CIS 1905

Lecture 3: Rust Projects

Larger Projects (Cargo, Modularization, Testing) and Data structures (Impl)

Recap

- Rust Data types
- Structs and Enums
- Ownership

Ownership Rules

1. Each value in Rust has a single owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

Logistics

- HW2 Markov Chain released tonight
- Joint Git bootcamp
  - **5-6pm this Sunday in Van Pelt Weigle 117**

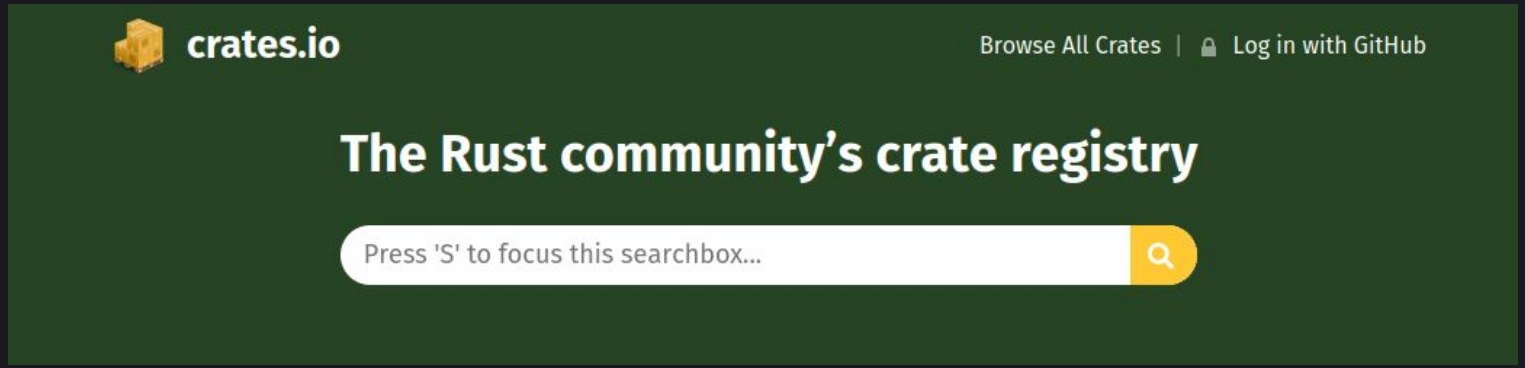
PollEv

https://pollev.com/alexanderrobertson109

Cargo

What is cargo?

- `cargo` is Rust's official *package manager*
- Create new projects
- Easily manage dependencies (crates)
- Run, test, and many other functionalities
- Publish to crates.io (official *package registry*)



Common Cargo Commands

*these should all be run from the project root directory (the one with Cargo.toml)

`cargo init`

Create new project in CWD

`cargo run`

Run the current project

`cargo clippy`

Check for errors and warnings

`cargo test`

Run all tests

`cargo doc`

Generate HTML docs

`cargo fmt`

Format current project

Cargo.toml: easily manage dependencies

- Single file to manage all your dependencies
- Uses TOML file format
- Can add dependencies by editing Cargo.toml or with `cargo add` (edits the file for you)

Simple syntax
`crate_name = “version”`

```
[package]
name = "rust-test"
version = "0.1.0"
edition = "2021"

# See more keys and their dei

[dependencies]
anyhow = "1.0.79"    ✔ 1.0.79
rand = "0.8.5"    ✔ 0.8.5
```

