What is rustdoc?

The standard Rust distribution ships with a tool called `rustdoc`. Its job is to generate documentation for Rust projects. On a fundamental level, Rustdoc takes as an argument either a crate root or a Markdown file, and produces HTML, CSS, and JavaScript.

Basic usage

Let's give it a try! Let's create a new project with Cargo:

```
$ cargo new docs
$ cd docs
```

In `src/lib.rs`, you'll find that Cargo has generated some sample code. Delete it and replace it with this:

```
/// foo is a function
fn foo() {}
```

Let's run `rustdoc` on our code. To do so, we can call it with the path to our crate root like this:

```
$ rustdoc src/lib.rs
```

This will create a new directory, `doc`, with a website inside! In our case, the main page is located in `doc/lib/index.html`. If you open that up in a web browser, you'll see a page with a search bar, and "Crate lib" at the top, with no contents. There's two problems with this: first, why does it think that our package is named "lib"? Second, why does it not have any contents?

The first problem is due to `rustdoc` trying to be helpful; like `rustc`, it assumes that our crate's name is the name of the file for the crate root. To fix this, we can pass in a command-line flag:

```
$ rustdoc src/lib.rs --crate-name docs
```

Now, `doc/docs/index.html` will be generated, and the page says "Crate docs."

For the second issue, it's because our function `foo` is not public; `rustdoc` defaults to generating documentation for only public functions. If we change our code...

```
/// foo is a function
pub fn foo() {}
```

... and then re-run `rustdoc`:

```
$ rustdoc src/lib.rs --crate-name docs
```

We'll have some generated documentation. Open up `doc/docs/index.html` and check it out! It should show a link to the `foo` function's page, which is located at `doc/docs/fn.foo.html`. On that page, you'll see the "foo is a function" we put inside the documentation comment in our crate.

Using rustdoc with Cargo

Cargo also has integration with `rustdoc` to make it easier to generate docs. Instead of the `rustdoc` command, we could have done this:

```
$ cargo doc
```

Internally, this calls out to `rustdoc` like this:

```
$ rustdoc --crate-name docs src/lib.rs -o <path>/docs/target/doc -L
dependency=<path>/docs/target/debug/deps
```

You can see this with `cargo doc --verbose`.

It generates the correct `--crate-name` for us, as well as pointing to `src/lib.rs` But what about those other arguments? `-o` controls the *o*utput of our docs. Instead of a top-level `doc` directory, you'll notice that Cargo puts generated documentation under `target`. That's the idiomatic place for generated files in Cargo projects. Also, it passes `-L`, a flag that helps rustdoc find the dependencies your code relies on. If our project used dependencies, we'd get documentation for them as well!

Using standalone Markdown files

`rustdoc` can also generate HTML from standalone Markdown files. Let's give it a try: create a `README.md` file with these contents:

~~~
# Docs

This is a project to test out `rustdoc`.

[Here is a link!](https://www.rust-lang.org)

## Subheading

```rust
fn foo() -> i32 {
    1 + 1
}
```
~~~

And call `rustdoc` on it:

```
$ rustdoc README.md
```

You'll find an HTML file in `docs/doc/README.html` generated from its Markdown contents.

Cargo currently does not understand standalone Markdown files, unfortunately.

Summary

This covers the simplest use-cases of `rustdoc`. The rest of this book will explain all of the options that `rustdoc` has, and how to use them.

How to write documentation

This chapter covers not only how to write documentation but specifically how to write **good** documentation. Something to keep in mind when writing documentation is that your audience is not just yourself but others who simply don't have the context you do. 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.

Basic structure

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 and, while you might think that a code example is trivial, the examples are really important because they can help your users to 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
~~~

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` is using the commonmark markdown specification. You might be interested into taking a look at their website to see what's possible to do.

Lints

To be sure that you didn't miss any item without documentation or code examples, you can take a look at the rustdoc lints here.

Command-line arguments

Here's the list of arguments you can pass to `rustdoc`:

`-h`/`--help`: help

Using this flag looks like this:

```
$ rustdoc -h
$ rustdoc --help
```

This will show `rustdoc`'s built-in help, which largely consists of a list of possible command-line flags.

Some of `rustdoc`'s flags are unstable; this page only shows stable options, `--help` will show them all.

`-V`/`--version`: version information

Using this flag looks like this:

```
$ rustdoc -V
$ rustdoc --version
```

This will show `rustdoc`'s version, which will look something like this:

```
rustdoc 1.17.0 (56124baa9 2017-04-24)
```

`-v`/`--verbose`: more verbose output

Using this flag looks like this:

```
$ rustdoc -v src/lib.rs
$ rustdoc --verbose src/lib.rs
```

This enables "verbose mode", which means that more information will be written to standard out. What is written depends on the other flags you've passed in. For example, with `--version`:

```
$ rustdoc --verbose --version
rustdoc 1.17.0 (56124baa9 2017-04-24)
binary: rustdoc
commit-hash: hash
commit-date: date
host: host-triple
release: 1.17.0
LLVM version: 3.9
```

`-r`/`--input-format`: input format

This flag is currently ignored; the idea is that `rustdoc` would support various input formats, and you could specify them via this flag.

Rustdoc only supports Rust source code and Markdown input formats. If the file ends in `.md` or `.markdown`, `rustdoc` treats it as a Markdown file. Otherwise, it assumes that the input file is Rust.

`-w`/`--output-format`: output format

This flag is currently ignored; the idea is that `rustdoc` would support various output formats, and you could specify them via this flag.

Rustdoc only supports HTML output, and so this flag is redundant today.

`-o`/`--output`: output path

Using this flag looks like this:

```
$ rustdoc src/lib.rs -o target/doc
$ rustdoc src/lib.rs --output target/doc
```

By default, `rustdoc`'s output appears in a directory named `doc` in the current working directory. With this flag, it will place all output into the directory you specify.

`--crate-name`: controlling the name of the crate

Using this flag looks like this:

