Study Notes from the (Rust book)[https://github.com/rust-lang/book]

MIT License

Chapter 1 - Installation

Use rustup.
linux - need a linker (e.g. that from gcc or clang).
windows - needs Visual Studio C++ build tools (e.g. 2019)
rustup doc opens local documentation copy

Hello World

rust source files end in .rs
rust code runs in a `main` function:

```
fn main() {

}
```

'Function' calls with a ! between the function name and bracketed args are Rust macros.
Statements in rust end in semi-colons. Otherwise they are expressions, e.g. for implicit return of the final argument from a function or code block.
You can compile stand alone files with `rustc`, but it's normal to use cargo.

Hello Cargo

`cargo new project_name` creates a new Hello World project, including dependencies (`Cargo.toml`), dependency versions, `Cargo.lock` and .gitstuff (other VCSs are supported too).
Once the version are figured out they are stored in `Cargo.lock` for future reference, giving automatically reproducible builds. Hence:
`Cargo.lock` is often checked into source control.
`cargo update` will recalculate the versions and update `Cargo.lock`.
`cargo build` to compile.
`cargo build --release` to compile optimised version.
`cargo run` to build and run.
`cargo check` checks compilation works without producing an executable.

Chapter 2 Guessing Game and Chapter 3 and a bit of 4

`Cargo.toml` contains a `[dependencies]` table.
Lines are: `name = x.y.z`, crate named name, semantic version: major x, minor y, patch z. E.g. `rand = "0.8.3"`
`use std::thing` imports thing from the standard library (prelude).
`use crate_name::TraitName` refers to a trait on the crate, a definition of methods that must be implemented.
`\\ comments are after \\` . There are documentation comments as well.
`fn` declares a new function:

```
fn function_name() -> ReturnType {
    statement;  // statements don't return a value.
    statement; // but don't necessarily end in ;, e.g. function definitions are statements.
    ...
    statement; // statements can contain expressions.
    expression // never end in semi-colon. implicitly returned by functions.  Optional.
}
```

Function calls and macro calls and scope blocks (`{ \\code }`) are expressions.
Naming convention is snake_case. Code body of functions must be in `{` and `}`. Functions are 'hoisted' like JS (can be declared below where they are called). Parameter types MUST be declared in function signatures. Args not declared with `&` will cause shadowing. kwargs aren't required.
`=` can bind a value to a variable. Naming convention is snake_case.
`&` indicates an argument is a *reference*.
`let var_name: Type = val` creates an `Type` immutable variable. This cannot be assigned to again after its definition.
`Type` annotations can be added (and e.g. for `const` must be), but Rust also has type inference.
`let mut var_name` creates a mutable variable.
`const var_name` creates a constant that can be evaluated at compile time, that must be followed by a type hint. A basic operation set is supported: https://doc.rust-lang.org/reference/const_eval.html. The naming convention for them is CAPITALS plus underscores.
`::` following a type refers to an 'associated function' of that type,
e.g. `let s = String::new()` defines an immutable empty String (UTF-8 encoded).
`io::stdin().read_line(& mut input)` reads a line from stdin, appending to a string input (including `\n`. Rust doesn't include `\r` in new lines).
`.expect(msg)` handles errors indicated by the `Result` retval of a lot of methods, e.g. `read_line`, by returning a string literal `msg` and crashing the program.
`Result`, the retval of `.read_line` above, is an "enumeration" (enum). Enumerations can be in multiple possible states called variants, e.g. `OK` and `Err` for `Result`.
`let var: type = old_var.trim().parse().expect()` likely follows `.read_line`,
especially with shadowing.
`.parse` tries to parse old_var to a `type`
`.trim` removes all leading and trailing whitespace, including "\n".
`"..."` enclose a string literal. Their values are known at compilation, so can be hardcoded into the binary, making them fast and efficient.
`'...'` enclose a single `char` character. `char`s have 4 bytes and can represent a Unicode Scalar Value.
`"{}"` is an empty format string. The `{}` can contain the name of a value to interpolate, and if inside `println!` or `format!` can be followed by those value(s).
`String` manages data on the heap (described below), and can grow. e.g. `let s = String::from("hello");`. If they are

`mut` they support the `push_str` method and others. Requests memory at run-time.