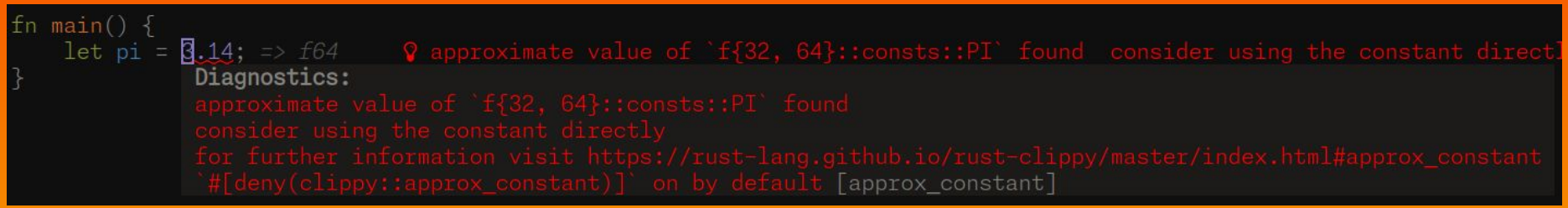
Aside:
Use `clippy` with `rust-analyzer`

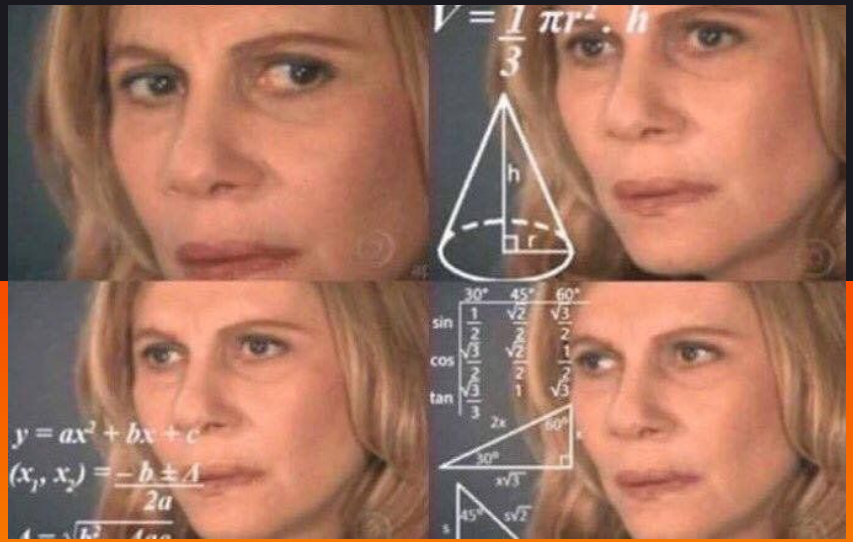
By default, `rust-analyzer` uses `cargo check`. You can configure it to use `cargo clippy` instead.

- VSCode: set config option "`rust-analyzer.check.command`" to "`clippy`"

https://code.visualstudio.com/docs/languages/rust

This way you won't be surprised by thousands of warnings when you finally get around to running `cargo clippy`.

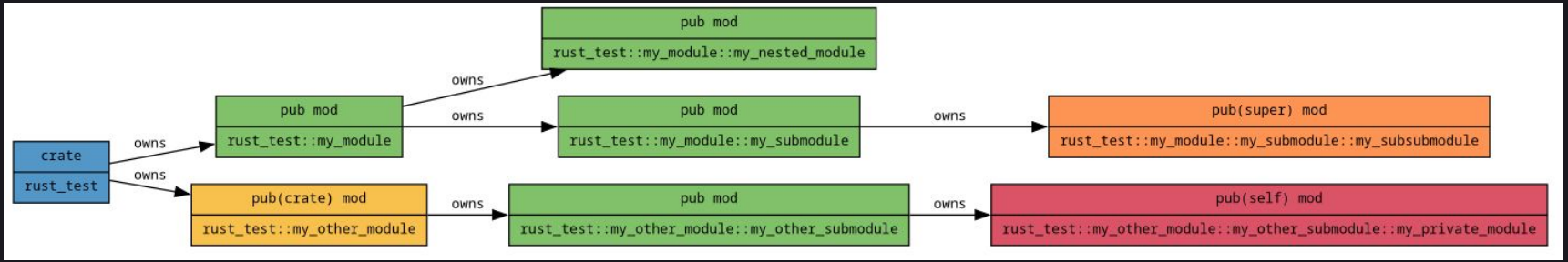



$V = \frac{1}{3}\pi r^2 \cdot h$
h
r
$y = ax^2 + bx + c$
$(x_1, x_2) = \frac{-b \pm \Delta}{2a}$
30°
45°
60°
sin
cos
tan
2x
60°
30°
x√3
45°
s
s√2

