Rust CheatSheet

Hello World!

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

Compile and run this with `rustc hello_world.rs; ./hello_world`

The `main` function is special: It is always the first code that runs in every program. For now, know that 'macros' have their names ending in `!`. Packages of code are referred to as *crates* —which are shipped with *cargo*, the build tool and package manager.
Let's see a full program viz a Number Guessing Game:

```rust
use std::io;   // Use the standard input-output library
// use rand::Rng; // Use Random Number Generator trait, needs "rand" crate
use std::cmp::Ordering; // Enumeration to denote result of comparisons

fn main() {
    println!("Guess the number!");
    // let secret_number = rand::thread_rng().gen_range(1, 101); // : 1..100
    let secret_number = 7;

    // The loop keyword creates an infinite loop.
    loop {
        println!("Please input your guess.");

        // mutable variable, bound to an empty String object
        let mut guess = String::new();

        // Get a handle to the standard input for your terminal.
        // Then, stick user input into (a mutable reference to) "guess".
        // If something goes wrong, show an informative message.
        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        // Instead of "expect", we can use "match" to move from
        // crashing on an error to instead handling the error...

        // Shadow the "guess" variable to avoid using a new variable name.
        // We make an unsigned integer out of the user's string input.
        // If no number is parsed, we ask the user to guess again.
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_)  => continue // Restarts the loop
        };

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => { println!("You win!"); break }, // Exit the loop
        }
        println!("You guessed: {}", guess); // Format string
    }
}
```

Types

Rust is a statically & strongly typed language: It knows the types of all variables at compile time —it can infer them if you don't write them— and it does no implicit type coercions.

Type	Explanation ; Example value
Integers `i32, u32`	Number without decimals ; `123`
Floats `f64`	Numbers with decimals ; `1.23`
Booleans `bool`	Truth values, 1 byte ; `true, false`
Characters `char`	Unicode chars, 4 bytes ; `'t', '☺'`
Tuples $(\tau_0, \tau_1, \ldots, \tau_{n-1})$	n things of different types ; `(1, 'a', 2.3)`
Arrays $[\tau; n]$	n things of the same type ; `[1, 2, 3]`
Functions $(\tau_1, \ldots, \tau_n) \to \tau$	Transform τ_i into a τ ; `fn first<A, B>(a:A, b:B) -> A { a }`
`Range`	Numbers in a sequence ; `3..7`
References $\&\tau$	Refer to an object, without owning it ; `let s = ⋯; &s`
Mutable references `&mut` τ	Borrow an object and change it ; `let mut s = ⋯; &mut s`

A *scalar* type represents a single value; e.g., numbers, booleans, and characters. Whereas *compound types* can group multiple values into one type; e.g., tuples and arrays.

Numbers have the expected arithmetic operations `+, -, /, %` (remainder). For integers, each signed variant can store numbers from -2^{n-1} to $2^{n-1}-1$ inclusive, where n is the number of bits that variant uses. Unsigned variants can store numbers from 0 to 2^{n-1}.

Declarations

Bindings are immutable by default, which means their values can't be changed; use `mut` to allow changes.

```rust
fn main() {
    let _x     = 10;
    let y: i32 = 20; // With explicit type anotation (integer 32-bit)
    let y      = 'a'; // 'Shadow' y
    let mut z  = 12; // A mutable (changeable) variable
    z = 13;
    let _ = println!("{0} and {1}", y, z); // Ignore result
}
```

Unused bindings are likely an error, so the compiler warns about them —*unless* a name starts with an underscore.

'Shadowing' (overriding) let's us *reuse* names: Mutable variables can change their value *only*, whereas shadowing means we can change the value *and* the type. Shadowing is done with `let`, whereas mutable variables can be assigned to directly.

With shadowing, `let x = f(x)`, we can perform a few transformations on a value but have the variable be immutable after those transformations have been completed.

