RUSTikales Rust for advanced coders

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Closures

Plan for today

1. Recap
2. Closures
3. Iterators

1. Recap

1. Recap

- Procedural Macros are powerful, but difficult to work with

1. Recap

- Procedural Macros are powerful, but difficult to work with
- Because they are written in normal Rust, we can use crates

1. Recap

- Procedural Macros are powerful, but difficult to work with
- Because they are written in normal Rust, we can use crates
- syn and quote are popular crates that help us develop procedural macros
  - Thanks to them, writing powerful procedural macros becomes almost trivial

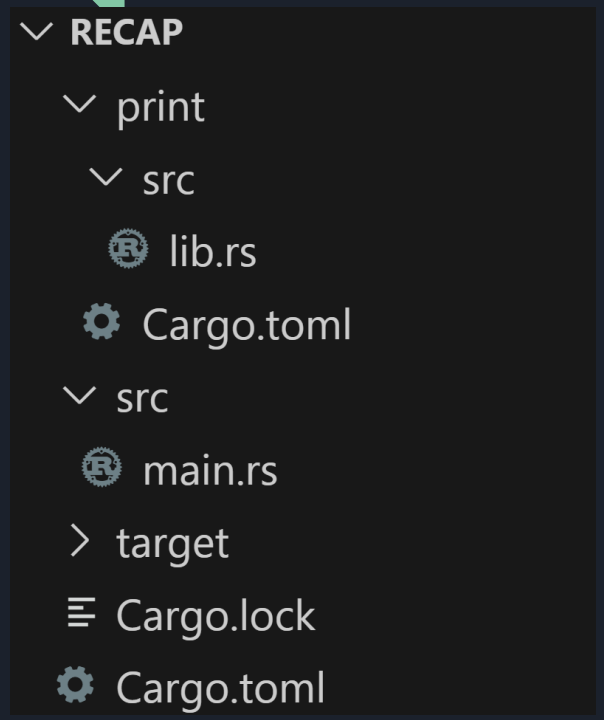
1. Recap

- Procedural Macros are powerful, but difficult to work with
- Because they are written in normal Rust, we can use crates
- syn and quote are popular crates that help us develop procedural macros
  - Thanks to them, writing powerful procedural macros becomes almost trivial
- syn is a Rust parser and converts TokenStreams into convenient data structures

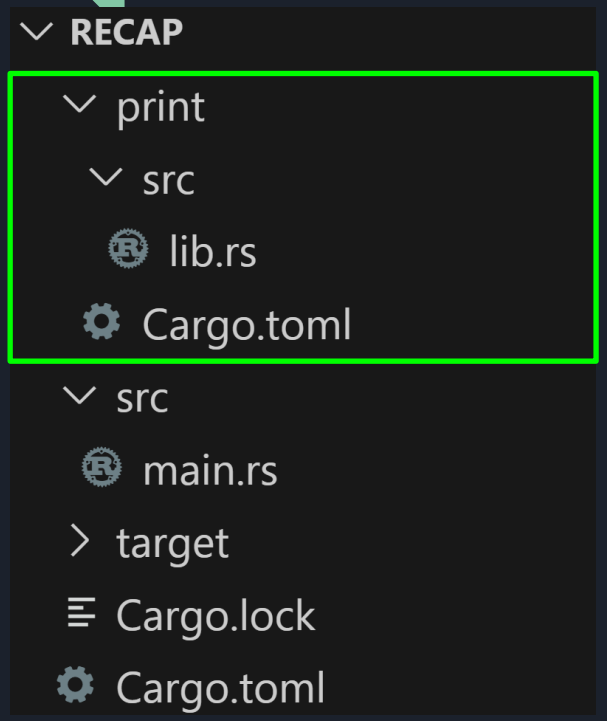
1. Recap

- Procedural Macros are powerful, but difficult to work with
- Because they are written in normal Rust, we can use crates
- syn and quote are popular crates that help us develop procedural macros
  - Thanks to them, writing powerful procedural macros becomes almost trivial
- syn is a Rust parser and converts TokenStreams into convenient data structures
- quote! uses Quasi-Quoting to generate code
  - We write code, the IDE can perform syntax-highlighting and auto-complete, but the macro uses it as data

1. Recap

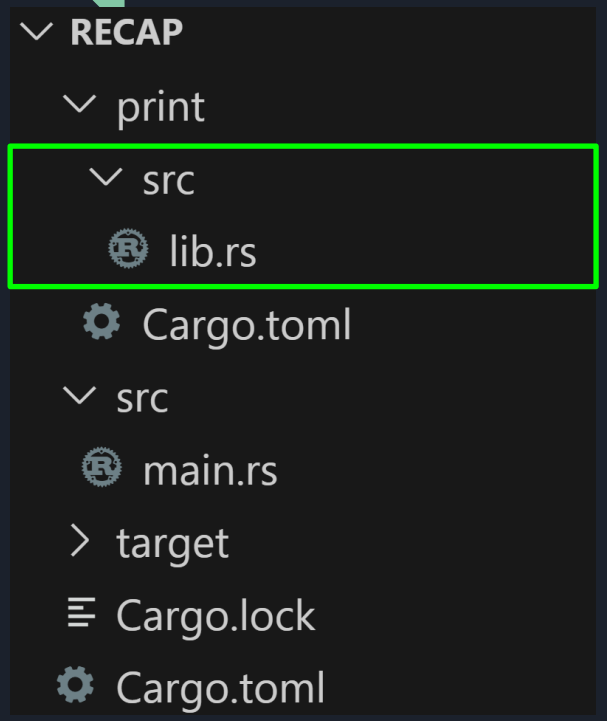

RECAP
print
src
lib.rs
Cargo.toml
src
main.rs
target
Cargo.lock
Cargo.toml

1. Recap



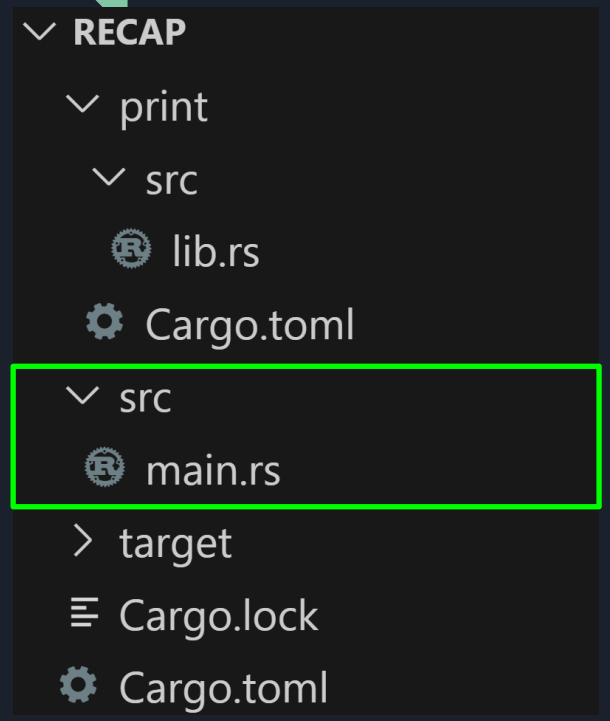
Idea:
→ print is a proc-macro crate

1. Recap



Idea:
→ print is a proc-macro crate
→ In there, we define a proc_macro_attribute which we want to use

1. Recap

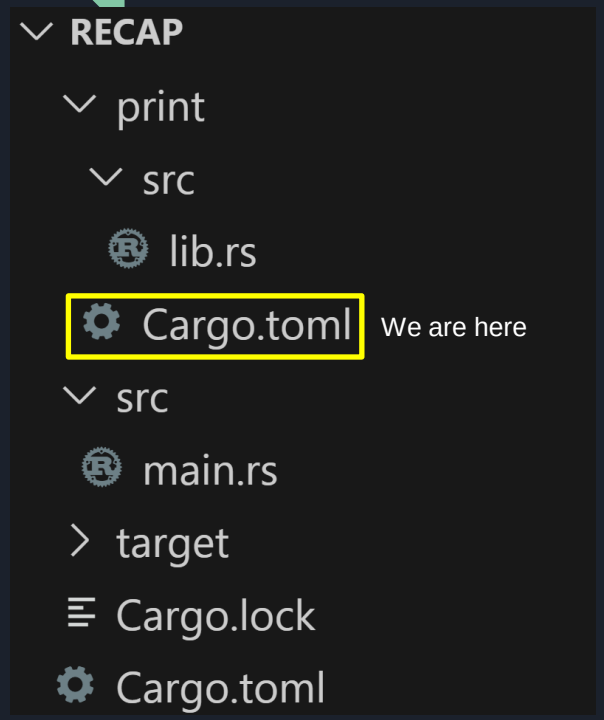


Idea:
→ print is a proc-macro crate
→ In there, we define a proc_macro_attribute which we want to use
→ We later use that macro in our main application

Usage: Annotate function, it then logs when we declare a variable using let

```
use print::log_variable;
#[log_variable]
▶ Run | Debug
fn main() {
```

1. Recap



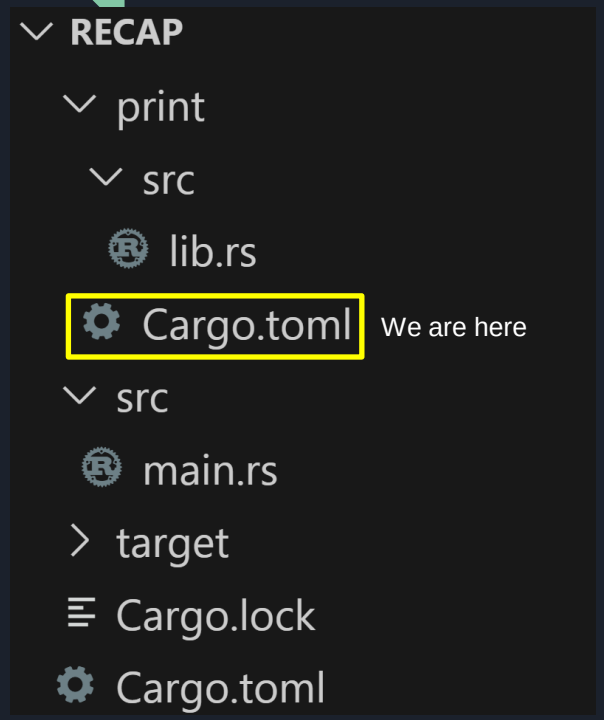

```
[package]
name = "print"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = { version="2.0.67", features=["full"] }
quote = "1.0.36"
```

