**Rust Quick Reference**

1. **Overview**

**1.1 Characteristics**

* Precompiled just like C.
* zero-cost abstractions: higher-level features that compile to lower-level code as fast as code written manually.
* Includes the official building system & packet manager **Cargo**, allowing the user to control and build dependencies.
* Snake case as the conventional style for function and variable names. In snake case, all letters are lowercase and underscores separate words. E.g. variable\_name;

**1.2 Developer tools in Rust**

* **Cargo**: the included dependency manager and build tool, makes adding, compiling, and managing dependencies painless and consistent across the Rust ecosystem.

Cargo has a mechanism that ensures you can rebuild the same artifact every build (Cargo.TOML). Cargo will use only the versions of the dependencies you specified until you indicate otherwise. To ignore manual versions, use cargo update which will update dependencies to latest versions.

* **Rustfmt**: ensures a consistent coding style across developers.
* The Rust **Language Server**: powers Integrated Development Environment (IDE) integration for code completion and inline error messages.
* **Crate**: collection of Rust source code files (Basically a library).
* The ***Prelude*** is a list of functionalities that Rust imports into every program, it includes traits of fundamental types, destructors and overloading, heap allocation, ownership, clone, comparison traits, generic conversions, iterators, heap allocated strings and vectors.

**1.3 Fields on Rust**

Command line tools, web services, DevOps tooling, embedded devices, audio and video analysis and transcoding, cryptocurrencies, bioinformatics, search engines, Internet of Things applications, machine learning, and even major parts of the Firefox web browser.

1. **Rust Fundamentals**
   1. **Variables**

let -> Creates a variable.

**{}**  -> Curly brackets are the format specifiers (called % in C) of Rust. In Rust they are just a place holder.

E.g. println! ("x = {} and y = {}", x, y);

2.1.1 Data Types

Rust is a statically typed language, which means that it must know the types of all variables at compile time. The compiler can usually infer what type we want to use based on the value and how we use it. In cases when many types are possible, such as when we converting a String to a numeric type using parse, we must add a type annotation.

E.g.let number = 13; //implicit data type

let number: i32 = 13; //explicit data type

| **Integer Types** | | |
| --- | --- | --- |
| **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 |

Rust’s **char** type is four bytes in size and represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII.

2.1.2 Arrays vs Vectors

Arrays are useful when you want your data allocated on the stack rather than the heap or when you want to ensure you always have a fixed number of elements. E.g. let a: [i32; 5] = [1, 2, 3, 4, 5]; //array with type

An array isn’t as flexible as the vector type, though. A vector is a similar collection type provided by the standard library that is allowed to grow or shrink in size

**Tuple**

A tuple is a general way of grouping together a number of values with a variety of types into one compound type. Tuples have a fixed length: once declared, they cannot grow or shrink in size.

E.g. let tup: (i32, f64, u8) = (500, 6.4, 1);

**Shadowing:** Shadowing lets us reuse variables with same name, rather than forcing us to create two unique variables. The second variable’s value is what appears when the variable is used. It’s also possible to change the type of the variable since we’re effectively creating a new variable. E.g. let x = 5; let x = x + 1; // x=6

* 1. **Traits**

--**mut**: Assigns mutable (modifiable, non-static content) attribute. In Rust variables are **immutable** by **default** in order to enforce safety and easy concurrency. E.g.: let mut guess = 5;

--**match:** allows us to compare a value against a series of patterns and then execute code based on which pattern matches. Returns an enum with 3 possible values: Less, Greater, Equal.

E.g. match number1.cmp(&number2) {

Ordering::Less => println!("Number 1 is smaller"),

Ordering::Greater => println!("Number 1 is larger!") }

--**const**: constants are ALWAYS immutable. Furthermore, constants require annotated type and can only be set to a constant expression computed in compile time. const MAX\_POINTS: u32 = 100\_000;

* 1. **Compiling**

|  |  |
| --- | --- |
| Instruction | Description |
| Cargo new {name} | Generates the packet manager folder for Cargo to manage your rust project |
| Cargo build  --release | Compiles rust program.  Compiles with optimizations (superfast code however is slower compilation time) |
| ./target/debug/{Cargo\_File} | Creates executable of the cargo project on the target |
| Cargo run |  |
| Cargo check | Check correct compiling without producing an executable (speed-up the process) |
| Use | Import library |

Figure 2.3.1 Compiling your project

* 1. **Input/output**

println!() -> Macro to print a string on screen.

* 1. **Strings**

The String type is the most common type that has ownership over the contents of the string. Growable and UTF-8 encoded.

* 1. **Methods**

Rust doesn’t care where you define your functions, only that they’re defined somewhere (unlike C, where you have to define a function before use it).

**object::method**() -> ´´method´´ is an associated function of an object type (static method).

E.g.: let guess = String::new(); //method that creates a new instance of a String

instance::method.submethod() -> Calls sub-method on method handle. E.g.: io::stdin().read\_line();

* 1. **Failures**

Result types are enumerations. For Result, the variants are Ok or Err. The Ok variant indicates the operation was successful, and inside Ok is the successfully generated value. The Err variant means the operation failed and Err contains information about how or why the operation failed. The purpose of these Result types is to encode error-handling information.

E.g.: .expect(“failed”) -> Expect Unwraps a result, yielding the content of an [Ok](https://doc.rust-lang.org/std/result/enum.Result.html" \l "variant.Ok). Otherwise panics and includes message and content of Err.

* 1. **Scope {}**

1. **Reference (&)**

**& ->** Get memory address: operator that gets the memory address (in hexadecimal) of a piece of data.

Appendix A: In deep

A1. Enumerators

Result Enumerator: is a type that represents either success ([Ok](https://doc.rust-lang.org/std/result/enum.Result.html" \l "variant.Ok)) or failure ([Err](https://doc.rust-lang.org/std/result/enum.Result.html" \l "variant.Err)). It helps to propagate errors.

pub enum Result<T, E> {

Ok(T),

Err(E),

}

Appendix C: Compiler

C1. Immutability:

The compiler guarantees that when you state immutable variables, the value really won’t change. This is done by getting compiler errors.

Note that mutating an instance in place is faster than copying and returning newly allocated instances.