Rust Programming Language Cheatsheet

A concise reference for the Rust programming language, covering syntax, data structures, control flow, and advanced features.

Basics & Syntax

Fundamental Syntax

Variable Declaration	`let variable_name: type = value;` `let mut mutable_variable = value;`

Functions	`fn function_name(parameter: type) -> return_type {`
	`  // Function body`
	`  return value;`
	`}`

Comments	// Single-line comment /* Multi-line comment */
Printing to Console	`println!("Hello, world!");` `println!("Value: {}", variable);`
Semicolons	Statements end with a semicolon `;`. Expressions that return a value do not need a semicolon at the end. Omitting the semicolon implicitly returns the last expression in a block.

Modules	`mod my_module {`
	`  // Module content`
	`}`
	
	`use my_module::my_function;`

Data Types

Scalar Types:

- `i32`, `i64` - Signed integers
- `u32`, `u64` - Unsigned integers
- `f32`, `f64` - Floating-point numbers
- `bool` - Boolean type (`true` or `false`)
- `char` - Character type (Unicode scalar values)

Compound Types:

- Tuples: `(i32, bool, f64)`
- Arrays: `[i32; 5]` (fixed size)
- Slices: dynamically sized views into a contiguous sequence

String Types:

- `String`: growable, owned, UTF-8 encoded string.
- `&str`: string slice, a view into a `String`.

Ownership and Borrowing

Rust's ownership system prevents memory errors.

Ownership Rules:

1. Each value has a variable that's its *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value is dropped.

Borrowing:

Creating references to data without taking ownership.

- `&`: immutable reference (multiple allowed)
- `&mut`: mutable reference (only one allowed at a time)

Lifetimes:

Annotated to ensure references are valid. Compiler infers most lifetimes.

Control Flow & Data Structures

Control Flow

If/Else	`if condition {`
	`  // Block of code`
	`} else if other_condition {`
	`  // Another block`
	`} else {`
	`  // Final block`
	`}`

Loops	`loop {`
	`  // Infinite loop, use 'break' to exit`
	`  if condition { break; }`
	`}`
	
	`while condition {`
	`  // While loop`
	`}`
	
	`for element in collection {`
	`  // For loop`
	`}`

Match	`match variable {`
	`  pattern1 => { /* Code */ },`
	`  pattern2 => { /* Code */ },`
	`  _ => { /* Default case */ },`
	`}`

Data Structures

Structs:

```
struct Person {
  name: String,
  age: u32,
}

let person = Person { name:
String::from("Alice"), age: 30 };
```

Enums:

```
enum Color {
  Red,
  Green,
  Blue,
}

let color = Color::Red;
```

Vectors:

Dynamically sized arrays.

```
let mut vector: Vec<i32> = Vec::new();
vector.push(1);
vector.push(2);
```

Hash Maps:

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

let mut map: HashMap<String, i32> =
HashMap::new();
map.insert(String::from("key"), 10);
```

Error Handling

`Result<T, E>`: Represents either success (`Ok(T)`) or failure (`Err(E)`).

```
fn fallible_function() -> Result<i32, String>
{
  if condition {
    Ok(value)
  } else {
    Err(String::from("Error message"))
  }
}
```

`panic!`: Unrecoverable error that terminates the program.

`?` **operator:** Propagates errors.

```
fn another_function() -> Result<i32, String> {
  let result = fallible_function()?;
  Ok(result + 1)
}
```

Advanced Features

Traits

Similar to interfaces in other languages. Define shared behavior.

```
trait Summary {
  fn summarize(&self) -> String;
}

struct NewsArticle {
  headline: String,
  author: String,
}

impl Summary for NewsArticle {
  fn summarize(&self) -> String {
    format!("{}, by {}", self.headline,
self.author)
  }
}
```

Trait Bounds: Specify that a generic type must implement a certain trait.

`fn notify<T: Summary>(item: &T) { ... }`

Generics

Generic Functions

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

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Generic Structs

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

let integer = Point { x: 5, y:
10 };
let float = Point { x: 1.0, y:
4.0 };
```

Closures

Anonymous functions that can capture their environment.

```
let add_one = |x: i32| x + 1;

println!("{}", add_one(5)); // Prints 6
```

Capturing variables:

- By reference: `&T`
- By mutable reference: `&mut T`
- By value: `T` (move semantics)

Concurrency

Threads:

```
use std::thread;

thread::spawn(|| {
  // Code to run in the new thread
});
```

Channels:

For message passing between threads.

```
use std::sync::mpsc;

let (tx, rx) = mpsc::channel();

thread::spawn(move || {
  tx.send(10).unwrap();
});

let received = rx.recv().unwrap();
println!("Got: {}", received);
```

Mutexes:

For protecting shared data.

```
use std::sync::{Mutex, Arc};

let counter = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..10 {
  let counter = Arc::clone(&counter);
  let handle = thread::spawn(move || {
    let mut num = counter.lock().unwrap();
    *num += 1;
  });
  handles.push(handle);
}

for handle in handles {
  handle.join().unwrap();
}

println!("Result: {}",
*counter.lock().unwrap());
```

Cargo & Crates

Cargo Commands

- `cargo new project_name` - Create a new project.
- `cargo build` - Build the current project.
- `cargo run` - Build and run the project.
- `cargo check` - Check the project for errors without building.
- `cargo test` - Run tests.
- `cargo doc` - Build documentation.
- `cargo publish` - Publish the crate to crates.io.

Cargo.toml

The manifest file for Rust projects. Specifies dependencies, metadata, etc.

```
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.8.0"
```

Crates

Packages of Rust code.

- crates.io: The official Rust package registry.
- Import crates using `extern crate crate_name;` (Rust 2015) or `use crate_name;` (Rust 2018+).

Common Crates:

- `rand`: Random number generation.
- `serde`: Serialization and deserialization.
- `tokio`: Asynchronous runtime.