Rust programming

Course – 9

Gavrilut Dragos

rev 9

Agenda for today

1. Modules (visibility concepts)
2. Modules and multiple files
3. Crates
4. Conditional Compilation
5. Building scripts (build.rs)
6. Tests
7. Documentation
8. Workspaces

Modules

Modules

Rust does not have namespaces, but it does have a way to separate code in modules. This separation implies some visibility restriction (within and/or outside the module).

The general rule is that within a module, everything (declared in that module) is visible for every code in that module.

A module can be declared in two ways:

1. Using **mod {…}**
2. Using **mod** <file>

Modules

Let's see a simple example:

Rust

```rust
mod math {
    fn add(x: i32, y: i32)-> i32 {
        x+y
    }
}
fn main() {
    let res = math::add(10,20);
    println!("{res}");
}
```

Error

```
error[E0603]: function `add` is private
 --> src\main.rs:7:21
  |
7 |     let res = math::add(10,20);
  |                     ^^^ private function
  |
note: the function `add` is defined here
 --> src\main.rs:2:5
  |
2 |     fn add(x: i32, y: i32)-> i32 {
  |
```

Notice that function add is not visible (cannot be called) from function main.

Function add is considered private outside its module.

Modules

Let's see a simple example:

Rust

```
mod math {
    fn add(x: i32, y: i32)-> i32 {
        x+y
    }
    fn test() {
        println!("{}",add(1,2));
    }
}
fn main() {
    println!("OK");
}
```

Output

```
OK
```

Notice that function test can call function add. This is because both of them are part of the same module (math) and as such each one of them is visible for the other one.

Modules

So … how can we change the visibility of a function/object so that it can be accessible outside its module ?

Rust has a special keyword called pub (short from public) that can be used for this scope.

"pub" keyword has several formats:

- pub
- pub (super)
- pub (crate)
- pub (self)
- pub (in <*name*>)

Modules

Let's see a simple example:

Rust

```
mod math {
    pub fn add(x: i32, y: i32)-> i32 {
        x+y
    }
}
fn main() {
    let res = math::add(10,20);
    println!("{res}");
}
```

Output

```
30
```

Now the code work.

Notice the usage of "pub" keywork in front of the definition for a function.

Modules

Inner modules have to be made public in order to be accessible from outside their module the **pub** keywork must be use in front of them, even if math::simple::add is defined as pub.

Rust

```rust
mod math {
    mod simple {
        pub fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    fn add(x: i32, y: i32, z: i32)-> i32 {
        simple::add(simple::add(x,y), z)
    }
}
fn main() {
    let res = math::simple::add(10, 20);
    println!("{res}");
}
```

Error

```
error[E0603]: module `simple` is private
  --> src\main.rs:14:21
   |
14 |     let res = math::simple::add(10, 20);
   |                     ^^^^^^ private module
   |
note: the module `simple` is defined here
  --> src\main.rs:2:5
   |
2  |     mod simple {
   |
```

Modules

One we add the **pub** specifier in front of the simple module, we can access math::simple::add . Keep in mind that we can not access math::add as it is not pub, but that function can access the add function from module ***simple***.

Rust

```
mod math {
    pub mod simple {
        pub fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    fn add(x: i32, y: i32, z: i32)-> i32 {
        simple::add(simple::add(x,y), z)
    }
}
fn main() {
    let res = math::simple::add(10, 20);
    println!("{res}");
}
```

Output

```
30
```

Modules

In this case, math::simple::add is not public while math::add is. This means that from main math::add is accessible, and from math::add, math::simple::add is accessible because it is ***public***.

Rust

```
mod math {
    mod simple {
        pub fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub fn add(x: i32, y: i32, z: i32)-> i32 {
        simple::add(simple::add(x,y), z)
    }
}
fn main() {
    let res = math::add(10, 20, 30);
    println!("{res}");
}
```

Output

```
60
```

Modules

At the same time, if we remove the **pub** from the math::simple::add, then it can not be access from math::add and as such the program will not compile.

Rust

```
mod math {
    mod simple {
        fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub fn add(x: i32, y: i32, z: i32)-> i32 {
        simple::add(simple::add(x,y), z)
    }
}
fn main() {
    let res = math::add(10, 20, 30);
    println!("{res}");
}
```

Error

```
error[E0603]: function `add` is private
 --> src\main.rs:8:17
  |
8 |         simple::add(simple::add(x,y), z)
  |                 ^^^ private function
  |
note: the function `add` is defined here
 --> src\main.rs:3:9
  |
3 |         fn add(x: i32, y: i32) -> i32 {
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add` is private
 --> src\main.rs:8:29
  |
8 |         simple::add(simple::add(x,y), z)
  |                             ^^^ private function
  |
note: the function `add` is defined here
 --> src\main.rs:3:9
  |
3 |         fn add(x: i32, y: i32) -> i32 {
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Modules

But what if we want to access a function define in another module that is not a child of the current module ?

Rust

```rust
mod math {
    pub mod simple {
        pub (super) fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub mod complex {
        pub fn add(x: i32, y: i32, z: i32) -> i32 {
            super::simple::add(super::simple::add(x, y), z)
        }
    }
}
fn main() {
    let res = math::complex::add(10, 20, 30);
    println!("{res}");
}
```

Output

```
60
```

Modules

But what if we want to access a function define in another module that is not a child of the current module ?

Rust

```rust
mod math {
    pub mod simple {
        pub fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub mod complex {
        pub fn add(x: i32, y: i32, z: i32) -> i32 {
            super::simple::add(super::simple::add(x, y), z)
        }
    }
}
fn main() {
    let res = math::complex::add(10, 20, 30);
    println!("{res}");
}
```

Output

60

Notice the usage of indicator "**super**" to explicitly address function "***add***" from module simple.

Modules

Another observation is that we need both “add” function to be public and the modules they are in to be public as well (simple and complex) for the next code to compile.

Rust

```
mod math {
    pub mod simple {
        pub fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub mod complex {
        pub fn add(x: i32, y: i32, z: i32) -> i32 {
            super::simple::add(super::simple::add(x, y), z)
        }
    }
}
fn main() {
    let res = math::complex::add(10, 20, 30);
    let res2 = math::simple::add(10, 20);
    println!("{res},{res2}");
}
```

Output
60,30

Modules

Another observation is that we need both "add" function to be public and the modules they are in to be public as well (simple and complex) for the next code to compile.

Rust

```
mod math {
    pub mod simple {
        pub fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub mod complex {
        pub fn add(x: i32, y: i32, z: i32) -> i32 {
            super::simple::add(super::simple::add(x, y), z)
        }
    }
}
fn main() {
    let res = math::complex::add(10, 20, 30);
    let res2 = math::simple::add(10, 20);
    println!("{res},{res2}");
}
```

But what if we want this function to be available in math module, but is uncacheable outside math module ?

Modules

The solution is to declare math::simple::add function as pub(super). This will make it unaccesible from outside math module.

Rust

```rust
mod math {
    pub mod simple {
        pub (super) fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub mod complex {
        pub fn add(x: i32, y: i32, z: i32) -> i32 {
            super::simple::add(super::simple::add(x, y), z)
        }
    }
}
fn main() {
    let res = math::complex::add(10, 20, 30);
    let res2 = math::simple::add(10, 20);
    println!("{res},{res2}");
}
```

Error

```
error[E0603]: function `add` is private
  --> src\main.rs:15:30
   |
15 |     let res2 = math::simple::add(10, 20);
   |                              ^^^ private function
   |
note: the function `add` is defined here
  --> src\main.rs:3:9
   |
3  |         pub (super) fn add(x: i32, y: i32) -> i32 {
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Modules

The solution is to declare math::simple::add function as pub(super). This will make it inaccessible from outside math module.

Rust

```
mod math {
    pub mod simple {
        pub (super) fn add(x: i32, y: i32) -> i32 {
            x + y
        }
    }
    pub mod complex {
        pub fn add(x: i32, y: i32, z: i32) -> i32 {
            super::simple::add(super::simple::add(x, y), z)
        }
    }
}
fn main() {
    let res = math::complex::add(10, 20, 30);
    println!("{res},{res2}");
}
```

Output

```
60
```

Modules

The rule is that <u>by default</u>, everything declared within a module is considered **private** (meaning that it can be access by everything declared in that module and its ancestors) but not from other locations (other modules).

OBS: for a struct, this rule includes its member and its implementation (if a struct its declared **public** but its members are not, then its members can not be accessed from another modules).

<u>**There are two exceptions from this rule**</u>:

1. If a trait is declared public, its associated items are public as well
2. If an enum is declared public, its variants are public as well

Modules

Let's see some examples and discuss how visibility rules work here:

Rust

```
mod a {
    mod b {
        fn add(x: i32, y: i32) -> i32 { x + y }
        mod c {
            fn inc(x: i32) -> i32 { x+1 }
            mod d {
                fn sub(x:i32, y:i32) -> i32 {
                    super::super::add(x,-y)
                }
            }
        }
    }
}
fn main() {
    println!("ok");
}
```

Output
ok

Modules

Let's see some examples and discuss how visibility rules work here:

Rust

```
mod a {
    mod b {
        fn add(x: i32, y: i32) -> i32 { x + y }
        mod c {
            fn inc(x: i32) -> i32 { x+1 }
            mod d {
                fn sub(x:i32, y:i32) -> i32 {
                    super::super::add(x,-y)
                }
            }
        }
    }
}
fn main() {
    println!("ok");
}
```

Output

ok

First "**super**" refers to function ***sub*** parent, and since function ***sub*** is located in module "d", it refers to "c" (module "d" parent)

Modules

Let’s see some examples and discuss how visibility rules work here:

Rust

```
mod a {
    mod b {
        fn add(x: i32, y: i32) -> i32 { x + y }
        mod c {
            fn inc(x: i32) -> i32 { x+1 }
            mod d {
                fn sub(x:i32, y:i32) -> i32 {
                    super::super::add(x,-y)
                }
            }
        }
    }
}
fn main() {
    println!("ok");
}
```

Output
ok

Second “**super**” refers to the parent of “c” (obtained from the previous super call) that is “b”

First “**super**” refers to function ***sub*** parent, and since function ***sub*** is located in module “d”, it refers to “c” (module “d” parent)

Modules

Let's see some examples and discuss how visibility rules work here:

Rust

```
mod a {
    mod b {
        fn add(x: i32, y: i32) -> i32 { x + y }
        mod c {
            fn inc(x: i32) -> i32 { x+1 }
            mod d {
                fn sub(x:i32, y:i32) -> i32 {
                    super::super::add(x,-y)
                }
            }
        }
    }
}
fn main() {
    println!("ok");
}
```

Output
ok

This means that "super::super::add(...)" refers to the "a::b::add(...)" method.

"a::b::add(...)" is **private**. However, it can be accessed by any ancestor of "b"

Modules

Let's see some examples and discuss how visibility rules work here:

Rust

```
mod a {
    mod b {
        fn add(x: i32, y: i32) -> i32 { x + y }
        mod c {
            fn inc(x: i32) -> i32 { x+1 }
            mod d {
                fn sub(x:i32, y:i32) -> i32 {
                    super::super::add(x,-y)
                }
            }
        }
    }
}
fn main() {
    printl
}
```

Output
ok

Alternativelly, you can use the following syntax to refer to the same thing: `crate::a::b::add(x,-y)`

A crate is a compilation unit in rust (but for the moment we can look at the concept as the ***root*** of all modules from the current library/application).

Modules

Let's modify the previous example so that we can access function sub from main.

Rust

```
mod a {
    mod b {
        fn add(x: i32, y: i32) -> i32 { x + y }
        mod c {
            fn inc(x: i32) -> i32 { x+1 }
            mod d {
                pub fn sub(x:i32, y:i32) -> i32 {
                    crate::a::b::add(x,-y)
                }
            }
        }
    }
}
fn main() {
    let x = a::b::c::d::sub(10,4);
    println!("{x}");
}
```

Error

```
error[E0603]: module `b` is private
  --> src\main.rs:15:16
   |
15 |     let x = a::b::c::d::sub(10,4);
   |               ^ private module
   |
note: the module `b` is defined here
  --> src\main.rs:2:5
   |
2  |     mod b {
   |     ^^^^^
```

The code will not compile, but the reason is not because sub function is not public, is because the path to sub function (b,c and d) are not public.

Module "a" does not have to be public as it is at the same level as function main.

Modules

Let's modify the previous example so that we can access function sub from main.

Rust

```
mod a {
    pub mod b {
        fn add(x: i32, y: i32) -> i32 { x + y }
        pub mod c {
            fn inc(x: i32) -> i32 { x+1 }
            pub mod d {
                pub fn sub(x:i32, y:i32) -> i32 {
                    crate::a::b::add(x,-y)
                }
            }
        }
    }
}
fn main() {
    let x = a::b::c::d::sub(10,4);
    println!("{x}");
}
```

Output
6

Not the code compiles and prints 6. Keep in mind that function sub also has to be public to be accessed.

Modules

Let's see how a structure works with modules:

Rust

```
mod a {
    struct MyClass {
        x: i32
    }
}
fn main() {
    let a = a::MyClass {x: 10};
    println!("{}",a.x);
}
```

Error

```
error[E0603]: struct `MyClass` is private
 --> src\main.rs:7:16
  |
7 |     let a = a::MyClass {x: 10};
  |                ^^^^^^^ private struct
  |
note: the struct `MyClass` is defined here
 --> src\main.rs:2:5
  |
2 |     struct MyClass {
  |
```

Notice that in this example, MyClass structure is **private** and as such trying to access it from outside its module is not possible.