Modules

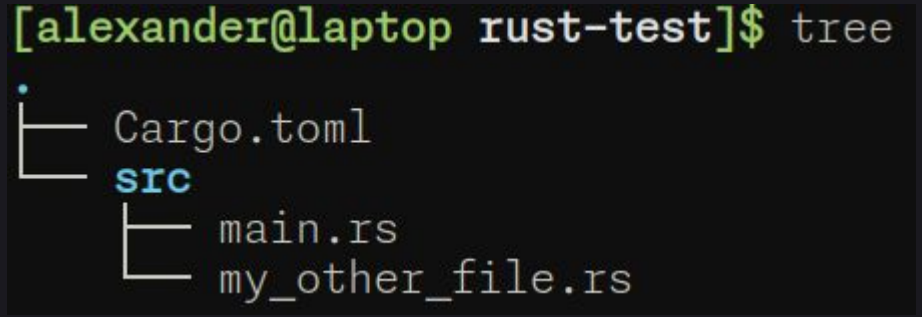
Why do we need modules?

- Projects are big
  - Stuffing everything in `main.rs` is a bad idea
- Encapsulation
- Crates are modules



What does the compiler see?

- By default, the compiler only “sees” `main.rs`
- You need to declare the existence of modules
  - If you forget to do this, your code will be “invisible”, and you won’t get any errors/warnings



Module trees

- crate
  - **main**
  - my_other_file
    - my_other_file::**hello**

Modules are declared with the mod keyword

If the compiler sees mod java_is_trash, it looks for it in two places two find it

1. File called “java_is_trash.rs” in the same directory as the current file
2. File called “mod.rs” in a subdirectory called “java_is_trash”

This process continues recursively for all modules in the project, forming a tree

```rust
// main.rs

mod my_other_file;

fn main() {
    my_other_file::hello();
}
```

```rust
// my_other_file.rs

pub fn hello() {
    println!("Hello, world!");
}
```

Example

- `crate`

Modules are children of the module they are declared in

`src/`

```
[alexander@laptop rust-test]$ tree
.
├── Cargo.toml
└── src
    ├── a.rs
    ├── b
    │   ├── c.rs
    │   ├── d.rs
    │   └── e
    │       └── mod.rs
    │   └── mod.rs
    └── main.rs
```

1. Compiler finds `main.rs` (module `crate`)

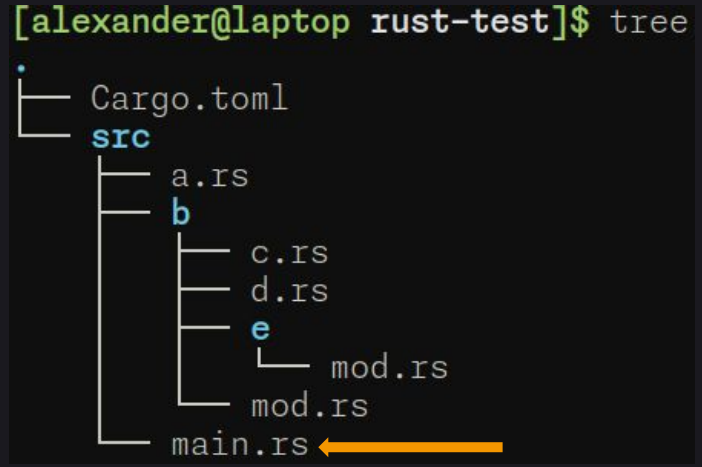
1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Example