Mark crate as proc-macro

1. Recap



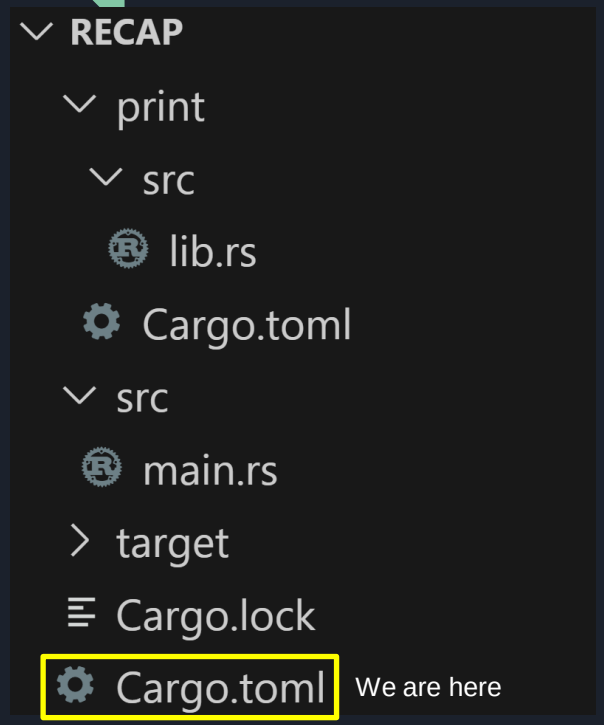

```
[package]
name = "print"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = { version="2.0.67", features=["full"] }
quote = "1.0.36"
```

Use syn and quote like any other crate

1. Recap

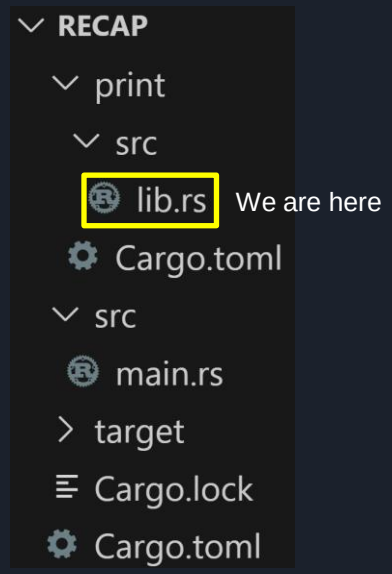



```
[package]
name = "recap"
version = "0.1.0"
edition = "2021"

[dependencies]
print = { path="./print" }
```

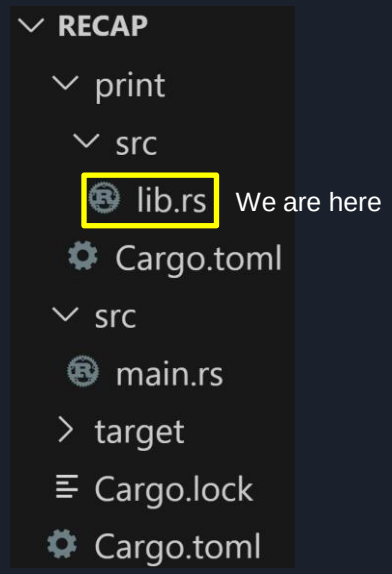
use proc-macro in our main application

1. Recap




```rust
#[proc_macro_attribute]
pub fn log_variable(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let ItemFn {
        attrs: Vec<Attribute>,
        vis: Visibility,
        sig: Signature,
        block: Box<Block>
    } = parse_macro_input!(item);
    let new_block: Block = replace_in_block(&block);
    quote! {
        #(#attrs)*
        #vis #sig #new_block
    }.into()
}
```

1. Recap

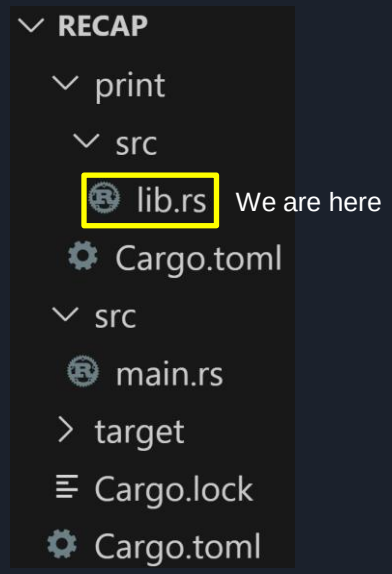



```rust
#[proc_macro_attribute]
pub fn log_variable(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let ItemFn {
        attrs: Vec<Attribute>,
        vis: Visibility,
        sig: Signature,
        block: Box<Block>
    } = parse_macro_input!(item);
    let new_block: Block = replace_in_block(&block);
    quote! {
        #(#attrs)*
        #vis #sig #new_block
    }.into()
}
```

syn allows us to easily parse whatever input we have into any form we want

→ If the given input is not a function, we throw a compiler error

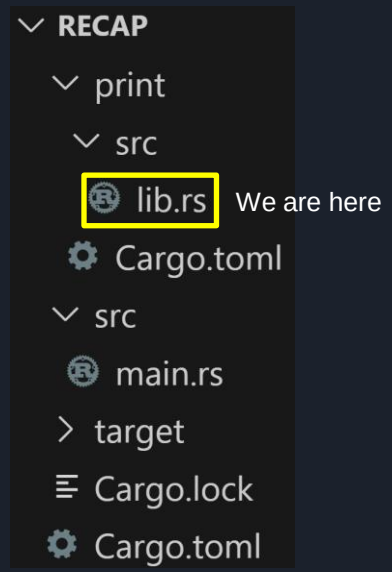
1. Recap




```rust
#[proc_macro_attribute]
pub fn log_variable(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let ItemFn {
        attrs: Vec<Attribute>,
        vis: Visibility,
        sig: Signature,
        block: Box<Block>
    } = parse_macro_input!(item);
    let new_block: Block = replace_in_block(&block);
    quote! {
        #(#attrs)*
        #vis #sig #new_block
    }.into()
}
```

syn transforms the input into a data structure,
which we can now easily work with

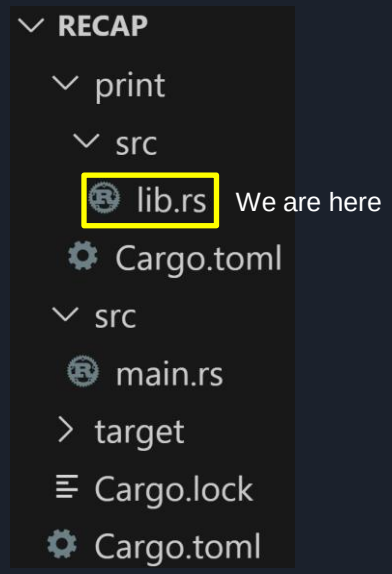
1. Recap




```rust
#[proc_macro_attribute]
pub fn log_variable(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let ItemFn {
        attrs: Vec<Attribute>,
        vis: Visibility,
        sig: Signature,
        block: Box<Block>
    } = parse_macro_input!(item);
    let new_block: Block = replace_in_block(&block);
    quote! {
        #(#attrs)*
        #vis #sig #new_block
    }.into()
}
```

quote! allows us to easily generate code using Quasi-Quoting
→ # interpolates variables defined in the proc-macro and inserts its value into the TokenStream
→ This macro invocation generates a function for us

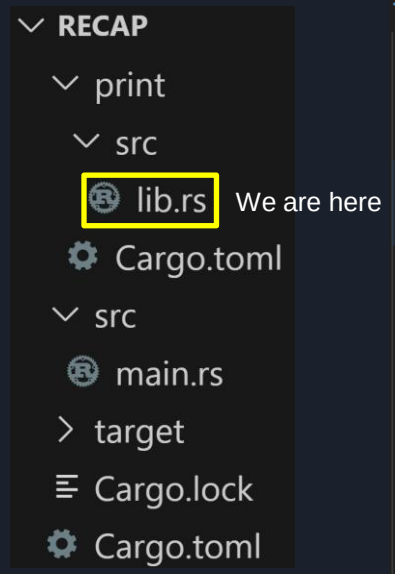
1. Recap




```rust
#[proc_macro_attribute]
pub fn log_variable(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let ItemFn {
        attrs: Vec<Attribute>,
        vis: Visibility,
        sig: Signature,
        block: Box<Block>
    } = parse_macro_input!(item);
    let new_block: Block = replace_in_block(&block);
    quote! {
        #(#attrs)*
        #vis #sig #new_block
    }.into()
}
```

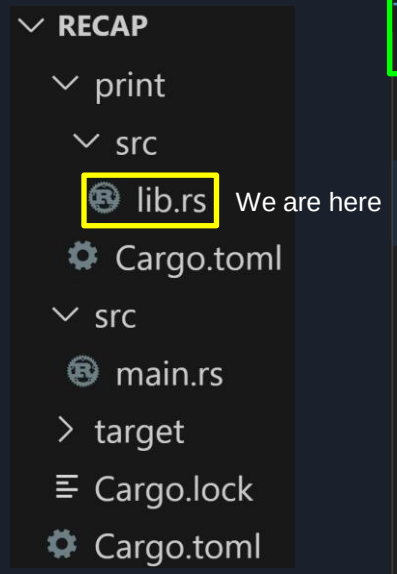
Combining both crates, we can now easily write macros

1. Recap




```rust
use syn::{parse_macro_input, Block, Expr, ExprBlock, ItemFn, Local, LocalInit, Pat, Stmt};
fn replace_in_block(block: &Block) -> Block {
    let mut new_stmts: Vec<Stmt> = vec![];
    for s: &Stmt in &block.stmts {
        match s {
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<E
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {…
            Stmt::Expr(Expr::Block(ExprBlock { attrs: a: &Vec<Attribute>, label: l: &Option<Label>,
                let new_inner: Block = replace_in_block(inner);
                new_stmts.push(Stmt::Expr(Expr::Block(ExprBlock { attrs: a.clone(), label: l.clone()
            }
            _ => new_stmts.push(s.clone()),
        }
    }
    Block { brace_token: block.brace_token, stmts: new_stmts }
} fn replace_in_block
```

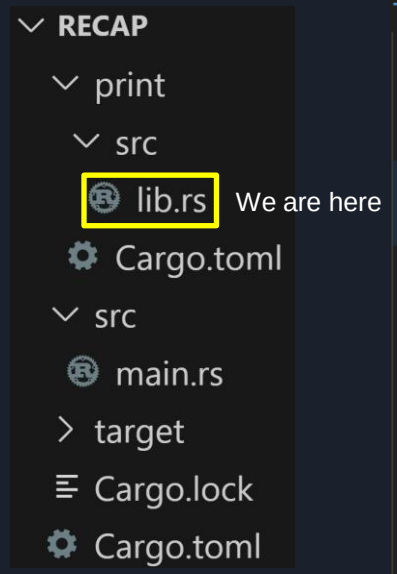
1. Recap




```rust
use syn::{parse_macro_input, Block, Expr, ExprBlock, ItemFn, Local, LocalInit, Pat, Stmt};
fn replace_in_block(block: &Block) -> Block {
    let mut new_stmts: Vec<Stmt> = vec![];
    for s: &Stmt in &block.stmts {
        match s {
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<E
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {…
            Stmt::Expr(Expr::Block(ExprBlock { attrs: a: &Vec<Attribute>, label: l: &Option<Label>,
                let new_inner: Block = replace_in_block(inner);
                new_stmts.push(Stmt::Expr(Expr::Block(ExprBlock { attrs: a.clone(), label: l.clone()
            }
            _ => new_stmts.push(s.clone()),
        }
    }
    Block { brace_token: block.brace_token, stmts: new_stmts }
} fn replace_in_block
```

