Modules and Macros in Rust

Jonathan Aldrich

17-363

Based heavily on the Rust book, Brown version

Crates

- A *crate* is one or more files that are compiled as a unit
    - The *crate root* is the file where compilation starts, e.g. `main.rs`
- A *binary crate* has a `main` function and is executable
- A *library crate* defines shared functionality to be used by multiple projects

Packages

- A package bundles one or more crates together
  - At most one library create
  - Any number of binary creates

- How to create a package:

```
cargo new my-project
```

- Package file system structure:
  - The `Cargo.toml` file describes how to build the crates
  - The `src` directory holds the source code
    - `main.rs` is the crate root of a binary crate with the same name as the package
    - `lib.rs` is the crate root of a library crate with the same name as the package

Modules

- A module is a unit of information hiding.
- Crates contain one or more modules
  - The crate root is the top-level module of a crate
- 3 ways to define a module:

```
mod foo { /* foo contents */ }
mod bar; // contents in bar.rs
mod baz; // contents in baz/mod.rs (less recommended)
```

- Submodules

```
mod boff; // in bar.rs; contents in bar/boff.rs
```

Module visibility

Visibility rules:

- You can access other declarations in your module, or those in a parent module
- You can only access things in a submodule if they are annotated pub

```
mod foo {
    pub mod child {
        pub fn bar() { return helper(); }
        fn helper() { return hidden::baz(); }
    }
    mod hidden {
        pub fn baz() { return 3; }
    }
}
foo::child::bar()     // OK
foo::hidden::baz()    // error: module `hidden` is private
```

Paths and Use

- You can use global *paths* (starting with `crate` for the current crate) to name any element that is visible to you

```
crate::garden::vegetables::Asparagus.eat()
```

- The use construct allows you to use a name without giving the whole path

```
use crate::garden::vegetables::Asparagus
Asparagus.eat()
```

- `self` is a name for the current module
- `super` is a name for the outer module

alias and pub use

- You can use `alias` to rename used things to avoid name clashes:

```
use std::fmt::Result;
use std::io::Result as IoResult;
```

- pub use is a convenient way to export things under a shorter path:

```
mod front_of_house {
    pub mod hosting { // inaccessible from outside top-level module
        pub fn add_to_waitlist() {} }
    }
pub use crate::front_of_house::hosting; // accessible as `hosting`
```