- `crate`

Modules are children of the module they are declared in

`src/`



1. Compiler finds `main.rs` (module `crate`)
   a. Declarations: `mod a`, `mod b`

1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Example

Modules are children of the module they are declared in

- `crate`
  - `crate::a`

`src/`

```
[alexander@laptop rust-test]$ tree
.
├── Cargo.toml
└── src
    ├── a.rs
    ├── b
    │   ├── c.rs
    │   ├── d.rs
    │   └── e
    │       └── mod.rs
    │   └── mod.rs
    └── main.rs
```

1. Compiler finds `main.rs` (module `crate`)
   a. Declarations: `mod a`, `mod b`
2. Compiler finds `a.rs` (module `crate::a`)

1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Example

Modules are children of the module they are declared in

- `crate`
  - `crate::a`
  - `crate::b`

`src/`

```
[alexander@laptop rust-test]$ tree
.
├── Cargo.toml
└── src
    ├── a.rs
    ├── b
    │   ├── c.rs
    │   ├── d.rs
    │   ├── e
    │   │   └── mod.rs
    │   └── mod.rs
    └── main.rs
```

1. Compiler finds `main.rs` (module `crate`)
   a. Declarations: `mod a`, `mod b`
2. Compiler finds `a.rs` (module `crate::a`)
3. Compiler finds `b/mod.rs` (module `crate::b`)

1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Example

Modules are children of the module they are declared in

- `crate`
  - `crate::a`
  - `crate::b`

`src/b/`

```
[alexander@laptop rust-test]$ tree
.
├── Cargo.toml
└── src
    ├── a.rs
    ├── b
    │   ├── c.rs
    │   ├── d.rs
    │   ├── e
    │   │   └── mod.rs
    │   └── mod.rs
    └── main.rs
```

1. Compiler finds `main.rs` (module `crate`)
   a. Declarations: `mod a`, `mod b`
2. Compiler finds `a.rs` (module `crate::a`)
3. Compiler finds `b/mod.rs` (module `crate::b`)
   a. Declarations: `mod c`, `mod d`, `mod e`

1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Example

Modules are children of the module they are declared in

- `crate`
  - `crate::a`
  - `crate::b`
    - `crate::b::c`

`src/b/`

```
[alexander@laptop rust-test]$ tree
.
├── Cargo.toml
└── src
    ├── a.rs
    ├── b
    │   ├── c.rs
    │   ├── d.rs
    │   └── e
    │       └── mod.rs
    │   └── mod.rs
    └── main.rs
```

1. Compiler finds `main.rs` (module `crate`)
   a. Declarations: `mod a`, `mod b`
2. Compiler finds `a.rs` (module `crate::a`)
3. Compiler finds `b/mod.rs` (module `crate::b`)
   a. Declarations: `mod c`, `mod d`, `mod e`
4. Compiler finds `c.rs` (module `crate::b::c`)

1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Example

Modules are children of the module they are declared in

- `crate`
  - `crate::a`
  - `crate::b`
    - `crate::b::c`
    - `crate::b::d`

`src/b/`

```
[alexander@laptop rust-test]$ tree
.
├── Cargo.toml
└── src
    ├── a.rs
    ├── b
    │   ├── c.rs
    │   ├── d.rs
    │   └── e
    │       └── mod.rs
    │   └── mod.rs
    └── main.rs
```

1. Compiler finds `main.rs` (module `crate`)
   a. Declarations: `mod a`, `mod b`
2. Compiler finds `a.rs` (module `crate::a`)
3. Compiler finds `b/mod.rs` (module `crate::b`)
   a. Declarations: `mod c`, `mod d`, `mod e`
4. Compiler finds `c.rs` (module `crate::b::c`)
5. Compiler finds `d.rs` (module `crate::b::d`)

1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Example

Modules are children of the module they are declared in

- `crate`
  - `crate::a`
  - `crate::b`
    - `crate::b::c`
    - `crate::b::d`
    - `crate::b::e`

