How to write documentation

Good documentation is not natural. There are opposing goals that make writing good documentation difficult. It requires expertise in the subject but also writing to a novice perspective. Documentation therefore often glazes over implementation detail, or leaves readers with unanswered questions.

There are a few tenets to Rust documentation that can help guide anyone through the process of documenting libraries so that everyone has an ample opportunity to use the code.

This chapter covers not only how to write documentation but specifically how to write **good** documentation. It is important to be as clear as you can, and as complete as possible. As a rule of thumb: the more documentation you write for your crate the better. If an item is public then it should be documented.

Getting Started

Documenting a crate should begin with front-page documentation. As an example, the `hashbrown` crate level documentation summarizes the role of the crate, provides links to explain technical details, and explains why you would want to use the crate.

After introducing the crate, it is important that the front-page gives an example of how to use the crate in a real world setting. Stick to the library's role in the example, but do so without shortcuts to benefit users who may copy and paste the example to get started.

`futures` uses inline comments to explain line by line the complexities of using a `Future`, because a person's first exposure to rust's `Future` may be this example.

The `backtrace` documentation walks through the whole process, explaining changes made to the `Cargo.toml` file, passing command line arguments to the compiler, and shows a quick example of backtrace in the wild.

Finally, the front-page can eventually become a comprehensive reference how to use a crate, like `regex`. In this front page, all requirements are outlined, the edge cases shown, and practical examples provided. The front page goes on to show how to use regular expressions then concludes with crate features.

Don't worry about comparing your crate, which is just beginning, to other more developed crates. To get the documentation to something more polished, start incrementally and put in an introduction, example, and features. Rome was not built in a day!

The first lines within the `lib.rs` will compose the front-page, and they use a different convention than the rest of the rustdocs. Lines should start with `//!` which indicate module-level or crate-level documentation. Here's a quick example of the difference:

```
//! Fast and easy queue abstraction.
//!
//! Provides an abstraction over a queue.  When the abstraction is used
//! there are these advantages:
//! - Fast
//! - [`Easy`]
//!
//! [`Easy`]: http://thatwaseasy.example.com

/// This module makes it easy.
pub mod easy {

    /// Use the abstraction function to do this specific thing.
    pub fn abstraction() {}

}
```

Ideally, this first line of documentation is a sentence without highly technical details, but with a good description of where this crate fits within the rust ecosystem. Users should know whether this crate meets their use case after reading this line.

Documenting components

Whether it is modules, structs, functions, or macros: the public API of all code should have documentation. Rarely does anyone complain about too much documentation!

It is recommended that each item's documentation follows this basic structure:

```
[short sentence explaining what it is]

[more detailed explanation]

[at least one code example that users can copy/paste to try it]

[even more advanced explanations if necessary]
```

This basic structure should be straightforward to follow when writing your documentation; while you might think that a code example is trivial, the examples are really important because they can help users understand what an item is, how it is used, and for what purpose it exists.

Let's see an example coming from the standard library by taking a look at the `std::env::args()` function:

```
Returns the arguments which this program was started with (normally passed
via the command line).
```

```
The first element is traditionally the path of the executable, but it can be
set to arbitrary text, and may not even exist. This means this property should
not be relied upon for security purposes.

On Unix systems shell usually expands unquoted arguments with glob patterns
(such as `*` and `?`). On Windows this is not done, and such arguments are
passed as-is.

# Panics

The returned iterator will panic during iteration if any argument to the
process is not valid unicode. If this is not desired,
use the [`args_os`] function instead.

# Examples

```
use std::env;

// Prints each argument on a separate line
for argument in env::args() {
    println!("{argument}");
}
```

[`args_os`]: ./fn.args_os.html
```

Everything before the first empty line will be reused to describe the component in searches and module overviews. For example, the function `std::env::args()` above will be shown on the `std::env` module documentation. It is good practice to keep the summary to one line: concise writing is a goal of good documentation.

Because the type system does a good job of defining what types a function passes and returns, there is no benefit of explicitly writing it into the documentation, especially since `rustdoc` adds hyper links to all types in the function signature.

In the example above, a 'Panics' section explains when the code might abruptly exit, which can help the reader prevent reaching a panic. A panic section is recommended every time edge cases in your code can be reached if known.

As you can see, it follows the structure detailed above: it starts with a short sentence explaining what the functions does, then it provides more information and finally provides a code example.

Markdown

`rustdoc` uses the CommonMark Markdown specification. You might be interested in taking a look at their website to see what's possible:

- CommonMark quick reference
- current spec

In addition to the standard CommonMark syntax, `rustdoc` supports several extensions:

Strikethrough

Text may be rendered with a horizontal line through the center by wrapping the text with two tilde characters on each side:

```
An example of ~~strikethrough text~~.
```

This example will render as:

> An example of ~~strikethrough text~~.

This follows the GitHub Strikethrough extension.

Footnotes

A footnote generates a small numbered link in the text which when clicked takes the reader to the footnote text at the bottom of the item. The footnote label is written similarly to a link reference with a caret at the front. The footnote text is written like a link reference definition, with the text following the label. Example:

```
This is an example of a footnote[^note].

[^note]: This text is the contents of the footnote, which will be rendered
    towards the bottom.
```

This example will render as:

> This is an example of a footnote[1].

The footnotes are automatically numbered based on the order the footnotes are written.

Tables

Tables can be written using pipes and dashes to draw the rows and columns of the table. These will be translated to HTML table matching the shape. Example:

```
| Header1 | Header2 |
|---------|---------|
| abc     | def     |
```

This example will render similarly to this:

[1] This text is the contents of the footnote, which will be rendered towards the bottom.

Header1	Header2
abc	def

See the specification for the GitHub Tables extension for more details on the exact syntax supported.

Task lists

Task lists can be used as a checklist of items that have been completed. Example:

```
- [x] Complete task
- [ ] Incomplete task
```

This will render as:

- ☒ Complete task
- ☐ Incomplete task

See the specification for the task list extension for more details.

Smart punctuation

Some ASCII punctuation sequences will be automatically turned into fancy Unicode characters:

ASCII sequence	Unicode
`--`	–
`---`	—
`...`	…
`"`	“ or ”, depending on context
`'`	‘ or ’, depending on context

So, no need to manually enter those Unicode characters!