Modules

Syntax:

Module :

    unsafe? mod IDENTIFIER ;
  | unsafe? mod IDENTIFIER {
      *InnerAttribute**
      *Item**
    }

A module is a container for zero or more items.

A *module item* is a module, surrounded in braces, named, and prefixed with the keyword `mod`. A module item introduces a new, named module into the tree of modules making up a crate. Modules can nest arbitrarily.

An example of a module:

```
mod math {
    type Complex = (f64, f64);
    fn sin(f: f64) -> f64 {
        /* ... */
    }
    fn cos(f: f64) -> f64 {
        /* ... */
    }
    fn tan(f: f64) -> f64 {
        /* ... */
    }
}
```

Modules and types share the same namespace. Declaring a named type with the same name as a module in scope is forbidden: that is, a type definition, trait, struct, enumeration, union, type parameter or crate can't shadow the name of a module in scope, or vice versa. Items brought into scope with `use` also have this restriction.

The `unsafe` keyword is syntactically allowed to appear before the `mod` keyword, but it is rejected at a semantic level. This allows macros to consume the syntax and make use of the `unsafe` keyword, before removing it from the token stream.

Module Source Filenames

A module without a body is loaded from an external file. When the module does not have a `path` attribute, the path to the file mirrors the logical module path. Ancestor module path components are directories, and the module's contents are in a file with the name of the module plus the `.rs` extension. For example, the following module structure can have this corresponding filesystem structure:

Module Path	Filesystem Path	File Contents
`crate`	`lib.rs`	`mod util;`
`crate::util`	`util.rs`	`mod config;`
`crate::util::config`	`util/config.rs`	

Module filenames may also be the name of the module as a directory with the contents in a file named `mod.rs` within that directory. The above example can alternately be expressed with `crate::util`'s contents in a file named `util/mod.rs`. It is not allowed to have both `util.rs` and `util/mod.rs`.

> **Note**: Prior to `rustc` 1.30, using `mod.rs` files was the way to load a module with nested children. It is encouraged to use the new naming convention as it is more consistent, and avoids having many files named `mod.rs` within a project.

The path attribute

The directories and files used for loading external file modules can be influenced with the `path` attribute.

For `path` attributes on modules not inside inline module blocks, the file path is relative to the directory the source file is located. For example, the following code snippet would use the paths shown based on where it is located:

```
#[path = "foo.rs"]
mod c;
```

Source File	`c`'s File Location	`c`'s Module Path
`src/a/b.rs`	`src/a/foo.rs`	`crate::a::b::c`
`src/a/mod.rs`	`src/a/foo.rs`	`crate::a::c`

For `path` attributes inside inline module blocks, the relative location of the file path depends on the kind of source file the `path` attribute is located in. "mod-rs" source files are root modules (such as `lib.rs` or `main.rs`) and modules with files named `mod.rs`. "non-mod-rs" source files are all other module files. Paths for `path` attributes inside inline module blocks in a mod-rs file are relative to the directory of the mod-rs file including the inline module components as directories. For non-mod-rs files, it is the same except the path starts with a directory with the name of the non-mod-rs module. For example, the following code snippet would use the paths shown based on where it is located:

```
mod inline {
    #[path = "other.rs"]
    mod inner;
}
```

Source File	`inner`'s File Location	`inner`'s Module Path
`src/a/b.rs`	`src/a/b/inline/other.rs`	`crate::a::b::inline::inner`
`src/a/mod.rs`	`src/a/inline/other.rs`	`crate::a::inline::inner`

An example of combining the above rules of `path` attributes on inline modules and nested modules within (applies to both mod-rs and non-mod-rs files):

```
#[path = "thread_files"]
mod thread {
    // Load the `local_data` module from `thread_files/tls.rs` relative to
    // this source file's directory.
    #[path = "tls.rs"]
    mod local_data;
}
```

Attributes on Modules

Modules, like all items, accept outer attributes. They also accept inner attributes: either after `{` for a module with a body, or at the beginning of the source file, after the optional BOM and shebang.

The built-in attributes that have meaning on a module are `cfg`, `deprecated`, `doc`, the lint check attributes, `path`, and `no_implicit_prelude`. Modules also accept macro attributes.