The first step we should do is to make it **public**.

Modules

Let's see how a structure works with modules:

Rust

```
mod a {
    pub struct MyClass {
        x: i32
    }
}
fn main() {
    let a = a::MyClass {x: 10};
    println!("{}",a.x);
}
```

Error

```
error[E0616]: field `x` of struct `MyClass` is private
 --> src\main.rs:8:21
  |
8 |     println!("{}",a.x);
  |                   ^ private field
```

However, even if we make struct MyClass **public**, the code will not compile as one of its fields that is required for initialization (MyStruct::x) is not public !

The solution is to make MyStruct::x public as well.

Modules

Let’s see how a structure works with modules:

Rust

```
mod a {
    pub struct MyClass {
        pub x: i32
    }
}
fn main() {
    let a = a::MyClass {x: 10};
    println!("{}",a.x);
}
```

Output

```
10
```

Now the code compiles as expected.

But what if we don’t want MyStruct::x to be accessible ?
Can we find a way to create an object of type MyStruct but without having access to data member “x” ?

Modules

Let's see how a structure works with modules:

Rust

```
mod a {
    pub struct MyClass {
        x: i32
    }
    impl MyClass {
        pub fn new(value: i32)->MyClass { MyClass { x: value } }
        pub fn get(&self)->i32 { self.x }
    }
}
fn main() {
    let a = a::MyClass::new(10);
    println!("{}",a.get());
}
```

Output

```
10
```

The solution is to create two method:

- *MyClass*::**new**(…) to create an object of type MyClass
- *MyClass*::**get**() to retrieve the value of data member "x"

Notice that both new and get methods from ***MyClass*** have the pub specifier. If we don't explicitly add that specifier, those methods will not be available outside module "a" and as such the code from ***main*** **will not compile**.

Modules

But what if we have a different scenario (for the same structure ***MyClass***).

We want to be able to access data member "x" at any time (both read and write) but we don't want to be able to create an object of type ***MyClass*** in the standard way. In other words, considering the following code:

```
mod a {
    pub struct MyClass { … }
}
fn main() {
    let a = a::MyClass {x: 10};
    println!("{}",a.x);
}
```

We would like for:

1. Line `let a = a::MyClass {x: 10};` to be impossible (**should not compile**)
2. Line `println!("{}",a.x);` to be possible (**should compile**)

Modules

The solution is to add an extra member to the structure MyClass (let's call it for the moment "extra") That member has to be private and as such instantiation of MyClass object will not possible. This is often referred to as a **phantom member**.

Rust

```
mod a {
    pub struct MyClass {
        pub x: i32,
        extra: i32
    }
}
fn main() {
    let a = a::MyClass {x: 10, extra: 0};
    println!("{}",a.x);
}
```

Error

```
error[E0451]: field `extra` of struct `MyClass` is private
 --> src\main.rs:8:32
  |
8 |     let a = a::MyClass {x: 10, extra: 0};
  |                                ^^^^^^^^ private field
```

This solution has a drawback → an object of type MyClass now has a size of 8 (instead of 4, its original size).

Modules

Let’s see how this solution works:

Rust

```
mod a {
    pub struct MyClass {
        pub x: i32,
        extra: i32
    }
    impl MyClass {
        pub fn new(value: i32) -> MyClass { MyClass { x: value, extra: 0 } }
    }
}
fn main() {
    let a = a::MyClass::new(10);
    println!("{}, size={}",a.x,std::mem::size_of::<a::MyClass>());
}
```

Output
10, size=8

Q: *What can we do to make MyClass of size 4 (its designed size) but still respect our constraints (in C++ we would have used a private constructor).*

Modules

The solution lies in creating a **ZST** (Zero Sized Type) that can be used for the type of the "_extra" data member:

Rust

```
mod a {

    struct MyEmptyStruct;

    pub struct MyClass {
        pub x: i32,
        _extra: MyEmptyStruct
    }
    impl MyClass {
        pub fn new(value: i32) -> MyClass { MyClass { x: value, _extra: MyEmptyStruct{} } }
    }
}
fn main() {
    let a = a::MyClass::new(10);
    println!("{}, size={}",a.x,std::mem::size_of::<a::MyClass>());
}
```

Output
10, size=4

Modules

A similar solution can be obtained via a special generic/template called PhantomData. This allows adding various data members within a structure that will are not public but have a size 0 and as such no impact on the final size of the structure.

Rust

```rust
mod a {
    use std::marker::PhantomData;
    pub struct MyClass {
        pub x: i32,
        _extra: PhantomData<MyClass>
    }
    impl MyClass {
        pub fn new(value: i32) -> MyClass { MyClass { x: value, _extra: PhantomData::default() } }
    }
}
fn main() {
    let a = a::MyClass::new(10);
    println!("{}, size={}",a.x,std::mem::size_of::<a::MyClass>());
}
```

Output
10, size=4

Modules

An even more simple solution is to use the void type (). The result is similar, and it does not require any use of another module or an ZST structure.

Rust

```
mod a {
    pub struct MyClass {
        pub x: i32,
        _extra: ()
    }
    impl MyClass {
        pub fn new(value: i32) -> MyClass {
            MyClass {
                x: value,
                _extra: ()
            }
        }
    }
}
fn main() {
    let a = a::MyClass::new(10);
    println!("{}, size={}",a.x,std::mem::size_of::<a::MyClass>());
}
```

Output

```
10, size=4
```

Modules

A similar solution can be obtained via a special generic/template called PhantomData. This allows adding various data members within a structure that will are not public but have a size 0 and as such no impact on the final size of the structure.

Rust

```rust
mod a {
    use std::marker::PhantomData;
    pub struct MyClass {
        pub x: i32,
        _extra: PhantomData<MyClass>
    }
    impl MyClass {
                             Class { x: value, _extra: PhantomData::default() } }

}
fn main() {
    let a = a::MyClass::new(10);
    println!("{}, size={}",a.x,std::mem::size_of::<a::MyClass>());
}
```

PhantonData generic is defined as follows:

```rust
pub struct PhantomData<T: ?Sized>;
```

Output

10, size=4

Modules

A similar solution can be obtained via a special generic/template called PhantomData. This allows adding various data members within a structure that will are not public but have a size 0 and as such no impact on the final size of the structure.

Rust

```
mod a {
    use std::marker::PhantomData;
    pub struct MyClas
        pub x: i32,
        _extra:
    }
    impl MyClass {
        pub fn new(value: i32) -> MyClass { MyClass { x: value, _extra: PhantomData::default() } }
    }
}
fn main() {
    let a = a::MyClass::new(10);
    println!("{}, size={}",a.x,std::mem::size_of::<a::MyClass>());
}
```

Output

10, size=4

Notice the "_" (underline) character in from of the name of this phantom variable. While not required, it is recomanded that private members (or members that will not be used to be prefixed by an underline so that Rust compiler will know not to throw warnings related to them (e.g. unused variable).

Modules

Let’s see how an **enum** works with modules:

Rust

```
mod a {
    #[derive(Debug)]
    pub enum Color {
        Red,
        Green,
        Blue
    }
}
fn main() {
    let c = a::Color::Red;
    println!("{:?}",c);
}
```

Output
Red

This is one exception (since we have an **enum** we don’t need to add pub for each of its variants, they are implicitly public when we define the **enum** public).

OBS: *The same logic applies for public traits in regard to their methods.*

Modules

We can also use modules to simulate a static data member for a structure.

Rust

```
mod a {
    static mut counter: i32 = 0;
    pub struct MyClass { pub x: i32 }
    impl MyClass {
        pub fn new() -> MyClass {
            unsafe {
                crate::a::counter += 1;
                MyClass { x: crate::a::counter }
            }
        }
        pub fn get(&self) -> i32 { self.x }
    }
}
fn main() {
    for i in 0..5 {
        let instance = a::MyClass::new();
        println!("{}", instance.get());
    }
}
```

Output
1
2
3
4
5

Modules

There are also cases where you might not want to use the full qualifier name every time (especially it is long) -> for example the next example:

```
mod a_very_long_module {
    pub struct MyClass { pub x: i32 }
}
fn main() {
    let obj = a_very_long_module::MyClass { x: 10 };
    println!("{}", obj.x);
}
```

Output
10

The solution is to use the “use” keyword in the following way:

```
use a_very_long_module::*;
mod a_very_long_module {
    pub struct MyClass { pub x: i32 }
}
fn main() {
    let obj = MyClass { x: 10 };
    println!("{}", obj.x);
}
```

or

```
use a_very_long_module::MyClass;
mod a_very_long_module {
    pub struct MyClass { pub x: i32 }
}
fn main() {
    let obj = MyClass { x: 10 };
    println!("{}", obj.x);
}
```

Output
10

Modules

It is also possible to re-export a structure or a function by using the "pub use" syntax within another module. In the next example, a::b::c::add(…) is not visible from main because b is not public. However, the use of pub use in module "a" makes it visible.

Rust

```
mod a {
    mod b {
        pub mod c {
            pub fn add(x:i32, y: i32)-> i32 { x+y }
        }
    }
    pub use self::b::c::add;
}
fn main() {
    let x = a::add(10,20);
    println!("{x}");
}
```

Output
30

Notice that we don't call the add function by its full qualifier a::b::c::add(…) but after its re-export qualifier: a::add(…)

Modules

It is also possible to re-export a structure or a function by using the "pub use" syntax within another module. In the next example, a::b::c::add(…) is not visible from main because b is not public. However, the use of pub use in module "a" makes it visible.

Rust

```
mod a {
    mod b {
        pub mod c {
            pub fn add(x:i32, y: i32)-> i32 { x+y }
        }
    }
    pub use self::
}
fn main() {
    let x = a::add(10,20);
    println!("{x}");
}
```

Notice the usage of self to refer to the current module. This is preferred when talking about relative paths

Output
30

Modules

It is also possible to re-export a structure or a function by using the "pub use" syntax within another module. In the next example, a::b::c::add(…) is not visible from main because b is not public. However, the use of pub use in module "a" makes it visible.

Rust

```
mod a {
    mod b {
        pub mod c {
            pub fn add(x:i32, y: i32)-> i32 { x+y }
        }
    }
    pub use self::b::c::add;
}
fn main() {
    let x = a::add(10,20);
    println!("{x}");
}
```

It is also important for "c" module and "add" function to be public so that the entire path is accessible from module "a"

Output
30

Modules & Multiple files

Modules & multiple files

Sometimes a program is too large to keep it in a single file. Even if that file is organized into multiple modules, it is still hard to navigate around it. The solution is to split that file into multiple ones and create a module like hierarchy around this.

Currently, Rust considers that each module should be form out of:

- A file with the name <module>.rs that contains module specific definitions, re-exports, etc
- An optional folder with the name <module> that will include sub-modules of the parent module (if any – hence optional).

Modules & multiple files

Let's consider the following hierarchy for a mathematical module:

- math → root module
    - simple → a module that contains simple operations (like sum of two number, multiplication of two numbers, etc)
    - complex → a module that contains more complex operations (like sum of all numbers from an array, or their product, …)
    - constants → a module that contains some mathematical constants (like Pi, E, etc)
    - other → logs and debug stuff
        - debug → debug methods for testing the result
        - log → log method

Observations:

- "**math::other**" module should not be accessible outside math module.
- Methods from "**math::other::log**" or "**math::other::debug**" must be visible within math module
- "**math**" module must have some configuration functions (to enable/disable debug/log)

Modules & multiple files

Step 1 → create a hierarchy of files and folders within the /src folder of our Rust program

- src
  - math
    - ***simple.rs***
    - ***complex.rs***
    - ***constants.rs***
    - other
      - ***debug.rs***
      - ***log.rs***
    - ***other.rs***
  - ***math.rs***
  - ***main.rs***

Modules & multiple files

Step 2 → Let's write the code for "debug" and "log" modules

- src
  - math
    - *simple.rs*
    - *complex.rs*
    - *constants.rs*
    - other
      - ***debug.rs***
      - *log.rs*
    - *other.rs*
  - *math.rs*
  - *main.rs*

Rust

```rust
static mut enabled: bool = false;
pub(in crate::math) fn msg(s: &str) {
    unsafe {
        if enabled {
            println!("{}", s);
        }
    }
}
pub(in crate::math) fn enable_debug_mode(value: bool) {
    unsafe {
        enabled = value;
    }
}
```

Modules & multiple files

Step 2 → Let's write the code for "debug" and "log" modules

- src
  - math
    - *simple.rs*
    - *complex.rs*
    - *constants.rs*
    - other
      - ***debug.rs***
      - *log.rs*
    - *other.rs*
  - *math.rs*
  - *main.rs*

Rust

```
static mut enabled: bool = false;
pub(in crate::math) fn msg(s: &str) {
    unsafe {
        if enabled {
            println!("{}
        }
    }
}
pub(in crate::math) fn enable_debug_mode(value: bool) {
    unsafe {
        enabled = value;
    }
}
```

Notice that both method are public (but only in crate::math !)
This means that they can be accessible with math module, but not outside it.