Idea: Build a new block out of the given input

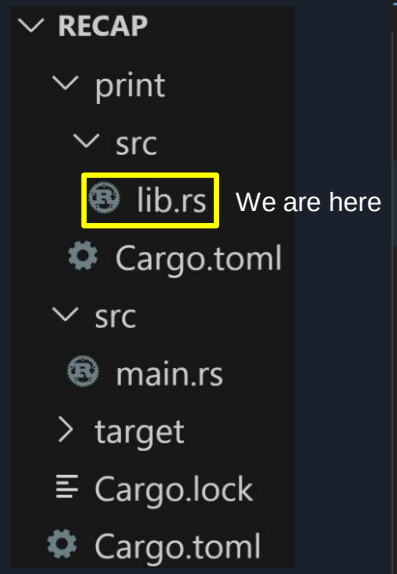
1. Recap




```rust
use syn::{parse_macro_input, Block, Expr, ExprBlock, ItemFn, Local, LocalInit, Pat, Stmt};
fn replace_in_block(block: &Block) -> Block {
    let mut new_stmts: Vec<Stmt> = vec![];
    for s: &Stmt in &block.stmts {
        match s {
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<E
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {…
            Stmt::Expr(Expr::Block(ExprBlock { attrs: a: &Vec<Attribute>, label: l: &Option<Label>,
                let new_inner: Block = replace_in_block(inner);
                new_stmts.push(Stmt::Expr(Expr::Block(ExprBlock { attrs: a.clone(), label: l.clone()
            }
            _ => new_stmts.push(s.clone()),
        }
    }
    Block { brace_token: block.brace_token, stmts: new_stmts }
} fn replace_in_block
```

Iterate over all statements, and replace relevant statements with custom behavior

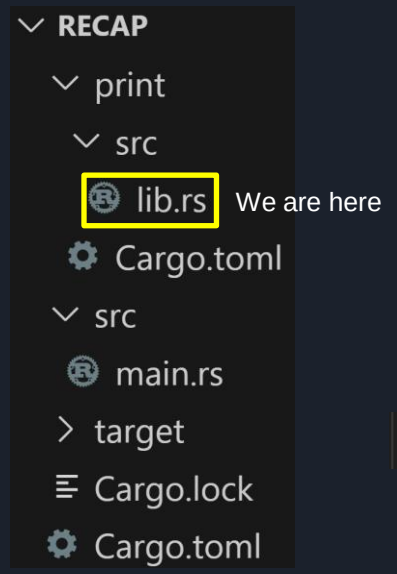
1. Recap




```rust
use syn::{parse_macro_input, Block, Expr, ExprBlock, ItemFn, Local, LocalInit, Pat, Stmt};
fn replace_in_block(block: &Block) -> Block {
    let mut new_stmts: Vec<Stmt> = vec![];
    for s: &Stmt in &block.stmts {
        match s {
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<E
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {…
            Stmt::Expr(Expr::Block(ExprBlock { attrs: a: &Vec<Attribute>, label: l: &Option<Label>,
                let new_inner: Block = replace_in_block(inner);
                new_stmts.push(Stmt::Expr(Expr::Block(ExprBlock { attrs: a.clone(), label: l.clone()
            }
            _ => new_stmts.push(s.clone()),
        }
    }
    Block { brace_token: block.brace_token, stmts: new_stmts }
} fn replace_in_block
```

Recursively replace all blocks we encounter

1. Recap



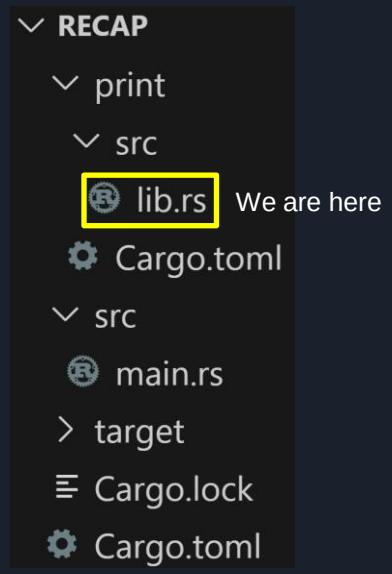

```
Stmt::
_ => n
brace_
```

- Expr(…)
- Item(…)
- Local(…)
- Macro(…)

```
_ => new_stmts.push(s.clone()),
```

Rust is expression-based:
→ Everything is an expression (except let)
→ We don‘t want to replace macros or items

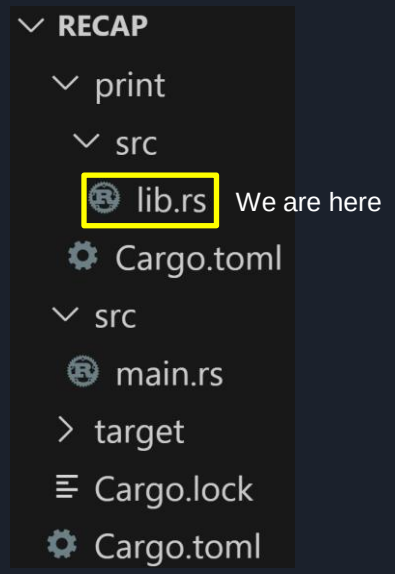
1. Recap




```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. })
    // let <pat> = <expr>;
    // <pat> is a normal identifier like `a` or `b`
    let log: TokenStream = quote! {
        let #var = {
            let _e = #expr;
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            let _print_yellow = |s: &str| -> String { format!("\x1b[93m{s}\x1b[0m") };
            println!("Setting {} to {}",
                _print_green(stringify!(#var)),
                _print_yellow(&format!("{} = {:?}", stringify!(#expr), _e))
            );
            _e
        };
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

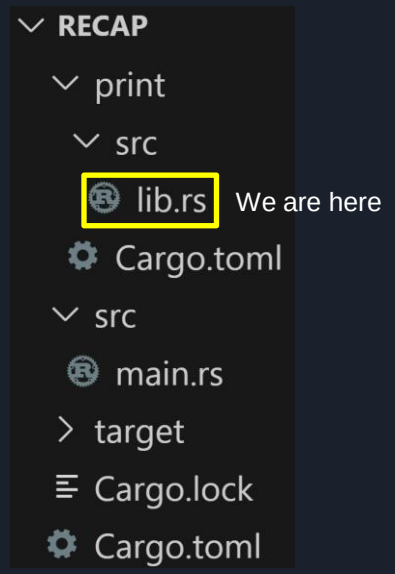
1. Recap

using syn and pattern matching makes developing procedural macros a breeze
→ If our input doesn‘t have this form, we simply don‘t match that case




```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. }) }
    // let <pat> = <expr>;
    // <pat> is a normal identifier like `a` or `b`
    let log: TokenStream = quote! {
        let #var = {
            let _e = #expr;
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            let _print_yellow = |s: &str| -> String { format!("\x1b[93m{s}\x1b[0m") };
            println!("Setting {} to {}",
                _print_green(stringify!(#var)),
                _print_yellow(&format!("{} = {:?}", stringify!(#expr), _e))
            );
            _e
        };
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

1. Recap

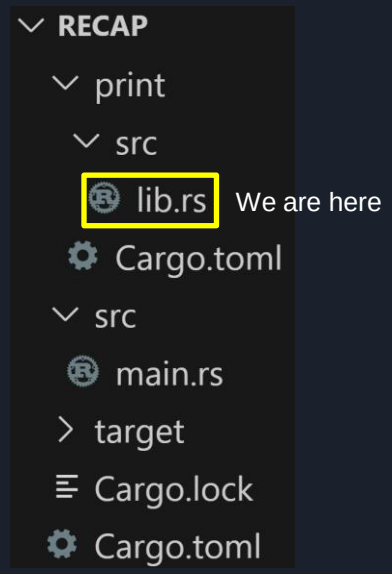



```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. })
    // let <pat> = <expr>;
    // <pat> is a normal identifier like `a` or `b`
    let log: TokenStream = quote! {
        let #var = {
            let _e = #expr;
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            let _print_yellow = |s: &str| -> String { format!("\x1b[93m{s}\x1b[0m") };
            println!("Setting {} to {}",
                _print_green(stringify!(#var)),
                _print_yellow(&format!("{} = {:?}", stringify!(#expr), _e))
            );
            _e
        };
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

Using quote! also really helps, here:

→ Replace original <expr> with a big block, where we print our custom info

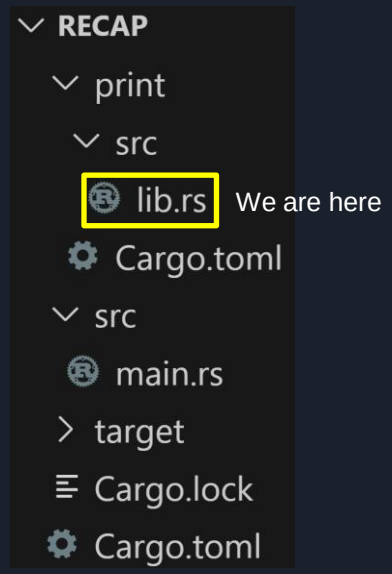
1. Recap




```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. })
    // let <pat> = <expr>;
    // <pat> is a normal identifier like `a` or `b`
    let log: TokenStream = quote! {
        let #var = {
            let _e = #expr;
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            let _print_yellow = |s: &str| -> String { format!("\x1b[93m{s}\x1b[0m") };
            println!("Setting {} to {}",
                _print_green(stringify!(#var)),
                _print_yellow(&format!("{} = {:?}", stringify!(#expr), _e))
            );
            _e
        };
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

We declare local helper variables
→ Will be visible when using cargo expand

1. Recap



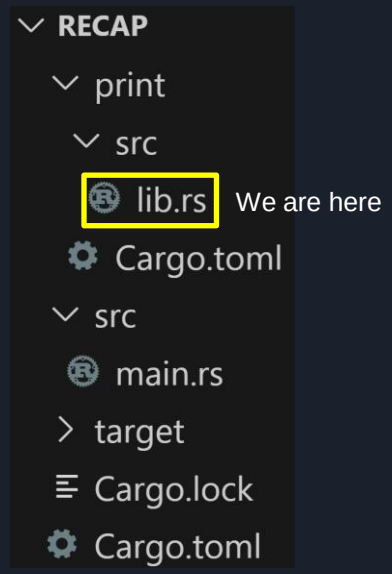

```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. })
    // let <pat> = <expr>;
    // <pat> is a normal identifier like `a` or `b`
    let log: TokenStream = quote! {
        let #var = {
            let _e = #expr;
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            let _print_yellow = |s: &str| -> String { format!("\x1b[93m{s}\x1b[0m") };
            println!("Setting {} to {}",
                _print_green(stringify!(#var)),
                _print_yellow(&format!("{} = {:?}", stringify!(#expr), _e))
            );
            _e
        };
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

Convention: Prefix helper variables with _

→ Disables unused-warnings

→ Prevents name clashes and variable shadowing

1. Recap



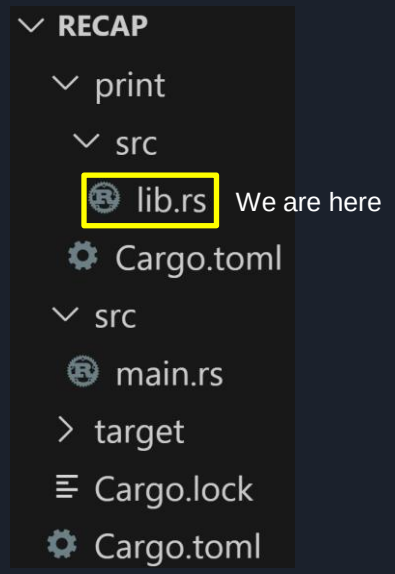
We are here

at runtime, we print this

