# Um projeto de Entrada e Saída

Diogo Silveira Mendonça

Slides Baseados no Capítulo 12 do Rust Book
https://doc.rust-lang.org/book/ch12-00-an-io-project.html

# Introdução

- Está capítulo é um resumo das habilidades aprendidas até agora, com a exploração de alguns recursos adicionais da biblioteca padrão.

- Será construído uma ferramenta de linha de comando que interage com entrada/saída de arquivos e comandos.

- Faremos nossa própria versão da clássica ferramenta de busca em linha de comando grep (globally search a regular expression and print).
  - Procura por uma string especificada em um arquivo especificado.
  - Recebe como argumentos um caminho de arquivo e uma string.
  - Lê o arquivo, encontra linhas nesse arquivo que contém o argumento da string e imprimir essas linhas.

- Os exemplo podem ser encontrados nesse link.

# Aceitando argumentos de linha de comando

- Criaremos um novo projeto chamado “minigrep”.

- A primeira tarefa é fazer o minigrep receber dois argumentos.
  - caminho para o arquivo.
  - string que será procurada.
  - cargo run -- searchstring example-filename.txt

- Apesar de existirem bibliotecas no crates.io que possibilitam o programa a aceitar argumentos, para fins de aprendizado, vamos desenvolver essa funcionalidade.

```
$ cargo new minigrep
    Created binary (application)
`minigrep` project
$ cd minigrep
```

# Lendo valores de argumentos

- Para permitir o minigrep ler os valores dos argumentos, precisaremos utilizar a função “std::env::args” a biblioteca padrão.

- Esta função retorna um iterador dos argumentos de linha de comando passados ao minigrep.

- Os iteradores produzem uma série de valores, e podemos chamar o método collect em um iterador para convertê-lo em uma coleção, como um vetor, que contém todos os elementos que o iterador produz.

# Lendo valores de argumentos

- Primeiro, trazemos o módulo std::env para o escopo com uma instrução use para que possamos usar sua função args.
- Note que a função std::env::args está aninhada em dois níveis de módulos.
- std::env::args gerará um panic se algum argumento contiver Unicode inválido.
- std::env::args_os pode receber um unicode inválido, gerando valores OsString, porém esses valores se diferem conforme a plataforma. Não utilizaremos por simplicidade.

src/main.rs:

```
use std::env;

fn main() {
        let args: Vec<String> =
                env::args().collect();
        dbg!(args);
}
```

# Lendo valores de argumentos

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
       Finished dev [unoptimized + debuginfo] target(s) in 0.61s
       Running `target/debug/minigrep`
[src/main.rs:5] args = [
       "target/debug/minigrep",
]
```

```
$ cargo run -- needle haystack
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
       Finished dev [unoptimized + debuginfo] target(s) in 1.57s
       Running `target/debug/minigrep needle haystack`
[src/main.rs:5] args = [
       "target/debug/minigrep",
       "needle",
       "haystack",
]
```

# Salvando valores dos argumentos em variáveis

- Agora que conseguimos acessar os valores dos argumentos, vamos armazenar em variáveis.

```
[src/main.rs:5] args = [
        "target/debug/minigrep",
        "needle",
        "haystack",
]
```

- O vetor tem os valores:
  - args[0] o nome do programa (target/debug/minigrep).
  - args[1] a string que vamos procurar (needle).
  - args[2] o caminho para o arquivo (haystack).

src/main.rs:

```
use std::env;

fn main() {
    let args: Vec<String> =
        env::args().collect();

    let query = &args[1];
    let file_path = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", file_path);
}
```

# Salvando valores dos argumentos em variáveis

- Executando o programa com os argumentos que desejamos.

- Note que o output imprime com sucesso os valores dos argumentos.

```
$ cargo run -- test sample.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
      Finished dev [unoptimized + debuginfo] target(s) in 0.0s
      Running `target/debug/minigrep test sample.txt`
Searching for test
In file sample.txt
```

# Lendo um arquivo

- Vamos criar um arquivo poem.txt e adicionar o poema de Emily Dickinson.

poem.txt:

I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!

# Lendo um arquivo

- Vamos utilizar std::fs para lidar com arquivos.
- A função fs::read_to_string recebe file_path, abre o arquivo e retorna um std::io::Result<String> com o conteúdo do arquivo.
- Imprimimos o valor de contents, para verificar se o código está funcionando.

Exemplo:

```
use std::env;
use std::fs;
fn main() {
    // --snip--
    println!("In file {}", file_path);

    let contents = fs::read_to_string(file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}
```

# Lendo um arquivo

```
$ cargo run -- the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

# Refatorando para melhorar a modularidade e tratamento de erros

- O código apesar de estar fazendo o que queremos, ele tem quatro problemas.