`x..y` is a "range expression"
`x..=y` is an inclusive "range expression" that includes its upper bound.
`(x..y).rev()` reversed range expression.
`Ordering` is an enum from `use std::cmp::Ordering;` with variants `Less`, `Greater` and `Equal`.

```
match var_1.cmp(&var_2) {
    Ordering::Less => {} //arm 1
    Ordering::Greater => expression //arm_2
    Ordering::Equal => expression //arm_2
}
```

Example syntax for match statements.
`let y = x` will make y "shadow" x, making x leave scope afterwards.
`i8, i16, i32, i64, i128, iDDD` DDD-bit signed integer types (on stack). Twos complement representation..
`u8, u16, u32, u64, u128, uDDD` DDD-bit unsigned integer types (on stack).
`isize, usize` are 32 bit or 64 bit depending on the architecture of the machine running the program.
Number literals can use _ as thousands separators.
`0x` Prefix of a hex literal.
`0o` Prefix of an octal literal.
`0b` Prefix of a binary literal.
`Integer overflow` by default integer wrapping will occur!
To handle these you can use `wrapping_`, `checked_`, `overflowing_` and `saturating_` methods.
`f32, f64` 32 bit and 64 bit floating-point number types (signed). IEEE-754. Single, Double.
`+` addition.
`-` subtraction.
`*` multiplication.
`/` division (int or float depends on the type of the operands).
`%` remainder.
`bool` Boolean type either `true` or `false`.
`let variable: ( , , ) = ( , , )`. Tuple. Can mix types. Fixed length. May be `mut`able. Can access elements with `.0`, `.1` etc. Can be unpacked using a pattern.
`()` "unit". Empty tuple.
`let variable: [type; 9] = [ , , ... , ]`. Array. All elements must have same type. Fixed length. Allocate contents to the stack. `[ ; ]` syntax can be used to initialise to repeated values. Access elements using [0], [1], etc. Accessing out of bounds indices will cause a panic.
`Vec`. Vectors. Like arrays but can grow. Implemented using generics.
`loop { }` creates an infinite loop (most terminals allow Ctrl-C to quit).
`break` breaks out of a loop. It can also return expressions from them: `let result = loop {... break retval;}` or break specific labelled loops:
`'loop_name: loop{ break 'loop_name}`. Otherwise inner most loop only is `break`ed out of.
`continue` skips to the next iteration of a loop.
`while condition { ... }`
`for element in collection`
Rust keywords are not valid variable or function names: https://doc.rust-lang.org/book/appendix-01-keywords.html as, async, await, break, const, continue, crate, dyn, else, enum, extern, false, fn, for, if, impl, in, let, loop, match, mod, move, mut, pub, ref, return, Self, self, static, struct, super, trait, true, type, union, unsafe, use, where, while, and for future use: abstract, become, box, do, final, macro, override, priv, try, typeof, unsized, virtual and yield.
Shadowing - reassign a variable name or make a new reference to it with `let` (e.g. that can be initialised to an expression involving the old shadowed variable). Mutable variables can change type. Immutables can't.
Shadowed variables on the heap leave scope. This is like shallow copying but as it also invalidates the first variable, it is called *moving*.
Cannot do `x = y = 6` in Rust.
`if condition_1 {} else if condition_2 {} ... else {}` if/else if/ else statements.
condition must be a `bool`. It does not automatically evaluate the truthiness of a variable.
`if...` is an expression. It can be used in a `let` statement as a Ternary, e.g. `let var_name = if condition {1} else {-1}`. However the resulting expressions of all arms must be the same type.

Chapter 4 Ownership

Rust has no automatic garbage collector (it has `drop`). Nor explicit allocation and deallocation of memory.
Stack : LIFO. Fast. Data on the stack must have a known size.
Heap : Not LIFO. A bit slower, but can grow - data that changes size can be stored. Uses a pointer.
Each value in Rust has an *owner*. Only one at a a time.
A value is dropped when its owner goes out of scope.
Rust uses Resource Acquisition is Initialisation (RAII).
Rust never automatically creates deep copies (this cna be done by
`.clone`), so automatic copying is inexpensive, performance wise.
Variables can either have the `Copy` trait or the `Drop` trait.
Return values of functions can transfer ownership if their arg doesn't use a `&` to reference the arg. So using `&` is common.
`*` is the dereferencing operator.
`&mut` must be used to alter a `mut` variable.
But a `mut`able variable can have only one `&mut`able reference at a time to it.
As many immutable references as desired are allowed, but having one precludes a mutable reference.
`&variable_name[a..b]` is a "slice". It is a reference to a portion of a String or an Array. They avoid problems with byte counting in strings.

Chapter 5 Structs

```
struct StructName {
    field_name_1 : FieldType1,
    field_name_2 : FieldType2,
    ...,
}
```

Structs can be defined before the main function.

```
let instance = StructName {
    field_name_1 : value_1,
    field_name_2 : value_2,
    ....,
};
```

`instance.field_name_1` accesses fields.
`instance.field_name_2 = new_value` assigns to fields of `mut`able instances only.