```
Setting b to a + 4 = 9
Setting c to b.abs() = 9
```

```rust
let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
let _print_yellow = |s: &str| -> String { format!("\x1b[93m{s}\x1b[0m") };
println!("Setting {} to {}",
    _print_green(stringify!(#var)),
    _print_yellow(&format!("{} = {:?}", stringify!(#expr), _e))
);
```

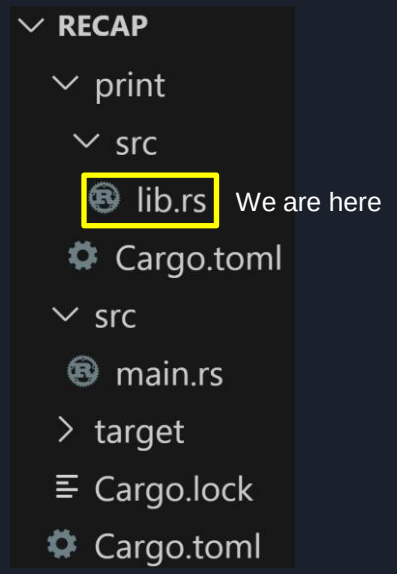
1. Recap




```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. })
    // let <pat> = <expr>;
    // <pat> is a normal identifier like `a` or `b`
    let log: TokenStream = quote! {
        let #var = {
            let _e = #expr;
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            let _print_yellow = |s: &str| -> String { format!("\x1b[93m{s}\x1b[0m") };
            println!("Setting {} to {}",
                _print_green(stringify!(#var)),
                _print_yellow(&format!("{} = {:?}", stringify!(#expr), _e))
            );
            _e
        };
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

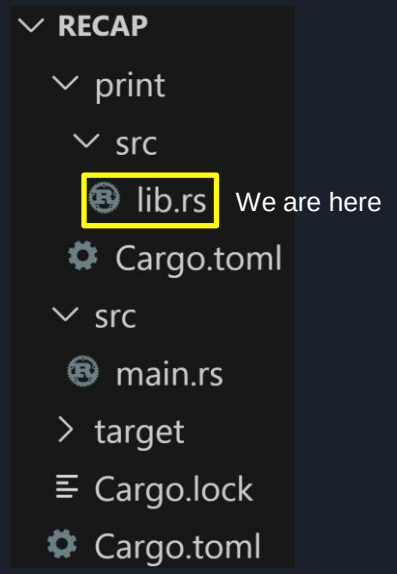
Parsing is a bit redundant, but we're building a vector of statements
→ Our generated code → We know it's correct → We can easily unwrap

1. Recap




```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {
    // let <pat>;
    // <pat> is a normal identifier like `a` or `b`
    new_stmts.push(s.clone());
    let log: TokenStream = quote! {
        {
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            println!("Declaring {}", _print_green(stringify!(#var)));
        }
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

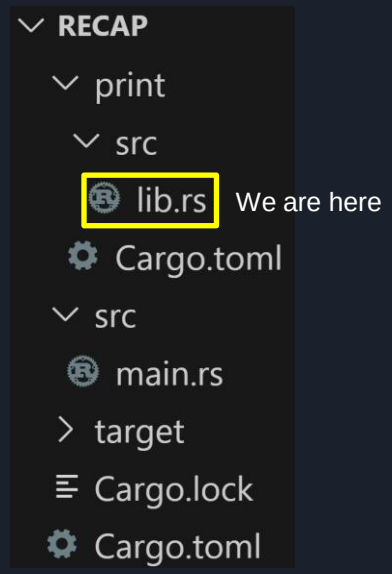
1. Recap




```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {
    // let <pat>;
    // <pat> is a normal identifier like `a` or `b`
    new_stmts.push(s.clone());
    let log: TokenStream = quote! {
        {
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            println!("Declaring {}", _print_green(stringify!(#var)));
        }
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

Simpler this time, we just log that we declared a variable

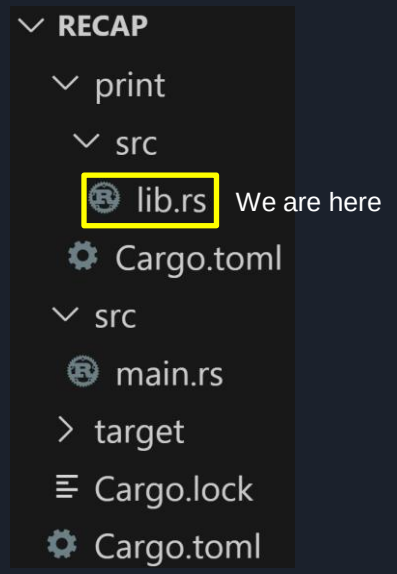
1. Recap




```
Declaring d
```

```
let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
println!("Declaring {}", _print_green(stringify!(#var)));
```

1. Recap




```rust
Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {
    // let <pat>;
    // <pat> is a normal identifier like `a` or `b`
    new_stmts.push(s.clone());
    let log: TokenStream = quote! {
        {
            let _print_green = |s: &str| -> String { format!("\x1b[92m{s}\x1b[0m") };
            println!("Declaring {}", _print_green(stringify!(#var)));
        }
    };
    let stmt: Stmt = syn::parse2::<Stmt>(tokens: log).unwrap();
    new_stmts.push(stmt);
}
```

Generate code

```rust
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, Block, Expr, ExprBlock, ItemFn, Local, LocalInit, Pat, Stmt};

fn replace_in_block(block: &Block) -> Block {
    let mut new_stmts: Vec<Stmt> = vec![];
    for s: &Stmt in &block.stmts {
        match s {
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. }), .. }) => {…
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {…
            Stmt::Expr(Expr::Block(ExprBlock { attrs: a: &Vec<Attribute>, label: l: &Option<Label>, block: inner: &Block, }), semi: …) => {…
            _ => new_stmts.push(s.clone()),
        }
    }
    Block { brace_token: block.brace_token, stmts: new_stmts }
} fn replace_in_block

#[proc_macro_attribute]
pub fn log_variable(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let ItemFn {
        attrs: Vec<Attribute>,
        vis: Visibility,
        sig: Signature,
        block: Box<Block>
    } = parse_macro_input!(item);
    let new_block: Block = replace_in_block(&block);
    quote! {
        #(#attrs)*
        #vis #sig #new_block
    }.into()
}
```

And that‘s it!

```rust
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, Block, Expr, ExprBlock, ItemFn, Local, LocalInit, Pat, Stmt};

fn replace_in_block(block: &Block) -> Block {
    let mut new_stmts: Vec<Stmt> = vec![];
    for s: &Stmt in &block.stmts {
        match s {
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: Some(LocalInit { expr: &Box<Expr>, .. }), .. }) => {…
            Stmt::Local(Local { pat: Pat::Ident(var: &PatIdent), init: None, .. }) => {…
            Stmt::Expr(Expr::Block(ExprBlock { attrs: a: &Vec<Attribute>, label: l: &Option<Label>, block: inner: &Block, }), semi: …) => {…
            _ => new_stmts.push(s.clone()),
        }
    }
    Block { brace_token: block.brace_token, stmts: new_stmts }
} fn replace_in_block

#[proc_macro_attribute]
pub fn log_variable(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let ItemFn {
        attrs: Vec<Attribute>,
        vis: Visibility,
        sig: Signature,
        block: Box<Block>
    } = parse_macro_input!(item);
    let new_block: Block = replace_in_block(&block);
    quote! {
        #(#attrs)*
        #vis #sig #new_block
    }.into()
}
```

And that‘s it!

→ We only needed 60 lines of code to write this macro
→ We can now log every variable declaration in any function!

```rust
use print::log_variable;
#[derive(Debug)]
2 implementations
struct Foo;
impl Foo {
    #[log_variable]
    fn new() -> Self {
        let new_foo: Foo = Foo;
        new_foo
    }
}
#[log_variable]
▶ Run | Debug
fn main() {
    let a: i32 = 5;
    let b: i32 = a + 4;
    let c: i32= b.abs();
    let d: i32;
    d = 1;
    {
        let f: i32 = a + b + c;
    }
    let foo: Foo = Foo::new();
}
```

```rust
use print::log_variable;
#[derive(Debug)]
2 implementations
struct Foo;
impl Foo {
    #[log_variable]
    fn new() -> Self {
        let new_foo: Foo = Foo;
        new_foo
    }
}
#[log_variable]
▶ Run | Debug
fn main() {
    let a: i32 = 5;
    let b: i32 = a + 4;
    let c: i32= b.abs();
    let d: i32;
    d = 1;
    {
        let f: i32 = a + b + c;
    }
    let foo: Foo = Foo::new();
}
```

cargo expand

```rust
let c = {
    let _e = b.abs();
    let _print_green = |s: &str| -> String {
        {
            let res = ::alloc::fmt::format(
                format_args!("\u{1b}[92m{0}\u{1b}[0m", s),
            );
            res
        }
    };
    let _print_yellow = |s: &str| -> String {
        {
            let res = ::alloc::fmt::format(
                format_args!("\u{1b}[93m{0}\u{1b}[0m", s),
            );
            res
        }
    };
    {
        ::std::io::_print(
            format_args!(
                "Setting {0} to {1}\n",
                _print_green("c"),
                _print_yellow(
                    &{
                        let res = ::alloc::fmt::format(
                            format_args!("{0} = {1:?}", "b.abs()", _e),
                        );
                        res
                    },
                ),
            ),
        );
    };
    _e
};
let d;
{
    let _print_green = |s: &str| -> String {
        {
            let res = ::alloc::fmt::format(
                format_args!("\u{1b}[92m{0}\u{1b}[0m", s),
            );
            res
        }
    };
    {
        ::std::io::_print(format_args!("Declaring {0}\n", _print_green("d")));
    };
}
d = 1;
```

```rust
use print::log_variable;
#[derive(Debug)]
2 implementations
struct Foo;
impl Foo {
    #[log_variable]
    fn new() -> Self {
        let new_foo: Foo = Foo;
        new_foo
    }
}
#[log_variable]
▶ Run | Debug
fn main() {
    let a: i32 = 5;
    let b: i32 = a + 4;
    let c: i32= b.abs();
    let d: i32;
    d = 1;
    {
        let f: i32 = a + b + c;
    }
    let foo: Foo = Foo::new();
}
```

cargo expand

```rust
fn new() -> Self {
    let new_foo = {
        let _e = Foo;
        let _print_green = |s: &str| -> String {
            {
                let res = ::alloc::fmt::format(
                    format_args!("\u{1b}[92m{0}\u{1b}[0m", s),
                );
                res
            }
        };
        let _print_yellow = |s: &str| -> String {
            {
                let res = ::alloc::fmt::format(
                    format_args!("\u{1b}[93m{0}\u{1b}[0m", s),
                );
                res
            }
        };
        {
            ::std::io::_print(
                format_args!(
                    "Setting {0} to {1}\n",
                    _print_green("new_foo"),
                    _print_yellow(
                        &{
                            let res = ::alloc::fmt::format(
                                format_args!("{0} = {1:?}", "Foo", _e),
                            );
                            res
                        },
                    ),
                ),
            );
        };
        _e
    };
    new_foo
}
```

```rust
use print::log_variable;
#[derive(Debug)]
2 implementations
struct Foo;
impl Foo {
    #[log_variable]
    fn new() -> Self {
        let new_foo: Foo = Foo;
        new_foo
    }
}
#[log_variable]
▶ Run | Debug
fn main() {
    let a: i32 = 5;
    let b: i32 = a + 4;
    let c: i32= b.abs();
    let d: i32;
    d = 1;
    {
        let f: i32 = a + b + c;
    }
    let foo: Foo = Foo::new();
}
```

cargo run

```
     Running `target\debug\recap.exe`
Setting b to a + 4 = 9
Setting c to b.abs() = 9
Declaring d
Setting f to a + b + c = 23
Setting new_foo to Foo = Foo
Setting foo to Foo :: new() = Foo
```

```rust
use print::log_variable;
#[derive(Debug)]
2 implementations
struct Foo;
impl Foo {
    #[log_variable]
    fn new() -> Self {
        let new_foo: Foo = Foo;
        new_foo
    }
}
#[log_variable]
▶ Run | Debug
fn main() {
    let a: i32 = 5;
    let b: i32 = a + 4;
    let c: i32= b.abs();
    let d: i32;
    d = 1;
    {
        let f: i32 = a + b + c;
    }
    let foo: Foo = Foo::new();
}
```

cargo run

```
     Running `target\debug\recap.exe`
Setting b to a + 4 = 9
Setting c to b.abs() = 9
Declaring d
Setting f to a + b + c = 23
Setting new_foo to Foo = Foo
Setting foo to Foo :: new() = Foo
```

a: i32 is Pat::Type, not Pat::Ident, so we don‘t log that :^)

