Rust cheatsheet

The Rust quick reference cheat sheet that aims at providing help on writing basic syntax and methods.

Getting Started

Hello_World.rs

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

Compiling and Running

```
$ rustc Hello_World.rs
$ ./Hello_World
Hello, World!
```

Primitive types

bool	Boolean (true / false)
char	character
f32, f64	32-bits, 64-bits floats
i64, i32, i16, i8	signed 16- ... integers
u64, u32, u16, u8	unsigned 16-bits, ... integers
isize	pointer-sized signed integers
usize	pointer-sized unsigned integers

See: Rust Types

Formatting

```
// Single Placeholder
println!("{}", 1);

// Multiple Placeholder
println!("{} {}", 1, 3);

// Positional Arguments
println!("{0} is {1} {2}, also {0} is a
{3} programming language", "Rust",
"cool", "language", "safe");

// Named Arguments
println!("{country} is a diverse nation
with unity.", country = "India");

// Placeholder traits :b for binary, :0x
is for hex and :o is octal
println!("Let us print 76 is binary which
is {:b} , and hex equivalent is {:0x} and
octal equivalent is {:o}", 76, 76, 76);

// Debug Trait
println!("Print whatever we want to here
using debug trait {:?}", (76, 'A', 90));

// New Format Strings in 1.58
let x = "world";
println!("Hello {x}!");
```

Printing Styles

```
// Prints the output
print!("Hello World\n");

// Appends a new line after printing
println!("Appending a new line");

// Prints as an error
eprint!("This is an error\n");

// Prints as an error with new line
eprintln!("This is an error with new line"
```

Variables

```
// Initializing and declaring a variable
let some_variable = "This_is_a_variable";

// Making a variable mutable
let mut mutable_variable = "Mutable";

// Assigning multiple variables
let (name, age) = ("ElementalX", 20);

// (Global) constant
const SCREAMING_SNAKE_CASE:i64 = 9;
```

Comments

```
// Line Comments
/*.............Block Comments */
/// Outer doc comments
//! Inner doc comments
```

See: Comment

Functions

```
fn test(){
  println!("This is a function!");
}

fn main(){
  test();
}
```

See: Functions

Rust Types

Integer

```
let mut a: u32 = 8;
let b: u64 = 877;
let c: i64 = 8999;
let d = -90;
```

Floating-Point

```
let mut sixty_bit_float: f64 = 89.90;
let thirty_two_bit_float: f32 = 7.90;
let just_a_float = 69.69;
```

Boolean

```
let true_val: bool = true;
let false_val: bool = false;
let just_a_bool = true;
let is_true = 8 < 5;  // => false
```

Character

```
let first_letter_of_alphabet = 'a';
let explicit_char: char = 'F';
let implicit_char = '8';
let emoji = "\u{1f600}";   // => 😀
```

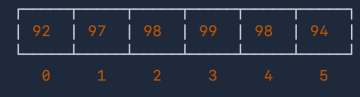
String Literal

```
let community_name = "AXIAL";
let no_of_members: &str = "ten";

println!("The name of the community is
{community_name} and it has
{no_of_members} members");
```

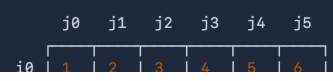
See: Strings

Arrays




```
let array: [i64; 6] = [92,97,98,99,98,94];
```

Multi-Dimensional Array



Mutable Array

```
let mut array: [i32 ; 3] = [2,6,10];

array[1] = 4;
```

Slices

```
let mut array: [ i64; 4] = [1,2,3,4];
let mut slices: &[i64] = &array[0..3] // l
```


```
let array: [[i64; 6] ;2] = [
            [1,2,3,4,5,6],
            [6,5,4,3,2,1]];
```

```
array[2] = 6;
```

Use the `mut` keyword to make it mutable.

```
println!("The elements of the slices are :
```

Vectors

```
let some_vector = vec![1,2,3,4,5];
```

A vector is declared using the `vec!` macro.

Tuples

```
let tuple = (1, 'A' , "Cool", 78, true);
```

Rust Strings

String Literal

```
let cs:&str = "cheat sheet";

// => Share cheat sheet for developers
println!("Share {cs} for developers");
```

String Object

```
// Creating an empty string object
let my_string = String::new;

// Converting to a string object
let S_string = a_string.to_string()

// Creating an initialized string object
let lang = String::from("Rust");
println!("First language is {lang}");
```

.capacity()

```
let rand = String::from("Random String");
rand.capacity()  // => 13
```

Calculates the capacity of the string in bytes.