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

"field init shorthand" syntax (omit repeating identically named and typed args)

```
    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
```

struct update syntax, for creating one instance from another. But this can Move variables.

```
struct Color(i32, i32, i32);
```

```
fn main() {
    let black = Color(0, 0, 0);
}
```

are tuple structs. Their fields are unnamed.

```
struct AlwaysEqual;

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

unit like structs.

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!("rect1 is {:?}", rect1);
}
```

Structs can opt into providing debugging information. The output from a format string field `{:#?}` can be easier to read.

Method syntax.

Methods are like functions obviously. They are defined within structs, enums or traits. Their first param is always self. For example:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

"impl" stands for implementation.
`&self` is short for `self: &Self`. Methods can borrow `self` mutably. It's rare for methods to take ownership, but for methods that transform instances it's possible.
`rect1.area()` method call.
Methods can share the names of fields - Rust differentiates by looking for brackets afterwards. Rust does not implement getters automatically for struct fields.
Rust doesn't have separate `->` and `.` like C and C++ (different to return type hints), it has *automatic referencing and dereferencing*.
All functions in the impl block are *"associated functions"*. but they need not all be a method with self as their first arg - e.g. factories can be included too.
Structs can have multiple impl blocks.

Chapter 6 Enums

Enums define a type by enumerating its possible variants.

```
enum EnumNameKind {
    VariantName1(Type1, ...),
    VariantName2(Type2, ...),
}
```

definition syntax.
Initialised e.g. as
`EnumNameKind::VariantName1(Type1::from(some_variable))`
There is no Null value in Rust. The Option enum should be used instead:

```
enum Option<T> {
    None,
    Some(T),
}
```

`<T>` is a generic type parameter.

match

The arms of a match statement need to be exhaustive. But catch alls can just use a simple variable, e.g. *other* in:

```
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        other => move_player(other),
    }
```

or to define 'nothing happens':

```
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => (),
    }
```

```
    let config_max = Some(3u8);
    match config_max {
        Some(max) => println!("The maximum is configured to be {}", max),
        _ => (),
    }
```

accessing the content of Some via pattern matching.

if let

```
   let config_max = Some(3u8);
    if let Some(max) = config_max {
        println!("The maximum is configured to be {}", max);
    }
```

Does same thing as above.
`if let` takes a pattern and an expression separated by an equal sign. It works like a match with one arm (and ignores all other values). The exhaustive checking that match enforces is lost.
`if let` can be followed by an `else`, which runs if there is no match.

Chapter 7 - Packages, Crates and Modules.

- Packages: A Cargo feature that lets you build, test, and share crates. A package contains a Cargo.toml file that describes how to build those crates. A package can contain at most one library crate.
- Crates: A tree of modules that produces a library or executable. A crate can be a binary crate or a library crate. Library crates don't have a main function, and they don't compile to an executable. They define functionality intended to be shared with multiple projects.
  The crate root is a source file that the Rust compiler starts from and makes up the root module of your crate
- 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

Cargo follows a convention that src/main.rs is the crate root of a binary crate with the same name as the package. Likewise, Cargo knows that if the package directory contains src/lib.rs, the package contains a library crate with the same name as the package, and src/lib.rs is its crate root.

`use`, `pub`, modules and paths.

- Start from the crate root file: `src/main.rs`, or `src/lib.rs` for a library crate.
- `mod module_name;` declares a module inside the crate root file. The compiler then looks for it in: {curly brackets after the declaration, on the same line}, `src/module_name.rs`, and in `src/module_name/mod.rs`.
- `mod sub_module_name;` within `src/module_name.rs` declared a submodule. The compiler looks for it on the same line as above,
  and in: `src/module_name/sub_module_name.rs` and `src/module_name/sub_module_name/mod.rs`
- You can then refer to code in submodules via: `crate::module_name::sub_module_name::TypeToImport`.
- `pub` before a `mod` (or any declaration) makes a module (and other items within it, including `struct` fields) public. i.e. accessible by its parent modules.
- `use` creates shortcuts to reduce repetitions of long paths.

Paths in the Module Tree.

A path can take two forms:

- An absolute path starts from a crate root by using a crate name (for code from an external crate) or a literal `crate` (for code from the current crate).
- A relative path starts from the current module and uses `self`, `super`, or an identifier in the current module.
  If `struct`s are public but have private fields, they need to provide a public factory to set private fields.
  A public `enum` automatically has all its variants public.
  To handle name clashes and names, `use crate_name::thing` supports `as NewName`.
  Both referring to items via their parent modules, and using aliases via `as` are considered idiomatic.
  `pub use` re-exports names brought into the current scope.