2. Closures

2. Closures

- Rust has various properties that make it a functional programming language

2. Closures

- Rust has various properties that make it a functional programming language
- Functional programming:

2. Closures

- Rust has various properties that make it a functional programming language
- Functional programming:
  - Programs are constructed by applying and composing functions

2. Closures

- Rust has various properties that make it a functional programming language
- Functional programming:
  - Programs are constructed by applying and composing functions
  - Functions are first-class citizen

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
▶ Run | Debug
fn main() {
    let p: fn print(i32) = print;
    p(arg: 5);
    let mut lookup: HashMap<&'static str, fn(i32) -> ()> = HashMap::new();
    lookup.insert(k: "print", v: print);
    if let Some(func: &fn(i32)) = lookup.get("print") {
        func(12);
    }
}
```

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
▶ Run | Debug
fn main() {
    let p: fn print(i32) = print;
    p(arg: 5);
    let mut lookup: HashMap<&'static str, fn(i32) -> ()> = HashMap::new();
    lookup.insert(k: "print", v: print);
    if let Some(func: &fn(i32)) = lookup.get("print") {
        func(12);
    }
}
```

- Functions in Rust are first-class citizen
- → We can do everything with functions that we can do with normal variables

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
▶ Run | Debug
fn main() {
    let p: fn print(i32) = print;
    p(arg: 5);
    let mut lookup: HashMap<&'static str, fn(i32) -> ()> = HashMap::new();
    lookup.insert(k: "print", v: print);
    if let Some(func: &fn(i32)) = lookup.get("print") {
        func(12);
    }
}
```

- Functions in Rust are first-class citizen
- → We can do everything with functions that we can do with normal variables

We can assign functions to variables

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
▶ Run | Debug
fn main() {
    let p: fn print(i32) = print;
    p(arg: 5);
    let mut lookup: HashMap<&'static str, fn(i32) -> ()> = HashMap::new();
    lookup.insert(k: "print", v: print);
    if let Some(func: &fn(i32)) = lookup.get("print") {
        func(12);
    }
}
```

- Functions in Rust are first-class citizen
- → We can do everything with functions that we can do with normal variables

We can put functions into data structures

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
▶ Run | Debug
fn main() {
    let p: fn print(i32) = print;
    p(arg: 5);
    let mut lookup: HashMap<&'static str, fn(i32) -> ()> = HashMap::new();
    lookup.insert(k: "print", v: print);
    if let Some(func: &fn(i32)) = lookup.get("print") {
        func(12);
    }
}
```

- Functions in Rust are first-class citizen
  - → We can do everything with functions that we can do with normal variables

both p and func are pointers to the original function
→ Indirect function call

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
▶ Run | Debug
fn main() {
    let p: fn print(i32) = print;
    p(arg: 5);
    let mut lookup: HashMap<&'static str, fn(i32) -> ()> = HashMap::new();
    lookup.insert(k: "print", v: print);
    if let Some(func: &fn(i32)) = lookup.get("print") {
        func(12);
    }
}
```

- Functions in Rust are first-class citizen
- → We can do everything with functions that we can do with normal variables

Type signature for functions

2. Closures

Type signature for functions

```
let mut lookup: HashMap<&'static str, fn(i32) -> ()> = HashMap::new();
let mut lookup: HashMap<&'static str, fn(i32)> = HashMap::new();
```

Those types are the same

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
fn wrong_arg(arg: u8) {}
fn call_func(func: fn(i32), arg: i32) {
    func(arg);
}
▶ Run | Debug
fn main() {
    call_func(func: print, arg: 5);
    call_func(func: wrong_arg, arg: 5);
}
```

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
fn wrong_arg(arg: u8) {}
fn call_func(func: fn(i32), arg: i32) {
    func(arg);
}
▶ Run | Debug
fn main() {
    call_func(func: print, arg: 5);
    call_func(func: wrong_arg, arg: 5);
}
```

We can pass functions as arguments

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
fn wrong_arg(arg: u8) {}
fn call_func(func: fn(i32), arg: i32) {
    func(arg);
}
```

Everything is checked at compile time

```rust
▶ Run | Debug
fn main() {
    call_func(func: print, arg: 5);
    call_func(func: wrong_arg, arg: 5);
}
```

```
error[E0308]: mismatched types
  --> src\main.rs:10:15
   |
10 |     call_func(wrong_arg, 5);
   |     --------- ^^^^^^^^^ expected fn pointer, found fn item
   |     |
   |     arguments to this function are incorrect
   |
   = note: expected fn pointer `fn(i32)`
                 found fn item `fn(u8) {wrong_arg}`
note: function defined here
  --> src\main.rs:5:4
   |
5  | fn call_func(func: fn(i32), arg: i32) {
   |    ^^^^^^^^^ -------------
```

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
fn call_func(func: fn(i32), arg: i32) {
    func(arg);
}
▶ Run | Debug
fn main() {
    call_func(func: print, arg: 5);
    let p: fn print(i32) = print;
    p(arg: 10);
}
```

```
     Running
arg is 5
arg is 10
```

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
fn call_func(func: fn(i32), arg: i32) {
    func(arg);
}
▶ Run | Debug
fn main() {
    call_func(func: print, arg: 5);
    let p: fn print(i32) = print;
    p(arg: 10);
}
```

```
    Running
arg is 5
arg is 10
```

2. Closures

```rust
fn print(arg: i32) {
    println!("arg is {}", arg);
}
fn call_func(func: fn(i32), arg: i32) {
    func(arg);
}
▶ Run | Debug
fn main() {
    call_func(func: print, arg: 5);
    let p: fn print(i32) = print;
    p(arg: 10);
}
```

```
    Running
arg is 5
arg is 10
```

2. Closures

```rust
fn filter<T>(elems: Vec<T>, predicate: fn(&T) -> bool) -> Vec<T> {
    let mut new_elems: Vec<T> = vec![];
    for e: T in elems {
        if predicate(&e) {
            new_elems.push(e);
        }
    }
    new_elems
}
fn is_even(n: &i32) -> bool { n % 2 == 0 }
▶ Run | Debug
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let modified: Vec<i32> = filter(elems: vec, predicate: is_even);
    println!("modified: {:?}", modified);
}
```

2. Closures

```rust
fn filter<T>(elems: Vec<T>, predicate: fn(&T) -> bool) -> Vec<T> {
    let mut new_elems: Vec<T> = vec![];
    for e: T in elems {
        if predicate(&e) {
            new_elems.push(e);
        }
    }
    new_elems
}
fn is_even(n: &i32) -> bool { n % 2 == 0 }
▶ Run | Debug
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let modified: Vec<i32> = filter(elems: vec, predicate: is_even);
    println!("modified: {:?}", modified);
}
```

By accepting functions as parameters, we can now build higher-level functions
→ filter: Take a list, and collect all elements which satisfy a given predicate

2. Closures

```rust
fn filter<T>(elems: Vec<T>, predicate: fn(&T) -> bool) -> Vec<T> {
    let mut new_elems: Vec<T> = vec![];
    for e: T in elems {
        if predicate(&e) {
            new_elems.push(e);
        }
    }
    new_elems
}
fn is_even(n: &i32) -> bool { n % 2 == 0 }
▶ Run | Debug
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let modified: Vec<i32> = filter(elems: vec, predicate: is_even);
    println!("modified: {:?}", modified);
}
```

modified contains all even elements of vec

```
original: [1, 2, 3, 4, 5]
modified: [2, 4]
```

2. Closures

```rust
fn map<T, E>(elems: Vec<T>, func: fn(&T) -> E) -> Vec<E> {
    let mut new_elems: Vec<E> = Vec::with_capacity(elems.len());
    for e: T in elems {
        new_elems.push(func(&e));
    }
    new_elems
}
fn square(n: &i32) -> i32 { n * n }
▶ Run | Debug
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let modified: Vec<i32> = map(elems: vec, func: square);
    println!("modified: {:?}", modified);
}
```

2. Closures

```rust
fn map<T, E>(elems: Vec<T>, func: fn(&T) -> E) -> Vec<E> {
    let mut new_elems: Vec<E> = Vec::with_capacity(elems.len());
    for e: T in elems {
        new_elems.push(func(&e));
    }
    new_elems
}
fn square(n: &i32) -> i32 { n * n }
▶ Run | Debug
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let modified: Vec<i32> = map(elems: vec, func: square);
    println!("modified: {:?}", modified);
}
```

By accepting functions as parameters, we can now build higher-level functions:
→ map: Take a list, and apply a function to all elements

2. Closures

```rust
fn map<T, E>(elems: Vec<T>, func: fn(&T) -> E) -> Vec<E> {
    let mut new_elems: Vec<E> = Vec::with_capacity(elems.len());
    for e: T in elems {
        new_elems.push(func(&e));
    }
    new_elems
}
fn square(n: &i32) -> i32 { n * n }
▶ Run | Debug
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let modified: Vec<i32> = map(elems: vec, func: square);
    println!("modified: {:?}", modified);
}
```

modified contains the first five square numbers

```
original: [1, 2, 3, 4, 5]
modified: [1, 4, 9, 16, 25]
```

2. Closures

```rust
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let even_square: Vec<i32> = map(elems: filter(elems: vec, predicate: is_even), func: square);
    println!("modified: {:?}", even_square);
}
```

2. Closures

Composition means passing the output of one function directly to the next function
→ Filter our original vector, only keep the even numbers, which we square directly after

```rust
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let even_square: Vec<i32> = map(elems: filter(elems: vec, predicate: is_even), func: square);
    println!("modified: {:?}", even_square);
}
```

2. Closures

Composition means passing the output of one function directly to the next function
→ Filter our original vector, only keep the even numbers, which we square directly after

```rust
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let even_square: Vec<i32> = map(elems: filter(elems: vec, predicate: is_even), func: square);
    println!("modified: {:?}", even_square);
}
```

```
original: [1, 2, 3, 4, 5]
modified: [4, 16]
```

2. Closures

This is not ideal though, we need to think about function names for our predicates
→ Rust does not support function overloading, we can only have one is_even and one square
→ What if the actual square-function should serve a different purpose?