`src/b/`

```
[alexander@laptop rust-test]$ tree
.
├── Cargo.toml
└── src
    ├── a.rs
    ├── b
    │   ├── c.rs
    │   ├── d.rs
    │   └── e
    │       └── mod.rs
    └── mod.rs
    └── main.rs
```

1. Compiler finds `main.rs` (module `crate`)
   a. Declarations: `mod a`, `mod b`
2. Compiler finds `a.rs` (module `crate::a`)
3. Compiler finds `b/mod.rs` (module `crate::b`)
   a. Declarations: `mod c`, `mod d`, `mod e`
4. Compiler finds `c.rs` (module `crate::b::c`)
5. Compiler finds `d.rs` (module `crate::b::d`)
6. Compiler finds `e/mod.rs` (module `crate::b::e`)

1. `<module_name>.rs`
2. `<module_name>/mod.rs`

Imports

You can import members from modules with the `use` keyword

Brings the name into scope, can still be accessed with `module::member` syntax

```rust
// main.rs

mod my_other_file;

fn main() {
    my_other_file::hello();
}
```

```rust
// main.rs

mod my_other_file;
use my_other_file::hello;

fn main() {
    hello();
}
```

`super`: refers to the parent crate

Publicity

By default, all members are private to that module, but you can make them public with `pub`

When declaring modules, you usually want `pub mod`

You can also make struct fields public/private

```
mod mod_1 {
    // private function
    fn do_something() {
        println!("called function do_something");
    }

    // public function
    pub fn print() {
        do_something();
    }
}

fn main() {
    mod_1::print();
}
```

```
pub mod encapsulation {
    // Private struct
    struct PrivateStruct {
        field_1: i32,
        field_2: i32,
    }

    // Public struct with private fields
    pub struct EncapsulatedStruct {
        field_1: i32,
        field_2: i32,
    }

    // Public struct with public fields
    pub struct PublicStruct {
        pub field_1: i32,
        pub field_2: i32,
    }
}
```

More refined publicity modifiers

What do these mean?

1. `pub`
2. `pub(crate)`
3. `pub(super)`
4. `pub(self)`
5. `pub(in module_path)`

Re-exports

What happens if you combine `pub` and `use`?