Modules & multiple files

Step 2 → Let's write the code for "debug" and "log" modules

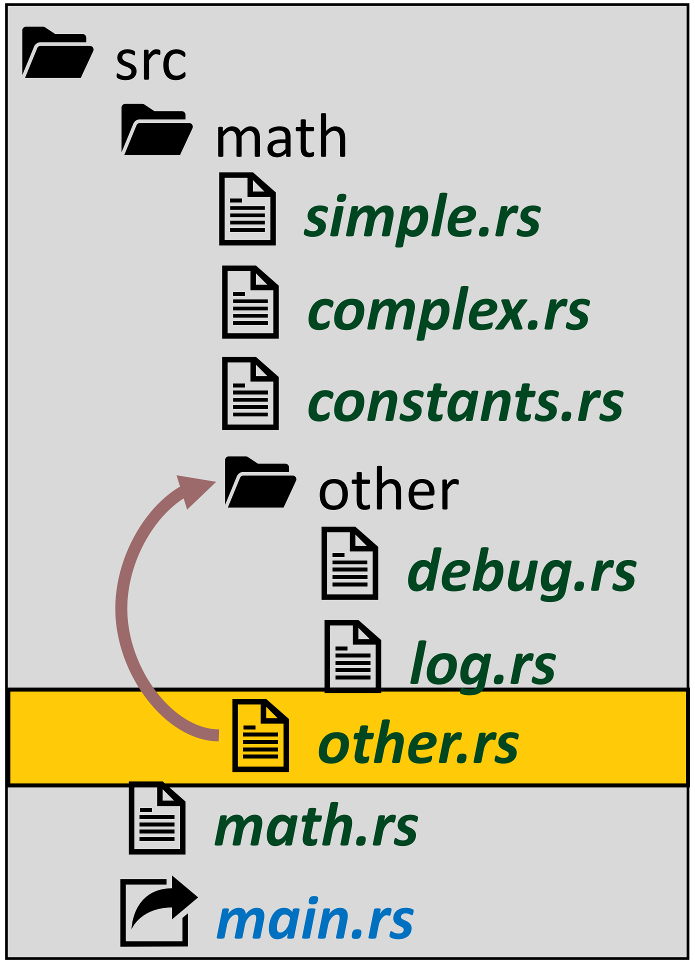
- src
  - math
    - *simple.rs*
    - *complex.rs*
    - *constants.rs*
    - other
      - *debug.rs*
      - ***log.rs***
    - *other.rs*
  - *math.rs*
  - *main.rs*

Rust

```rust
static mut enabled: bool = false;
pub(in crate::math) fn msg(function: &str, s: &str) {
    unsafe {
        if enabled {
            println!("Logging: (Function: {}) -> {}", function, s);
        }
    }
}
pub(in crate::math) fn enable_log_mode(value: bool) {
    unsafe {
        enabled = value;
    }
}
```

Modules & multiple files

Step 3 → Let's write the code for "other" module



Rust

```rust
pub (in crate::math) mod debug;
pub (in crate::math) mod log;
```

In this case the code is quite simple. File "other.rs" just creates two new sub-modules (debug and log) that corresponds to the files debug.rs and log.rs and that are public (in **crate::math**) → meaning that they are visible within **math** module, but not outside it.

Modules & multiple files

Step 4 → Let's write the code for "constants" module

- src
  - math
    - *simple.rs*
    - *complex.rs*
    - ***constants.rs***
    - other
      - *debug.rs*
      - *log.rs*
    - *other.rs*
  - *math.rs*
  - *main.rs*

Rust

```rust
pub const PI: f32 = 3.14;
pub const E: f32 = 2.71;
```

In this case both constants are declared as public (meaning that they can be used from outside module main).

Modules & multiple files

Step 5 → Let's write the code for "complex" module

- src
  - math
    - *simple.rs*
    - ***complex.rs***
    - *constants.rs*
    - other
      - *debug.rs*
      - *log.rs*
    - *other.rs*
  - *math.rs*
  - *main.rs*

Rust

```
pub fn sum(v: &[i32]) -> i32 {
    let mut elements_sum = 0;
    for elem in v {
        elements_sum += elem;
    }
    elements_sum
}
pub fn prod(v: &[i32]) -> i32 {
    let mut elements_prod = 1;
    for elem in v {
        elements_prod *= elem;
    }
    elements_prod
}
```

Notice that both "sum" and "prod" functions are declared as public (so that they will be accessible from outside math module).

Modules & multiple files

Step 6 → Let's write the code for "simple" module

- src
  - math
    - ***simple.rs***
    - *complex.rs*
    - *constants.rs*
    - other
      - *debug.rs*
      - *log.rs*
    - *other.rs*
  - *math.rs*
  - *main.rs*

Rust

```rust
pub fn add(x: i32, y: i32) -> i32 {
    crate::math::other::debug::msg("add two numbers");
    crate::math::other::log::msg("simple::add","add two numbers");
    x + y
}
pub fn mul(x: i32, y: i32) -> i32 {
    crate::math::other::debug::msg("multiply two numbers");
    crate::math::other::log::msg("simple::mul","multiply two numbers");
    x * y
}
```

Notice that both "add" and "mul" functions are declared as public. Before each one of them is called, they call function from both "debug" and "log" module that if enable will print a message on the screen.

Modules & multiple files

Step 7 → Let's write the code for "math" module

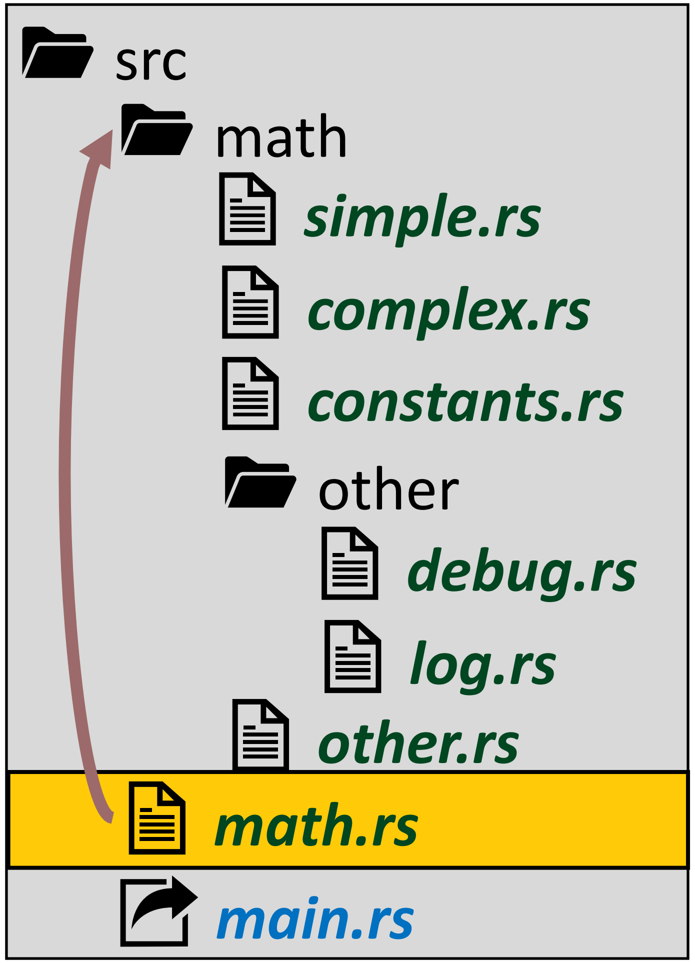
```
src
    math
        simple.rs
        complex.rs
        constants.rs
        other
            debug.rs
            log.rs
        other.rs
    math.rs
    main.rs
```

Rust

```rust
pub mod simple;
pub mod complex;
pub mod constants;
mod other;

#[derive(PartialEq)]
pub enum InfoMode {
    None,
    Log,
    Debug
}

pub fn set_info_mode(mode: InfoMode) {
    other::debug::enable_debug_mode(mode == InfoMode::Debug);
    other::log::enable_log_mode(mode == InfoMode::Log);
}
```

Modules & multiple files

Step 7 → Let's write the code for "math" module



Rust

```rust
pub mod simple;
pub mod complex;
pub mod constants;
mod other;

#[derive(PartialEq)]
pub enum InfoMode {
    None,
    Log,
    Debug
}

pub fn set_info_mode(mode: InfoMode) {
    other::debug::enable_debug_mode(mode == InfoMode::Debug);
    other::log::enable_log_mode(mode == InfoMode::Log);
}
```

Notice that module simple, complex and constants are defined here and are declared as ***public*** (so that they can be accessible from outside math module).

Modules & multiple files

Step 7 → Let's write the code for "math" module

- src
  - math
    - *simple.rs*
    - *complex.rs*
    - *constants.rs*
    - other
      - *debug.rs*
      - *log.rs*
    - *other.rs*
  - ***math.rs***
  - *main.rs*

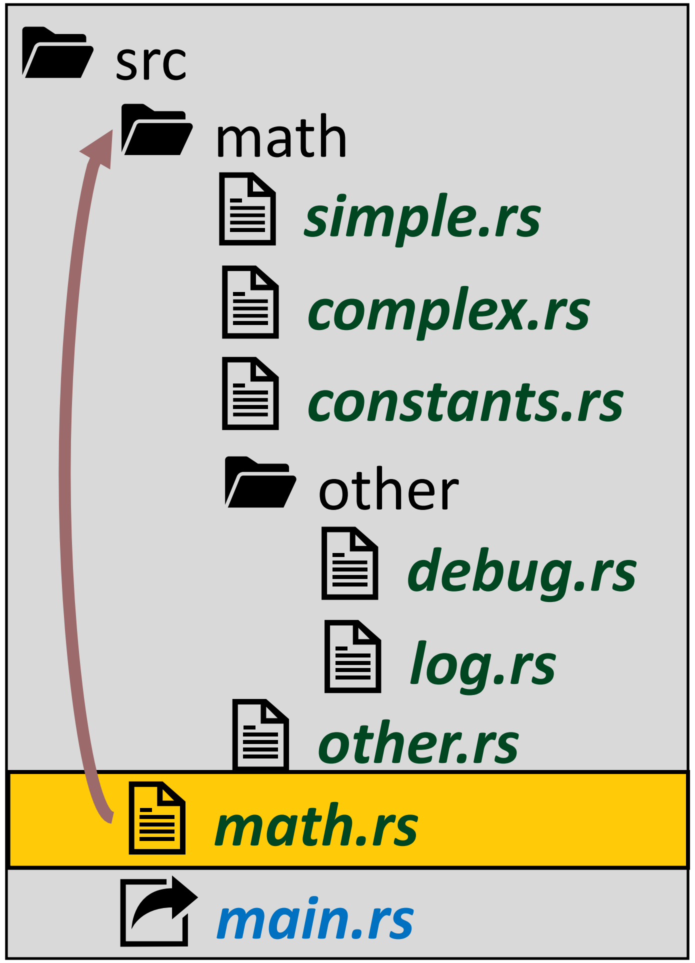
Rust

```
pub mod simple;
pub mod complex;
pub mod consta ts
mod other;

#[derive(PartialE
pub enum InfoMode
    None,
    Log,
    Debug
}

pub fn set_info_mode(mode: InfoMode) {
    other::debug::enable_debug_mode(mode == InfoMode::Debug);
    other::log::enable_log_mode(mode == InfoMode::Log);
}
```

Notice that module other, is declare as ***private***. This means that it can be accessed from here but not outside math module. This combine with pub (in crate::math) declaration for functions/modules from other, makes those functions accessible in math but not outside it.

Modules & multiple files

Step 7 → Let's write the code for "math" module



Rust

```
pub mod simple;
pub mod complex;
pub mod constants;
mod other;

#[derive(PartialEq)]
pub enum InfoMode {
    None,
    Log,
    Debug
}

pub fn set_info_mode(mode: InfoMode) {
    other::debug::enable_debug_mode(mode == InfoMode::Debug);
    other::log::enable_log_mode(mode == InfoMode::Log);
}
```

We need this **enum** to explain what kind of logging method we want to have. As such, this **enum** is declared ***public*** (and subsequentially, its variants are ***public*** as well).

Modules & multiple files

Step 7 → Let's write the code for "math" module

- src
  - math
    - *simple.rs*
    - *complex.rs*
    - *constants.rs*
    - other
      - *debug.rs*
      - *log.rs*
    - *other.rs*
  - *math.rs*
  - *main.rs*

Rust

```rust
pub mod simple;
pub mod complex;
pub mod constants;
mod other;

#[derive(Partia
pub enum InfoMo
    None,
    Log,
    Debug
}

pub fn set_info_mode(mode: InfoMode) {
    other::debug::enable_debug_mode(mode == InfoMode::Debug);
    other::log::enable_log_mode(mode == InfoMode::Log);
}
```

Finally, module math also exports (besides its submodules, a method to enable/disable loging mode).