```
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3, 4, 5];
    println!("original: {:?}", vec);
    let even_square: Vec<i32> = map(elems: filter(elems: vec, predicate: is_even), func: square);
    println!("modified: {:?}", even_square);
}
```

2. Closures

- Closures are special expressions that declare anonymous functions
  - Anonymous function → Closures have no name

2. Closures

- Closures are special expressions that declare anonymous functions
  - Anonymous function → Closures have no name

```
fn main() {
    let square: impl Fn(i32) -> i32 = |n: i32| n * n;
    let n: i32 = square(5);
    println!("n={}", n);
}
```

2. Closures

- Closures are special expressions that declare anonymous functions
  - Anonymous function → Closures have no name

```
fn main() {
    let square: impl Fn(i32) -> i32 = |n: i32| n * n;
    let n: i32 = square(5);
    println!("n={}", n);
}
```

This is a closure

2. Closures

- Closures are special expressions that declare anonymous functions
  - Anonymous function → Closures have no name

Parameters are declared between the pipes

```
fn main() {
    let square: impl Fn(i32) -> i32 = |n: i32| n * n;
    let n: i32 = square(5);
    println!("n={}", n);
}
```

2. Closures

- Closures are special expressions that declare anonymous functions
  - Anonymous function → Closures have no name

```
fn main() {
    let square: impl Fn(i32) -> i32 = |n: i32| n * n;
    let n: i32 = square(5);
    println!("n={}", n);
}
```

Closure body
→ Can also be a block {}

2. Closures

- Closures are special expressions that declare anonymous functions
  - Anonymous function → Closures have no name

We can now bind the closure to a variable, and call it like a normal function

```
fn main() {
    let square: impl Fn(i32) -> i32 = |n: i32| n * n;
    let n: i32 = square(5);
    println!("n={}", n);
}
```

2. Closures

- Closures are special expressions that declare anonymous functions
  - Anonymous function → Closures have no name

We can also directly call the closure

```
fn main() {
    let n: i32 = (|n: i32|n * n)(5);
    println!("n={}", n);
}
```

2. Closures

```rust
fn main() {
    let n: i32 = 7;
    let add_n: impl Fn(i32) -> i32 = |x: i32| x + n;
    let start: i32 = 3;
    let result: i32 = add_n(start);
    println!("{}+{}={}", start, n, result);
}
```

2. Closures

```rust
fn main() {
    let n: i32 = 7;
    let add_n: impl Fn(i32) -> i32 = |x: i32| x + n;
    let start: i32 = 3;
    let result: i32 = add_n(start);
    println!("{}+{}={}", start, n, result);
}
```

Closures are more powerful than function pointers
→ They allow us to capture and use local variables

2. Closures

```rust
fn main() {
    let n: i32 = 7;
    let add_n: impl Fn(i32) -> i32 = |x: i32| x + n;
    let start: i32 = 3;
    let result: i32 = add_n(start);
    println!("{}+{}={}", start, n, result);
}
```

Closure computes x + 7

```
3+7=10
```

2. Closures

Closures implement a special trait that defines their behavior

```
fn main() {
    let n: i32 = 7;
    let add_n: impl Fn(i32) -> i32 = |x: i32| x + n;
    let start: i32 = 3;
    let result: i32 = add_n(start);
    println!("{}+{}={}", start, n, result);
}
```

```
3+7=10
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let add_size_of_s: impl Fn(usize) -> usize = |x: usize| x + s.len();
    println!("{}", add_size_of_s(5));
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

If we only ever immutably borrow outer variables,
the closure implements Fn

```
fn main() {
    let mut s: String = String::from("Big Data");
    let add_size_of_s: impl Fn(usize) -> usize = |x: usize| x + s.len();
    println!("{}", add_size_of_s(5));
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

If we only ever immutably borrow outer variables, the closure implements Fn

```
fn main() {
    let mut s: String = String::from("Big Data");
    let add_size_of_s: impl Fn(usize) -> usize = |x: usize| x + s.len();
    println!("{}", add_size_of_s(5));
}
```

In this range we immutably borrow s

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let add_size_of_s: impl Fn(usize) -> usize = |x: usize| x + s.len();
    s.push(ch: 'c');
    println!("{}", add_size_of_s(5));
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```rust
fn main() {
    let mut s: String = String::from("Big Data");
    let add_size_of_s: impl Fn(usize) -> usize = |x: usize| x + s.len();
    s.push(ch: 'c');
    println!("{}", add_size_of_s(5));
}
```

Immutable borrow here

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let add_size_of_s: impl Fn(usize) -> usize = |x: usize| x + s.len();
    s.push(ch: 'c');
    println!("{}", add_size_of_s(5));
}
```

Mutable borrow here

Immutable borrow here

2. Closures

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
  --> src\main.rs:49:5
   |
48 |     let add_size_of_s = |x| x + s.len();
   |                         ---     - first borrow occurs due to use of `s` in closure
   |                         |
   |                         immutable borrow occurs here
49 |     s.push('c');
   |     ^^^^^^^^^^ mutable borrow occurs here
50 |     println!("{}", add_size_of_s(5));
   |                    ------------- immutable borrow later used here
```

```rust
fn main() {
    let mut s: String = String::from("Big Data");
    let add_size_of_s: impl Fn(usize) -> usize = |x: usize| x + s.len();
    s.push(ch: 'c');
    println!("{}", add_size_of_s(5));
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```rust
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    push_to_str(what: ", now expanded!");
    println!("{}", s);
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```rust
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    push_to_str(what: ", now expanded!");
    println!("{}", s);
}
```

If we mutably borrow any outer variables, the closure implements FnMut

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    push_to_str(what: ", now expanded!");
    println!("{}", s);
}
```

We modify s → We mutably borrow s

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    push_to_str(what: ", now expanded!");
    println!("{}", s);
}
```

If we want to call a FnMut closure, the variable needs to be mutable

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    push_to_str(what: ", now expanded!");
    println!("{}", s);
}
```

In this range we mutably borrow s

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```rust
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    println!("{}", s);
    push_to_str(what: ", now expanded!");
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    println!("{}", s);
    push_to_str(what: ", now expanded!");
}
```

Mutable borrow here

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    println!("{}", s);
    push_to_str(what: ", now expanded!");
}
```

Immutable borrow here

Mutable borrow here

2. Closures

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed as mutable
  --> src\main.rs:56:20
   |
53 |     let mut push_to_str = |what: &str| {
   |                           ------------ mutable borrow occurs here
54 |         s.push_str(what)
   |         - first borrow occurs due to use of `s` in closure
55 |     };
56 |     println!("{}", s);
   |                    ^ immutable borrow occurs here
57 |     push_to_str(", now expanded!");
   |     ----------- mutable borrow later used here
   |
```

```rust
fn main() {
    let mut s: String = String::from("Big Data");
    let mut push_to_str: impl FnMut(&str) = |what: &str| {
        s.push_str(string: what)
    };
    println!("{}", s);
    push_to_str(what: ", now expanded!");
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
}
```

If we move any outer variables, the closure implements FnOnce

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
}
```

If we move any outer variables, the closure implements FnOnce

Because we move values, FnOnce closures can only be called once

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
}
```

String doesn‘t implement Copy, s is moved here

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```rust
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
    println!("{}", s);
}
```

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
    println!("{}", s);
}
```

s moved here

2. Closures

- There are three traits that a closure can implement → Fn, FnMut, FnOnce

```
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
    println!("{}", s);
}
```

s moved here

Can‘t use s here

2. Closures

```
error[E0382]: borrow of moved value: `s`
  --> src\main.rs:66:20
   |
60 |     let s = String::from("Big Data");
   |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
61 |     let move_str = || {
   |                    -- value moved into closure here
62 |         let x = s;
   |                 - variable moved due to use in closure
...
66 |     println!("{}", s);
   |                    ^ value borrowed here after move
   |
```

```rust
fn main() {
    let s: String = String::from("Big Data");
    let move_str: impl FnOnce() = || {
        let x: String = s;
        println!("{}", x);
    };
    move_str();
    println!("{}", s);
}
```

2. Closures

- Those traits have additional trait boundaries
  → FnMut: FnOnce → Every FnMut is also FnOnce
  → Fn: FnMut → Every Fn is also FnMut → Every Fn is also FnOnce

2. Closures

- Those traits have additional trait boundaries
  → FnMut: FnOnce → Every FnMut is also FnOnce
  → Fn: FnMut → Every Fn is also FnMut → Every Fn is also FnOnce

2. Closures

- Those traits have additional trait boundaries
  → FnMut: FnOnce → Every FnMut is also FnOnce
  → Fn: FnMut → Every Fn is also FnMut → Every Fn is also FnOnce
  → FnOnce is the most generic closure, every closure can be called at least once

2. Closures

- Those traits have additional trait boundaries
  → FnMut: FnOnce → Every FnMut is also FnOnce
  → Fn: FnMut → Every Fn is also FnMut → Every Fn is also FnOnce
  → FnOnce is the most generic closure, every closure can be called at least once
- When writing functions that accept closures, it's recommended to always use the most generic trait

2. Closures

- Those traits have additional trait boundaries
  → FnMut: FnOnce → Every FnMut is also FnOnce
  → Fn: FnMut → Every Fn is also FnMut → Every Fn is also FnOnce
  → FnOnce is the most generic closure, every closure can be called at least once
- When writing functions that accept closures, it's recommended to always use the most generic trait
  → Function only expects to call the closure once → Mark parameter as FnOnce → Can still pass FnMut

2. Closures

- Those traits have additional trait boundaries
  → FnMut: FnOnce → Every FnMut is also FnOnce
  → Fn: FnMut → Every Fn is also FnMut → Every Fn is also FnOnce
  → FnOnce is the most generic closure, every closure can be called at least once
- When writing functions that accept closures, it's recommended to always use the most generic trait
  → Function only expects to call the closure once → Mark parameter as FnOnce → Can still pass FnMut
  → Closure *might* mutate its environment → Mark parameter as FnMut → Can still pass Fn

2. Closures

```rust
fn filter_2<T, F>(elems: Vec<T>, mut predicate: F) -> Vec<T>
where F: FnMut(&T) -> bool {
    let mut new_elems: Vec<T> = vec![];
    for e: T in elems {
        if predicate(&e) {
            new_elems.push(e);
        }
    }
    new_elems
}
```

2. Closures

```rust
fn filter_2<T, F>(elems: Vec<T>, mut predicate: F) -> Vec<T>
where F: FnMut(&T) -> bool {
    let mut new_elems: Vec<T> = vec![];
    for e: T in elems {
        if predicate(&e) {
            new_elems.push(e);
        }
    }
    new_elems
}
```

We accept every closure that can be called multiple times
→ We don‘t care if it modifies its environment