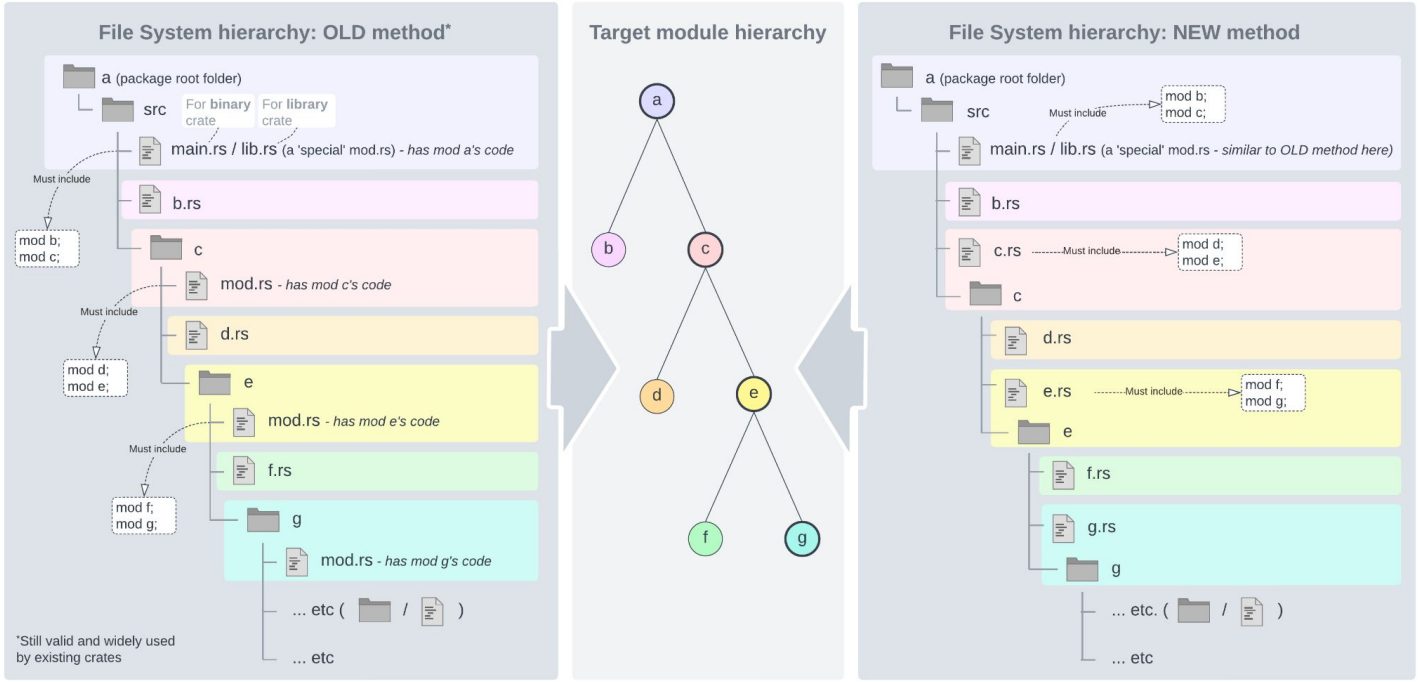
Whatever comes after the `use` looks like it belongs to the current module (both names are equally valid)

```
pub mod outer {
    pub use inner::MyStruct;

    pub mod inner {
        pub struct MyStruct;
    }
}

fn main() {
    let x = outer::MyStruct; => MyStruct
    let y = outer::inner::MyStruct; => MyStruct
}
```

Two ways to create a module hierarchy with files and folders in Rust



Example access 'paths'

To call `my_fn_g()` in module `g` from mod a (works in binary or library crate): `c::e::g::my_fn_g()`

To call `my_fn_c()` in module `c` from mod b (works in binary or library crate): `crate::c::my_fn_c()`