Tuples and Arrays

Both tuples and arrays collect multiple values and have fixed (non-growable) sizes; the only difference is that the values in a tuple can be of a different type whereas an array requires them to all have the same type.

```rust
fn main() {
    // Create: Tuples, different types allowed
    let p = ('a', 1, 2.3, "bye");
    let q: (char, i32) = ('a', 12); // Explicit type annotation

    // Create: Arrays, can only be same type
    let a = [1, 2, 3];
    let b : [char; 2] = ['x', 'y']; // Explicit type annotation

    // Read: Tuples, with " .i " notation
    // Read: Arrays, with " [i] " notation
    println!("{0}, {1}, {2}", q.0, p.3, a[1]); // p.4, a[4] ⇒ Error!

    // Both can be *destructured*, but must match in length
    let (_, me, _, you) = p; // ≈  let me = p.1; let you = p.3;
    let [_, them, _]    = a; // ≈  let them = a[1];

    // Example: A super simple way to swap
    let x = 1; let y = 2;
    assert!(x == 1 && y == 2);
    let (y, x) = (x, y);        // Shadowing
    assert!(x == 2 && y == 1);

    // Shorthand for constant arrays
    assert!([2, 2, 2, 2] == [2; 4]);
}
```

`[τ; n]` is the type of arrays of length n with elements from type τ.
If τ is a value, then there is only one possible array: `[τ, τ, ..., τ]` (n-many times).

Imports

Dots are generally used to get field from an object such as `p.0` for pairs; the double-colon is used similarly but for namespaces as in `library::file::method`.

```rust
fn main() {
    assert!(std::cmp::min(3, 8) == 3);
}
```

- `use` directives can be used to "bring into scope" names from other namespaces.

```rust
// use std::cmp::min;         // Single import
// use std::cmp::{max, min}; // Multiple imports
use std::cmp::*;              // Imports everything
fn main() {
    assert!(min(5, 5) == max(5, 5));
}
```

- Types are namespaces too, and methods can be called as regular functions:

```rust
fn main() {
    assert!("hola".len() == str::len("hola"));
}
```

Block, Expressions, and Functions

Blocks are chunks of code surrounded by curly braces. They introduce a *new scope* (shadowing any variables in the parent scope) and are *expressions* and so have a value —being the value of the final expression within the block.

```rust
fn main() {
    let z = {
        let (x, y) = (1, 2);
        x + y
    };
    assert!(z == 3);
}
```

If the block's expression ends in a semi-colon `;`, then it becomes a statement and so the final line would become: `assert!(z == ());`.

Statements are instructions that perform some action and do not return a value —which is expressed by `()`, an empty tuple. *Expressions* evaluate to a resulting value; e.g., blocks `{}` are used to create new scopes but are expressions, as are function (and macro) calls.

A *function* is a block that is *named, parameterised, and typed.* It is declared with the `fn` keyword; its arguments *must* be typed and the resulting type of the function is declared with `->` (omitted when void). The types are required so that Rust can infer them at use sites.

Requiring type annotations in function definitions means the compiler almost never needs you to use them elsewhere in the code to figure out what you mean.

```rust
// Functions can have multiple type parameters
fn first<A, B>(a: A, b: B) -> A {
    a
}
// To require 'A' implements interfaces 'I₁' and 'I₂',
// we write 'first<A: I₁ + I₂, B>'.

fn main() {
    assert!('a' == first('a', 1));
    assert!('a' == first::<char, i32>('a', 1)); // Explicit type application
}
```

- Rust doesn't care where you define your functions, only that they're defined somewhere.
  - We could have defined `first` *after* `main`.
- A function's return value, like a block, is the final expression. But one can use `return expr;` to exit "in the middle" of a function.

Structs

```rust
fn main() {
    // Structs are declared with the struct keyword:
    struct Number {
        odd: bool,
        value: i32,
    }

    // They can be initialized using struct literals:
    let n = Number {
        odd: true,
```

```
        value: 11,
    };

    // Structs, like tuples, can be destructured.
    let Number { odd, .. } = n;

    // Projection
    assert!(odd && n.value == 11);

    // Mutable updates
    let mut _m = Number { ..n }; // Copy n's data
    _m.value = 32;

    // let patterns can be used as conditions in "if"
    if let Number { odd: true, .. } = n {
        "yay"
    } else {
        "nay"
    };

    // match arms are also patterns, just like if let:
    // match n { Number {odd, ..} => "has an ODD field", _ => "nope"};
}
```

You can declare methods on your own types:

```
struct Number { odd: bool,
        value: i32,
    }

impl Number {
    fn is_strictly_positive(self) -> bool {
        self.value > 0
    }
}

fn main() {
    let n = Number {
        odd: true,
        value: 11,
    };
    assert!(n.is_strictly_positive());
    println!("HOLA");
}
```

Structs can be generic too:

```
struct Pair<A, B> {
    first: A,
    second: B,
}

fn main() {
    assert!(
        (Pair {
            first: 1,
            second: 'b'
        })
        .first
            == 1
    );
}
```

