# 🦀 Rust Learning Notes - Absolute Beginner to Pro

## ✅ 1. Getting Started

* main() is the **entry point** of every Rust executable.
* Rust is **compiled** and **statically typed** (like C).
* Compilation output is **platform-dependent**, just like C.
* Compile manually using:
* rustc file.rs  
  ./file # or file.exe on Windows
* rustfmt formats a single file.
* Use println!() to print — it’s a **macro**, not a function (hence the !).
  + Macros are like functions, but more powerful and processed before compilation.
  + New in Rust 1.58: you can use named arguments:
  + println!("this year, my garden has {apples} apples", apples = 10);
* Interpolating variables in println!:
* println!("this year, my garden has {} apples", apples);  
  println!("this year, my garden has {} apples and {} oranges", apples, oranges);  
  println!("this year, my garden has {0} apples and {1} oranges", apples, oranges);  
  println!("this year, my garden has {0} apples and {1} oranges, I can't believe I have {0} apples", apples, oranges);

## ✅ 2. Cargo & Project Structure

* cargo = Rust’s official **build system + package manager**.
* Create a new project:
* cargo new my\_project
* Structure:
* my\_project/  
  ├── Cargo.toml  
  └── src/  
   └── main.rs
* Commands:
  + cargo build or cargo b → compiles in debug mode
  + cargo build --release → optimized for production
  + cargo run or cargo r → builds and runs (default debug mode)
  + cargo run --quiet → runs and suppresses build output
  + cargo check → only type-checks to quickly find errors
  + cargo fmt → formats all files in the project
  + cargo clean → deletes build artifacts for a fresh build

## ✅ 3. Cargo.toml & TOML

* Cargo.toml = **manifest** file for metadata & dependencies.
* Written in **TOML** (Tom’s Obvious, Minimal Language).
  + Clear, minimal syntax:
  + [package]  
    name = "hello\_world"  
    version = "0.1.0"  
    edition = "2021"  
      
    [dependencies]  
    rand = "0.8"
  + Strongly typed: supports arrays, booleans, strings, integers.
  + Similar to package.json, but stricter.

## ✅ 4. Crates, Packages, and Modules

### 📦 Crate

* A **crate** is the **smallest unit of compilation**.
* Two types:
  + **Binary Crate**: has main.rs, produces an executable.
    - A **standalone application**.
  + **Library Crate**: has lib.rs, reusable code without main().
    - Meant to be used by binary crates or other libraries.

### 📦 Package

* A **package** is a **set of crates** controlled by a Cargo.toml.
* Default layout:
* my\_app/  
  ├── Cargo.toml  
  └── src/  
   └── main.rs
* A package can have multiple crates using [[bin]] or workspaces.

### 📂 Modules

* Modules organize code inside a crate using mod keyword.
* Example:
* mod network;  
    
  fn main() {  
   network::connect();  
  }
* // in network.rs  
  pub fn connect() {  
   println!("Connected!");  
  }

## ✅ 5. How Rust Compiles to Platform-Dependent Binaries 🛠

* Rust uses rustc to compile source code into native machine code.
* Compilation targets are **platform-specific**:
  + Windows → .exe
  + Linux → ELF binary
  + macOS → Mach-O binary
* You can cross-compile using:
* rustup target add x86\_64-pc-windows-gnu  
  cargo build --target x86\_64-pc-windows-gnu
* Similar to C/C++ where you must specify target triple for cross-platform support.

## ✅ 6. Variables, Constants & Shadowing

### 📌 Variables & Mutability

* Variables are **immutable by default**.
* Use mut for mutability:
* let mut x = 5;  
  x = 6;
* Variable names should be in snake\_case.
* Use \_ before variable name if it’s intentionally unused.

### 📌 Variable Shadowing

* You can **reuse** the same variable name with new type or value:
* let grams: &str = "100.35";  
  let grams: f64 = 100.35;  
  let grams: i32 = 100;
* Shadowing allows transformation without mutability.
* Different from JavaScript (in Rust, shadowing is scoped and type-safe).

### 📌 Constants

* Defined with const, always typed and UPPER\_SNAKE\_CASE:
* const TAX\_RATE: f64 = 7.25;
* Must be known at **compile time**.
* Can be declared globally (outside functions).

### 📌 Type Alias

* You can create a type alias for readability:
* type Meters = i32;  
  let distance: Meters = 1600;

### 📌 Compiler Directives

* Directives are annotations for the compiler:
  + #[allow(unused\_variables)] → suppress warning for next line/function
  + #![allow(unused\_variables)] → suppress warning for whole file

### 📌 Rust Error Help

* Rust gives detailed errors in terminal.
* To get an explanation:
* rustc --explain E0384
* Error index: <https://doc.rust-lang.org/error_codes/error-index.html>

## ✅ 7. Data Types & Formatting

### 📌 Declaring Data Types

* You can specify a type like this:
* let x: i32 = 20;  
  let y = 20i32;
* Use underscores to separate large digits:
* let salary: i32 = 1\_00\_000;
* usize and isize types depend on system architecture:
  + On 32-bit systems: 32-bit size
  + On 64-bit systems: 64-bit size

### 📌 String Escapes

* \ escapes special characters like:
  + \n → newline
  + \t → tab
  + \" → escaped double quote
* Use raw strings with r prefix to avoid escape processing:
* let filepath: &str = r"c:\mydoc\new\videos";

### 📌 Format Specifiers

* You can control how data is printed with specifiers:
* let pi: f64 = 3.14159265359;  
  println!("This current value of pi is {:.4}", pi); // prints: 3.1416

## ✅ 8. Visibility and Access Control 📂

### Keywords:

* pub — make the item public.
* self — refer to current module.
* super — refer to parent module.
* crate — refer to root of the crate.

### Examples:

mod outer {  
 pub mod inner {  
 pub fn say\_hello() {  
 println!("Hi from inner!");  
 }  
 }  
  
 pub fn call\_inner() {  
 // relative path  
 self::inner::say\_hello();  
 // from parent  
 super::helper();  
 }  
}  
  
fn helper() {  
 println!("I'm helper in root module.");  
}

## ✅ 9. use, mod, crate Mechanics 🔧

* mod xyz; → Tells compiler to include a module file xyz.rs or xyz/mod.rs
* use → Bring items into scope
* crate:: → Start from root of the crate

### Example:

// lib.rs or main.rs  
mod utils;  
  
use crate::utils::math::add;  
  
fn main() {  
 println!("{}", add(2, 3));  
}

// utils/math.rs  
pub fn add(a: i32, b: i32) -> i32 {  
 a + b  
}