.contains()

```
let name = String::from("ElementalX");
name.contains("Element") // => true
```

Checks if the substring is contained inside the original string or not.

Pushing a single character

```
let mut half_text = String::from("Hal");
half_text.push('f');    // => Half
```

Pushing an entire String

```
let mut hi = String::from("Hey there...");
hi.push_str("How are you doing??");

// => Hey there...How are you doing??
println!("{hi}");
```

Rust Operators

Comparison Operators

`e == f`	e is equal to f
`e != f`	e is NOT equal to f
`e < f`	e is less than f
`e > f`	e is greater f
`e <= f`	e is less than or equal to f
`e >= f`	e is greater or equal to f

```
let (e, f) = (1, 100);

let greater = f > e;        // => true
let less = f < e;           // => false
let greater_equal = f >= e; // => true
let less_equal = e <= f;    // => true
let equal_to = e == f;      // => false
let not_equal_to = e != f;  // => true
```

Arithmetic Operators

`a + b`	a is added to b
`a - b`	b is subtracted from a
`a / b`	a is divided by b
`a % b`	Gets remainder of a by dividing with b
`a * b`	a is multiplied with b

```
let (a, b) = (4, 5);

let sum: i32 = a + b;            // => 9
let subtractions: i32 = a - b;   // => -1
let multiplication: i32 = a * b; // => 20
let division: i32 = a / b;       // => 0
let modulus: i32 = a % b;        // => 4
```

Bitwise Operators

`g & h`	Binary AND
`g | h`	Binary OR
`g ^ h`	Binary XOR
`g ~ h`	Binary one's complement
`g << h`	Binary shift left
`g >> h`	Binary shift right

```
let (g, h) = (0x1, 0x2);

let bitwise_and = g & h;  // => 0
let bitwise_or = g | h;   // => 3
let bitwise_xor = g ^ h;  // => 3
let right_shift = g >> 2; // => 0
let left_shift = h << 4;  // => 32
```

Logical Operators

`c && d`	Both are true (AND)
`c || d`	Either is true (OR)
`!c`	c is false (NOT)

```
let (c, d) = (true, false);

let and = c && d;  // => false
let or  = c || d;  // => true
let not = !c;      // => false
```

Compound Assignment Operator

```
let mut k = 9;
let mut l = k;
```

`k += 1`	Add a value and assign, then k=9
`k -= 1`	Substrate a value and assign, then k=18
`k /= 1`	Divide a value and assign, then k=9
`k *= 1`	Multiply a value and assign, then k=81
`k |= 1`	Bitwise OR and assign, then k=89

Rust Flow Control

If Expression

```
let case1: i32 = 81;
let case2: i32 = 82;

if case1 < case2 {
```

If...Else Expression

```
let case3 = 8;
let case4 = 9;

if case3 >= case4 {
```

If...Else...if...Else Expression

```
let foo = 12;
let bar = 13;

if foo == bar {
```

```
  println!("case1 is greater than case2");
}
```

```
  println!("case3 is better than case4");
} else {
  println!("case4 is greater than case3");
}
```

```
  println!("foo is equal to bar");
} else if foo < bar {
  println!("foo less than bar");
} else if foo != bar {
  println!("foo is not equal to bar");
} else {
  println!("Nothing");
}
```

If...Let Expression

```
let mut arr1:[i64 ; 3] = [1,2,3];
if let[1,2,_] = arr1{
    println!("Works with array");
}

let mut arr2:[&str; 2] = ["one", "two"];
if let["Apple", _] = arr2{
    println!("Works with str array too");
}
```

```
let tuple_1 = ("India", 7, 90, 90.432);
if let(_, 7, 9, 78.99) = tuple_1{
    println!("Works with tuples too");
}

let tuple_2 = ( 9, 7, 89, 12, "Okay");
if let(9, 7,89, 12, blank) = tuple_2 {
    println!("Everything {blank} mate?");
}

let tuple_3 = (89, 90, "Yes");
if let(9, 89, "Yes") = tuple_3{
    println!("Pattern did match");
}
else {
    println!("Pattern did not match");
}
```

Match Expression

```
let day_of_week = 2;
  match day_of_week {
    1 => {
      println!("Its Monday my dudes");
    },
    2 => {
      println!("It's Tuesday my dudes");
    },
    3 => {
      println!("It's Wednesday my dudes");
    },
    4 => {
      println!("It's Thursday my dudes");
    },
    5 => {
      println!("It's Friday my dudes");
    },
    6 => {
      println!("It's Saturday my dudes");
    },
    7 => {
      println!("It's Sunday my dudes");
    },
    _ => {
      println!("Default!")
    }
  };
}
```

