Rust Cheatsheet

Abyan Majid, 2023

Cargo	
Concept	**Snippets/Command/Explanation/Examples**
Create a new project	`$ cargo new <project_name>`
Compile into an executable	`$ cargo build`
Compile into an executable and then run	`$ cargo run`
Check if code will compile, without building an executable	`$ cargo check`
Build for release	`$ cargo build --release`
Update dependencies	`$ cargo update`
Build the documentation for your dependencies	`$ cargo doc --open`

Variables	
Concept	**Snippets/Command/Explanation/Examples**
Declare immutable variable	`// can only be declared in functions` `let x = 5;`
Declare mutable variable	`// can only be declared in functions` `let mut x = 5;`
Constants	`// can be in functions or global scope` `const x: u32 = 5;`
Shadowing	`let x = 5;` `let x = "hello";`
Statements vs. Expressions	`// statements end with a semicolon, expressions do not.` `// besides a syntactic scope, {/* */} also denotes an expression.` `let y = {` `  let x = 3; // statement` `  x + 1 // expression` `}; // statement`

Data Types

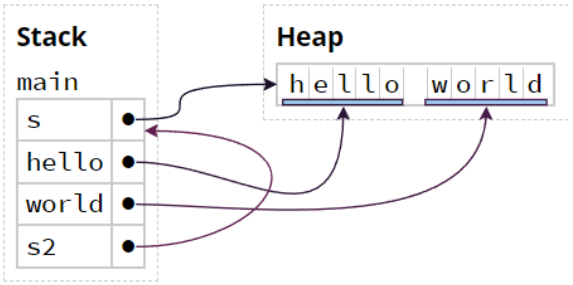
Concept	Snippets/Command/Explanation/Examples
(SCALAR) Integer	**Options:** Length / Signed / Unsigned 8-bit / `i8` / `u8` 16-bit / `i16` / `u16` 32-bit / `i32` / `u32` 64-bit / `i64` / `u64` 128-bit / `i128` / `u128` arch / `isize` / `usize` **Default:** i32
(SCALAR) Float	**Options:** f32 (single precision), f64 (double precision) **Default:** f64
(SCALAR) Boolean	**Options:** true, false `let x = true;` `let y: bool = false; // with explicit type annotation`
(SCALAR) Char	`// use single quotes` `let c = 'z';` `let z: char = 'ℤ'; // with explicit type annotation`
(COMPOUND) Tuple	`// fixed size; cannot grow or shrink, the elements may have different types` `let tup: (i32, f64, u8) = (500, 6.4, 1);` `// destructure a tuple` `let (x, y, z) = tup;` `// indexing tuples` `let a = tup.0; // a = 500` `// empty tuple "()" is called a unit`
(COMPOUND) Array	`// fixed size; cannot grow or shrink, the elements must have the same type` `let arr = [1, 2, 3, 4, 5];` `// with type and length annotation` `let arr: [i32; 5] = [1, 2, 3, 4, 5]; // type: i32, length: 5` `// initialise an array of the same values` `let arr = [3; 5]; // the same as arr = [3, 3, 3, 3, 3]` `// indexing arrays` `let first = arr[0]'`

(COMPOUND) String

```rust
let x = String::from("hello, world!"); // from std, vector of bytes
let y: &str = "hello, world!"; // string literal
```

(SLICE) String slice

```rust
let s = String::from("hello world");
let hello: &str = &s[0..5];
let world: &str = &s[6..11];
let s2: &String = &s;
```



Struct

```rust
// defining a struct
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

// creating a User instance
fn main() {
    let user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
}

// struct update syntax
let user2 = User {
    email: String::from("another@example.com"),
    ..user1 // inherit every other field from instance `user1`
};

// tuple struct
struct RGB(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = RGB(0, 0, 0);
    let origin = Point(0, 0, 0);
}

// unit-like struct (no fields)
struct UnitLikeStruct;

fn main() {
    let subject = UnitLikeStruct;
}
```