2. Closures

```rust
fn filter_2<T, F>(elems: Vec<T>, mut predicate: F) -> Vec<T>
where F: FnMut(&T) -> bool {
    let mut new_elems: Vec<T> = vec![];
    for e: T in elems {
        if predicate(&e) {
            new_elems.push(e);
        }
    }
    new_elems
}
```

We accept every closure that can be called multiple times
→ We don‘t care if it modifies its environment
→ Can‘t accept FnOnce, because we call the closure more than once

2. Closures

```rust
fn main() {
    let vec: Vec<i32> = vec![6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
    let evens: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 0);
    let odds: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 1);
    let mut not_prime: Vec<i32> = vec![];
    let primes: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| {
        for i: i32 in 2..*n {
            if n % i == 0 {
                not_prime.push(*n);
                return false;
            }
        }
        true
    });
    println!("original:  {:?}", vec);
    println!("even:      {:?}", evens);
    println!("odds:      {:?}", odds);
    println!("not_prime: {:?}", not_prime);
    println!("primes:    {:?}", primes);
}
```

2. Closures

```rust
fn main() {
    let vec: Vec<i32> = vec![6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
    let evens: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 0);
    let odds: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 1);
    let mut not_prime: Vec<i32> = vec![];
    let primes: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| {
        for i: i32 in 2..*n {
            if n % i == 0 {
                not_prime.push(*n);
                return false;
            }
        }
        true
    });
    println!("original:  {:?}", vec);
    println!("even:      {:?}", evens);
    println!("odds:      {:?}", odds);
    println!("not_prime: {:?}", not_prime);
    println!("primes:    {:?}", primes);
}
```

We can now call our filter-function with any closure that‘s FnMut

2. Closures

```rust
fn main() {
    let vec: Vec<i32> = vec![6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
    let evens: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 0);
    let odds: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 1);
    let mut not_prime: Vec<i32> = vec![];
    let primes: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| {
        for i: i32 in 2..*n {
            if n % i == 0 {
                not_prime.push(*n);
                return false;
            }
        }
        true
    });
    println!("original:  {:?}", vec);
    println!("even:      {:?}", evens);
    println!("odds:      {:?}", odds);
    println!("not_prime: {:?}", not_prime);
    println!("primes:    {:?}", primes);
}
```

Fn-closures

2. Closures

```rust
fn main() {
    let vec: Vec<i32> = vec![6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
    let evens: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 0);
    let odds: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 1);
    let mut not_prime: Vec<i32> = vec![];
    let primes: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| {
        for i: i32 in 2..*n {
            if n % i == 0 {
                not_prime.push(*n);
                return false;
            }
        }
        true
    });
    println!("original:  {:?}", vec);
    println!("even:      {:?}", evens);
    println!("odds:      {:?}", odds);
    println!("not_prime: {:?}", not_prime);
    println!("primes:    {:?}", primes);
}
```

This closure modifies the environment (it pushes elements to not_prime), so it is FnMut

2. Closures

```rust
fn main() {
    let vec: Vec<i32> = vec![6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
    let evens: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 0);
    let odds: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| n % 2 == 1);
    let mut not_prime: Vec<i32> = vec![];
    let primes: Vec<i32> = filter_2(elems: vec.clone(), predicate: |n: &i32| {
```

```
original:  [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
even:      [6, 8, 10, 12, 14]
odds:      [7, 9, 11, 13, 15]
not_prime: [6, 8, 9, 10, 12, 14, 15]
primes:    [7, 11, 13]
```

```rust
    println!("original:  {:?}", vec);
    println!("even:      {:?}", evens);
    println!("odds:      {:?}", odds);
    println!("not_prime: {:?}", not_prime);
    println!("primes:    {:?}", primes);
}
```

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

We can also return closures from functions

2. Closures

```
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

However, Fn only borrows variables
→ Closure would outlive num

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    move |n: i32| n * num
}
```

Instead of borrowing, we explicitly move all variables into the closure

```rust
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    move |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

Composing functions

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    move |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

Those function calls return closures

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    move |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

n * 7

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    move |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

3 * 7

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    move |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

3 * 7 * 5

2. Closures

```rust
fn multiply_by(num: i32) -> impl Fn(i32) -> i32 {
    move |n: i32| n * num
}
▶ Run | Debug
fn main() {
    let result: i32 = multiply_by(num: 5)(multiply_by(num: 7)(3));
    println!("{}", result);
}
```

3 * 7 * 5

```
105
```

3. Iterators

3. Iterators

- The next step in making closures useful is implementing data structures that work with them

3. Iterators

- The next step in making closures useful is implementing data structures that work with them
- Those data structures are called iterators

3. Iterators

- The next step in making closures useful is implementing data structures that work with them
- Those data structures are called iterators
- Iterators only serve one purpose → Repeatedly emit the next item of a collection, until it is exhausted

3. Iterators

- The next step in making closures useful is implementing data structures that work with them
- Those data structures are called iterators
- Iterators only serve one purpose → Repeatedly emit the next item of a collection, until it is exhausted
- Every iterator implements the Iterator trait in which they define how the next item is obtained

3. Iterators

There are three variations of iterators

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1, 2, 3];
    let ref_iter: Iter<i32> = collection.iter();
    let ref_mut_iter: IterMut<i32> = collection.iter_mut();
    let owned_iter: IntoIter<i32> = collection.into_iter();
}
```

3. Iterators

There are three variations of iterators
→ iter() creates an iterator over immutable references of the elements

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1, 2, 3];
    let ref_iter: Iter<i32> = collection.iter();
    let ref_mut_iter: IterMut<i32> = collection.iter_mut();
    let owned_iter: IntoIter<i32> = collection.into_iter();
}
```

3. Iterators

There are three variations of iterators
→ iter() creates an iterator over immutable references of the elements
→ iter_mut() creates an iterator over mutable references of the elements

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1, 2, 3];
    let ref_iter: Iter<i32> = collection.iter();
    let ref_mut_iter: IterMut<i32> = collection.iter_mut();
    let owned_iter: IntoIter<i32> = collection.into_iter();
}
```

3. Iterators

There are three variations of iterators
→ iter() creates an iterator over immutable references of the elements
→ iter_mut() creates an iterator over mutable references of the elements
→ into_iter() creates an iterator over owned elements

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1, 2, 3];
    let ref_iter: Iter<i32> = collection.iter();
    let ref_mut_iter: IterMut<i32> = collection.iter_mut();
    let owned_iter: IntoIter<i32> = collection.into_iter();
}
```

3. Iterators

Iterators are lazy → They only compute elements when you consume them

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1; 100_000_000];
    let ref_iter: Iter<i32> = collection.iter();
}
```

3. Iterators

Iterators are lazy → They only compute elements when you consume them

Creating an iterator over 100 million elements
→ Super fast because we're not doing anything with it

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1; 100_000_000];
    let ref_iter: Iter<i32> = collection.iter();
}
```

3. Iterators

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1; 100_000_000];
    let mut ref_iter: Iter<i32> = collection.iter();
    ref_iter.map(|n: &i32|n * n);
}
```

3. Iterators

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1; 100_000_000];
    let mut ref_iter: Iter<i32> = collection.iter();
    ref_iter.map(|n: &i32|n * n);
}
```

Iterators come with many different methods
→ map and filter are some of them

3. Iterators

```rust
fn main() {
    let mut collection: Vec<i32> = vec![1; 100_000_000];
    let mut ref_iter: Iter<i32> = collection.iter();
    ref_iter.map(|n: &i32|n * n);
}
```

Iterators come with many different methods
→ map and filter are some of them

map() returns an iterator over all elements of the original collection when applied to the given closure
→ map() is also lazy, it does not do anything until consumed

3. Iterators

```rust
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

Because many iterator-methods are lazy and return iterators, we can easily chain those calls
→ Iterator adaptors
→ Function composition

3. Iterators

```
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum(); sum() consumes the iterator, now we start computing elements
    println!("{}", sum);
}
```

3. Iterators

```rust
fn main() {
    let mut sum: i128 = 0;
    for n in (1..100_000_000) {
        if n % 2 == 1 {
            sum += n * n;
        }
    }
    println!("{}", sum);
}
```

This iterator-chain is equivalent to this for-loop

```rust
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```rust
fn main() {
    let mut sum: i128 = 0;
    for n in (1..100_000_000) {
        if n % 2 == 1 {
            sum += n * n;
        }
    }
    println!("{}", sum);
}
```

This iterator-chain is equivalent to this for-loop

```rust
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```rust
fn main() {
    let mut sum: i128 = 0;
    for n in (1..100_000_000) {
        if n % 2 == 1 {
            sum += n * n;
        }
    }
    println!("{}", sum);
}
```

for also implicitly calls into_iter() for us
→ Moves the collection
→ That‘s why we need to borrow collections

This iterator-chain is equivalent to this for-loop

```rust
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```rust
fn main() {
    let mut sum: i128 = 0;
    for n in (1..100_000_000) {
        if n % 2 == 1 {
            sum += n * n;
        }
    }
    println!("{}", sum);
}
```

This iterator-chain is equivalent to this for-loop

```rust
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```rust
fn main() {
    let mut sum: i128 = 0;
    for n in (1..100_000_000) {
        if n % 2 == 1 {
            sum += n * n;
        }
    }
    println!("{}", sum);
}
```

This iterator-chain is equivalent to this for-loop

```rust
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```rust
fn main() {
    let mut sum: i128 = 0;
    for n in (1..100_000_000) {
        if n % 2 == 1 {
            sum += n * n;
        }
    }
    println!("{}", sum);
}
```

This iterator-chain is equivalent to this for-loop

```rust
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter() Range<i128>
        .filter(|n: &i128| n % 2 == 1) impl Iterator<Item = i128>
        .map(|n: i128|n * n) impl Iterator<Item = i128>
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```rust
use rayon::iter::{IntoParallelIterator, ParallelIterator};
▶ Run | Debug
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_par_iter() Iter<i128>
        .filter(filter_op: |n: &i128| n % 2 == 1) Filter<It
        .map(map_op: |n: i128|n * n) Map<Filter<Iter<i128>,
        .sum();
    println!("{}", sum);
}
```

3. Iterators

```rust
use rayon::iter::{IntoParallelIterator, ParallelIterator};
▶ Run | Debug
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_par_iter() Iter<i128>
        .filter(filter_op: |n: &i128| n % 2 == 1) Filter<It
        .map(map_op: |n: i128|n * n) Map<Filter<Iter<i128>,
        .sum();
    println!("{}", sum);
}
```

Because all elements of iterators are decoupled, we can easily parallelize them

3. Iterators

```rust
use rayon::iter::{IntoParallelIterator, ParallelIterator};
▶ Run | Debug
fn main() {
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_par_iter() Iter<i128>
        .filter(filter_op: |n: &i128| n % 2 == 1) Filter<It
        .map(map_op: |n: i128|n * n) Map<Filter<Iter<i128>,
        .sum();
    println!("{}", sum);
}
```

The crate rayon makes it super easy to add parallelism to our iterators
→ Easy extra performance by just changing a single line

Because all elements of iterators are decoupled, we can easily parallelize them

3. Iterators

```rust
fn main() {
    let now: Instant = std::time::Instant::now();
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter().filter(|n: &i128| n % 2 == 1).map(|n: i128|n * n).sum();
    println!("{}", sum);
    println!("Normal: {}", now.elapsed().as_secs_f64());
    let now: Instant = std::time::Instant::now();
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_par_iter().filter(filter_op: |n: &i128| n % 2 == 1).map(map_op: |n: i128|n * n).sum();
    println!("{}", sum);
    println!("Parallel: {}", now.elapsed().as_secs_f64());
}
```

