Linking to items by name

Rustdoc is capable of directly linking to other rustdoc pages using the path of the item as a link. This is referred to as an 'intra-doc link'.

For example, in the following code all of the links will link to the rustdoc page for `Bar`:

```
/// This struct is not [Bar]
pub struct Foo1;

/// This struct is also not [bar](Bar)
pub struct Foo2;

/// This struct is also not [bar][b]
///
/// [b]: Bar
pub struct Foo3;

/// This struct is also not [`Bar`]
pub struct Foo4;

/// This struct *is* [`Bar`]!
pub struct Bar;
```

Unlike normal Markdown, `[bar][Bar]` syntax is also supported without needing a `[Bar]: ...` reference link.

Backticks around the link will be stripped, so ``[`Option`]`` will correctly link to `Option`.

Valid links

You can refer to anything in scope, and use paths, including `Self`, `self`, `super`, and `crate`. Associated items (functions, types, and constants) are supported, but not for blanket trait implementations. Rustdoc also supports linking to all primitives listed in the standard library documentation.

You can also refer to items with generic parameters like `Vec<T>`. The link will resolve as if you had written ``[`Vec<T>`](Vec)``. Fully-qualified syntax (for example, `<Vec as IntoIterator>::into_iter()`) is not yet supported, however.

```
use std::sync::mpsc::Receiver;

/// This is a version of [`Receiver<T>`] with support for [`std::future`].
///
/// You can obtain a [`std::future::Future`] by calling [`Self::recv()`].
pub struct AsyncReceiver<T> {
    sender: Receiver<T>
}

impl<T> AsyncReceiver<T> {
    pub async fn recv() -> T {
        unimplemented!()
    }
}
```

Rustdoc allows using URL fragment specifiers, just like a normal link:

```
/// This is a special implementation of [positional parameters].
///
/// [positional parameters]: std::fmt#formatting-parameters
struct MySpecialFormatter;
```

Namespaces and Disambiguators

Paths in Rust have three namespaces: type, value, and macro. Item names must be unique within their namespace, but can overlap with items in other namespaces. In case of ambiguity, rustdoc will warn about the ambiguity and suggest a disambiguator.

```
/// See also: [`Foo`](struct@Foo)
struct Bar;

/// This is different from [`Foo`](fn@Foo)
struct Foo {}

fn Foo() {}
```

These prefixes will be stripped when displayed in the documentation, so `[struct@Foo]` will be rendered as `Foo`.

You can also disambiguate for functions by adding `()` after the function name, or for macros by adding `!` after the macro name:

```
/// This is different from [`foo!`]
fn foo() {}

/// This is different from [`foo()`]
macro_rules! foo {
  () => {}
}
```

Warnings, re-exports, and scoping

Links are resolved in the scope of the module where the item is defined, even when the item is re-exported. If a link from another crate fails to resolve, no warning is given.

```
mod inner {
    /// Link to [f()]
    pub struct S;
    pub fn f() {}
}
pub use inner::S; // the link to `f` will still resolve correctly
```

When re-exporting an item, rustdoc allows adding additional documentation to it. That additional documentation will be resolved in the scope of the re-export, not the original, allowing you to link to items in the new crate. The new links will still give a warning if they fail to resolve.

```rust
/// See also [foo()]
pub use std::process::Command;

pub fn foo() {}
```

This is especially useful for proc-macros, which must always be defined in their own dedicated crate.

Note: Because of how `macro_rules!` macros are scoped in Rust, the intra-doc links of a `macro_rules!` macro will be resolved relative to the crate root, as opposed to the module it is defined in.

If links do not look 'sufficiently like' an intra-doc link, they will be ignored and no warning will be given, even if the link fails to resolve. For example, any link containing `/` or `[]` characters will be ignored.