Branching

```
fn main() {
    let b = if 0 < 1 { true } else { false };
    assert!(b == (0 < 1));

    let it = 3;
    let res = match it {
        1 => 'a',
        2 => 'b',
        3 => 'c',
        _ => 'd', // default case
    };
    assert!(res == 'c')
}
```

A *function* is a block that is *named, parameterised, and typed.* It is declared with the `fn` keyword; its arguments must be typed and the resulting type of the function is declared with `->` (omitted when void).

```
fn do_the_thing(x: i32, y: i32) -> i32 {
    x + y
}

fn main() {
    let z = do_the_thing(1, 2);
    assert!(z == 3);
}
```

- `if` can be used without an `else` but must uses {braces} for the branches.
  - The condition *must* be a `bool`.
  - When used as an expression, the bodies must have the same type.
    - `let number = if condition { 5 } else { "six" };` crashes since no type can be assigned to `number` *at compile time.*
  - Avoid nested `else if`'s by using a `match` instead.
- Rust has three kinds of loops: `loop`, `while`, and `for`. Let's try each one.
- The `loop` keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop.
  - You can place the `break` keyword within the loop to tell the program when to stop executing the loop.
  - When `loop` is used as an *expression*, say in a `let`, then `break` returns the result of the loop: `let x = loop { ⋯; break expr; ⋯}`.
  - `while cond {⋯} ≈ loop {⋯; if ! cond {break;}}`

- Loop through a collection with `for`.

```
fn main() {
    let xs = [10, 20, 30, 40, 50];
    for i in 0..xs.len()-1 { println!("{}", xs[i]) }
    // Better:
    for x in xs.iter() { println!("{}", x) }
}
```

Understanding Ownership

Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector. Therefore, it's important to understand how ownership works in Rust. In this chapter, we'll talk about ownership as well as several related features: borrowing, slices, and how Rust lays data out in memory.

my/doc/garbage-collection All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that constantly looks for no longer used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. None of the ownership features slow down your program while it's running.

Ownership Rules

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped ("it's memory is garbage collected").

We've already seen string literals, where a string value is hardcoded into our program. String literals are convenient, but they aren't suitable for every situation in which we may want to use text. One reason is that they're immutable. Another is that not every string value can be known when we write our code: for example, what if we want to take user input and store it? For these situations, Rust has a second string type, String. This type is allocated on the heap and as such is able to store an amount of text that is unknown to us at compile time. You can create a String from a string literal using the from function, like so:

- let s = String::from("hello");

```
fn main() {
    let mut s = String::from("hello");
    s.push_str(", world!"); // push_str() appends a literal to a String
    assert!(s == "hello, world!")
}
```

Just as `let mut x = 12` means we can later declare `x = 13`, the *variable is mutable* but we did not change the literal "12". Likewise, `str` are values that cannot be changed and are constructed by writing " then any text then "; that is all. (Other languages call these 'symbols'.)

For scalar variables `x` and `y`, the declaration `x = y` means "copy the value of `y` and stick it in `x`." For compound structures, `x = y` means "since there can only be one owner, make `x` refer to the object that `y` refereed to, and make `y` refer to nothing", and one says "[the object pointed to by] y has **moved** to x" —use of `y` is now a compile-time error. Copying is very expensive for objects, and sharing an object via two aliases can be confusing, so ownership is changed instead. If you really do want to (deeply) copy an object: `x = y.clone()`.

Tuples/arrays of scalars *copy*, but tuples/arrays involving non-scalars *move*.

Ownership and Functions. Passing a variable to a function will move or copy, just as assignment does. That is, `f(x)` will take ownership if `x` is a variable `x` of a compound type, and otherwise a copy of `x` is performed. In particular, `f(x); x` will result in an error for compound `x`, since `x`'s object *moved* when `f` was called and now `x` is invalid for use. In contrast, `ok` below takes ownership of `b` but then returns ownership to (a shadowing of) `b`.

```
// Three ways to get the length of a string
fn main() {
    let a = String::from("a"); theft(a); // assert!(a == "a"); // Error!
    let b = String::from("b"); let (b, _) = akward(b); assert!(b == "b"); // Okay
    let c = String::from("c"); let n = best(&c); assert!(c.len() == n); // Yay!
}

fn theft(a : String) -> usize { a.len() }
fn akward(b : String) -> (String, usize) { let n = b.len(); (b, n) }
fn best(c : &String) -> usize {c.len()}
```

Taking ownership and then returning ownership with every function is a bit tedious. What if we want to let a function use a value but not take ownership? It's quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well. Enter **references**: Method `count` wants to refer to a `String` *without* taking ownership, and so the method call expects a reference, which are constructed with `&`.