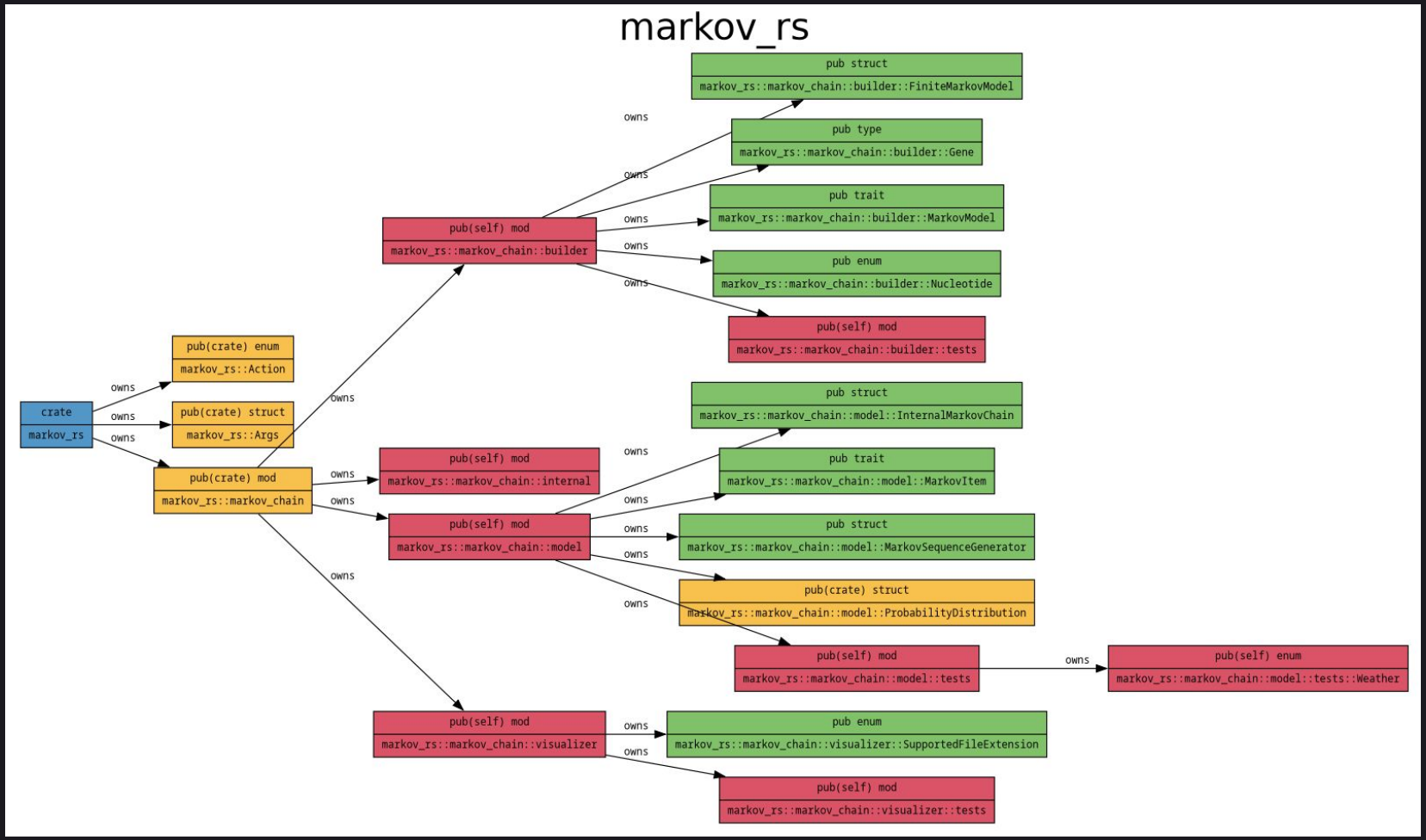
To call `my_fn_c()` in module `c` from external crate (if c is in binary crate module hierarchy): NOT POSSIBLE

To call `my_fn_c()` in module `c` from external crate (if c is in library crate module hierarchy): `a::c::my_fn_c()`

Legend

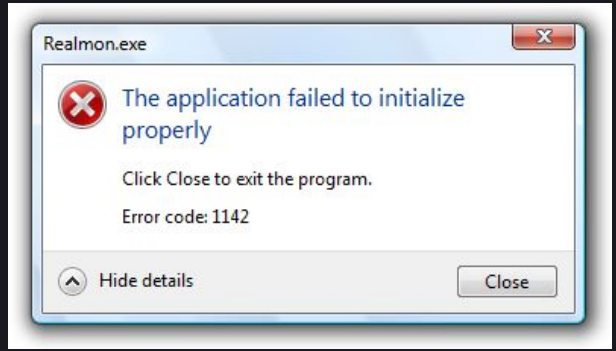
(x) 'Leaf' module. Still possible to add sub-modules *in code* with nested `mod` declarations. With NEW method can convert to 'internal node' module by adding new directory with the same name.

(x) 'Internal node' module. Can have child 'file-based' modules.



cargo-modules (crates.io)

Testing


Realmon.exe
The application failed to initialize properly
Click Close to exit the program.
Error code: 1142
Hide details
Close

Testing is easy in rust

Testing is a built-in feature to the language *(no external dependencies)*

1. Declare a module called `tests` with `#[cfg(test)]`
2. Tag any test functions with `#[test]`
3. Run them with `cargo test`

```rust
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2); ← (left, right) ⇒ usize
        assert_eq!(result, 4);
    }
}
```