Modules & multiple files

Step 8 → Let's write the code for "main.rs" file

- src
  - math
    - *simple.rs*
    - *complex.rs*
    - *constants.rs*
    - other
      - *debug.rs*
      - *log.rs*
    - *other.rs*
  - *math.rs*
  - ***main.rs***

Rust

Notice that we still have to define math as a module ! If we don't do this, its functions and sub-modules will not be accessible !

```rust
mod math;

fn main() {
    // usage of math::simple
    println!("1+2 = {}",math::simple::add(1, 2));
    println!("2*4 = {}",math::simple::mul(2, 4));

    // usage of math::complex
    let v = [1,2,3,4,5];
    println!("Sum of all elements from {:?} is {}",&v, math::complex::sum(&v));
    println!("Product of all elements from {:?} is {}",&v, math::complex::prod(&v));

    // usage of math::constants
    println!("Pi = {}, E = {}", math::constants::PI, math::constants::E);

    // log modes
    math::set_info_mode(math::InfoMode::Debug);
    println!("1+2 = {}",math::simple::add(1, 2));
    math::set_info_mode(math::InfoMode::Log);
    println!("1+2 = {}",math::simple::add(1, 2));
    math::set_info_mode(math::InfoMode::None);
    println!("1+2 = {}",math::simple::add(1, 2));
}
```

Modules & multiple files

Upon execution, the next program should print the following:

- src
  - math
    - ***simple.rs***
    - ***complex.rs***
    - ***constants.rs***
    - other
      - ***debug.rs***
      - ***log.rs***
    - ***other.rs***
  - ***math.rs***
  - ***main.rs***

Rust

```
mod math;

fn main() {
    // usage of
    println!("1
    println!("2

    // usage of
    let v = [1,
    println!("S
    println!("P                                                    ));

    // usage of
    println!("P

    // log mode
    math::set_i
    println!("1
    math::set_i
    println!("1+2 = {}",math::simple::add(1, 2));
    math::set_info_mode(math::InfoMode::None);
    println!("1+2 = {}",math::simple::add(1, 2));
}
```

Output

```
1+2 = 3
2*4 = 8
Sum of all elements from [1, 2, 3, 4, 5] is 15
Product of all elements from [1, 2, 3, 4, 5] is 120
Pi = 3.14, E = 2.71
add two numbers
1+2 = 3
Logging: (Function: simple::add) -> add two numbers
1+2 = 3
1+2 = 3
```

Crates

Crates

A ***crate*** in Rust is a compilation unit (e.g a library). In terms of binary output, a crate can correspond to a .dll file in Windows or a .so file in Unix based systems or a .dylib in OSX.

To create a ***crate***, use the following cargo command:

```
cargo new --lib <library_name>
```

Example: running the next command: "cargo new –lib my_math_lib" will create:

- ***Folder***: my_new_math_lib
  - ***Folder***: src
    - ***File***: lib.rs ➔ this is the main entry for the new library.
  - ***File***: Cargo.toml

Crates

Usually, the code from lib.rs will indicate the functions/modules that are exported:

Rust

```
pub fn add(x: i32, y: i32) -> i32 { x + y }
pub fn sub(x: i32, y: i32) -> i32 { x - y }
pub fn mul(x: i32, y: i32) -> i32 { x * y }
pub fn div(x: i32, y: i32) -> i32 { x / y }
pub fn rem(x: i32, y: i32) -> i32 { x % y }
```

If the library is more complex, other modules can be built (similar to what we have discussed on the previous chapters) and lib.rs just links all modules together.

Notice that if you want to export something out of the crate you have to mark it as public.

Crates

But what happens when we build a crate → it depends on what is its purpose. There are a couple of reasons a crate is being created:

1. To act as a **library for another Rust program** (the default case). This implies static linkage between another Rust program and this crate
2. A **dynamic linked library** ([Windows: .dll, Linux: .so, OSX: .dylib]). This implies other programs (maybe written in a different language can use this library)
3. A **static library** for other programs written in a different language.

While each one of these cases, relies on the existing of lib.rs, the difference is made from cargo.toml file

Crates

Case 1 → a static library for another Rust program

cargo.toml

```toml
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021"

[dependencies]
```

The cargo.toml file contains just the regular fields: **name**, **version** and **edition**. Optionally if other similar crates are being used in this one, they are refered in the section ***dependencies***.

The execution of cargo build will not produce a binary, but an object file named: lib<name>.rlib (in our case it will be named ***libmy_math_lib.rlib***)

Crates

Case 1 → a static library for another Rust program

cargo.toml

```toml
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021"

[dependencies]
```

The cargo.toml file contains just the regular fields: **name**, **version** and **edition**. Optionally if other similar crates are being used in this one, they are refered in the section ***dependencies***.

The execution of cargo build will not produce a binary, but an object file named: lib<name>.rlib (in our case it will be named ***libmy_math_lib.rlib***)

Crates

Case 1 → a static library for another Rust program

You can use such a crate in other Rust programs in different ways:

1. You can publish it in crates.io and then use it as a dependency
2. You can upload it to a git repository and link it directly from there
3. You can keep it locally and link it directly from its local folder
4. You can use your own registry

Crates

Case 1 → a static library for another Rust program

1. Publish to crates.io and use it as a dependency

Before you publish it, you need to make sure that your cargo.toml has the following fields:

cargo.toml

```
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021"
authors = ["author1", …]
description = "some description on what crate is doing"
license = "MIT"
keywords = ["keyword1", …]
categories = ["category1", …]
repository = "https://github.com/...."
readme = "README.md"
```

Crates

Case 1 → a static library for another Rust program

<u>1. Publish to crates.io and use it as a dependency</u>

Before you publish it, you need to make sure that your cargo.toml has the following fields:

cargo.toml

```
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021"
authors = ["author1", …]
description = "some description on what crate is doing"
license = "MIT"
keywords = ["keyword1", …]
categories = ["category1", …]
repository = "https://github.com/...."
readme = "README.md"
```

Required for compatibility reasons (an edition is a release in time where backwards compatibility is not maintained anymore). Currently, there are three editions: 2015, 2018 and 2021

Crates

Case 1 → a static library for another Rust program

1. Publish to crates.io and use it as a dependency

Before you publish it, you need to make sure that your cargo.toml has the following fields:

cargo.toml

```
[package]
name = "my_math_lib"
description = "some description on what crate is doing"
license = "MIT"
keywords = ["keyword1", …]
categories = ["category1", …]
repository = "https://github.com/...."
readme = "README.md"
```

A specific category that crates.io uses to filter down crates
(ex: **development-tools::procedural-macro-helpers**)
A list of all supported categories can be found here: https://crates.io/categories/

Crates

Case 1 → a static library for another Rust program

1. Publish to crates.io and use it as a dependency

Before you publish it, you need to make sure that your cargo.toml has the following fields:

cargo.toml

```
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021“
authors = ["author1", …]
d                                    t crate is doing"
l
keywords = [ keywo    1 , …]
categories = ["cate  ory1", …]
repository = "https://github.com/...."
readme = "README.md"
```

You need to upload your crate to a public repo

Crates

Case 1 → a static library for another Rust program

1. Publish to crates.io and use it as a dependency

Before you publish it, you need to make sure that your cargo.toml has the following fields:

cargo.toml

```
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021“
authors = ["author1", …]
                   on on what crate is doing"

categories   ["category1", …]
repository = "https://github.com/...."
readme = "README.md"
```

Make sure that you have a readme file (markdown format .md) in your repo where you describe how that crate should be used.

Crates

Case 1 → a static library for another Rust program

1. Publish to crates.io and use it as a dependency

After you finish completing your cargo.toml file , you can run the following command to publish your crate:

```
cargo publish
```

You can also run `cargo publish --dry-run` to test for errors before uploading a new version.

OBS: *You have to create an account on crates.io first !*

OBS: *Make sure that you increment your version (in cargo.toml) file before you are uploading to crates.io*

Crates

Case 1 → a static library for another Rust program

1. Publish to crates.io and use it as a dependency

Once you finish uploading , you (or someone else) can use it by simply adding your crate in his Rust application cargo.toml dependencies section.

cargo.toml (for another application)

```
[package]
name = "another_app"
version = "1.2.3"
edition = "2021"

[dependencies]
my_math_lib = "0.1.0"
```

In this case, another application is using "my_math_lib" version 0.1.0 in its dependencies.

Crates

Case 1 → a static library for another Rust program

2. Link your crate from a git repository

Once you crate is completed, uploaded to a link repo and link it in another application in the following way:

cargo.toml (for another application)

```
[package]
name = "another_app"
version = "1.2.3"
edition = "2021"

[dependencies]
my_math_lib = {
                git = "<a git URI>" ,
                branch = "branch_name"
              }
```

In this context, <a git URI> can be an URI towards any ***git based*** system (e.g. github, bitbucket from Atlassian, etc). The branch name is optional.

Crates

Case 1 → a static library for another Rust program

3. Link from your local hard drive

In this case, it is easier, just specify the location of the crate in your hard drive in the following way:

cargo.toml (for another application)

```
[package]
name = "another_app"
version = "1.2.3"
edition = "2021"

[dependencies]
my_math_lib = {
                path = "<path to crate>"
              }
```

In this context, <path to crate> is a local folder where the crate is located. Relative paths ("../../.....") are accepted as well.

Crates

Case 2 → a dynamic library that another program can use

cargo.toml

```
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021"

[dependencies]
…

[lib]
crate-type = ["cdylib"]
```

The first step is to add a [lib] section and specify that the crate type (cdylib [**C** **dy**namic **Lib**rary])

Crate-type supports other types as well:

- cdylib
- dylib
- rlib
- staticlib
- proc-macro
- …

Crates

Case 2 → a dynamic library that another program can use

We will also need to change the way we write our exported functions:

Rust

```
#[no_mangle]
pub extern "C" fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

1. The “#[no_mangle]” attribute is required as Rust mangles symbols and as such other applications can’t use them.
2. The “extern "C" “ specifier is required so that Rust exports that function in a way a program like C/C++ can understand.

Crates

Case 2 → a dynamic library that another program can use

Upon compiling, a .dll or a .so or a .dylib will be created that export the function add.

That library can be used in a C/C++ program in the following way:

C/C++ program (for Windows)

```
#include <Windows.h>
#include <stdio.h>

typedef int32_t (* FNADD)(int32_t,int32_t);
void main() {
    auto handle = LoadLibraryA("my_math_lib.dll");
    auto add = (FNADD)GetProcAddress(handle,"add");
    printf("%d",add(1,2));
}
```

OBS: *We consider my_math_lib.dll the binary result obtain when running cargo build.*

Crates

Case 3 → a static library that another program can use

cargo.toml

```
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021"

[dependencies]
...

[lib]
crate-type = ["staticlib"]
```

The first step is to add a [lib] section and specify that the crate type (static)

The execution of cargo build will produce a static library file (*.lib) (in our case it will be named ***my_math_lib.lib***)

That library file can further be used in a C/C++ program for linkage (option -l from C/C++ compiler).

Conditional Compilation

Conditional Compilation

In rust, a Conditional compilation means building different branches of code based on some condition (features) that can be activated or not.

The closest similarity in C/C++ is #ifdef or #ifndef compounds.

In Rust a similar functionality is achieved with:

1. cfg proc macro attribute: `#[cfg(…)]`
2. cfg macro: `cfg!(…)`

Conditional compilation works based on features (that can be considered a ***Boolean*** value), that can be:

1. Custom (defined by user). **ALL CUSTOM FEATURES MUST** be defined in cargo.toml file
2. Predefined (e.g. current operation system).

Conditional Compilation

The following steps must be performed when defining custom features:

1. Add a list of features in cargo.toml file into the section ***features***
2. Use that feature (via #[cfg(…)] or cfg!(...)) in your program
3. Either:
   1. enable that feature directly in ***cargo.toml*** (via default key)
      or
   2. use --features <name_1, name_2 , name_n > with cargo command line to enable one or multiple features

OBS: Notice that unlike C/C++ where a feature that enables conditional programming in a program does not have to be defined but can be used with the parameter "–D" from command line , in Rust all feature **MUST** be defined in cargo.toml file in order to be used with cargo command line.

Conditional Compilation

The general format of #[cfg(…)] attribute implies a logical condition (expression) that if evaluated with true will enable the next block from the program in the compilation phase. An expression from #[cfg(<expression>)] can be:

- A simple key="value" expression (where key is usually **feature**). Example: #[cfg(feature="ABC")]
- Another expression that uses not, and or any to create a more complex expression

The compound after the #[cfg(…)] can be:

1. A function/method. #[cfg(…)] fn <name>(…) { … }
2. A block #[cfg(…)] { … }
3. A condition/loop #[cfg(…)] if condition { … }
4. A variable definition #[cfg(…)] let x = …
5. A type/struct definition #[cfg(…)] struct <name> {…}
6. A module #[cfg(…)] pub mod <name> {…}
7. And others …

Conditional Compilation

Let's see a simple example (with code and cargo.toml).

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
METHOD_A = []
METHOD_B = []
```

Rust

```rust
#[cfg(feature = "METHOD_A")]
fn foo() {
    println!("Method A");
}
#[cfg(feature = "METHOD_B")]
fn foo() {
    println!("Method B");
}
fn main() {
    foo();
}
```

C++ (equivalent code)

```cpp
#ifdef METHOD_A
    void foo() {
        printf("Method A");
    }
#endif

#ifdef METHOD_B
    void foo() {
        printf("Method B");
    }
#endif

void main() {
    foo();
}
```

There are two options to run this program:

Conditional Compilation

Let's see a simple example (with code and cargo.toml).