```
$ rustdoc src/lib.rs --crate-name mycrate
```

By default, `rustdoc` assumes that the name of your crate is the same name as the `.rs` file. `--crate-name` lets you override this assumption with whatever name you choose.

`-L`/`--library-path`: where to look for dependencies

Using this flag looks like this:

```
$ rustdoc src/lib.rs -L target/debug/deps
$ rustdoc src/lib.rs --library-path target/debug/deps
```

If your crate has dependencies, `rustdoc` needs to know where to find them. Passing `--library-path` gives `rustdoc` a list of places to look for these dependencies.

This flag takes any number of directories as its argument, and will use all of them when searching.

`--cfg`: passing configuration flags

Using this flag looks like this:

```
$ rustdoc src/lib.rs --cfg feature="foo"
```

This flag accepts the same values as `rustc --cfg`, and uses it to configure compilation. The example above uses `feature`, but any of the `cfg` values are acceptable.

`--extern`: specify a dependency's location

Using this flag looks like this:

```
$ rustdoc src/lib.rs --extern lazy-static=/path/to/lazy-static
```

Similar to `--library-path`, `--extern` is about specifying the location of a dependency. `--library-path` provides directories to search in, `--extern` instead lets you specify exactly which dependency is located where.

`-C`/`--codegen`: pass codegen options to rustc

Using this flag looks like this:

```
$ rustdoc src/lib.rs -C target_feature=+avx
$ rustdoc src/lib.rs --codegen target_feature=+avx

$ rustdoc --test src/lib.rs -C target_feature=+avx
$ rustdoc --test src/lib.rs --codegen target_feature=+avx

$ rustdoc --test README.md -C target_feature=+avx
$ rustdoc --test README.md --codegen target_feature=+avx
```

When rustdoc generates documentation, looks for documentation tests, or executes documentation tests, it needs to compile some rust code, at least part-way. This flag allows you to tell rustdoc to provide some extra codegen options to rustc when it runs these compilations. Most of the time, these options won't affect a regular documentation run, but if something depends on target features to be enabled, or documentation tests need to use some additional options, this flag allows you to affect that.

The arguments to this flag are the same as those for the `-C` flag on rustc. Run `rustc -C help` to get the full list.

`--passes`: add more rustdoc passes

Using this flag looks like this:

```
$ rustdoc --passes list
$ rustdoc src/lib.rs --passes strip-priv-imports
```

An argument of "list" will print a list of possible "rustdoc passes", and other arguments will be the name of which passes to run in addition to the defaults.

For more details on passes, see the chapter on them.

See also `--no-defaults`.

`--no-defaults`: don't run default passes

Using this flag looks like this:

```
$ rustdoc src/lib.rs --no-defaults
```

By default, `rustdoc` will run several passes over your code. This removes those defaults, allowing you to use `--passes` to specify exactly which passes you want.

For more details on passes, see the chapter on them.

See also `--passes`.

`--test`: run code examples as tests

Using this flag looks like this:

```
$ rustdoc src/lib.rs --test
```

This flag will run your code examples as tests. For more, see the chapter on documentation tests.

See also `--test-args`.

`--test-args`: pass options to test runner

Using this flag looks like this:

```
$ rustdoc src/lib.rs --test --test-args ignored
```

This flag will pass options to the test runner when running documentation tests. For more, see the chapter on documentation tests.

See also `--test`.

`--target`: generate documentation for the specified target triple

Using this flag looks like this:

```
$ rustdoc src/lib.rs --target x86_64-pc-windows-gnu
```

Similar to the `--target` flag for `rustc`, this generates documentation for a target triple that's different than your host triple.

All of the usual caveats of cross-compiling code apply.

`--markdown-css`: include more CSS files when rendering markdown

Using this flag looks like this:

```
$ rustdoc README.md --markdown-css foo.css
```

When rendering Markdown files, this will create a `<link>` element in the `<head>`

section of the generated HTML. For example, with the invocation above,

```
<link rel="stylesheet" type="text/css" href="foo.css">
```

will be added.

When rendering Rust files, this flag is ignored.

`--html-in-header`: include more HTML in

Using this flag looks like this:

```
$ rustdoc src/lib.rs --html-in-header header.html
$ rustdoc README.md --html-in-header header.html
```

This flag takes a list of files, and inserts them into the `<head>` section of the rendered documentation.

`--html-before-content`: include more HTML before the content

Using this flag looks like this:

```
$ rustdoc src/lib.rs --html-before-content extra.html
$ rustdoc README.md --html-before-content extra.html
```

This flag takes a list of files, and inserts them inside the `<body>` tag but before the other content `rustdoc` would normally produce in the rendered documentation.

`--html-after-content`: include more HTML after the content

Using this flag looks like this:

```
$ rustdoc src/lib.rs --html-after-content extra.html
$ rustdoc README.md --html-after-content extra.html
```

This flag takes a list of files, and inserts them before the `</body>` tag but after the other content `rustdoc` would normally produce in the rendered documentation.

`--markdown-playground-url`: control the location of the playground

Using this flag looks like this:

```
$ rustdoc README.md --markdown-playground-url https://play.rust-
lang.org/
```

When rendering a Markdown file, this flag gives the base URL of the Rust Playground, to use for generating `Run` buttons.

`--markdown-no-toc`: don't generate a table of contents

Using this flag looks like this:

```
$ rustdoc README.md --markdown-no-toc
```

When generating documentation from a Markdown file, by default, `rustdoc` will generate a table of contents. This flag suppresses that, and no TOC will be generated.

`-e`/`--extend-css`: extend rustdoc's CSS

Using this flag looks like this:

```
$ rustdoc src/lib.rs -e extra.css
$ rustdoc src/lib.rs --extend-css extra.css
```

With this flag, the contents of the files you pass are included at the bottom of Rustdoc's `theme.css` file.

While this flag is stable, the contents of `theme.css` are not, so be careful! Updates may break your theme extensions.

`--sysroot`: override the system root

Using this flag looks like this:

```
$ rustdoc src/lib.rs --sysroot /path/to/sysroot
```

Similar to `rustc --sysroot`, this lets you change the sysroot `rustdoc` uses when compiling your code.

`--edition`: control the edition of docs and doctests

Using this flag looks like this:

```
$ rustdoc src/lib.rs --edition 2018
$ rustdoc --test src/lib.rs --edition 2018
```

This flag allows `rustdoc` to treat your rust code as the given edition. It will compile doctests with the given edition as well. As with `rustc`, the default edition that `rustdoc` will use is `2015` (the first edition).

`--theme`: add a theme to the documentation output

Using this flag looks like this:

```
$ rustdoc src/lib.rs --theme /path/to/your/custom-theme.css
```

`rustdoc`'s default output includes two themes: `light` (the default) and `dark`. This flag allows you to add custom themes to the output. Giving a CSS file to this flag adds it to your documentation as an additional theme choice. The theme's name is determined by its filename; a theme file named `custom-theme.css` will add a theme named `custom-theme` to the documentation.

`--check-theme`: verify custom themes against the default theme

Using this flag looks like this:

```
$ rustdoc --check-theme /path/to/your/custom-theme.css
```

While `rustdoc`'s HTML output is more-or-less consistent between versions, there is no guarantee that a theme file will have the same effect. The `--theme` flag will still allow you to add the theme to your documentation, but to ensure that your theme works as expected, you can use this flag to verify that it implements the same CSS rules as the official `light` theme.

`--check-theme` is a separate mode in `rustdoc`. When `rustdoc` sees the `--check-theme` flag, it discards all other flags and only performs the CSS rule comparison operation.

`--crate-version`: control the crate version

Using this flag looks like this:

```
$ rustdoc src/lib.rs --crate-version 1.3.37
```

When `rustdoc` receives this flag, it will print an extra "Version (version)" into the sidebar of the crate root's docs. You can use this flag to differentiate between different versions of your library's documentation.

The `#[doc]` attribute

The `#[doc]` attribute lets you control various aspects of how `rustdoc` does its job.

The most basic function of `#[doc]` is to handle the actual documentation text. That is, `///` is syntax sugar for `#[doc]`. This means that these two are the same:

```
/// This is a doc comment.
#[doc = " This is a doc comment."]
```

(Note the leading space in the attribute version.)

In most cases, `///` is easier to use than `#[doc]`. One case where the latter is easier is when generating documentation in macros; the `collapse-docs` pass will combine multiple `#[doc]` attributes into a single doc comment, letting you generate code like this:

```
#[doc = "This is"]
#[doc = " a "]
#[doc = "doc comment"]
```

Which can feel more flexible. Note that this would generate this:

```
#[doc = "This is\n a \ndoc comment"]
```

but given that docs are rendered via Markdown, it will remove these newlines.

The `doc` attribute has more options though! These don't involve the text of the output, but instead, various aspects of the presentation of the output. We've split them into two kinds below: attributes that are useful at the crate level, and ones that are useful at the item level.

At the crate level

These options control how the docs look at a crate level.

`html_favicon_url`

This form of the `doc` attribute lets you control the favicon of your docs.

```
#![doc(html_favicon_url = "https://example.com/favicon.ico")]
```

This will put `<link rel="shortcut icon" href="{}">` into your docs, where the string for the attribute goes into the `{}`.

If you don't use this attribute, there will be no favicon.

`html_logo_url`

This form of the `doc` attribute lets you control the logo in the upper left hand side of the docs.

```
#![doc(html_logo_url = "https://example.com/logo.jpg")]
```

This will put `<a href='index.html'><img src='{}' alt='logo' width='100'></a>` into your docs, where the string for the attribute goes into the `{}`.

If you don't use this attribute, there will be no logo.

`html_playground_url`

This form of the `doc` attribute lets you control where the "run" buttons on your documentation examples make requests to.

```
#![doc(html_playground_url = "https://playground.example.com/")]
```

Now, when you press "run", the button will make a request to this domain.

If you don't use this attribute, there will be no run buttons.

`issue_tracker_base_url`

This form of the `doc` attribute is mostly only useful for the standard library; When a feature is unstable, an issue number for tracking the feature must be given. `rustdoc` uses this number, plus the base URL given here, to link to the tracking issue.

```
#![doc(issue_tracker_base_url = "https://github.com/rust-lang/rust
/issues/")]
```

`html_root_url`

The `#[doc(html_root_url = "…")]` attribute value indicates the URL for generating links to external crates. When rustdoc needs to generate a link to an item in an external crate, it will first check if the extern crate has been documented locally on-disk, and if so link directly to it. Failing that, it will use the URL given by the `--extern-html-root-url` command-line flag if available. If that is not available, then it will use the `html_root_url` value in the extern crate if it is available. If that is not available, then the extern items will not be linked.

```
#![doc(html_root_url = "https://docs.rs/serde/1.0")]
```

`html_no_source`

By default, `rustdoc` will include the source code of your program, with links to it in the docs. But if you include this:

```
#![doc(html_no_source)]
```

it will not.

`test(no_crate_inject)`

By default, `rustdoc` will automatically add a line with `extern crate my_crate;` into each doctest. But if you include this:

```
#![doc(test(no_crate_inject))]
```

it will not.

`test(attr(...))`

This form of the `doc` attribute allows you to add arbitrary attributes to all your doctests. For example, if you want your doctests to fail if they produce any warnings, you could add this:

```
#![doc(test(attr(deny(warnings))))]
```

At the item level

These forms of the `#[doc]` attribute are used on individual items, to control how they are documented.

`#[doc(no_inline)]`/`#[doc(inline)]`

These attributes are used on `use` statements, and control where the documentation shows up. For example, consider this Rust code:

```rust
pub use bar::Bar;

/// bar docs
pub mod bar {
    /// the docs for Bar
    pub struct Bar;
}
```

The documentation will generate a "Re-exports" section, and say `pub use bar::Bar;`, where `Bar` is a link to its page.

If we change the `use` line like this:

```rust
#[doc(inline)]
pub use bar::Bar;
```

Instead, `Bar` will appear in a `Structs` section, just like `Bar` was defined at the top level, rather than `pub use`'d.

Let's change our original example, by making `bar` private:

```rust
pub use bar::Bar;

/// bar docs
mod bar {
    /// the docs for Bar
    pub struct Bar;
}
```

Here, because `bar` is not public, `Bar` wouldn't have its own page, so there's nowhere to link to. `rustdoc` will inline these definitions, and so we end up in the same case as the `#[doc(inline)]` above; `Bar` is in a `Structs` section, as if it were defined at the top level. If we add the `no_inline` form of the attribute:

```
#[doc(no_inline)]
pub use bar::Bar;

/// bar docs
mod bar {
    /// the docs for Bar
    pub struct Bar;
}
```

Now we'll have a `Re-exports` line, and `Bar` will not link to anywhere.

One special case: In Rust 2018 and later, if you `pub use` one of your dependencies, `rustdoc` will not eagerly inline it as a module unless you add `#[doc(inline)]`.

`#[doc(hidden)]`

Any item annotated with `#[doc(hidden)]` will not appear in the documentation, unless the `strip-hidden` pass is removed.

`#[doc(primitive)]`

Since primitive types are defined in the compiler, there's no place to attach documentation attributes. This attribute is used by the standard library to provide a way to generate documentation for primitive types.

Documentation tests

`rustdoc` supports executing your documentation examples as tests. This makes sure that examples within your documentation are up to date and working.

The basic idea is this:

````
/// # Examples
///
/// ```
/// let x = 5;
/// ```
````

The triple backticks start and end code blocks. If this were in a file named `foo.rs`, running `rustdoc --test foo.rs` will extract this example, and then run it as a test.

Please note that by default, if no language is set for the block code, `rustdoc` assumes it is `Rust` code. So the following:

~~~
```rust
let x = 5;
```
~~~

is strictly equivalent to:

~~~
```
let x = 5;
```
~~~

There's some subtlety though! Read on for more details.

Passing or failing a doctest

Like regular unit tests, regular doctests are considered to "pass" if they compile and run without panicking. So if you want to demonstrate that some computation gives a certain result, the `assert!` family of macros works the same as other Rust code:

```rust
let foo = "foo";

assert_eq!(foo, "foo");
```

This way, if the computation ever returns something different, the code panics and the doctest fails.

Pre-processing examples

In the example above, you'll note something strange: there's no `main` function! Forcing you to write `main` for every example, no matter how small, adds friction. So `rustdoc` processes your examples slightly before running them. Here's the full algorithm rustdoc uses to preprocess examples:

1. Some common `allow` attributes are inserted, including `unused_variables`, `unused_assignments`, `unused_mut`, `unused_attributes`, and `dead_code`. Small examples often trigger these lints.
2. Any attributes specified with `#![doc(test(attr(...)))]` are added.
3. Any leading `#![foo]` attributes are left intact as crate attributes.
4. If the example does not contain `extern crate`, and `#![doc(test(no_crate_inject))]` was not specified, then `extern crate <mycrate>;` is inserted (note the lack of `#[macro_use]`).
5. Finally, if the example does not contain `fn main`, the remainder of the text is wrapped in `fn main() { your_code }`.

For more about that caveat in rule 4, see "Documenting Macros" below.

Hiding portions of the example

Sometimes, you need some setup code, or other things that would distract from your example, but are important to make the tests work. Consider an example block that looks like this:

```
/// /// Some documentation.
/// # fn foo() {} // this function will be hidden
/// println!("Hello, World!");
```

It will render like this:

```
/// Some documentation.
println!("Hello, World!");
```

Yes, that's right: you can add lines that start with `# `, and they will be hidden from the output, but will be used when compiling your code. You can use this to your advantage. In this case, documentation comments need to apply to some kind of function, so if I want to show you just a documentation comment, I need to add a little function definition below it. At the same time, it's only there to satisfy the compiler, so hiding it makes the example more clear. You can use this technique to explain longer examples in detail, while still preserving the testability of your documentation.

For example, imagine that we wanted to document this code:

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

We might want the documentation to end up looking like this:

> First, we set `x` to five:
>
> ```
> let x = 5;
> ```
>
> Next, we set `y` to six:
>
> ```
> let y = 6;
> ```
>
> Finally, we print the sum of `x` and `y`:
>
> ```
> println!("{}", x + y);
> ```

To keep each code block testable, we want the whole program in each block, but we don't want the reader to see every line every time. Here's what we put in our source code:

~~~
First, we set `x` to five:

```
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

Next, we set `y` to six:

```
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

Finally, we print the sum of `x` and `y`:

```
# let x = 5;
# let y = 6;
println!("{}", x + y);
```
~~~

By repeating all parts of the example, you can ensure that your example still compiles, while only showing the parts that are relevant to that part of your explanation.

The `#`-hiding of lines can be prevented by using two consecutive hashes `##`. This only needs to be done with the first `#` which would've otherwise caused hiding. If we have a string literal like the following, which has a line that starts with a `#`:

```
let s = "foo
# bar # baz";
```

We can document it by escaping the initial `#`:

```
/// let s = "foo
/// ## bar # baz";
```

Using `?` in doc tests

When writing an example, it is rarely useful to include a complete error handling, as it would add significant amounts of boilerplate code. Instead, you may want the following:

```
/// ```
/// use std::io;
/// let mut input = String::new();
/// io::stdin().read_line(&mut input)?;
/// ```
```

The problem is that `?` returns a `Result<T, E>` and test functions don't return anything, so this will give a mismatched types error.

You can get around this limitation by manually adding a `main` that returns `Result<T, E>`, because `Result<T, E>` implements the `Termination` trait:

```
/// A doc test using ?
///
/// ```
/// use std::io;
///
/// fn main() -> io::Result<()> {
///     let mut input = String::new();
///     io::stdin().read_line(&mut input)?;
///     Ok(())
/// }
/// ```
```

Together with the `# ` from the section above, you arrive at a solution that appears to the reader as the initial idea but works with doc tests:

```
/// ```
/// use std::io;
/// # fn main() -> io::Result<()> {
/// let mut input = String::new();
/// io::stdin().read_line(&mut input)?;
/// # Ok(())
/// # }
/// ```
```

As of version 1.34.0, one can also omit the `fn main()`, but you will have to disambiguate the error type:

```
/// ```
/// use std::io;
/// let mut input = String::new();
/// io::stdin().read_line(&mut input)?;
/// # Ok::<(), io::Error>(())
/// ```
```

This is an unfortunate consequence of the `?` operator adding an implicit conversion, so type inference fails because the type is not unique. Please note that you must write the `(())` in one sequence without intermediate whitespace so that rustdoc understands you want an implicit `Result`-returning function.

Documenting macros

Here's an example of documenting a macro:

```
/// Panic with a given message unless an expression evaluates to true.
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, “Math is broken.”);
/// # }
/// ```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, “I’m broken.”);
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic!
($($rest),+); } });
}
```

You'll note three things: we need to add our own `extern crate` line, so that we can add the `#[macro_use]` attribute. Second, we'll need to add our own `main()` as well (for reasons discussed above). Finally, a judicious use of `#` to comment out those two things, so they don't show up in the output.

Attributes

There are a few annotations that are useful to help `rustdoc` do the right thing when testing your code:

```
/// ```ignore
/// fn foo() {
/// ```
```

The `ignore` directive tells Rust to ignore your code. This is almost never what you want, as it's the most generic. Instead, consider annotating it with `text` if it's not code, or using `#` s to get a working example that only shows the part you care about.

```
/// ```should_panic
/// assert!(false);
/// ```
```

`should_panic` tells `rustdoc` that the code should compile correctly, but not actually pass as a test.

```
/// ```no_run
/// loop {
///     println!("Hello, world");
/// }
/// ```
```

The `no_run` attribute will compile your code, but not run it. This is important for examples such as "Here's how to retrieve a web page," which you would want to ensure compiles, but might be run in a test environment that has no network access.

```
/// ```compile_fail
/// let x = 5;
/// x += 2; // shouldn't compile!
/// ```
```

`compile_fail` tells `rustdoc` that the compilation should fail. If it compiles, then the test will fail. However please note that code failing with the current Rust release may work in a future release, as new features are added.

```
/// Only runs on the 2018 edition.
///
/// ```edition2018
/// let result: Result<i32, ParseIntError> = try {
///     "1".parse::<i32>()?
///         + "2".parse::<i32>()?
///         + "3".parse::<i32>()?
/// };
/// ```
```

`edition2018` tells `rustdoc` that the code sample should be compiled using the 2018 edition of Rust. Similarly, you can specify `edition2015` to compile the code with the 2015 edition.

Syntax reference

The *exact* syntax for code blocks, including the edge cases, can be found in the Fenced Code Blocks section of the CommonMark specification.

Rustdoc also accepts *indented* code blocks as an alternative to fenced code blocks: instead of surrounding your code with three backticks, you can indent each line by four or more spaces.

```
    let foo = "foo";
    assert_eq!(foo, "foo");
```

These, too, are documented in the CommonMark specification, in the Indented Code Blocks section.

However, it's preferable to use fenced code blocks over indented code blocks. Not only are fenced code blocks considered more idiomatic for Rust code, but there is no way to use directives such as `ignore` or `should_panic` with indented code blocks.

Include items only when collecting doctests

Rustdoc's documentation tests can do some things that regular unit tests can't, so it can sometimes be useful to extend your doctests with samples that wouldn't otherwise need to be in documentation. To this end, Rustdoc allows you to have certain items only appear when it's collecting doctests, so you can utilize doctest functionality without forcing the test to appear in docs, or to find an arbitrary private item to include it on.

When compiling a crate for use in doctests (with `--test` option), rustdoc will set `cfg(doctest)`. Note that they will still link against only the public items of your crate; if you need to test private items, you need to write a unit test.

In this example, we're adding doctests that we know won't compile, to verify that our struct can only take in valid data:

```
/// We have a struct here. Remember it doesn't accept negative numbers!
pub struct MyStruct(pub usize);

/// ```compile_fail
/// let x = my_crate::MyStruct(-5);
/// ```
#[cfg(doctest)]
pub struct MyStructOnlyTakesUsize;
```

Note that the struct `MyStructOnlyTakesUsize` here isn't actually part of your public crate API. The use of `#[cfg(doctest)]` makes sure that this struct only exists while rustdoc is collecting doctests. This means that its doctest is executed when `--test` is passed to rustdoc, but is hidden from the public documentation.

Another possible use of `cfg(doctest)` is to test doctests that are included in your README file without including it in your main documentation. For example, you could write this into your `lib.rs` to test your README as part of your doctests:

```
#![feature(extern_doc)]

#[doc(include="../README.md")]
#[cfg(doctest)]
pub struct ReadmeDoctests;
```

This will include your README as documentation on the hidden struct `ReadmeDoctests`, which will then be tested alongside the rest of your doctests.

Lints

`rustdoc` provides lints to help you writing and testing your documentation. You can use them like any other lints by doing this:

```
#![allow(missing_docs)] // allowing the lint, no message
#![warn(missing_docs)] // warn if there is missing docs
#![deny(missing_docs)] // rustdoc will fail if there is missing docs
```

Here is the list of the lints provided by `rustdoc`:

intra_doc_link_resolution_failure

This lint **warns by default** and is **nightly-only**. This lint detects when an intra-doc link fails to get resolved. For example:

```
/// I want to link to [`Inexistent`] but it doesn't exist!
pub fn foo() {}
```

You'll get a warning saying:

```
error: `[`Inexistent`]` cannot be resolved, ignoring it...
```

missing_docs

This lint is **allowed by default**. It detects items missing documentation. For example:

```
#![warn(missing_docs)]

pub fn undocumented() {}
```

The `undocumented` function will then have the following warning:

```
warning: missing documentation for a function
  --> your-crate/lib.rs:3:1
   |
 3 | pub fn undocumented() {}
   | ^^^^^^^^^^^^^^^^^^^^^
```

missing_doc_code_examples

This lint is **allowed by default**. It detects when a documentation block is missing a code example. For example:

```
#![warn(missing_doc_code_examples)]

/// There is no code example!
pub fn no_code_example() {}
```

The `no_code_example` function will then have the following warning:

```
warning: Missing code example in this documentation
  --> your-crate/lib.rs:3:1
   |
LL | /// There is no code example!
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

To fix the lint, you need to add a code example into the documentation block:

```

/// There is no code example!
///
/// ```
/// println!("calling no_code_example...");
/// no_code_example();
/// println!("we called no_code_example!");
/// ```
pub fn no_code_example() {}
```

private_doc_tests

This lint is **allowed by default**. It detects documentation tests when they are on a private item. For example:

```
#![warn(private_doc_tests)]

mod foo {
    /// private doc test
    ///
    /// ```
    /// assert!(false);
    /// ```
    fn bar() {}
}
```

Which will give:

```
warning: Documentation test in private item
  --> your-crate/lib.rs:4:1
   |
 4 | /     /// private doc test
 5 | |     ///
 6 | |     /// ```
 7 | |     /// assert!(false);
 8 | |     /// ```
   | |___________^
```

Passes

Rustdoc has a concept called "passes". These are transformations that `rustdoc` runs on your documentation before producing its final output.

In addition to the passes below, check out the docs for these flags:

- `--passes`
- `--no-defaults`

Default passes

By default, rustdoc will run some passes, namely:

- `strip-hidden`
- `strip-private`
- `collapse-docs`
- `unindent-comments`

However, `strip-private` implies `strip-priv-imports`, and so effectively, all passes are run by default.

`strip-hidden`

This pass implements the `#[doc(hidden)]` attribute. When this pass runs, it checks each item, and if it is annotated with this attribute, it removes it from `rustdoc`'s output.

Without this pass, these items will remain in the output.

`unindent-comments`

When you write a doc comment like this:

```
/// This is a documentation comment.
```

There's a space between the `///` and that `T`. That spacing isn't intended to be a part of the output; it's there for humans, to help separate the doc comment syntax from the text of the comment. This pass is what removes that space.

The exact rules are left under-specified so that we can fix issues that we find.

Without this pass, the exact number of spaces is preserved.

`collapse-docs`

With this pass, multiple `#[doc]` attributes are converted into one single documentation string.

For example:

```
#[doc = "This is the first line."]
#[doc = "This is the second line."]
```

Gets collapsed into a single doc string of

```
This is the first line.
This is the second line.
```

`strip-private`

This removes documentation for any non-public items, so for example:

```
/// These are private docs.
struct Private;

/// These are public docs.
pub struct Public;
```

This pass removes the docs for `Private`, since they're not public.

This pass implies `strip-priv-imports`.

`strip-priv-imports`

This is the same as `strip-private`, but for `extern crate` and `use` statements instead of items.

Advanced Features

The features listed on this page fall outside the rest of the main categories.

`#[cfg(doc)]`: Documenting platform-/feature-specific information

For conditional compilation, Rustdoc treats your crate the same way the compiler does. Only things from the host target are available (or from the given `--target` if present), and everything else is "filtered out" from the crate. This can cause problems if your crate is providing different things on different targets and you want your documentation to reflect all the available items you provide.

If you want to make sure an item is seen by Rustdoc regardless of what platform it's targeting, you can apply `#[cfg(doc)]` to it. Rustdoc sets this whenever it's building documentation, so anything that uses that flag will make it into documentation it generates. To apply this to an item with other `#[cfg]` filters on it, you can write something like `#[cfg(any(windows, doc))]`. This will preserve the item either when built normally on Windows, or when being documented anywhere.

Please note that this feature is not passed to doctests.

Example:

```rust
/// Token struct that can only be used on Windows.
#[cfg(any(windows, doc))]
pub struct WindowsToken;
/// Token struct that can only be used on Unix.
#[cfg(any(unix, doc))]
pub struct UnixToken;
```

Here, the respective tokens can only be used by dependent crates on their respective platforms, but they will both appear in documentation.

Unstable features

Rustdoc is under active development, and like the Rust compiler, some features are only available on nightly releases. Some of these features are new and need some more testing before they're able to be released to the world at large, and some of them are tied to features in the Rust compiler that are unstable. Several features here require a matching `#![feature(...)]` attribute to enable, and thus are more fully documented in the Unstable Book. Those sections will link over there as necessary.

Nightly-gated functionality

These features just require a nightly build to operate. Unlike the other features on this page, these don't need to be "turned on" with a command-line flag or a `#![feature(...)]` attribute in your crate. This can give them some subtle fallback modes when used on a stable release, so be careful!

Error numbers for `compile-fail` doctests

As detailed in the chapter on documentation tests, you can add a `compile_fail` attribute to a doctest to state that the test should fail to compile. However, on nightly, you can optionally add an error number to state that a doctest should emit a specific error number:

~~~
```compile_fail,E0044
extern { fn some_func<T>(x: T); }
```
~~~

This is used by the error index to ensure that the samples that correspond to a given error number properly emit that error code. However, these error codes aren't guaranteed to be the only thing that a piece of code emits from version to version, so this is unlikely to be stabilized in the future.

Attempting to use these error numbers on stable will result in the code sample being interpreted as plain text.

Linking to items by type

As designed in RFC 1946, Rustdoc can parse paths to items when you use them as links. To resolve these type names, it uses the items currently in-scope, either by declaration or by `use` statement. For modules, the "active scope" depends on whether the documentation is written outside the module (as `///` comments on the `mod` statement) or inside the module (at `//!` comments inside the file or block). For all other items, it uses the enclosing module's scope.