Assertions

`assert!(x)`

Assert that `x` is true

`assert_eq!(a, b)`

Assert that `a` and `b` are equal

`assert_ne!(a, b)`

Assert that `a` and `b` are *not* equal

Example

Here is a simple example with unsigned integer subtraction

- Mark tests as failing with `#[should_panic]`
  - Do not confuse this with `Result::Err`, which is a valid return value

```
[alexander@laptop rust-test]$ cargo test
   Compiling rust-test v0.1.0 (/home/alexander/Coding/rust-test)
    Finished test [unoptimized + debuginfo] target(s) in 0.38s
     Running unittests src/lib.rs (target/debug/deps/rust_test-7b3b7b90ed5d9271)

running 3 tests
test tests::test_sub_neg - should panic ... ok
test tests::test_sub_pos ... ok
test tests::test_sub_zero ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Doc-tests rust-test

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

[alexander@laptop rust-test]$
```

```rust
pub fn sub(left: usize, right: usize) -> usize {
    if right > left {
        panic!("overflow");
    }
    left - right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_sub_zero() {
        assert_eq!(sub(0, 0), 0);
    }

    #[test]
    fn test_sub_pos() {
        assert_eq!(sub(4, 3), 1);
    }

    #[test]
    #[should_panic(expected = "overflow")]
    fn test_sub_neg() {
        sub(2, 7); ← (left, right)
    }
}
```

Quickcheck

Work smart not hard

https://github.com/BurntSushi/quickcheck

```
quickcheck! {
    fn prop_reverse_reverse(xs: Vec<usize>) -> bool {
        let rev: Vec<_> = xs.clone().into_iter().rev().collect();
        let revrev: Vec<_> = rev.into_iter().rev().collect();
        xs == revrev
    }
};
```

SURPRISE!
Impl Blocks

Associated Functions and Methods

- Implementation blocks define the behavior of types
  - You can impl a struct or enum

```
enum Nucleotide {
    A,
    T,
    C,
    G,
}

impl Nucleotide {
    fn flip(&self) -> Nucleotide {
        match self {
            Nucleotide::A => Nucleotide::T,
            Nucleotide::T => Nucleotide::A,
            Nucleotide::C => Nucleotide::G,
            Nucleotide::G => Nucleotide::C,
        }
    }
}
```

Methods are just functions

What is a method really?

- A function that takes `self` as the first parameter (like Java `this`)
- Special syntax sugar
- Impl blocks also define the `Self` type
- **Can access private struct fields**

```
impl Color {
    fn invert(&self) -> Color { ···
    fn invert_desugared(self: &Color) -> Color { ···
    fn invert_self(&self) -> Self { ···
}
```

`Self` vs. `self`

`Self` vs. `self`

`Self` — Type

`self` — Value

Method call syntax

```
impl Data {
    fn by_val(self) {}
    fn by_ref(&self) {}
    fn by_mut_ref(&mut self) {}
}
```

```
x.by_val();     = Data::by_val(x);
x.by_ref();     = Data::by_ref(&x);
x.by_mut_ref(); = Data::by_mut_ref(&mut x);
```

Normal Fully qualified

Example:
Vector3

**Similar to OOP, but without inheritance*

```rust
struct Vector3 {
    x: f32,
    y: f32,
    z: f32,
}

impl Vector3 {
    fn new(x: f32, y: f32, z: f32) -> Self {
        Self { x, y, z }
    }

    fn length(&self) -> f32 {
        (self.x * self.x + self.y * self.y + self.z * self.z).sqrt()
    }

    fn normalize(self) -> Self {
        let length = self.length(); => f32
        Self {
            x: self.x / length,
            y: self.y / length,
            z: self.z / length,
        }
    }
}

fn main() {
    let v = Vector3::new(0.0, 3.0, 4.0); <- (x, y, z) => Vector3

    assert_eq!(v.length(), 5.0);

    let v_normalized = v.normalize(); => Vector3

    assert_eq!(v_normalized.x, 0.0);
    assert_eq!(v_normalized.y, 0.6);
    assert_eq!(v_normalized.z, 0.8);
}
```

Impl: Live Coding