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
METHOD_A = []
METHOD_B = []
```

Rust

```
#[cfg(feature = "METHOD_A")]
fn foo() {
    println!("Method A");
}
#[cfg(feature = "METHOD_B")]
fn foo() {
    println!("Method B");
}
fn main() {
    foo();
}
```

C++ (equivalent code)

```
#ifdef METHOD_A
    void foo() {
        printf("Method A");
    }
#endif

#ifdef METHOD_B
    void foo() {
        printf("Method B");
    }
#endif

void main() {
    foo();
}
```

There are two options to run this program:

1. Run "cargo run --features METHOD_A"

Conditional Compilation

Let's see a simple example (with code and cargo.toml).

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
default = ["METHOD_A"]
METHOD_A = []
METHOD_B = []
```

Rust

```rust
#[cfg(feature = "METHOD_A")]
fn foo() {
    println!("Method A");
}
#[cfg(feature = "METHOD_B")]
fn foo() {
    println!("Method B");
}
fn main() {
    foo();
}
```

C++ (equivalent code)

```cpp
#ifdef METHOD_A
    void foo() {
        printf("Method A");
    }
#endif

#ifdef METHOD_B
    void foo() {
        printf("Method B");
    }
#endif

void main() {
    foo();
}
```

There are two options to run this program:

1. Run "cargo run --features METHOD_A"
2. Set a default field with a list of features to be automatically enabled upon execution and run the program with a "cargo run"

Conditional Compilation

Let's see a simple example (with code and cargo.toml).

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
default = ["METHOD_A"]
METHOD_A = []
METHOD_B = []
```

Rust

```rust
#[cfg(feature = "METHOD_A")]
fn foo() {
    println!("Method A");
}
#[cfg(feature = "METHOD_B")]
fn foo() {
    println!("Method B");
}
fn main() {
    foo();
}
```

Error

```
error[E0428]: the name `foo` is defined multiple times
 --> src\main.rs:6:1
  |
2 | fn foo() {
  | -------- previous definition of the value `foo` here
...
6 | fn foo() {
  | ^^^^^^^^ `foo` redefined here
```

Those two methods, if combine are NOT exclusive (meaning that if we run: "cargo run --features METHOD_B" then both METHOD_A and METHOD_B will be **enabled** and the code will not compile as there are two foo methods).

Conditional Compilation

Let's see a simple example (with code and cargo.toml).

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
default = ["METHOD_A"]
METHOD_A = []
METHOD_B = []
```

Rust

```
#[cfg(feature = "METHOD_A")]
fn foo() {
    println!("Method A");
}
#[cfg(feature = "METHOD_B")]
fn foo() {
    println!("Method B");
}
fn main() {
    foo();
}
```

Output
Method B

The solution in this case is to use --no-default-features parameter in the cargo command line followed by the enablement of another feature via --features parameter:

cargo run --no-default-features --features METHOD_B

Conditional Compilation

But what if we don't want to define two separate features, but instead we want to define one and if not set have another action/function defined that we can use (something like #ifdef … #else … #endif from C/C++). To do this we can negate the cfg attribute in the following way: #[cfg(not(…))]

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
METHOD_A = []
```

Rust

```rust
#[cfg(feature = "METHOD_A")]
fn foo() {
    println!("Method A");
}

#[cfg(not(feature = "METHOD_A"))]
fn foo() {
    println!("Method B");
}

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

C++ (equivalent code)

```cpp
#ifdef METHOD_A
    void foo() {
        printf("Method A");
    }
#else
    void foo() {
        printf("Method B");
    }
#endif

void main() {
    foo();
}
```

Conditional Compilation

Similarly, any and all can also be combined to check is one feature has been set up or if all features have been set up. The general format is:

- ANY → #[cfg(**any**(feature="feature$_1$" , feature="feature$_2$" , … feature="feature$_n$"))]
- ALL → #[cfg(**all**(feature="feature$_1$" , feature="feature$_2$" , … feature="feature$_n$"))]
- NOT in various combinations such as → #[cfg(**not**(all(feature="feature$_1$" … feature="feature$_n$")))]

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
FEAT_A = []
FEAT_B = []
```

Rust

```rust
#[cfg(any(feature = "FEAT_A", feature = "FEAT_B"))]
fn one_of_A_or_B() {
    println!("One of FEAT_A or FEAT_B");
}
#[cfg(all(feature = "FEAT_A", feature = "FEAT_B"))]
fn both_of_A_or_B() {
    println!("Both FEAT_A and FEAT_B");
}
#[cfg(not(all(feature = "FEAT_A", feature = "FEAT_B")))]
fn not_both_of_A_or_B() {
    println!("Not both FEAT_A and FEAT_B");
}
```

Conditional Compilation

In fact, any and all and not can be used to create a complex conditional compilation expression. In the next example we condition the existence of method "foo" with the following expression: `#[cfg(not(any(all(feature="FEAT_A", feature="FEAT_B"), not(feature="FEAT_C"))))]`

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
default = ["FEAT_C"]
FEAT_A = []
FEAT_B = []
FEAT_C = []
```

Rust

```rust
#[cfg(not(any(all(feature="FEAT_A", feature="FEAT_B"), not(feature="FEAT_C"))))]
fn foo() {
    println!("foo");
}

fn main() {
    foo()
}
```

Output

```
foo
```

Conditional Compilation

Each one of the features from [features] section can have dependencies (e.g. a feature might pe dependent on another one). The list of dependencies is described as the value of a specific feature as follows:

- `feature = ["dependency`$_1$`", "dependency`$_2$`",… "dependency`$_n$`"]`

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
default = ["FEAT_A"]
FEAT_A = ["FEAT_B" , "FEAT_C"]
FEAT_B = []
FEAT_C = []
```

Rust

```
#[cfg(all(feature = "FEAT_B", feature = "FEAT_C"))]
fn foo() {
    println!("foo");
}

fn main() {
    foo()
}
```

Output

```
foo
```

Conditional Compilation

Each one of the features from [features] section can have dependencies (e.g. a feature might pe dependent on an
of a specific feature as foll

- `feature = ["depend`

Notice that for the functon foo to be compiled, both **FEAT_B** and **FEAT_C** have to be set up. By default, **FEAT_A** is setup up in the key *default*. And as **FEAT_A** dependencies include **FEAT_B** and **FEAT_C**, then both **FEAT_B** and **FEAT_C** are being set up the moment FEAT_A is enabled.

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
default = ["FEAT_A"]
FEAT_A = ["FEAT_B" , "FEAT_C"]
FEAT_B = []
FEAT_C = []
```

Rust

```rust
#[cfg(all(feature = "FEAT_B", feature = "FEAT_C"))]
fn foo() {
    println!("foo");
}

fn main() {
    foo()
}
```

Output

```
foo
```

Conditional Compilation

Features can also be used to conditionally compile dependencies.

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[dependencies]
ABC = { version = "0.1.2", optional = true }

[features]
FEAT_A = ["dep:ABC"]
```

Conditional Compilation

Features can also be used to conditionally compile dependencies.

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[dependencies]
ABC = { version = "0.1.2", optional = true }

[features]
FEAT_A = ["dep:ABC"]
```

Adding the "**optional = true**" implies that ABC module will not be compiled by default.

By using **"dep:"** prefix, you make a dependency like module **ABC** being compiled upon building.

Conditional Compilation

At the same time, when using a dependency, one can enable features from that dependency:

cargo.toml

```
[dependencies]
ABC = { version = "0.1.2", default = ["FEAT_X"] }
```

This states that ABC module will be compiled with **FEAT_X** from that module enabled

It is also possible to disable the default features setup from a library (for cases where we want to compile a dependency with a specific setup via "default-features" key).

cargo.toml

```
[dependencies]
ABC = { version = "0.1.2", default-feature = false, default = ["FEAT_X"] }
```

This means that for library ABC we will disable all features defined in the default key, and enable **FEAT_X** and its de dependencies upon building.

Conditional Compilation

You can also define a features that forces some features from dependencies to be enabled upon compilation:

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[dependencies]
ABC = { version = "0.1.2", default-feature = false }

[features]
ABC_WITH_XY = ["ABC/FEAT_X", "ABC/FEAT_Y"]
ABC_WITH_DEFAULT = ["ABC/default"]
```

Conditional Compilation

You can also define a features that forces some features from dependencies to be enabled upon compilation:

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[dependencies]
ABC = { version = "0.1.2", default-feature = false }

[features]
ABC_WITH_XY = ["ABC/FEAT_X", "ABC/FEAT_Y"]
ABC_WITH_DEFAULT = ["ABC/default"]
```

This states that ABC module will be compiled without the default features enabled.

Conditional Compilation

You can also define a features that forces some features from dependencies to be enabled upon compilation:

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[depe
ABC =                                ture = false }

[features]
ABC_WITH_XY = ["ABC/FEAT_X", "ABC/FEAT_Y"]
ABC_WITH_DEFAULT = ["ABC/default"]
```

Notice the format for such dependencies:
<crate>/<feature>

This states that if feature ABC_WITH_XY is enabled, then **ABC** crate will be built with **FEAT_X** and **FEAT_Y** from **ABC** crate enabled.

Conditional Compilation

You can also define a features that forces some features from dependencies to be enabled upon compilation:

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[dependencies]
ABC = { version = "0.1.2", default-feature = false }

[features]
ABC_WITH_XY = ["ABC/FEAT_X", "ABC/FEAT_
ABC_WITH_DEFAULT = ["ABC/default"]
```

This states that if feature ABC_WITH_DEFAULT is enabled, then **ABC** crate will be built with its default features enabled (as they are described in ABC crate [features] section for key default).

Conditional Compilation

Besides features, there are a couple of attributes (that have values – not Boolean values like features) that are already set up by Rust compiler. They can be used in the following way:

#[cfg(<attribute$_1$>=“value” , <attribute$_2$>=“value” , … <attribute$_n$>=“value”)]

Attribute	Values
target_arch	x86, x86_64, mips, powerpc, powerpc64, arm, aarch64
target_feature	avx, avx2, crt-static, rdrand, sse, sse2, sse4.1
target_os	windows, macos, ios, linux, android, freebsd, dragonfly, openbsd, netbsd
target_family	unix, windows, wasm
target_env	gnu, msvc, musl, sgx
target_endian	big, little
target_pointer_width	16, 32, 64
target_vendor	apple, fortanix, pc, unknwon
target_has_atomic	8, 16, 32, 64, 128, ptr

Conditional Compilation

Let's see an example where we use target_os to set up the name of the current operating system.

Rust

```
#[cfg(target_os = "windows")]
fn get_os_name()->&'static str {
    "Windows"
}
#[cfg(target_os = "linux")]
fn get_os_name()->&'static str {
    "Linux"
}
fn main() {
    println!("Current OS is {}",get_os_name());
}
```

Output (possible)
Current OS is Windows

Conditional Compilation

Besides custom features and attributes, there are some predefined / preset features that can be used:

Predefine feature	Scope
debug_assertions	Enabled when compiling in debug mode (without optimizations)
unix	Equivalent with target_family = "unix"
windows	Equivalent with target_family = “windows"
test	Usually used with modules to indicate that a module is used for unit testing
proc_macro	If a crate is being compiled with proc_macro flag

Conditional Compilation

Let's see an example that use debug_assertions to enable a method only in debug mode:

Rust

```rust
#[cfg(debug_assertions)]
fn foo() {
    println!("foo");
}

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

A) Normal execution (debug mode)

Output (normal execution)

```
foo
```

B) Compiled in release mode (**cargo run -r**)

Error

```
error[E0425]: cannot find function `foo` in this scope
 --> src\main.rs:6:5
  |
6 |     foo();
  |     ^^^ not found in this scope
```

Conditional Compilation

Additionally, Rust provide another macro called cfg_attr, that can be used to add other attributes based on a condition. The general format is:

#[cfg_attr(condition, Attribute_1, Attribute_2, … Attribute_n)]

Several such conditions can be applied over the same block.

cargo.toml

```toml
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
MAKE_INLINE = []
TEST_MODE = []
```

Rust

```rust
#[cfg_attr(feature="MAKE_INLINE",inline(always))]
#[cfg_attr(feature="TEST_MODE",test)]
fn foo() {
    println!("foo");
}
fn main() {
    foo();
}
```

Conditional Compilation

Additionally, Rust ... add other attributes b...

#[cfg_attr ...

Several such cond...

Assuming both **MAKE_INLINE** and **TEST_MODE** features are **enabled**, the foo method will be translated into the following form (before compilation):

```
#[inline(always)]
#[test]
fn foo() { println!("foo"); }
```

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
MAKE_INLINE = []
TEST_MODE = []
```

Rust

```
#[cfg_attr(feature="MAKE_INLINE",inline(always))]
#[cfg_attr(feature="TEST_MODE",test)]
fn foo() {
    println!("foo");
}
fn main() {
    foo();
}
```

Conditional Compilation

This technique is particularly useful for modules (e.g. if you want to have different modules with the same name but different file names for every OS specific build.

Rust

```rust
#[cfg_attr(target_os = "windows", path = "windows.rs")]
#[cfg_attr(target_os = "linux", path = "linux.rs")]
#[cfg_attr(target_os = "macos", path = "macos.rs")]
mod OS_methods;
```

For example, in the previous context, the module ***OS_methods*** will be defined in 3 different files (windows.rs, linux.rs and macos.rs). The compiler will choose what file to compile based on the **path** attribute of the module ***OS_methods***. The **path** attribute is selected based on the operating system (meaning that if we run on Windows, the previous code will look like the following one:

Rust

```rust
#[path = "windows.rs"]
mod OS_methods;
```

Conditional Compilation

"cfg" can also be used in cargo.toml (as a way to specify additional properties depending on the selected configuration). The most common one is to add additional dependencies based on the operating system.