Enum	```// enum without associated data enum IpAddrKind { V4, V6, } struct IpAddr { kind: IpAddrKind, address: String, } let home = IpAddr { kind: IpAddrKind::V4, address: String::from("127.0.0.1"), }; // enum with associated data enum IpAddr { V4(u8, u8, u8, u8), V6(String), } let home = IpAddr::V4(127, 0, 0, 1); let loopback = IpAddr::V6(String::from("::1")); // special `Option<T>` enum to denote presence or absence of value enum Option<T> { None, Some(T), } // `T` is type let some_number = Some(5); let some_char = Some('e'); let absent_number: Option<i32> = None;```

Numeric Operations	
Concept	**Snippets/Command/Explanation/Examples**
Addition	```let sum = 5 + 10;```
Subtraction	```let difference = 95.5 - 4.3;```
Multiplication	```let product = 4 * 30;```
Division	```// division truncates towards zero let quotient = 56.7 / 32.2; let truncated = -5 / 3; // Results in -1```
Remainder	```let remainder = 43 % 5;```

Function & Method Syntaxes

Concept	Snippets/Command/Explanation/Examples
Function	

```rust
// if not a return type is not specified, all functions default to returning a unit/empty tuple ie. ()

fn plus_one(arg:i32) -> i32 {
    return arg + 1;
}

fn main() {
    let x = plus_one(5);
    println!("The value of x is {x}."); // The value of x is 6.
}
```

Method Syntax

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

// `impl` is a keyword to denote methods defined on the Rectangle struct
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    // method that accepts another Rectangle instance
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area() // call method `area`
    );
}

// IMPL CAN ALSO BE DONE ON AN ENUM
```

Control Flow

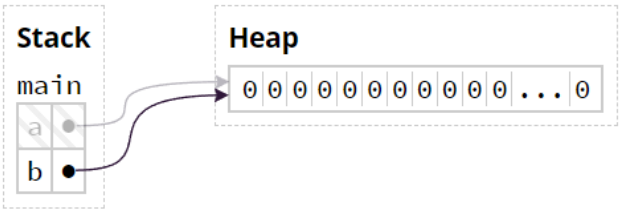
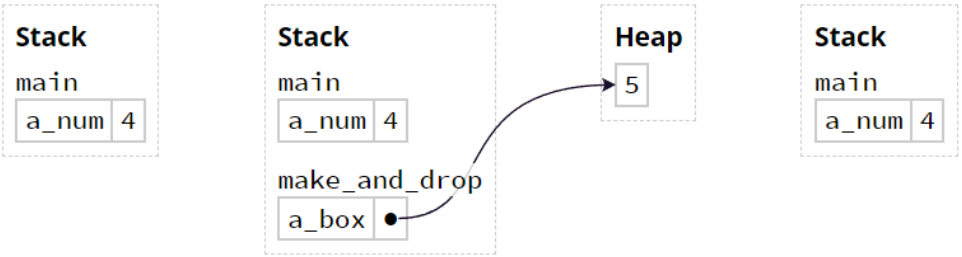
Concept	Snippets/Command/Explanation/Examples
If-Else If-Else Blocks	

