Add for Point {
    type Output = Point;
    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}
```

SuperTraits

```
use std::fmt;
trait Log: fmt::Display {
    fn log(&self) {
        let output = self.to_string();
        println!("Logging: {}", output);
    }
}
```

Lifetimes in function signatures

```
fn long<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes in struct defination

```
struct email<'a> {
  email_address: &'a str,
}
```

Static lifetimes

```
let test: &'static str = "Welcome to RustMY!";
```

Functions, Function Pointers & Closures

Static lifetimes

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    // Associated function
    fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }

    // Method
    fn get_x(&self) -> i32 {
        self.x
    }
}
```

Function Pointers

```
fn do_tw(f: fn(i32) -> i32, v: i32) -> i32 {
  f(v) + f(v)
}
```

Creating Closures

```
let add_one = |num: u32| -> u32 {
  num + 1
};
```

Returning closures

```
fn add_one() -> impl Fn(i32) -> i32 {
    |x| x + 1
}

fn add_or_subtract(x: i32) -> Box<dyn Fn(i32) ->
i32> {
    if x > 10 {
        Box::new(move |y| y + x)
    } else {
        Box::new(move |y| y - x)
    }
}
```

Closure traits

• FnOnce - consumes the variables it captures from its enclosing scope.
• FnMut - mutably borrows values from its enclosing scope.
• Fn - immutably borrows values from its enclosing scope.

Closure traits

```
struct Cacher<T>
where
  T: Fn(u32) -> u32,
{
  calculation: T,
  value: Option<u32>,
}
```

Function that accepts closure or function pointer

```
fn do_twice<T>(f: T, x: i32) -> i32
  where T: Fn(i32) -> i32
{
  f(x) + f(x)
}
```

Pointers

References

```
let mut num = 5;
let r1 = &num; // immutable reference
let r2 = &mut num; // mutable reference
```

Raw Pointers

```
let mut num = 5;
// immutable raw pointer
let r1 = &num as *const i32;
// mutable raw pointer
let r2 = &mut num as *mut i32;
```

Smart Pointers

```
Box<T> - for allocating values on the heap
let b = Box::new(5);

Rc<T> - multiple ownership with reference
counting
let a = Rc : : new(5);
let b = Rc : : clone(&a);

Ref<T>, RefMut<T>, and RefCell<T> - enforce
borrowing rules at runtime instead of compile
time.

let num = 5;
let r1 = RefCell::new(5);
// Ref - immutable borrow
let r2 = r1.borrow();
// RefMut - mutable borrow
let r3 = r1.borrow_mut();
// RefMut - second mutable borrow
let r4 = r1.borrow_mut();
```

Multiple owners of mutable data

```
let x = Rc::new(RefCell::new(5));
```

PACKAGES, CRATES & MODULES

Defination

• Packages - A Cargo feature that lets you build,test, and share crates.
• Crates - A tree of modules that produces a library or executable.
• Modules and use - Let you control the organization, scope, and privacy of paths.
• Paths - A way of naming an item, such as a struct, function, or module.

Creating a new package with a binary crate

$ cargo new my-project

Creating a new package with a library crate

$ cargo new my-project –lib

Defining & using modules

```
fn some_function() {}

mod outer_module {
    // private module by default
    pub mod inner_module {
        // public inner module
        pub fn inner_public_function() {
            super::super::some_function(); //
calling a function from the top level
        }
        fn inner_private_function() {}
    }
}

fn main() {
    // Absolute path from crate root

crate::outer_module::inner_module::inner_public_func
    // Relative path from current scope

outer_module::inner_module::inner_public_function();
    // Bringing the module into scope with use
    use_outer(); // helper below
}

// Helper to show use
fn use_outer() {
    use crate::outer_module::inner_module;
    inner_module::inner_public_function();
}
```

Renaming with as keyword

```
use std::fmt::Result;
use std::io::Result as IoResult;
```

Re-exporting with pub use

```
mod outer_module {
  pub mod inner_module {
    pub fn inner_public_function() {}
  }
}
pub use crate::outer_module::inner_module;
```

Defining modules in separate files

17

```
// src/lib.rs
mod my_module;
pub fn some_function() {
  my_module::my_function();
}

// src/my_module.rs
pub fn my_function() {}
```