Cargo.toml

```
[target.'cfg(unix)'.dependencies]
Linux_library = "<some version>"

[target.'cfg(windows)'.dependencies]
Windows_library = "<some version>"
```

In this case we specify that different libraries need to be linked depending on the target operating system. The general format for dependencies is:

[target.'**<cfg rule>**'.dependencies]

Conditional Compilation

Because of this format, we can write even more complex conditions for conditional compilation. Let's analyze the following example:

Cargo.toml

```
[target.'cfg(all(unix, target_pointer_width = "64"))'.dependencies]
Pointer_optimizer_for_unix = "<some version>"
```

This translated into the following compilation logic:

- If the target operating system is unix
  and
- If the pointer size for that operating system is stored on 64 bits
  then
- Add e dependency in the current project to "Pointer_optimizer_for_unix" crate.

Conditional Compilation

The cfg! macro works similar with #[cfg(...)], with the difference is that it returns **true** or **false**. It is important to notice that even if the code is not enabled, if the cfg! macro is used with an **if** compound , the block of that if must compile.

Rust

```rust
fn foo() {
    let x = if cfg!(feature = "METHOD_B") { 1 } else { 2 };
    println!("x = {}", x);
}

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

Output (possible).

```
x = 2
```

In this case, cfg!(feature = "METHOD_B") will translate to either **true** or **false** , and as such **x** will be 1 or 2.

Building scripts (build.rs)

Building scripts

Sometimes, before building a program, additional pre-processing is required to:

- Generate new files (e.g. from a template)
- Create a wrapper (e.g. for a file written in a different language to be easily accessible from Rust)
- Link resources with rust code
- Modify existing files
- Download or upload some components (to make sure that they are using the latest versions)
- etc

Building scripts

For these tasks Rust provides an internal mechanism based on another rust file called **build.rs**

- src
  - ***main.rs***
  - *..other files*
- ***build.rs***
- ***Cargo.toml***

If the compiler finds a build.rs file in the root of the project, it will compile that file into an executable, then run that executable and then recompile the rest of the project. If build.rs exists it will **always** be executed before compiling the project.

Building scripts

For these tasks Rust provides an internal mechanism based on another rust file called **build.rs**

- src
  - *main.rs*
  - *..other files*
- ***build.rs***
- *Cargo.toml*

Rust

```rust
use std::fs;

fn main()
{
    println!("cargo:warning=Running code generation...");

    // code that performs pre-build actions
}
```

Building scripts

For these tasks Rust provides an internal mechanism based on another rust file called **build.rs**

- src
  - *main.rs*
  - *..other files*
- *build.rs*
- *Cargo.toml*

Cargo.toml

```
[build-dependencies]
Dependenct-1 = { version = "1.0", features = [“…"] }
Dependency-2 = "1.0.145"
Dependency-3 = "0.9.8"
```

Optionally, in the project cargo.toml you can add a new section called **[build-dependencies]** where you can add various dependencies that are needed by the build script.

Building scripts

The ***build.rs*** file can communicate with the compiler by using the println! , eprintln! , print! or eprint! macros with some special strings that the compiler recognizes.

The format is “**cargo**::*<command>*=***<value>***”

A detailed list of all commands that the compiler recognizes can be found on :
https://doc.rust-lang.org/cargo/reference/build-scripts.html

Building scripts

Some of the most common commands:

Command	Meaning
cargo::warning=MESSAGE	Prints a warning in the compiler CLI (warning is the MESSAGE parameter)
cargo::error=MESSAGE	Prints an error in the compiler CLI (error is the MESSAGE parameter)
cargo::rustc-flags=FLAGS	Tells the compiler to use certain flags for the projects
cargo::rustc-link-arg=FLAG	Tells the compiler to use a specific set of flags in the linkers
cargo::rustc-link-arg-cdylib=FLAG cargo::rustc-link-arg-tests=FLAG cargo::rustc-link-arg-bins=FLAG	Tells the compiler to use certain flags for specific parts of the project (e.g. libraries, tests, binaries, etc)
cargo::rerun-if-changed=PATH	This tells cargo to rerun the build script if the file (PATH) last modified time has changed. You can use this to disable the normal execution of build.rs by writing **println!("cargo::rerun-if-changed=build.rs")**
cargo::rerun-if-env-changed=VAR	This tells cargo to rerun if a specific environment variable has changed.

Building scripts

One common scenario is that you might need to manually run the build script (e.g. for example if the build script uses an external resource – like the content of an URL that it can not know if has changed or not).

The most common solution is to use a feature defined in **cargo.toml** and check that function directly in the **build.rs** code.

Then, simply run `cargo build –features <name_of_the_feature>` to enable the build script.

Building scripts

Example to use a feature to decide when to generate something:

cargo.toml

```toml
[package]
name = "…"
version = "0.1.0"
edition = "2021"

[dependencies]
…

[features]
generate = []
```

Rust (build.rs)

```rust
fn main() {
    if cfg!(feature = "generate") {
        println!("cargo:warning=Running code generation...");
        // code that generates other files
        // or creates wrappers
        // or modifies existing files
    } else {
        println!("cargo:warning=Skipping code generation... ");
    }
}
```

To generate code run: `cargo build --features generate`

Building scripts

If you need a more complex script (formed out of multiple files):

- src
  - ***main.rs***
  - *..other files*
- my_scripts
  - *..other files*
- ***my_scripts.rs***
- ***build.rs***
- ***Cargo.toml***

You can group all of your scripts into a module (in this case 'my_scripts') and create a module file my_scripts.rs near the ***build.rs*** file and use that module as part of the build.rs code.

Building scripts

If you need a more complex script (formed out of multiple files):

- src
  - *main.rs*
  - *..other files*
- my_scripts
  - *..other files*
- *my_scripts.rs*
- *build.rs*
- *Cargo.toml*

Rust

```rust
mod my_scripts;

fn main()
{
    println!("cargo:warning=Running code generation...");

    // code that performs pre-build actions
}
```

We reference my_script in the code of build.rs

```rust
mod my_scripts;
```

Building scripts

If you need a more complex script (formed out of multiple files):

- src
  - *main.rs*
  - *..other files*
- my_scripts
  - *..other files*
- *my_scripts.rs*
- *build.rs*
- *Cargo.toml*

- my_scripts
  - *a.rs*
  - *b.rs*
  - *c.rs*

In a.rs / b.rs and c.rs you can add your larger code.

Rust (my_scripts.rs)

```rust
pub mod a;
pub mod b;
pub mod c;
```

Building scripts

The build.rs script is run in the root of the project. This means that you can use relative path to generate files:

Rust (build.rs)

```rust
use std::fs;


fn main() {
    println!("cargo:warning=Running code generation...");
    // read some data from the src folder
    let content = fs::read_to_string("src\\my_data.json").unwrap();

    // process the content and generate some rust files
    let rust_file_content = ...;

    // write the new rust file in the src folder
    fs::write("src/my_code.rs", rust_file_content).unwrap();
}
```

Building scripts

Any panic that appears while running the build script will be translated as a compiler error and the compilation will stop.

This also means that you can use panic! macro to stop the compilation from the build script with a specific message.

Keep in mind that you can only modify OUT_DIR and subdirectories from the build script if you want cargo to automatically detect changes and update them. You can modify other locations, but cargo will not update anything.

Tests

Tests

Working with library implies having some sort of unit test to run before releasing it. While in other languages this is done via a 3rd party application, Rust ecosystem can do this within the same integration (via cargo).

Let’s consider the following problem → we need to export a function that sums up two u8 values, but returns an Option for cases where an integer overflow appears (e.g. adding 200+200 will result in an integer overflow for u8 scenario).

Tests

Let's see how we can do this.

Step 1 → lets create a new library via cargo by running:

cargo new --lib addlibrary

Step 2 → lets add an add method:

Rust (lib.rs)

```rust
pub fn add(x: u8, y: u8) -> Option<u8> {
    let result = (x as u32) + (y as u32);
    if result > 255 {
        return None;
    }
    return Some(result as u8);
}
```

Tests

Let’s see how we can do this.

Step 3 → add some methods to test if function “add” works properly.

Rust (lib.rs)

```rust
pub fn add(x: u8, y: u8) -> Option<u8> {…}
#[test]
fn check_add() {
    assert!(add(1,2)==Some(3));
    assert_eq!(add(100,155),Some(255));
    assert_ne!(add(0,0),Some(1));
}
#[test]
fn check_overflow() {
    assert_eq!(add(200,200),None);
    assert_ne!(add(100,100),None);
}
```

Tests

Let’s see how we can do this.

Step 3 → add some methods to test if function “add” works properly.

Rust (lib.rs)

```
pub fn add(x: u8, y: u8) -> Option<u8> {…}
#[test]
fn check_add() {
    assert!(add(
    assert_eq
    assert_ne!(a
}
#[test]
fn check_overflow() {
    assert_eq!(add(200,200),None);
    assert_ne!(add(100,100),None);
}
```

Notice the #[test] attribute. This indicates that those function are meant for unit testing and are not part of the normal development of this library.

Tests

Let's see how we can do this.

Step 3 → add some methods to test if function "add" works properly.

Rust (lib.rs)

```
pub fn add(x: u8, y: u8) -> Option<u8> {…}
#[test]
fn check_add() {
    assert!(add(1,2)==Some(3));
    assert_eq!(add(100,155),Some(255));
    assert_ne!(add(0,0),Some(1));
}
#[test]
fn check_overflow() {
    assert_eq!(add(200,200),None);
    assert_ne!(add(100,100),None);
}
```

It is also worth mentioning that any panic encountered in this functions will translate in a fail for that tested function.

Notice various macros (assert!, assert_eq!, assert_ne!) that can be used to check if a function works as expected.

Tests

Let’s see how we can do this.

Step 3 → add some methods to test if function “add” works properly.

Rust (lib.rs)

```
pub fn add(x: u8, y: u8) -> Option<u8> {…}

#[test]
fn check_add() {…}
#[test]
fn check_overflow() {…}

#[test]
#[should_panic]
fn check_panic_test() {
    let x = add(200,200).unwrap();
}
```

If you need to test if a function panics, there is an attribute that can be added #[should_panic] that can be used for this purpose.

Tests

Let’s see how we can do this.

Now that the testing was performed, it can be run in various ways:

- All tests: ➔ cargo test ➔ run all tests

```
running 3 tests
test check_add ... Ok
test check_overflow ... Ok
test check_panic_test - should panic ... Ok

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

- One test: ➔ cargo test <name> ➔ run a test with a name
  Ex: “***cargo test add***” will produce the following result

```
running 1 test
test check_add ... Ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Tests

It is a common practice to group all test in a separate module (often name test). For example, in our case, the code could be organized in the following way:

Rust (lib.rs)

```
pub fn add(x: u8, y: u8) -> Option<u8> {…}
mod tests {
    #[test]
    fn check_add() {…}

    #[test]
    fn check_overflow() {…}

    #[test]
    #[should_panic]
    fn check_panic_test() {…}
}
```

Tests

It is also recommended that modules that are designed for tests to have a special attribute (#[cfg(test)]) that will only compile then if tests are being run.

Rust (lib.rs)