Using external packages

- In `Cargo.toml`

```
rand = "0.8.5"
```

- In `main.rs`

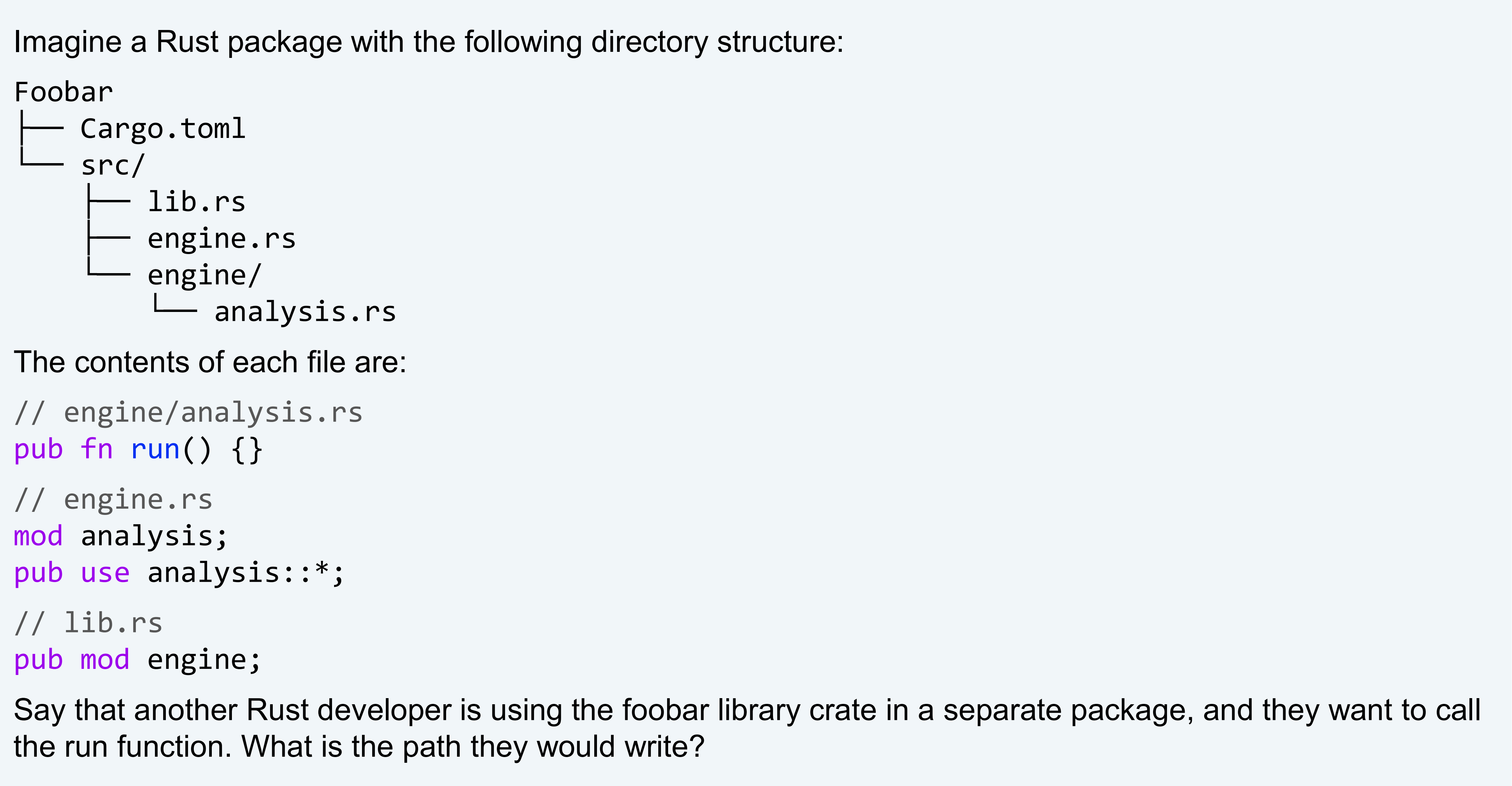
```
use rand::Rng;
fn main() {
  let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

In-Class Exercise 1

```rust
pub mod parent {
  pub fn a() {}
  fn b() {}
  pub mod child {
    pub fn c() {}
  }
}
fn main() {
  use parent::{*, child as alias};
  // ...
}
```

Inside main, what is the total number of paths that can refer to a, b, or c (not including those that use self, super, or crate)? Write your answer as a digit such as 0 or 1. For example, if the only two valid paths were a and `parent::b`, then the answer would be 2.

In-Class Exercise 2



Imagine a Rust package with the following directory structure:

```
Foobar
├── Cargo.toml
└── src/
    ├── lib.rs
    ├── engine.rs
    └── engine/
        └── analysis.rs
```

The contents of each file are:

```rust
// engine/analysis.rs
pub fn run() {}
```

```rust
// engine.rs
mod analysis;
pub use analysis::*;
```

```rust
// lib.rs
pub mod engine;
```

Say that another Rust developer is using the foobar library crate in a separate package, and they want to call the run function. What is the path they would write?

Macros

- Run at compile time – so they can transform code
- Are more flexible than functions – e.g. can take any # of arguments

```
println!("hello {}", name)
```

Declarative Macros

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
let v: Vec<u32> = vec![1, 2, 3];
```

Generated code for `vec![1, 2, 3]`:

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

Procedural macros

- Lower-level but extremely flexible implementation interface

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream { ... }

let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Derive macro (extended example)

```
pub trait HelloMacro { fn hello_macro(); }
struct Pancakes;
impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}
fn main() {
    Pancakes::hello_macro();
}
```

We want to generate this impl

Derive macro in action

```rust
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Much cleaner!

Defining the Hello derive macro (1)

```
use proc_macro::TokenStream;
use quote::quote;
#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

Defining the Hello derive macro (2)

```rust
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

Attribute macros

- Let you modify code based on an “attribute” that you define

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream)
           -> TokenStream { ... }

#[route(GET, "/")]
fn index() { ... }
```