For example, in the following code:

```rust
/// Does the thing.
pub fn do_the_thing(_: SomeType) {
    println!("Let's do the thing!");
}

/// Token you use to [`do_the_thing`].
pub struct SomeType;
```

The link to ``[`do_the_thing`]`` in `SomeType`'s docs will properly link to the page for `fn do_the_thing`. Note that here, rustdoc will insert the link target for you, but manually writing the target out also works:

```
pub mod some_module {
    /// Token you use to do the thing.
    pub struct SomeStruct;
}

/// Does the thing. Requires one [`SomeStruct`] for the thing to work.
///
/// [`SomeStruct`]: some_module::SomeStruct
pub fn do_the_thing(_: some_module::SomeStruct) {
    println!("Let's do the thing!");
}
```

For more details, check out the RFC, and see the tracking issue for more information about what parts of the feature are available.

Extensions to the `#[doc]` attribute

These features operate by extending the `#[doc]` attribute, and thus can be caught by the compiler and enabled with a `#![feature(...)]` attribute in your crate.

Documenting platform-/feature-specific information

Because of the way Rustdoc documents a crate, the documentation it creates is specific to the target rustc compiles for. Anything that's specific to any other target is dropped via `#[cfg]` attribute processing early in the compilation process. However, Rustdoc has a trick up its sleeve to handle platform-specific code if it *does* receive it.

Because Rustdoc doesn't need to fully compile a crate to binary, it replaces function bodies with `loop {}` to prevent having to process more than necessary. This means that any code within a function that requires platform-specific pieces is ignored. Combined with a special attribute, `#[doc(cfg(...))]`, you can tell Rustdoc exactly which platform something is supposed to run on, ensuring that doctests are only run on the appropriate platforms.

The `#[doc(cfg(...))]` attribute has another effect: When Rustdoc renders documentation for that item, it will be accompanied by a banner explaining that the item is only available on certain platforms.

For Rustdoc to document an item, it needs to see it, regardless of what platform it's currently running on. To aid this, Rustdoc sets the flag `#[cfg(doc)]` when running on your crate. Combining this with the target platform of a given item allows it to appear when building your crate normally on that platform, as well as when building documentation anywhere.

For example, `#[cfg(any(windows, doc))]` will preserve the item either on Windows or during the documentation process. Then, adding a new attribute `#[doc(cfg(windows))]` will tell Rustdoc that the item is supposed to be used on Windows. For example:

```rust
#![feature(doc_cfg)]

/// Token struct that can only be used on Windows.
#[cfg(any(windows, doc))]
#[doc(cfg(windows))]
pub struct WindowsToken;

/// Token struct that can only be used on Unix.
#[cfg(any(unix, doc))]
#[doc(cfg(unix))]
pub struct UnixToken;
```

In this sample, the tokens will only appear on their respective platforms, but they will both appear in documentation.

`#[doc(cfg(...))]` was introduced to be used by the standard library and currently requires the `#![feature(doc_cfg)]` feature gate. For more information, see its chapter in the Unstable Book and its tracking issue.

Exclude certain dependencies from documentation

The standard library uses several dependencies which, in turn, use several types and traits from the standard library. In addition, there are several compiler-internal crates that are not considered to be part of the official standard library, and thus would be a distraction to include in documentation. It's not enough to exclude their crate documentation, since information about trait implementations appears on the pages for both the type and the trait, which can be in different crates!

To prevent internal types from being included in documentation, the standard library adds an attribute to their `extern crate` declarations: `#[doc(masked)]`.

This causes Rustdoc to "mask out" types from these crates when building lists of trait implementations.

The `#[doc(masked)]` attribute is intended to be used internally, and requires the `#![feature(doc_masked)]` feature gate. For more information, see its chapter in the Unstable Book and its tracking issue.

Include external files as API documentation

As designed in RFC 1990, Rustdoc can read an external file to use as a type's documentation. This is useful if certain documentation is so long that it would break the flow of reading the source. Instead of writing it all inline, writing `#[doc(include = "sometype.md")]` will ask Rustdoc to instead read that file and use it as if it were written inline.

`#[doc(include = "...")]` currently requires the `#![feature(external_doc)]` feature gate. For more information, see its chapter in the Unstable Book and its tracking issue.

Add aliases for an item in documentation search

This feature allows you to add alias(es) to an item when using the `rustdoc` search through the `doc(alias)` attribute. Example:

```
#![feature(doc_alias)]

#[doc(alias = "x")]
#[doc(alias = "big")]
pub struct BigX;
```

Then, when looking for it through the `rustdoc` search, if you enter "x" or "big", search will show the `BigX` struct first.

Unstable command-line arguments

These features are enabled by passing a command-line flag to Rustdoc, but the flags in question are themselves marked as unstable. To use any of these options, pass `-Z unstable-options` as well as the flag in question to Rustdoc on the command-line. To do this from Cargo, you can either use the `RUSTDOCFLAGS` environment variable or the `cargo rustdoc` command.

`--markdown-before-content`: include rendered Markdown before the content

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --markdown-before-content
extra.md
$ rustdoc README.md -Z unstable-options --markdown-before-content
extra.md
```

Just like `--html-before-content`, this allows you to insert extra content inside the `<body>` tag but before the other content `rustdoc` would normally produce in the rendered documentation. However, instead of directly inserting the file verbatim, `rustdoc` will pass the files through a Markdown renderer before inserting the result into the file.

`--markdown-after-content`: include rendered Markdown after the content

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --markdown-after-content
extra.md
$ rustdoc README.md -Z unstable-options --markdown-after-content
extra.md
```

Just like `--html-after-content`, this allows you to insert extra content before the `</body>` tag but after the other content `rustdoc` would normally produce in the rendered documentation. However, instead of directly inserting the file verbatim, `rustdoc` will pass the files through a Markdown renderer before inserting the result into the file.