```
pub fn add(x: u8, y: u8) -> Option<u8> {…}

#[cfg(test)]
mod tests {
    #[test]
    fn check_add() {…}

    #[test]
    fn check_overflow() {…}

    #[test]
    #[should_panic]
    fn check_panic_test() {…}
}
```

This attribute says that all methods from test module should be compiled only if tests are required.

Documentation

Documentation

Every project needs documentation. And there are usually two forms of documentation:

- **Function**/**Module** documentation (very useful when using an IDE and you trying to use a method/function)
  Example (for Vector): https://doc.rust-lang.org/std/vec/struct.Vec.html

- **Project** documentation (that usually explain a library / a project / general way of using that project / etc). Usually these types of documentations are presented as books.
  Example: https://doc.rust-lang.org/book/

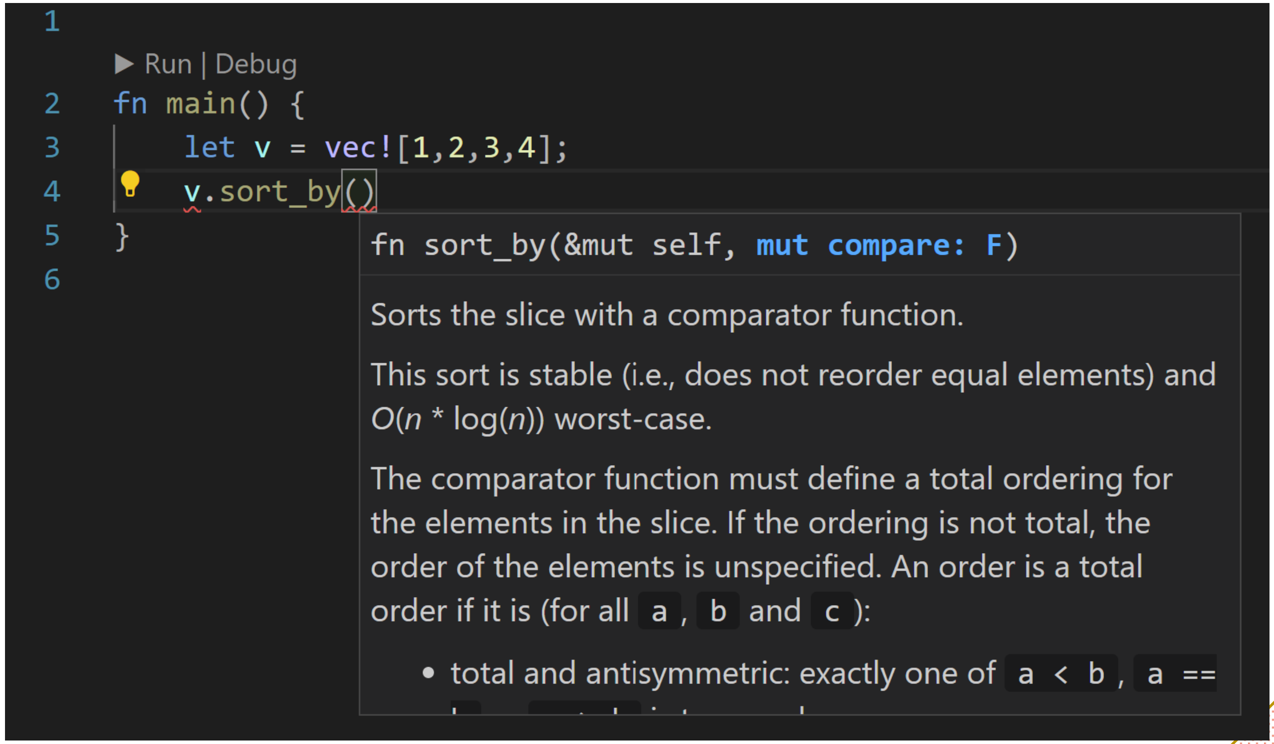
In both cases, Rust uses Markdown to write documentation:
https://en.wikipedia.org/wiki/Markdown

Documentation

Every project needs documentation. And there are usually two forms of documentation:

- **Function**/**Module** documentation (very useful when using an IDE and you trying to use a method/function)

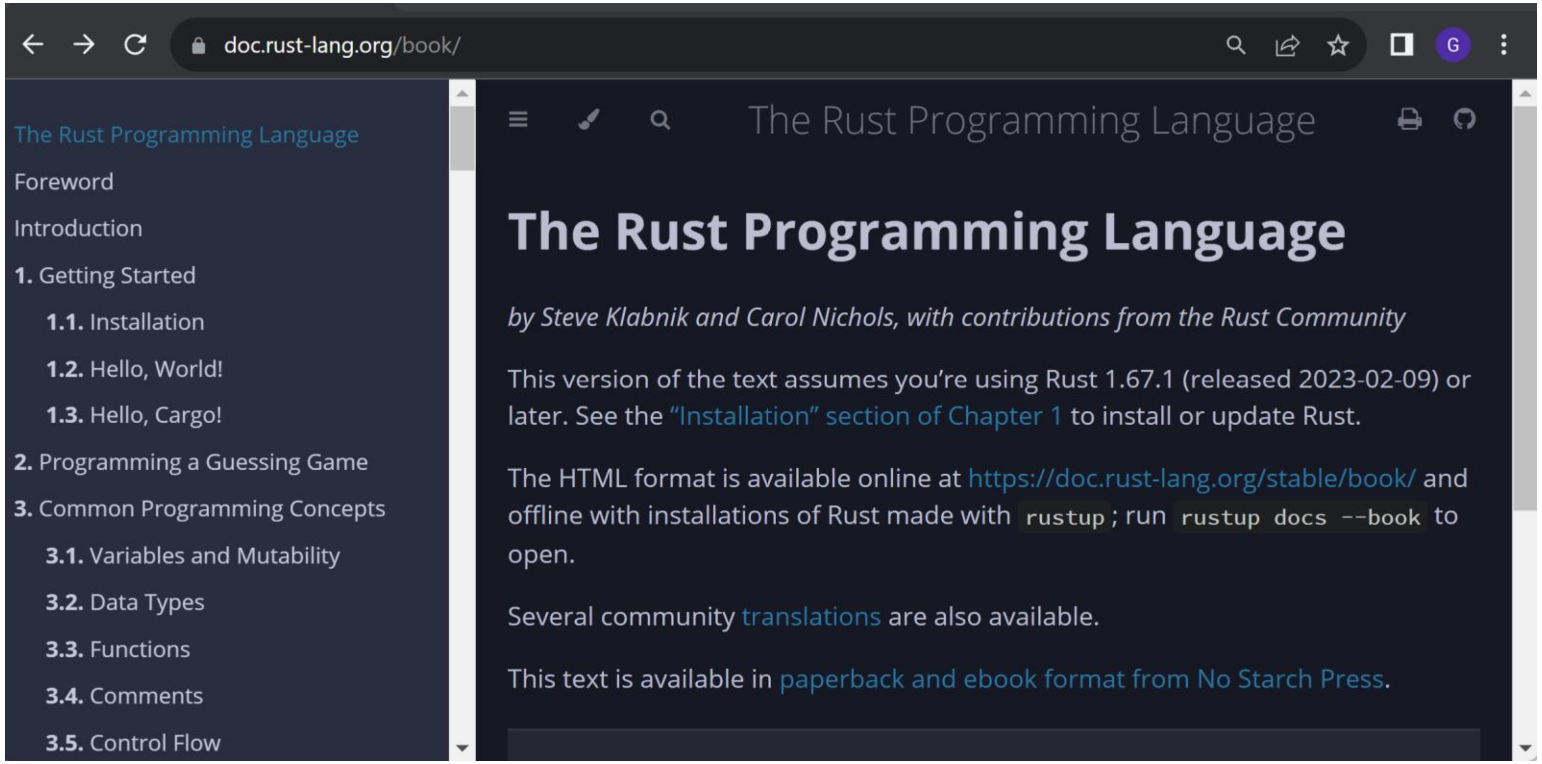
Example of documentation for Vector::sort(…) is being used in an IDE.



Documentation

Every project needs documentation. And there are usually two forms of documentation:

- **Project** documentation (Rust book):

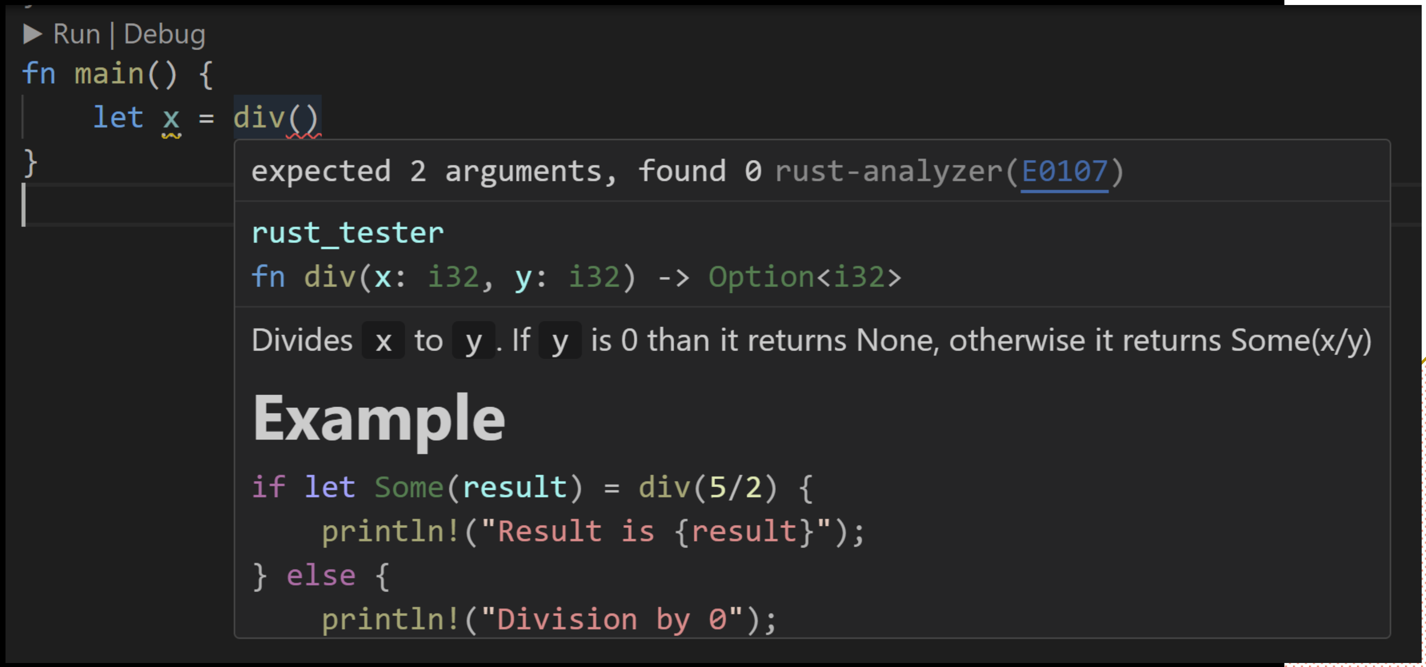


Documentation (Function/Module)

To write a documentation, use /// characters (on multiple lines) to explain (in Markdown format what that function / module is doing).

Rust

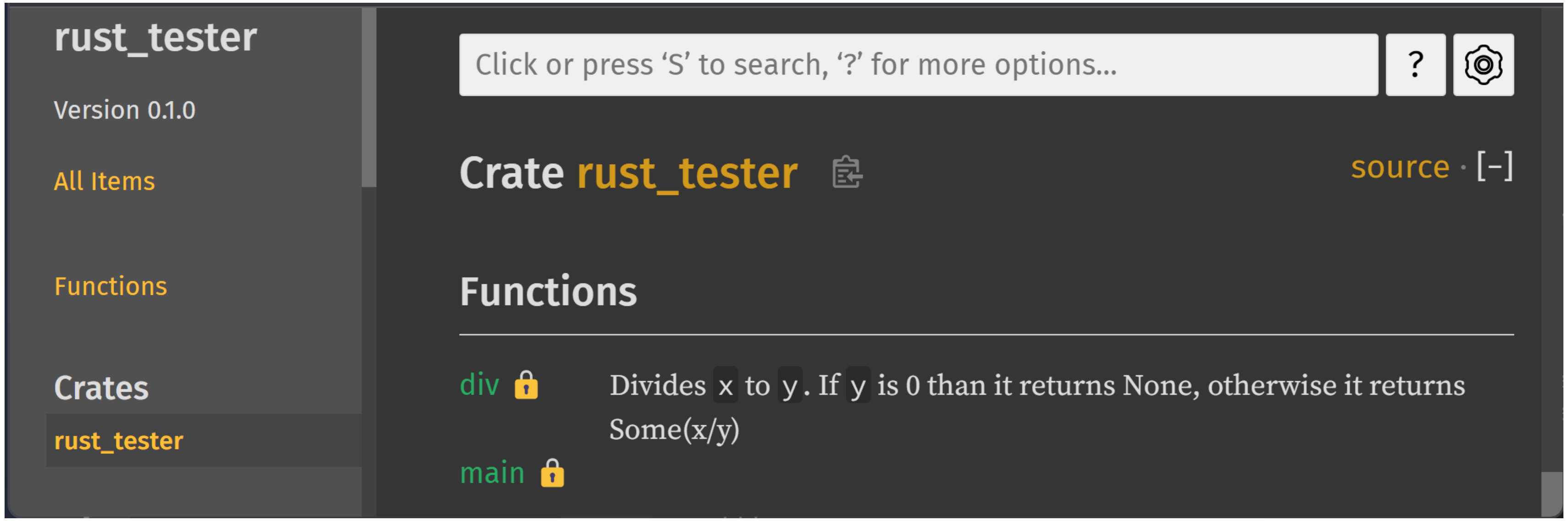
````rust
/// Divides `x` to `y`. If `y` is 0 than it returns None,
/// otherwise it returns Some(x/y)
///
/// # Example
///
/// ```
/// if let Some(result) = div(5/2) {
///     println!("Result is {result}");
/// } else {
///     println!("Division by 0");
/// }
fn div(x: i32, y: i32) -> Option<i32> {
    if y != 0 {
        Some(x / y)
    } else {
        None
    }
}
````



Documentation (Function/Module)

If a documentation is provided, this documentation will be built when a crate is uploaded to crates.io and is being visible on docs.rs (for example for the random library you can find the documentation on https://docs.rs/random/latest/random/).

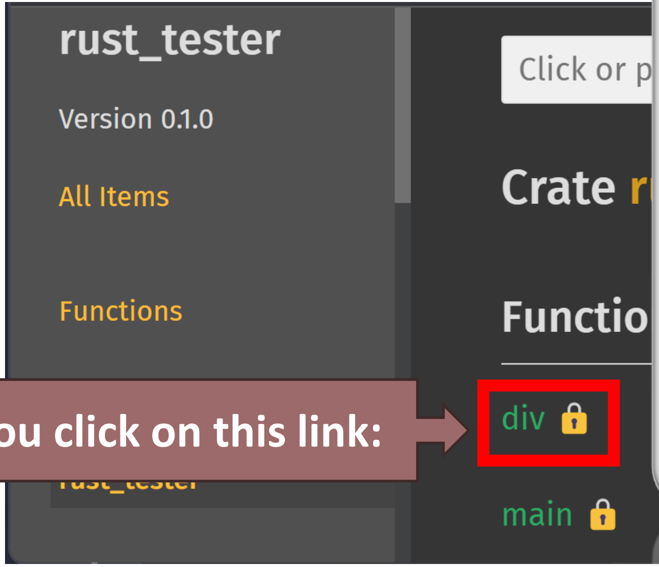
To view the documentation locally, run the following command: `cargo doc --open`
For the previous example this should open the browser to something that looks like this:



Documentation (Function/Module)

If a documentation is provided, this documentation will be built when a crate is uploaded to crates.io and is bein… library you can find the documen…

To view the documentation local… For the previous example this sh… this:



If you click on this link:

Documentation (Function/Module)

The use of /// characters is in fact a syntax sugar for #[doc = "…"] attribute. For example, the following two cases are the same:

Rust