Nested paths.

`use std::{cmp::Ordering, io};` is the same as:

```
use std::cmp::Ordering;
use std::io;
```

`use std::io::{self, Write};` is the same as:

```
use std::io;
use std::io::Write;
```

Glob Operator

use std::collections::*;

Modules in different files.

`mod module_name` declares a module. The compiler looks for `module_name.rs` or `src/module_name/mod.rs`
`mod` is not an include - it only needs to declare an option once, and then other files in the project can refer to it via a path to that declaration (path as above).

Chapter 8. Collections.

Data structures in *collections* store data on the heap. The main 3 are:

- vector,
- string,
- hash map.
  Sets and a binary Heap are available too, as are variations of vector and hash map.

Vectors

`let v: Vec<i32> = Vec::new();` example declaration, with no initial values.
All elements contained by a Vector must have the same data type, but known different types can be stored together as variants of an Enum. unknown different types require a trait.
`let v = vec![1, 2, 3];` creating a vector using the `vec!` macro.
`Vec`s support `.push(val_to_append)`, `.pop`, `.insert`, `.len`, `.clear`, `.drain`, `.is_empty`, `.truncate` and `.append` methods.
`Vec`s can be sliced with ranges.
Vectors drop all their elements when leaving a scope.
`&v[i]` is a reference to the (i+1)th element of v. IT will cause a panic if i is out of bounds.
`v.get(i)` returns an Option<&T>, to allow handling of out of bounds indices.
An immutable reference of an element of a mutable vector cannot be created.

```
for el in &v {
    println!("{}", el);
}
```

iterates over the values in v.

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Mutating the vector. Here, * is the dereferencing operator.

Strings

`String`s can grow - they support `.push_str` to append string slices without taking ownership of the arg. `.push` takes a single `char`.
`String` is the main type. `str` is a string slice, usually seen as `&str`, e.g. string literals (hardcoded in the binary).

```
let data = "initial contents";
let s = data.to_string();
```

and
`let s = "initial contents".to_string();` both create a String from a string literal. `.to_string` is available on anything that implements the `Display` trait.
`+` concatenates strings. It calls the add method, which takes ownership of the LHS, first argument, self.
`.add` only adds an `&str` to a `String`, but the compiler can coerce an `%String` arg into an `&str` (deref coercion).

The format! macro

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
let s = s1 + "-" + &s2 + "-" + &s3;
```

returns `tic-tac-toe`, but takes ownership of s1.
`let s = format!("{}-{}-{}", s1, s2, s3);` doesn't take ownership of any args.

Indexing into Strings

Strings cannot be indexed with integers (different unicode code points need different numbers of Bytes under `utf-8`).
`String` is a wrapper over `Vec<u8>`.
Using ranges to create string slices can crash your program.

Iteration over Strings

The best way to operate on pieces of strings is to be explicit about whether you want characters or bytes.

```
for c in "नमस्ते".chars() {
    println!("{}", c);
}

for b in "नमस्ते".bytes() {
    println!("{}", b);
}
```

Crates are available on crates.io for grapheme clusters (letters).

Hash Maps

`HashMap<K, V>` stores a mapping of keys of type `K` to values of type `V` using a *hashing function*. The key can be of any type.
They are growable.
`use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);` Example use.
HashMaps are not automatically in the prelude - they need to be brought in.

```
let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let mut scores: HashMap<_, _> =
    teams.into_iter().zip(initial_scores.into_iter()).collect();
```

Creating a HashMap from a vector of keys and a vector of values.
`<_,_ >` means Rust will infer the types the HashMap contains based on the data.
HashMaps will own owned values like String, both as keys and values.
References can be inserted into a HashMap but their values must be valid while the HashMap is.
HashMaps support the `.get` method. An Option is returned.

```
for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

Iteration over a HashMap scores.

HashMaps are not ordered.
HashMaps have a special API, `entry`, for checking if a key is in a HashMap (has a value) and if not, giving it one:

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

    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);

    scores.entry(String::from("Yellow")).or_insert(50);
    scores.entry(String::from("Blue")).or_insert(50);

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

`.or_insert` returns a mutable reference, which can be used to change the val without taking ownership. For example, a word counter:

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

    let text = "hello world wonderful world";

    let mut map = HashMap::new();

    for word in text.split_whitespace() {
        let count = map.entry(word).or_insert(0);
        *count += 1;
    }

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

prints: `{"world": 2, "hello": 1, "wonderful": 1}`
The default hashing function *SipHash* of a HashMap is resistant to DDOS attacks but is a little slower. It can be swapped to a different *hasher*, e.g. from crates.io.