1 Foundations

1.1 Memory

There are three main memory regions:

- The Stack
- The Heap
- Static memory

The **stack** contains the memory necessary for each function call and the variables that it uses, each time a function is called it gets a place on top of the stack and once it finishes the memory is de-allocated.

The **heap** contains memory that is not related with any function or any threat so it can be accessed at any time from any function. To use the heap the most common structure to use is the **Box<T>**.

The static memory contains the binary code for the program and the static values that are set at compile time and last all the life time of the program. The life time *static* indicates that a variable lives until the program end but doesn't need to be initialized at the start of the program.

1.2 Ownership

A *value* is owned by a *variable* that stores it on a memory region, when the value is assigned to an other variable or moved to an other region of memory (push to a vector, box it...) then the ownership is moved and the original variable no longer owns the value so it can no longer access it.

Some types are special and don't drop from the original variable instead are copied to the new one, so an exact copy is created on the corresponding memory region. This types have the **Copy** trait, primitive types like *i32*, *char*..., have this trait.

1.3 Borrowing and Lifetimes

You can borrow multiple shareable instances of a variable **&T** but just one that is a mutable reference to the variable **&mut T** and the compiler will make sure that when a mutable reference is alive no other reference to the variable is accessed.

2 Types

2.1 Alignment

Variables are stored in memory depending on their size, but they occupy at least the same space as the bytes on the CPU architecture in order to go faster on calculations.

Also a generic type is aligned with the biggest of its variables type size. Rust allows to use the C style memory for generic types, which preserves the order of the variables, but you can let the compiler do some optimizations on memory.

Rust allows also to not align with the architecture at cost of performance if memory is so critical.

2.1.1 Dynamically Sized Types

DTS's are also known as fat / wide pointers and they store the pointer to the memory on the heap and the size of the object.

2.2 Traits

A trait is a generic method for a given type. Can implement a default behaviour for type's that don't implement it.

2.2.1 Trait Bounds

And can be used as a type constraint for functions, so only types implementing the trait can be called as variables to the function.

2.2.2 Marker Traits

Marker traits are traits have not methods or are associated with types, they only indicate if a give type can or cannot be used in a certain way.

2.2.3 Trait implementation

At compile time Rust constructs the code for each trait implemented and for each type that is used for those traits, so on the compiled code only appears the code that is needed on run time.

The process of going from a generic type for a given trait to many non-generic types is called *monophormization*, and makes Rust optimize for each type the trait is implemented, but makes the compiler slower.

To avoid the monophormization behaviour one can make a trait dynamic by replacing the key work *impl* with *&dyn*, so the user must provide a pointer to the type or trait to be used.

3 Designing interfaces

Any project has an API. So there are some good practices that a project should follow so it can scale better and it can be used by anyone as easily as possible.

3.1 Unsurprising

The Principle of Least Surprise. Where possible your interfaces should be intuitive so when a user has to guess, guesses right.

3.1.1 Naming

The namig should be intuitive and if any name is used in more than one place it should align with each other so the user can infer what the function, type, trait... produces.

3.1.2 Traits

The types that the interface contains should implement a list of traits if possible so the users don't get confused by not having them:

1. Debug
2. Send & Sync
3. Clone & Default
4. PartialEq, PartialOrd, Hash, Eq, Ord

Some might not apply but if they do, should be there. Also the interface should handle *From* and *Into* so the types can be **wrap** and **unwrap**.

3.2 Flexibility

"Each piece of code can be thought as a contract, with a set of restrictions and promises, where some inputs with some properties are expected and an output with some properties is returned."

A good rule of thump is not be too restrictive and only do promises you can keep. Adding restrictions or removing promises usually breaks backward compatibility, on the other hand, relaxing the restrictions or adding more promises tends to be more save.

Rust restrictions and promises are usually on the signature of a function, where it is specified which type or trait should the input have and which characteristics the return will present.