```rust
/// Ads two integer values (x and y)
/// # Example
/// ```
/// let y = add(2,3);
/// ````
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Rust

```rust
#[doc = "Ads two integer values (x and y)"]
#[doc = "# Example"]
#[doc = "```"]
#[doc = "let y = add(2,3);"]
#[doc = "````"]
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Documentation (Function/Module)

The #[doc …] attribute can be used for multiple purposes:

1. Load the documentation from an external document
   `#[doc = include_str!("<path to external documentation file>")]`
2. Set the favicon for the documentation
   `#![doc(html_favicon_url = "<url to favicon>")]`
3. Set the playground URL for your documentation (this allows you documentation to add a <Play> button to your examples so that you can test them:
   `#![doc(html_playground_url = "<url to playground site>")]`
4. Move some part of the documentation in a separate document or not (via **inline** or **no_inline** attributes)
   `#[doc(inline)]` or `#[doc(no_inline)]`

More details on these attributes:
https://doc.rust-lang.org/rustdoc/write-documentation/the-doc-attribute.html

Documentation (Function/Module)

Its also important to distinguish between function/method documentation and documentation for the entire crate or module.

- Use `///` for function / module documentation
- Use `//!` for crate / module documentation (this type of documentation is usually added at the beginning of lib.rs file

Rust

```rust
//! math module can be used for simple math operations
//! such as add or sub

/// math module with 2 methods
pub mod math {
    /// ads two numbers
    pub fn add(x: i32, y: i32) -> i32 { x + y }
    /// substracts two numbers
    pub fn sub(x: i32, y: i32) -> i32 { x - y }
}
fn main() {
    math::add(1,2);
}
```

Crate rust_tester

[−] math module can be used for simple math operations such as add or sub

Modules

math   math module with 2 methods

Functions

main

Documentation (Function/Module)

Another key aspect of Rust documentation is that examples that are added in the documentation can be tested as well. This makes sure that if you change something to a function, you will be notified if the new functionality is not correct according to what the documentation says.

Rust

```
/// Divides two numbers
/// # Example:
///
/// ```
/// use rust_tester_lib::div;
/// assert_eq!(div(10,2),Some(5));
/// assert_eq!(div(5,0),None);
/// ```
pub fn div(x: i32, y: i32)->Option<i32> {
    if y!=0 { Some(x/y) } else { None }
}
```

Under the hood, Rust creates a similar function to the one below (by copying the code provided in the example section (the code between the markers ``` and then it uses the test system to see if it works as expected.

```
#[test]
fn test_function() {
    use rust_tester_lib::div;
    assert_eq!(div(10,2),Some(5));
    assert_eq!(div(5,0),None);
}
```

Documentation (Function/Module)

Another key aspect of Rust documentation is that examples that are added in the documentation can be tested as well. This makes sure that if you change something to a function, you will be notified if the new functionality is not correct according to what the documentation says.

Rust

```rust
/// Divides two numbers
/// # Example:
///
/// ```
/// use rust_tester_lib::div;
/// assert_eq!(div(10,2),Some(5));
/// assert_eq!(div(5,0),None);
/// ```
pub fn div(x: i32, y: i32)->Option<i32> {
    if y!=0 { Some(x/y) } else { None }
}
```

Because Rust build a function out of this code, it needs to reference the div function so that the test function (described in the previous step) can be compiled. **However, this code does not look that good in the documentation !**

```
rust_tester_lib
pub fn div(x: i32, y: i32) -> Option<i32>
```

Divides two numbers

Example:

```rust
use rust_tester_lib::div;
assert_eq!(div(10,2),Some(5));
assert_eq!(div(5,0),None);
```

Documentation (Function/Module)

The solution is to use /// # to hide some lines in the example that should not be presented in the documentation (but still have to be used when running tests).

Rust

```rust
/// Divides two numbers
/// # Example:
///
/// ```
/// # use rust_tester_lib::div;
/// assert_eq!(div(10,2),Some(5));
/// assert_eq!(div(5,0),None);
/// ```
pub fn div(x: i32, y: i32)->Option<i32> {
    if y!=0 { Some(x/y) } else { None }
}
```

```
rust_tester_lib
pub fn div(x: i32, y: i32) -> Option<i32>
```

Divides two numbers

Example:

```
assert_eq!(div(10,2),Some(5));
assert_eq!(div(5,0),None);
```

Notice that the line `use rust_tester_lib::div;` does not appear in the documentation anymore.

Documentation (Function/Module)

You can use /// # to define functions and other variables that you might need to test a function. If you define a function around you test code, Rust will not create another one around your code. This can also allow you to use special operators such as ? in your documentation code.

Rust

```
/// Divides two numbers
/// # Example:
///
/// ```
/// # use rust_tester_lib::div;
/// # fn my_test() {
/// assert_eq!(div(10,2),Some(5));
/// assert_eq!(div(5,0),None);
/// # }
/// ```
pub fn div(x: i32, y: i32)->Option<i32> {
    if y!=0 { Some(x/y) } else { None }
}
```

Approximative result.

```
use rust_tester_lib::div;
#[test]
fn my_test() {
    assert_eq!(div(10,2),Some(5));
    assert_eq!(div(5,0),None);
}
```

Documentation (Function/Module)

However, testing code from example is not without challenges. This is because the code from an example might reflect an error or something that is not recommended to be done. As such, there is a need to add some additional attributes to explain how that particular example should be tested.

This is done my adding **attributes** (keywords) after the ``` characters (several attributes can be added if separated by ,).

You can find more on example test on: https://doc.rust-lang.org/rustdoc/write-documentation/documentation-tests.html

Documentation (Function/Module)

Let's see some examples:

- ```should_panic ➔ this attribute indicates that the code will panic

```
/// ```should_panic
/// assert_eq!(5,4)
/// ```
```

- ```no_run ➔ this indicates that the code should not be run (but should compile). For example, infinite loops or code that needs a lot of time to run should be prefixed with this attribute.

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

- ```compile_fail ➔ this indicates that the code should not compile

```
/// ```compile_fail
/// if while x do y and z
/// ```
```

Documentation (Function/Module)

There are several additional configurations that can be added when describing how documentation should look like such as:

- Ignoring a code
- Specifying an edition to be used for testing
- Specifying some compiling attributes
- The use of ***cfg*** attribute for documentation

More on this topic can be found on:
https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html

Documentation (Book)

If you want to write a project documentation (a book), Rust provides a utility called **mdbook** that helps you here.

To build a book you have to perform the following steps:

1. Install mdbook → use this command: `cargo install mdbook`
2. Create a file in the root folder of your project (named: `book.toml`):

```toml
[book]
authors = ["Gavrilut Dragos"]
language = "en"
multilingual = false
src = "docs"

[build]
build-dir = "html"
```

The most important fields in **book.toml** are:

- "src" → the source folder for your documentation
- "build-dir" → a folder where the resulted documentation (in html format) will be outputted

3. Create a `SUMMARY.md` file in the src file (this will be the entry point)
4. Run `mdbook build` and open the index.html file from the destination folder

Documentation (Book)

Since our course is organized using a mdbook format, let's take a deeper view into it:

```
rust_course_fii
    docs
        SUMMARY.md
        courses.md
        administrative.md
        labs
            lab01
                lab01.md
                menu.png
            labs.md
        courses
            Course-1.pdf
    book.toml
```

TOML

```toml
[book]
authors = ["Gavrilut Dragos"]
language = "en"
multilingual = false
src = "docs"

[build]
build-dir = "html"
```

Documentation (Book)

Since our course is organized using a mdbook format, let's take a deeper view into it:

- rust_course_fii
  - docs
    - ***SUMMARY.md***
    - ***courses.md***
    - ***administrative.md***
    - labs
      - lab01
        - ***lab01.md***
        - ***menu.png***
      - ***labs.md***
    - courses
      - ***Course-1.pdf***
  - ***book.toml***

Markdown

```markdown
# Rust course 2023 - 2024 (FII)

- [Administrative](administrative.md)
- [Courses](courses.md)
- [Labs](labs/labs.md)
  - [Lab 01](labs/lab01/lab01.md)
  - [Lab 02](labs/lab02/lab02.md)
  - [Lab 03](labs/lab03/lab03.md)
  - [Lab 04](labs/lab04/lab04.md)
  - [Lab 05](labs/lab05/lab05.md)
  - [Lab 06](labs/lab06/lab06.md)
…
```

Documentation (Book)

Since our course is organized using a mdbook format, let's take a deeper view into it:

- rust_course_fii
  - docs
    - ***SUMMARY.md***
    - ***courses.md***
    - ***administrative.md***
    - labs
      - lab01
        - ***lab01.md***
        - ***menu.png***
      - ***labs.md***
    - courses
      - ***Course-1.pdf***
  - ***book.toml***

Markdown

```
# Courses

1. Introduction
    - Basic types
    - Variables
    - Operators
    - Functions & Expression statements
    - Basic statement (if, while, loop, ...)
    - Link: [Course 1](courses/Course-1.pdf)

2. Ownership
    - Prerequisite: String type
    - Ownership management
    - Borrowing & References
    - Optimizations
    - Link: [Course 2](courses/Course-2.pdf)
…
```

Documentation (Book)

Since our course is organized using a mdbook format, let's take a deeper view into it:

- rust_course_fii
  - docs
    - *SUMMARY.md*
    - *courses.md*
    - *administrative.md*
    - labs
      - lab01
        - *lab01.md*
        - *menu.png*
      - *labs.md*
    - courses
      - *Course-1.pdf*
  - *book.toml*

Markdown

```
# Setting up the first Rust project

## 1. Prerequisites

On Windows: Install Visual Studio Community
…

Other optional extensions:
- Error lens (id: usernamehw.errorlens)
- crates (id: serayuzgur.crates)

![Menu](menu.png)

## 4. Create a new project

Cargo is a tool that helps building and packaging Rust applications.
The usual workflow is:
- open a terminal
```

Workspaces

Workspaces

Larger project usually imply a smaller set of projects (libraries, utilities, examples, books, etc). To organize all of these, Rust uses a concept of workspace.

A workspace also have a cargo.toml file in its root, but it also has a cargo.toml file for each of the projects from that workspace.

The cargo.toml file from the root contains the references of other projects from the workspace. It also contains default dependencies for all projects. A project can however overwrite them if needed. The main advantage is that if a workspace has multiple projects, you have one place to keep all your common dependencies (and you can change them from that place).

Workspaces

Let’s see an example to better understand how a workspace works.

- Workspace root
  - Project-1
    - src
    - *cargo.toml*
  - Project-2
    - src
    - *cargo.toml*
  - Documentation
    - docs
    - *book.toml*
  - Examples
    - *cargo.toml*
  - ***cargo.toml***

TOML

```toml
[workspace]

members = [
  "Project-1",
  "Project-2",
  "examples“
  …
]

[workspace]
common_dependency_1 = "version"
common_dependency_2 = "version"
common_dependency_3 = "version"
…
```

Workspaces

Let's see an example to better understand how a workspace works.

- Workspace root
  - Project-1
    - src
    - ***cargo.toml***
  - Project-2
    - src
    - ***cargo.toml***
  - Documentation
    - docs
    - ***book.toml***
  - Examples
    - ***cargo.toml***
  - ***cargo.toml***

TOML

```toml
[package]
name = "<name>"
version = "1.0.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
common_dependency_1 = "1.0.7"
common_dependency_2.workspace = true

[features]
default = []
SOME_BOUNDERIES = []
```

In this case we a different version of common_dependency_1 then the one described in the workspace ***cargo.toml*** and common_dependency_2 from the workspace.

Workspaces

Let’s see an example to better understand how a workspace works.

- Workspace root
  - Project-1
    - src
    - *cargo.toml*
  - Project-2
    - src
    - *cargo.toml*
  - Documentation
    - docs
    - *book.toml*
  - Examples
    - ***cargo.toml***
  - *cargo.toml*

TOML

```toml
[package]
name = "examples"
version = "0.0.0"
publish = false
edition = "2021"

[dev-dependencies]
Project-1 = { version = "1.0.0", path = "../project-1" }

[[example]]
name = "example-1"
path = "example-1.rs"

[[example]]
name = "example-2"
path = "example-2.rs"
```

Q & A