3. Iterators

```rust
fn main() {
    let now: Instant = std::time::Instant::now();
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter().filter(|n: &i128| n % 2 == 1).map(|n: i128|n * n).sum();
    println!("{}", sum);
    println!("Normal: {}", now.elapsed().as_secs_f64());
    let now: Instant = std::time::Instant::now();
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_par_iter().filter(filter_op: |n: &i128| n % 2 == 1).map(map_op: |n: i128|n * n).sum();
    println!("{}", sum);
    println!("Parallel: {}", now.elapsed().as_secs_f64());
}
```

The only difference in this iterator chain

3. Iterators

```
     Running `target\debug\
166666666666666650000000
Normal: 2.8685183
166666666666666650000000
Parallel: 0.8238326
```

```
     Running `target\release\
166666666666666650000000
Normal: 0.0467487
166666666666666650000000
Parallel: 0.008967
```

```rust
fn main() {
    let now: Instant = std::time::Instant::now();
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_iter().filter(|n: &i128| n % 2 == 1).map(|n: i128|n * n).sum();
    println!("{}", sum);
    println!("Normal: {}", now.elapsed().as_secs_f64());
    let now: Instant = std::time::Instant::now();
    let sum: i128 = (1..100_000_000) Range<i128>
        .into_par_iter().filter(filter_op: |n: &i128| n % 2 == 1).map(map_op: |n: i128|n * n).sum();
    println!("{}", sum);
    println!("Parallel: {}", now.elapsed().as_secs_f64());
}
```

3. Iterators

```rust
struct Container(i32);
▶ Run | Debug
fn main() {
    let collection: [Container; 3] = [Container(1), Container(2), Container(3)];
    let mut element: Option<&Container> = None;
    for c: &Container in &collection {
        if c.0 == 3 {
            element = Some(c);
            break;
        }
    }
    println!("Found container with a number of 3: {:?}", element);
}
```

3. Iterators

```rust
struct Container(i32);
▶ Run | Debug
fn main() {
    let collection: [Container; 3] = [Container(1), Container(2), Container(3)];
    let mut element: Option<&Container> = None;
    for c: &Container in &collection {
        if c.0 == 3 {
            element = Some(c);
            break;
        }
    }
    println!("Found container with a number of 3: {:?}", element);
}
```

Find the first element that has a value of 3

3. Iterators

```rust
struct Container(i32);
▶ Run | Debug
fn main() {
    let collection: [Container; 3] = [Container(1), Container(2), Container(3)];
    let element: Option<&Container> = collection.iter().find(|c: &&Container| c.0 == 3);
    println!("Found container with a number of 3: {:?}", element);
}
```

3. Iterators

```rust
struct Container(i32);
▶ Run | Debug
fn main() {
    let collection: [Container; 3] = [Container(1), Container(2), Container(3)];
    let element: Option<&Container> = collection.iter().find(|c: &&Container| c.0 == 3);
    println!("Found container with a number of 3: {:?}", element);
}
```

Find the first element that has a value of 3
→ Iterators allow us to write more concise and readable code

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}
▶ Run | Debug
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}
```

▶ Run | Debug

```rust
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

Iterators can go on forever, which means we can implement infinite data structures

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}
▶ Run | Debug
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

This iterator goes over every number that fits in an u128, and prints every prime

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}

▶ Run | Debug
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

To check if a number is prime:
Take all odd numbers

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}

▶ Run | Debug
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

To check if a number is prime:
Take all odd numbers that are smaller than the square root of the candidate

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}
```

To check if a number is prime:
Take all odd numbers that are smaller than the square root of the candidate and check that they don't divide the candidate

▶ Run | Debug

```rust
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}
▶ Run | Debug
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

all is short-circuiting
→ It stops iterating once it finds an element that returns false

3. Iterators

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}
▶ Run | Debug
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

If this iterator hits the end of u128, two things can happen

→ Debug mode: Panic

→ Release mode: Overflow and start at 0 again

3. Iterators

At the same time, iterators are memory efficient, at any point we only store two numbers:
The prime candidate, and the number we're currently checking against

```rust
fn is_prime(num: &u128) -> bool {
    if *num == 2 { return true; }
    else if num % 2 == 0 { return false; }
    (1..).map(|n: u128|2*n+1).take_while(|n: &u128|n*n<=*num).all(|n: u128|num % n != 0)
}
▶ Run | Debug
fn main() {
    let all_numbers: RangeFrom<u128> = 2..;
    all_numbers.filter(is_prime).for_each(|e: u128|println!("{e}"));
}
```

some prime numbers:

```
261775807
261775823
261775841
261775867
261775889
261775897
261775907
261775919
261775931
261775933
261775981
261775993
261776003
261776017
261776023
261776027
261776051
```

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
▶ Run | Debug
fn main() {
    let containers: Vec<Container> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i).unwrap()) impl Iterator<Item = Container>
        .collect::<Vec<_>>();
}
```

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
```

Implement Container::try_from(i32), which only accepts numbers above 10

```rust
▶ Run | Debug
fn main() {
    let containers: Vec<Container> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i).unwrap()) impl Iterator<Item = Container>
        .collect::<Vec<_>>();
}
```

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
▶ Run | Debug
fn main() {
    let containers: Vec<Container> = (1..20)
        .map(|i: i32|Container::try_from(i).unwrap())
        .collect::<Vec<_>>();
}
```

Create an iterator to easily build a vector of containers

Range<i32>

impl Iterator<Item = Container>

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
▶ Run | Debug
fn main() {
    let containers: Vec<Container> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i).unwrap()) impl Iterator<Item = Container>
        .collect::<Vec<_>>();
}
```

Problem: .unwrap() is not good practice, we'd like to report the error

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
```

▶ Run | Debug

```rust
fn main() {
    let containers: Vec<Result<Container, String>> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i)) impl Iterator<Item = Result<…, …>>
        .collect::<Vec<_>>();
}
```

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
```

.unwrap() is gone, but we now have a Vector of Results!

```rust
▶ Run | Debug
fn main() {
    let containers: Vec<Result<Container, String>> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i)) impl Iterator<Item = Result<…, …>>
        .collect::<Vec<_>>();
}
```

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
▶ Run | Debug
fn main() {
    let containers: Vec<Result<Container, String>> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i)) impl Iterator<Item = Result<…, …>>
        .collect::<Vec<_>>();
}
```

.unwrap() is gone, but we now have a Vector of Results!
→ collect() comes with a lot of useful overloads

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
▶ Run | Debug
fn main() {
    let containers: Result<Vec<Container>, String> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i)) impl Iterator<Item = Result<…, …>>
        .collect::<Result<Vec<_>, _>>();
}
```

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
▶ Run | Debug
fn main() {
    let containers: Result<Vec<Container>, String> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i)) impl Iterator<Item = Result<…, …>>
        .collect::<Result<Vec<_>, _>>();
}
```

collect() can be called with any generic types that implement the FromIterator trait
→ Result<T, E> does implement that trait!

3. Iterators

```rust
struct Container(i32);
impl TryFrom<i32> for Container {
    type Error = String;
    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value < 10 { Err(format!("Numbers below 10 are reserved, got {}", value)) }
        else { Ok(Self(value)) }
    }
}
▶ Run | Debug
fn main() {
    let containers: Result<Vec<Container>, String> = (1..20) Range<i32>
        .map(|i: i32|Container::try_from(i)) impl Iterator<Item = Result<…, …>>
        .collect::<Result<Vec<_>, _>>();
}
```

collect() can be called with any generic types that implement the FromIterator trait

→ Result<T, E> does implement that trait!

→ If the iterator contains an Err()-element, we stop directly and return the Err()

→ If all elements are Ok(), we transform Vec<Ok> into Ok<Vec>

3. Iterators

```rust
fn main() -> Result<(), String> {
    let containers = (1..20) Range<i32>
        .inspect(|n|println!("Number: {}", n)) im
        .map(|i|Container::try_from(i)) impl Iter
        .inspect(|v|println!("Result: {:?}", v))
        .collect::<Result<Vec<_>, _>>()?;
    for c in &containers {
        println!("{:?}", c);
    }
    Ok(())
}
```

3. Iterators

```rust
fn main() -> Result<(), String> {
    let containers = (1..20) Range<i32>
        .inspect(|n|println!("Number: {}", n)) im
        .map(|i|Container::try_from(i)) impl Iter
        .inspect(|v|println!("Result: {:?}", v))
        .collect::<Result<Vec<_>, _>>()?;
    for c in &containers {
        println!("{:?}", c);
    }
    Ok(())
}
```

inspect() allows us to see what element the iterator is currently processing

3. Iterators

```rust
fn main() -> Result<(), String> {
    let containers = (1..20) Range<i32>
        .inspect(|n|println!("Number: {}", n)) im
        .map(|i|Container::try_from(i)) impl Iter
        .inspect(|v|println!("Result: {:?}", v))
        .collect::<Result<Vec<_>, _>>()?;
    for c in &containers {
        println!("{:?}", c);
    }
    Ok(())
}
```

Because we're now collecting into a Result, we can use the ?-operator

3. Iterators

```rust
fn main() -> Result<(), String> {
    let containers = (1..20) Range<i32>
        .inspect(|n|println!("Number: {}", n)) im
        .map(|i|Container::try_from(i)) impl Iter
        .inspect(|v|println!("Result: {:?}", v))
        .collect::<Result<Vec<_>, _>>()?;
    for c in &containers {
        println!("{:?}", c);
    }
    Ok(())
}
```

Our first element directly caused an error, so we did not advance the iterator further

```
     Running `target\debug\iterators.exe`
Number: 1
Result: Err("Numbers below 10 are reserved, got 1")
Error: "Numbers below 10 are reserved, got 1"
error: process didn't exit successfully: `target\deb
```

3. Iterators

Change range to something more reasonable

```rust
fn main() -> Result<(), String> {
    let containers = (15..20) Range<i32>
        .inspect(|n|println!("Number: {}", n)) ir
        .map(|i|Container::try_from(i)) impl Iter
        .inspect(|v|println!("Result: {:?}", v))
        .collect::<Result<Vec<_>, _>>()?;
    for c in &containers {
        println!("{:?}", c);
    }
    Ok(())
}
```

3. Iterators

```rust
fn main() -> Result<(), String> {
    let containers = (15..20) Range<i32>
        .inspect(|n|println!("Number: {}", n)) ir
        .map(|i|Container::try_from(i)) impl Iter
        .inspect(|v|println!("Result: {:?}", v))
        .collect::<Result<Vec<_>, _>>()?;
    for c in &containers {
        println!("{:?}", c);
    }
    Ok(())
}
```

We successfully iterate over all numbers and build containers!

```
Number: 15
Result: Ok(Container(15))
Number: 16
Result: Ok(Container(16))
Number: 17
Result: Ok(Container(17))
Number: 18
Result: Ok(Container(18))
Number: 19
Result: Ok(Container(19))
Container(15)
Container(16)
Container(17)
Container(18)
Container(19)
```

4. Next time

- Multithreading