Nested...If Expression

```
let nested_conditions = 89;
if nested_conditions == 89 {
    let just_a_value = 98;
    if just_a_value >= 97 {
        println!("Greater than 97");
    }
}
```

For Loop

```
for mut i in 0..15 {
  i-=1;
  println!("The value of i is : {i}");
}
```

While Loop

```
let mut check =  0;
while check < 11{
  println!("Check is : {check}");
  check+=1;
  println!("After incrementing: {check}");

  if check == 10{
    break; // stop while
  }
}
```

Loop keyword

```
loop {
  println!("hello world forever!");
}
```

The infinite loop indicated.

Break Statement

```
let mut i = 1;
loop {
  println!("i is {i}");
  if i > 100 {
    break;
  }
  i *= 2;
}
```

Continue Statement

```
for (v, c) in (0..10+1).enumerate(){
  println!("The {c} number loop");
  if v == 9{
    println!("Here we go continue?");
    continue;
  }
  println!{"The value of v is : {v}"};
}
```

Rust Functions

Basic function

```
fn print_message(){
  println!("Hello, QuickRef.ME!");
}

fn main(){
  //Invoking a function in Rust.
  print_message();
}
```

Pass by Value

```
fn main()
{
  let x:u32 = 10;
  let y:u32 = 20;

  // => 200
  println!("Calc: {}", cal_rect(x, y));
}
fn cal_rect(x:u32, y:u32) -> u32
{
  x * y
}
```

Pass by Reference

```
fn main(){
  let mut by_ref = 3;       // => 3
  power_of_three(&mut by_ref);
  println!("{by_ref}");  // => 9
}

fn power_of_three(by_ref: &mut i32){
  // de-referencing is important
  *by_ref = *by_ref * *by_ref;
  println!("{by_ref}");  // => 9
}
```

Returns

```
fn main(){
  let (mut radius, mut pi) = (3.0, 3.14);
  let(area, _perimeter) = calculate (
      &mut radius,
      &mut pi
  );
  println!("The area and the perimeter of
the circle are: {area} & {_perimeter}");
}

fn calculate(radius : &mut f64, pi: &mut
f64) -> (f64, f64){
  let perimeter = 2.0 * *pi * *radius;
```

Arrays as Arguments

```
fn main(){
  let mut array: [i32 ; 5] = [1,2,3,4,6];
  print_arrays(array);
  println!("The elements: {array:?}");
}

fn print_arrays(mut array:[i32; 5]) {
  array[0] = 89;
  array[1] = 90;
  array[2] = 91;
  array[3] = 92;
  array[4] = 93;
  println!("The elements: {array:?}");
```

Returning Arrays

```
fn main(){
  let mut arr:[i32; 5] = [2,4,6,8,10];
  multiply(arr);
  println!("The array is : {:?}", multiply
}

fn multiply (mut arr: [i32 ; 5]) -> [i32 ;
  arr[2] = 90;
  for mut i in 0..5 {
      arr[i] = arr[i] * arr[2];
  }
  return arr;
```

```
  let area = *pi * *radius * *radius;
  return (area, perimeter);
}
```

Misc

Type Casting

```
let a_int = 90; // int
// int to float
let mut type_cast = (a_int as f64);
```

```
let orginal: char = 'I';
// char to int => 73
let type_casted: i64 = orginal as i64;
```

To perform type-casting in Rust one must use the `as` keyword.

Borrowing

```
let mut foo = 4;
let mut borrowed_foo = &foo;
println!("{borrowed_foo}");
```

```
let mut bar = 3;
let mut mutable_borrowed_bar = &mut bar;
println!("{mutable_borrowed_bar}");
```

Here borrowed value borrows the value from value one using & operator.

De-referencing

```
let mut borrow = 10;
let deref = &mut borrow;

println!("{}", *deref);
```

De-referencing in rust can be done using the * operator

Variable Scope

```
{
  // The scope limited to this braces
  let a_number = 1;
}
println!("{a_number}");
```

This will produce error as the scope of the variable `a_number` ends at the braces

Also see

The Rust Document (doc.rust-lang.org)
The Rust Reference (doc.rust-lang.org)
Rust Cheatsheet (phaiax.github.io)