little languages

lecture 18:

io tutorial

Start-up the VM!

```
git pull upstream master
cd <repo>/lecture/18-files-and-crates
```

Today's Tutorial

- Today's lecture is a project-style tutorial implementing the `cat` utility

- It's a utility that illustrates how to read inputs line-by-line
  - We'll explore how to read files line-by-line
  - As well as standard input

- Both of these input sources implement the same interfaces and thus can share processing code
  - The same interface is shared by Rust's networking libraries, as well, so you could extend today's demo to read URLs over the internet

Implementing `cat`

- The `cat` utility's official purpose is to *concatenate* files to **stdout**
- It's most frequently used to print a single file's contents to stdout, but its signature is actually:
- `cat option* file*`
- Zero-or-more files?!
  - When 0 files are provided, cat reads from stdin rather than file paths.
  - Try it: cat <enter> hello <enter> Ctrl+C to quit

Creating a new project

- Establish a new project:
  - `cargo new thcat`
  - `cd thcat`

- Add a structopt (command line option parser) crate dependency:
  - vim Cargo.toml
  - `[dependencies ]`
  - `structopt = "0.2"`

- Build to precompile dependencies:
  - `cargo build`

- Open main file and silence warnings about unused symbols (for now):
  - **`#![allow(unused)]`**

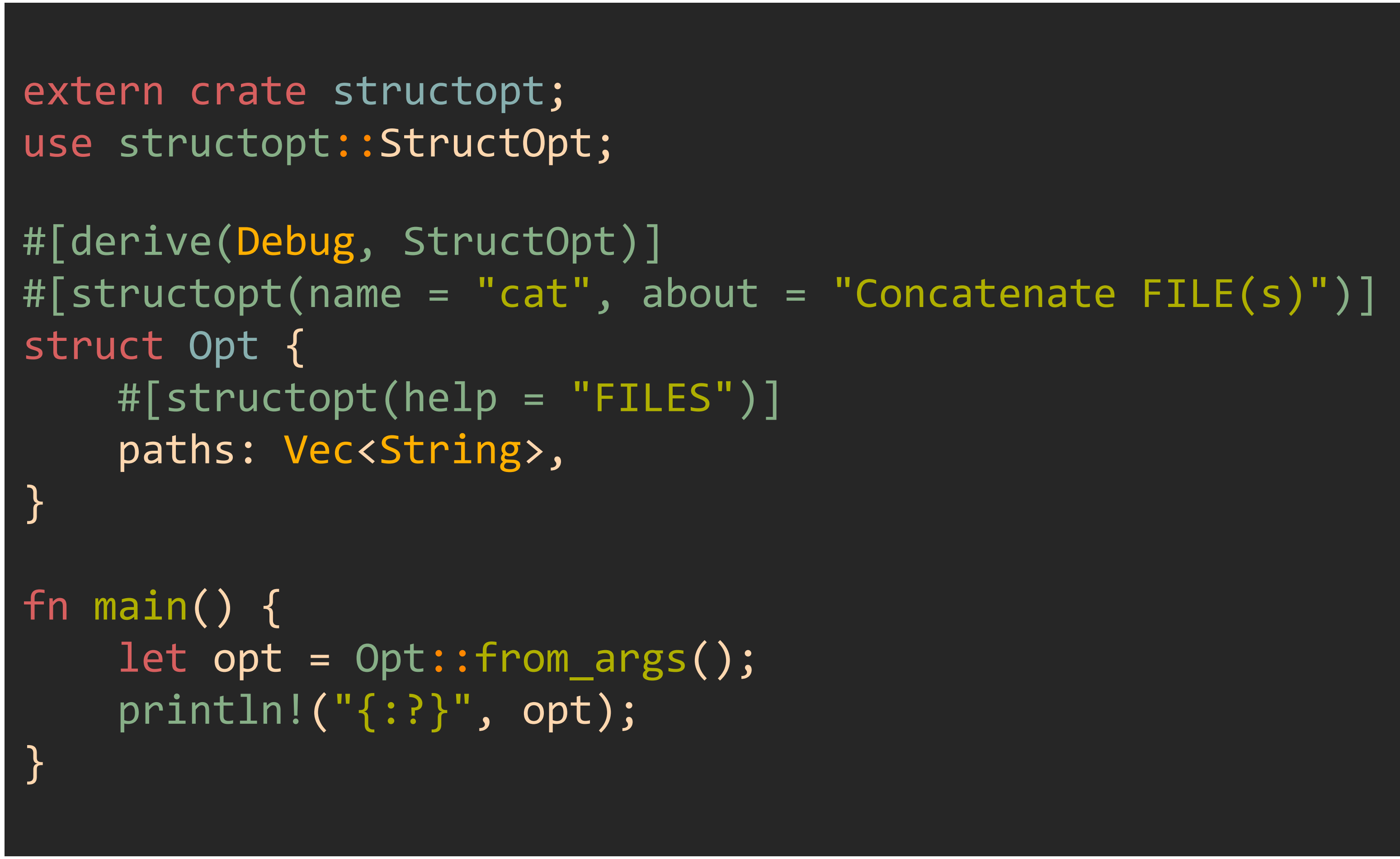
Libraries for Parsing Command-line Options