`--playground-url`: control the location of the playground

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --playground-url
https://play.rust-lang.org/
```

When rendering a crate's docs, this flag gives the base URL of the Rust Playground, to use for generating `Run` buttons. Unlike `--markdown-playground-url`, this argument works for standalone Markdown files *and* Rust crates. This works the same way as adding `#![doc(html_playground_url = "url")]` to your crate root, as mentioned in the chapter about the `#[doc]` attribute. Please be aware that the official Rust Playground at https://play.rust-lang.org does not have every crate available, so if your examples require your crate, make sure the playground you provide has your crate available.

If both `--playground-url` and `--markdown-playground-url` are present when rendering a standalone Markdown file, the URL given to `--markdown-playground-url` will take precedence. If both `--playground-url` and `#![doc(html_playground_url = "url")]` are present when rendering crate docs, the attribute will take precedence.

`--sort-modules-by-appearance`: control how items on module pages are sorted

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --sort-modules-by-appearance
```

Ordinarily, when `rustdoc` prints items in module pages, it will sort them alphabetically (taking some consideration for their stability, and names that end in a number). Giving this flag to `rustdoc` will disable this sorting and instead make it print the items in the order they appear in the source.

`--resource-suffix`: modifying the name of CSS/JavaScript in crate docs

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --resource-suffix suf
```

When rendering docs, `rustdoc` creates several CSS and JavaScript files as part of the output. Since all these files are linked from every page, changing where they are can be cumbersome if you need to specially cache them. This flag will rename all these files in the output to include the suffix in the filename. For example, `light.css` would become `light-suf.css` with the above command.

`--display-warnings`: display warnings when documenting or running documentation tests

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --display-warnings
$ rustdoc --test src/lib.rs -Z unstable-options --display-warnings
```

The intent behind this flag is to allow the user to see warnings that occur within their library or their documentation tests, which are usually suppressed. However, due to a bug, this flag doesn't 100% work as intended. See the linked issue for details.

`--extern-html-root-url`: control how rustdoc links to non-local crates

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --extern-html-root-url some-
crate=https://example.com/some-crate/1.0.1
```

Ordinarily, when rustdoc wants to link to a type from a different crate, it looks in two places: docs that already exist in the output directory, or the `#![doc(doc_html_root)]` set in the other crate. However, if you want to link to docs that exist in neither of those places, you can use these flags to control that behavior. When the `--extern-html-root-url` flag is given with a name matching one of your dependencies, rustdoc use that URL for those docs. Keep in mind that if those docs exist in the output directory, those local docs will still override this flag.

`-Z force-unstable-if-unmarked`

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z force-unstable-if-unmarked
```

This is an internal flag intended for the standard library and compiler that applies an `#[unstable]` attribute to any dependent crate that doesn't have another stability attribute. This allows `rustdoc` to be able to generate documentation for the compiler crates and the standard library, as an equivalent command-line argument is provided to `rustc` when building those crates.

`--index-page`: provide a top-level landing page for docs

This feature allows you to generate an index-page with a given markdown file. A good example of it is the rust documentation index.

With this, you'll have a page which you can custom as much as you want at the top of your crates.

Using `index-page` option enables `enable-index-page` option as well.

`--enable-index-page`: generate a default index page for docs

This feature allows the generation of a default index-page which lists the generated crates.

`--static-root-path`: control how static files are loaded in HTML output

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --static-root-path '/cache/'
```

This flag controls how rustdoc links to its static files on HTML pages. If you're hosting a lot of crates' docs generated by the same version of rustdoc, you can use this flag to cache rustdoc's CSS, JavaScript, and font files in a single location, rather than duplicating it once per "doc root" (grouping of crate docs generated into the same output directory, like with `cargo doc`). Per-crate files like the search index will still load from the documentation root, but anything that gets renamed with

`--resource-suffix` will load from the given path.

`--persist-doctests`: persist doctest executables after running

Using this flag looks like this:

```
$ rustdoc src/lib.rs --test -Z unstable-options --persist-doctests
target/rustdoctest
```

This flag allows you to keep doctest executables around after they're compiled or run. Usually, rustdoc will immediately discard a compiled doctest after it's been tested, but with this option, you can keep those binaries around for farther testing.

`--show-coverage`: calculate the percentage of items with documentation

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --show-coverage
```

If you want to determine how many items in your crate are documented, pass this flag to rustdoc. When it receives this flag, it will count the public items in your crate that have documentation, and print out the counts and a percentage instead of generating docs.

Some methodology notes about what rustdoc counts in this metric:

- Rustdoc will only count items from your crate (i.e. items re-exported from other crates don't count).
- Docs written directly onto inherent impl blocks are not counted, even though their doc comments are displayed, because the common pattern in Rust code is to write all inherent methods into the same impl block.
- Items in a trait implementation are not counted, as those impls will inherit any docs from the trait itself.
- By default, only public items are counted. To count private items as well, pass `--document-private-items` at the same time.

Public items that are not documented can be seen with the built-in `missing_docs` lint. Private items that are not documented can be seen with Clippy's `missing_docs_in_private_items` lint.

`--enable-per-target-ignores`: allow `ignore-foo` style filters for doctests

Using this flag looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --enable-per-target-ignores
```

This flag allows you to tag doctests with compiltest style `ignore-foo` filters that prevent rustdoc from running that test if the target triple string contains foo. For example:

```
///```ignore-foo,ignore-bar
///assert!(2 == 2);
///```
struct Foo;
```

This will not be run when the build target is `super-awesome-foo` or `less-bar-awesome`. If the flag is not enabled, then rustdoc will consume the filter, but do nothing with it, and the above example will be run for all targets. If you want to preserve backwards compatibility for older versions of rustdoc, you can use

```
///```ignore,ignore-foo
///assert!(2 == 2);
///```
struct Foo;
```

In older versions, this will be ignored on all targets, but on newer versions `ignore-gnu` will override `ignore`.

`--runtool`, `--runtool-arg`: program to run tests with; args to pass to it

Using these options looks like this:

```
$ rustdoc src/lib.rs -Z unstable-options --runtool runner --runtool-arg
--do-thing --runtool-arg --do-other-thing
```

These options can be used to run the doctest under a program, and also pass arguments to that program. For example, if you want to run your doctests under valgrind you might run

```
$ rustdoc src/lib.rs -Z unstable-options --runtool valgrind
```

Another use case would be to run a test inside an emulator, or through a Virtual Machine.