3.3 Borrowed vs. Owned

Functions should take references unless they need the object in memory for some reason, this helps keeping the memory more clean and do not perform unnecessary copies of memory.

3.4 Destructors

It should be clear who takes care of cleaning and object like a I/O stream before dropping it so the memory it occupies frees.

4 Documentation

The best thing for a project that is big enough or is shared by multiple users is to have a good documentation.

Documentation should explain the obvious things that are in the interface like types, traits, functions... but more important it should contain all the unexpected behavior like errors...

Also it should contain no obvious examples of the interface, so the users can start their projects with an example and modifying things until they get the result they were looking for.

4.1 Type guidance

A useful thing to do is to create dummy types to use in the function as a boolean instead of a *true* or *false*, so the signatures and the usage of the function is more intuitive.

For example if an input can be a string so the function does something different depending on the string, would be interesting to have an enum with types so its more natural the options that are available.

5 Error Handling

Functions that can error in rust must return a *Result<T, E>*, where the *E* represents de error that is returned. Errors can be handled in tow main ways

5.1 Enumerate Errors

If the error can have different type and response depending on where it is generated, it's useful to return an *enum* indicating the kind of error so the called can identify it and decide how to handle it.

In this case the types on the enumeration can can have different traits depending on their characteristics, but **all** must implement the *std::error::Error* trait, and as long as possible the *Display* and *Debug* traits, and if possible the *Sync* and *Send* traits are also important.

5.2 Opaque Errors

If the errors on the other hand just provide information that will be useful to the user but not to the program in order to perform different things, we should use the a single opaque type error that contains the traits *Error, Send, Debug* and *Display*.

5.3 Propagating Errors

When a function can throw an error and the caller function don't have to take care of it we can just propagate it back.

Rust's *?* operator acts as a shorthand for *unwrap or return early*. It helps working easily with errors so they can be propagated back without to verbose.

6 Project Structure

6.1 Features

Features are flags that Cargo uses to add optional functionality to your project and its dependencies.

In a project some part of the code can be left as optional behind a feature in order to make the compilation and de dependencies easier for the user. This can be done by:

```
    #[cfg(feature = "some-feature")]
```

6.2 Work-spaces

The project can be split in different parts in a workspace to reduce the compile times and the usability, so one change is made only the modified part of the project will be compiled. To do it add this to the *Cargo.toml* file:

```
    [workspace]
    members = [
        "foo",
        "bar/one",
        "bar/two",
    ]
```

6.3 Project Configuration

On the *Cargo.toml* file we can add some metadata about the author, path to the project or to the documentation, etc.

Also the build configuration can be specified from here so the compiler knows what to do for every kind of build that encounters. We can specify the optimization level, the debug options or how the program should **panic**. We can even set different optimization options to the different dependencies.

6.4 Conditional Compilation

We can specify which parts of the project will be compiled depending on some options like, features, operating systems, context (debug, test. . .) or cpu architecture. This conditioning can also be applied to dependencies.

6.5 Versioning

There are three levels of versioning,

1. Breaking changes, that require a major version change
2. Additions, that require a minor version change
3. Bug fix, that require a patch version change

On top of that if the project is public we should respect the **MSRV** which stands for *Minimum Supported Rust Version*, for our project and should be supported for at least 6 to 12 month.

To not make the version number too large we can name with a suffix like *-alpha.1* the unreleased versions.

7 Testing

In Rust the majority of testing is done using the attribute #[test] and the *tests/* directory.

7.1 Rust Testing mechanism

To execute the test of the project use the command *cargo test* this enables all the code flagged with *cfg(test)*. Then generates the **test harness**.

7.1.1 Test Harness

The test harness is created by the compiler and consists of a main function that invokes all the #[test] functions of the code.

One can opt out from the harness to create the tests.

The integration test (i.e. the tests on the directory textittests/) are build on a separate crate so they can only access the public interface of the code.

The #[cfg(test)] allows to build code only to run the tests, also can be combined with other