```rust
fn main() {
    let number = 6;
```

	``rust if number % 4 == 0 { println!("number is divisible by 4"); } else if number % 3 == 0 { println!("number is divisible by 3"); } else if number % 2 == 0 { println!("number is divisible by 2"); } else { println!("number is not divisible by 4, 3, or 2"); } } ```
In-line If-Else	```rust fn main() { let condition = true; let number = if condition { 5 } else { 6 }; // types must be the same } ```
If-Let	```rust // USEFUL FOR ENUMS // with Option<T> let config_max = Some(3u8); if let Some(max) = config_max { println!("The maximum is configured to be {}", max); } // with other enums let mut count = 0; if let Coin::Quarter(state) = coin { println!("State quarter from {:?}!", state); } else { count += 1; } ```
Loop (Infinite)	```rust // loops indefinitely - the only way to exit is through a break statement (you can also write a value to return beside the break statement). fn main() { let mut counter = 0; let result = loop { counter += 1; if counter == 10 { break counter * 2; } }; println!("The result is {result}"); } // break from a parent loop through labels fn main() { let mut count = 0; 'counting_up: loop { println!("count = {count}"); let mut remaining = 10; loop { println!("remaining = {remaining}"); ```

	`` if remaining == 9 { break; } if count == 2 { break 'counting_up; } remaining -= 1; } count += 1; } println!("End count = {count}"); } ```
While loop	``` // Loop for as long as a condition is true fn main() { let mut number = 3; while number != 0 { println!("{number}!"); number -= 1; } println!("LIFTOFF!!!"); } ```
For loop	``` // iterate over a collection fn main() { let a = [10, 20, 30, 40, 50]; for element in a { println!("the value is: {element}"); } } // loop for a specified number of times fn main() { let a = [10, 20, 30, 40, 50]; for element in a { println!("the value is: {element}"); } } fn main() { // 1..4 is exclusive of the upper bound, so it counts from 1 to 3. // ascending count for number in 1..4 { println!("{number}!"); } // descending count for number in (1..4).rev() { println!("{number}") } } ```

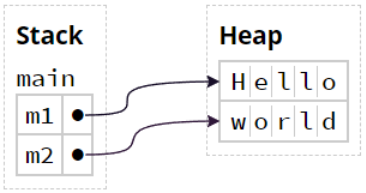
Ownership, References, Borrowing

Concept	Snippets/Command/Explanation/Examples
Box deallocation principle	If a variable owns a box, when Rust deallocates the variable's frame, then Rust deallocates the box's heap memory. (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
Moved heap data principle	If a variable x moves ownership of heap data to another variable y, then x cannot be used after the move. (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
Pointer safety principle	Data should never be aliased and mutated at the same time. (https://rust-book.cs.brown.edu/ch04-02-references-and-borrowing.html)
Put data on the heap with Box + basic ownership	`fn main() {` `  let a = Box::new([0; 1_000_000]); // a owns the array` `  let b = a; // ownership is transferred from a to b` `} // array is deallocated only once on behalf of its owner, b`  (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
Automatic heap deallocation	`fn main() {` `  let a_num = 4;` `  make_and_drop();` `}` `fn make_and_drop() {` `  let a_box = Box::new(5);` `} // a_box gets out of scope, so Rust automatically deallocates 5 from the heap.`  (https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html)
References	References are non-owning pointers that borrows an owning pointer **Example:** `fn main() {` `  let m1 = String::from("Hello");`

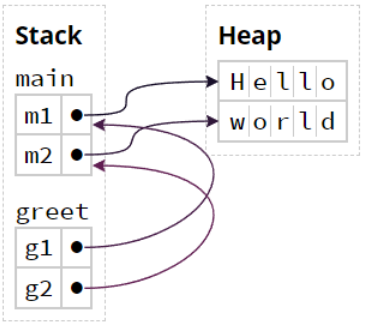
```rust
    let m2 = String::from("world"); // L1
    greet(&m1, &m2); // note the ampersands
    let s = format!("{} {}", m1, m2); // L3
}

fn greet(g1: &String, g2: &String) { // note the ampersands
    println!("{} {}!", g1, g2); // L2
}
```

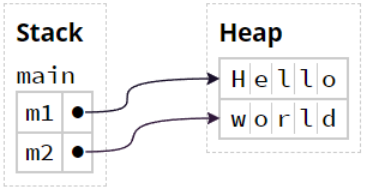
L1



L2



L3



(https://rust-book.cs.brown.edu/ch04-02-references-and-borrowing.html)