Notice that in the body of `best`, we use the argument as if it were a normal `String`.

We call having references as function parameters **borrowing**. As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back.

Just as variables are immutable by default, so are references. We're not allowed to modify something we have a reference to.

```
fn main() {
    let a = String::from("a"); let b = &a; let _c = &a; assert!(b == "a"); // OK
    // Mutable variables cannot be borrowed more than once
    // let mut x = String::from("a"); let y = &mut x; let z = &mut x; assert!(y ==

    println!("{:?}", b);
}
```

Mutable references can only be made from mutable variables.

As always, we can use curly brackets to create a new scope, allowing for multiple mutable references, just not simultaneous ones:

```
    let mut s = String::from("hello");
```

```
    {
        let r1 = &mut s;
    } // r1 goes out of scope here, so we can make a new reference with no problems.

    let r2 = &mut s;
```

We also cannot have a mutable reference while we have an immutable one. Users of an immutable reference don't expect the values to suddenly change out from under them! However, multiple immutable references are okay because no one who is just reading the data has the ability to affect anyone else's reading of the data.

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // BIG PROBLEM

    println!("{}, {}, and {}", r1, r2, r3);
}
```

Note that a reference's scope starts from where it is introduced and continues through the last time that reference is used. For instance, this code will compile because the last usage of the immutable references occurs before the mutable reference is introduced:

```
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{} and {}", r1, r2);
    // r1 and r2 are no longer used after this point

    let r3 = &mut s; // no problem
    println!("{}", r3);
```

The scopes of the immutable references r1 and r2 end after the println! where they are last used, which is before the mutable reference r3 is created. These scopes don't overlap, so this code is allowed.

Rust prevents dangling pointers.

The Rules of References Let's recap what we've discussed about references:

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

Next, we'll look at a different kind of reference: slices.

Slices

Suppose we're looking for the first "hello" in a string. If we use an index, then we have to keep the string and the index variable in-sync: Whenever the string changes, we must change the index variable. This is tedious and error prone. A better solution is *immutable string slices*: If we have an immutable reference to that part of the string, then by Rust's ownership rules, the string cannot be altered there.

A string slice is a reference to part of a String, and it looks like this:

```
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
```

This is similar to taking a reference to the whole String but with the extra [0..5] bit. Rather than a reference to the entire String, it's a reference to a portion of the String.

Slice abbreviations

"up to n"	`&s[0..`n`]`	≈	`&s[..`n`]`
"n onwards"	`&s[`n`..s.len()]`	≈	`&s[`n`..]`
"all, as slice"	`&s[0..s.len()]`	≈	`&s[..]`

The type that signifies "string slice" is written as `&str`.

String Literals Are Slices

Recall that we talked about string literals being stored inside the binary. Now that we know about slices, we can properly understand string literals:

```
let s = "Hello, world!";
```

The type of s here is &str: it's a slice pointing to that specific point of the binary. This is also why string literals are immutable; &str is an immutable reference.

When defining "string functions", it's best to take `&str` as parameters since then such functions will work with string literals (elements of type `&str`) and with `String` values `s` by passing in `&s[..]` to convert them to string slices.

String slices, as you might imagine, are specific to strings. But there's a more general slice type, too. Just as we might want to refer to a part of a string, we might want to refer to part of an array. We'd do so like this:

```
#![allow(unused)]
fn main() {
let array     : [i32; 5]  = [1, 2, 3, 4, 5];
let reference : &[i32;5]  = &array;
let slice     : &[i32]    = &array[1..3];

let a : &[i32; 5] = &[1, 2, 3, 4 ,5];
let b: &[&str] = &["one", "two", "three"];
// Why is the type of "a" not "&[i32]"?
}
```

Slices are a view into a block of memory represented as a pointer and a length.

```
fn main () {
    // Explicitly mention sizes of slices
    let a: &[i32; 5]  = &[1, 2, 3, 4, 5];
    let b: &[&str; 3] = &["one", "two", "three"];
    println!("{:?} and {:?}", a, b);

    // Or not, that's okay too
    let x: &[i32] = &[1, 2, 3, 4 ,5];
    let y: &[&str] = &["one", "two", "three"];
    println!("{:?} and {:?}", x, y);
```

```
// Mutable slices
let m : &mut[i32; 1] = &mut [1];
 m[0] = 7;

let it = String::from("it");
steal( it );
it;
}
```