- Most languages have popular libraries for abstracting away the problem of parsing command-line options and generating helpful documentation.

- In Rust, the structopt crate is the most common choice.

- It takes an annotated struct (like you're seeing to the right) and automatically parses command-line inputs into the struct, generates help and version information, and is flexible for other kinds of options.

- https://docs.rs/structopt/0.2.14/structopt/



```rust
extern crate structopt;
use structopt::StructOpt;

#[derive(Debug, StructOpt)]
#[structopt(name = "cat", about = "Concatenate FILE(s)")]
struct Opt {
    #[structopt(help = "FILES")]
    paths: Vec<String>,
}

fn main() {
    let opt = Opt::from_args();
    println!("{:?}", opt);
}
```

Setting up for File I/O

```
fn main() {
    let opt = Opt::from_args();
    let result = print_files(&opt);
    if let Err(e) = result {
        eprintln!("{}", e);
    }
}

use std::fs::File;
use std::io::BufRead;
use std::io;

fn print_files(opt: &Opt) -> io::Result<()> {
    Ok(())
}
```

Input/Output always introduces the possibility of errors external to the system (such as file not found). As such, you need to handle the errors.

We'll need to import a few symbols for the demo. BufRead is a trait that's explicitly imported for reasons we'll discuss when we get to traits.

Our initial goal here is just getting skeleton code to compile. We'll fill in the details next.

Reading from Files

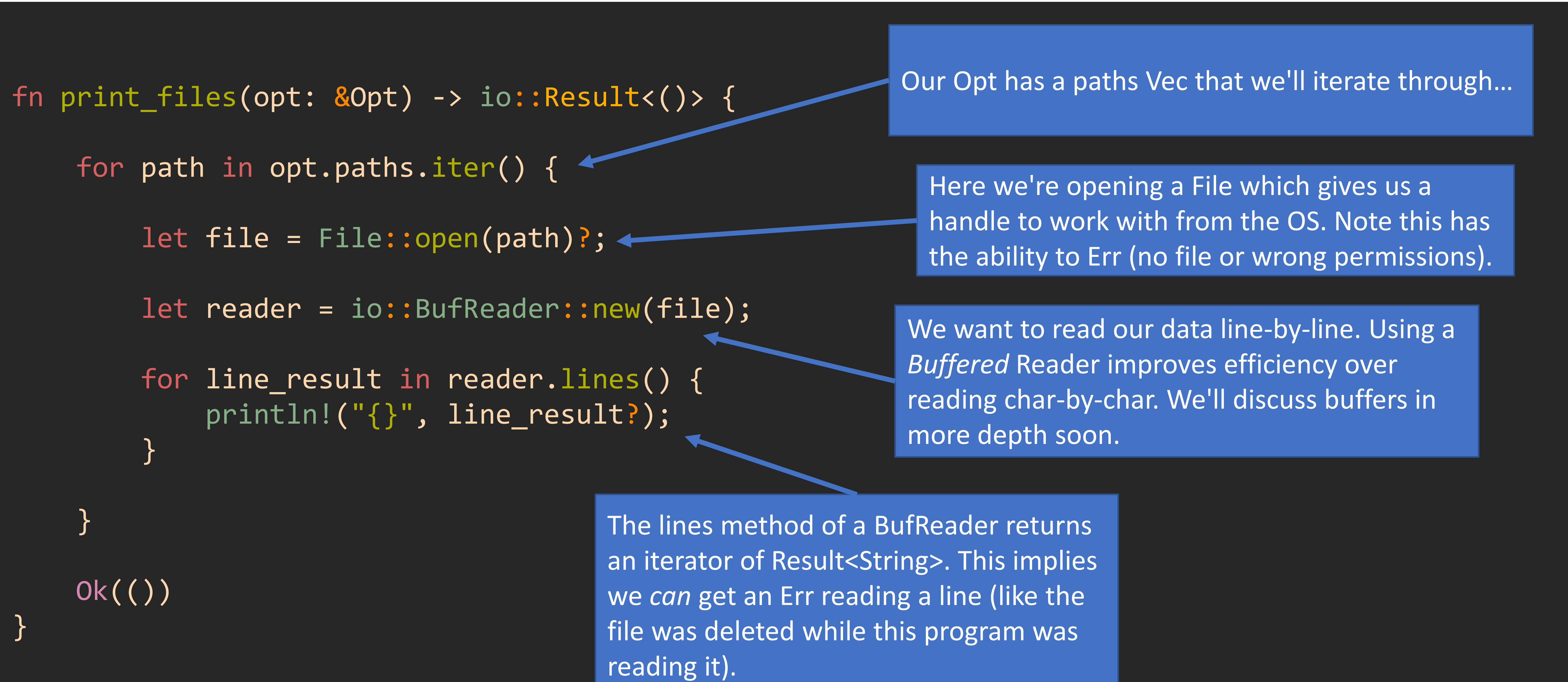
- The general process for reading input from a file is:

1. Ask the operating system for a file handle
    - This is a descriptor the operating system keeps track of specific to your process.
    - Your program will use it in subsequent calls to ask for data and close the file.
    - The operating system uses it to keep track of how its resources are allocated.

2. Ask the operating system for more contents of the file by its handle
    - Behind the scenes systems calls are happening with the file descriptor "hey, give me the next chunk of this file"

3. Tell the operating system to close the file handle
    - When a process is done reading a file, it lets the OS know to conserve resources
    - In Rust, this happens automatically for you when the file handle's lifetime expires and is dropped
    - If your program exits, whether normally or during panic, the operating system handles the cleanup of closing out a process' open file handles

Iterating through the Paths and Reading Each File



```rust
fn print_files(opt: &Opt) -> io::Result<()> {

    for path in opt.paths.iter() {

        let file = File::open(path)?;

        let reader = io::BufReader::new(file);

        for line_result in reader.lines() {
            println!("{}", line_result?);
        }

    }

    Ok(())
}
```

Our Opt has a paths Vec that we'll iterate through...

Here we're opening a File which gives us a handle to work with from the OS. Note this has the ability to Err (no file or wrong permissions).

We want to read our data line-by-line. Using a *Buffered* Reader improves efficiency over reading char-by-char. We'll discuss buffers in more depth soon.

The lines method of a BufReader returns an iterator of Result<String>. This implies we *can* get an Err reading a line (like the file was deleted while this program was reading it).

Adding Support for Using cat with Standard Input

```
// fn main...
    let result = if opt.paths.len() > 0 {
        print_files(&opt)
    } else {
        print_stdin()
    };
```

```
fn print_stdin() -> io::Result<()> {
    let stdin = io::stdin();
    let reader = stdin.lock();
    for line_result in reader.lines() {
        println!("{}", line_result?);
    }
    Ok(())
}
```

- You can also establish a buffered line reader for standard input.
- To do so, a "lock" is acquired on the standard input's file descriptor.
  - File descriptor?!? More on this, soon, but a great design innovation of Unix was treating *everything* as a "file".
- Notice that reading lines from stdin can also error (this usually only happens when a "pipe breaks" and a previous program closes an output or crashes).

Notice there's redundancy in each of these loops:

```rust
// fn print_files
let reader = io::BufReader::new(file);
for line_result in reader.lines() {
    println!("{}", line_result?);
}
```

```rust
// fn print_stdin
let reader = stdin.lock();
for line_result in reader.lines() {
    println!("{}", line_result?);
}
```

- BufReader implements the BufRead trait (with the lines() method)
- Standard Input's lock() returns a StdInLock which also implements BufRead

- The logic inside the loop for `cat` is *super straightforward* but you could imagine (and will see in `thgrep`) doing more with each line and having more redundancy

- Let's look at how to make use of a generic function to process these BufReads

Processing Input from Different Sources Generically

```
fn print_lines<R: BufRead>(reader: R) -> io::Result<()> {
    for line in reader.lines() {
        println!("{}", line?);
    }
    Ok(())
}
```

Notice we've abstracted out the printing loop that iterates over lines()

```
// fn print_files
let reader = io::BufReader::new(file);
print_lines(reader)?;
```

```
// fn print_stdin
let stdin = io::stdin();
let reader = stdin.lock();
print_lines(reader)
```

We can now make use of this function from both print_files and print_stdin.