- A nossa função main ela realiza duas funções, separa os argumentos e lê um arquivo. Ela só ficará mais complexa, então é melhor criar uma lib.rs para separar as funcionalidades.

- Quanto maior a função main fica, mais variáveis serão utilizadas, dificultando o entendimento do código. É melhor separar as variáveis que configuram nosso programa em uma struct.
  - Variáveis como query e file_path são de configuração
  - Variáveis como contents são para a programação lógica.

# Refatorando para melhorar a modularidade e tratamento de erros

- Usamos um expect para imprimir a mensagem “Should have been able to read the file”, que não diz tanto sobre o porquê não conseguiu ler o arquivo. É melhor tentar tratar todos os erros com mensagens específicas para cada erro.

- Seria melhor também mover todo o código de tratamento de erro em um lugar separado, para facilitar futuramente a manutenção do código.

# Separação de responsabilidades para projetos binários

- Para resolver o problema da função main com diversas funcionalidades:
  - Separar o programa main.rs em main.rs e lib.rs e mover a lógica da programação para lib.rs
  - Como a funcionalidade de separação da linha de comando é simples ela pode ficar em main.rs
  - Se ela ficar complicada, também deveremos mover ela para lib.rs.

- A responsabilidade da função main deve se limitar a:
  - Chamar a lógica que analisa a linha de comando com os valores dos argumentos;
  - Configurar qualquer outra configuração;
  - Chamar a função run em lib.rs;
  - Tratar do erro, caso a função run retorne algum erro.

- Esse padrão é sobre separação de funcionalidade, main.rs executa o programa e lib.rs tem a logica do programa.

- Como main.rs não pode ser diretamente testada, essa estrutura separa também o que é interessante de ser diretamente testado em lib.rs.

# Extraindo o separador de argumentos

- Vamos separar a funcionalidade da separação dos argumentos do comando em uma nova função parse_config.
- No início da função main temos a chamada de parse_config.
- Ainda criamos as variáveis query e file_path em main, mas main não tem mais a responsabilidade de separar os argumentos do comando.

Exemplo:

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let (query, file_path) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let file_path = &args[2];

    (query, file_path)
}
```

# Agrupando valores de configuração

- Podemos melhorar a abstração do código, criando uma nova estrutura, para a armazenar os valores de query e file_path.

- Chamar o método clone nos valores pode ser ineficiente, porém torna mais fácil, pois não precisamos gerenciar os tempos de vida das referências.

Exemplo:

```
struct Config {
        query: String,
        file_path: String,
}

fn parse_config(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
}
```

# Agrupando valores de configuração

- A variável args em main é a proprietária dos valores dos argumentos e apenas permite que a função parse_config faça empréstimos deles.

- O código agora expressa mais claramente que query e file_path estão relacionados e que seu propósito é configurar como o programa funcionará.

Exemplo:

```
use std::env;
use std::fs;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

    println!("With text:\n{contents}");
}
```

# Criando um construtor para config

- Podemos alterar parse_config de uma função simples para uma função chamada new associada à struct Config.
- Fazer essa alteração tornará o código mais idiomático.
- Poderemos criar instâncias de Config chamando Config::new.

Exemplo:

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::new(&args);

    // --snip--
}

// --snip--

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let file_path = args[2].clone();

        Config { query, file_path }
    }
}
```

# Ajustando os tratamentos de erros

- Vamos agora trabalhar na correção do tratamento de erros.

- Tente executar o programa sem nenhum argumento.

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
       Finished dev [unoptimized + debuginfo] target(s) in 0.0s
       Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is 1',
src/main.rs:27:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- A mensagem de erro "line index out of bounds: the len is 1 but the index is 1" é destinada aos programadores. Isso não ajuda tanto os usuários finais a entender o erro.

# Melhorando as mensagens de erro

- O código ao lado irá entrar em pânico e fornecer uma mensagem de erro, caso o comando não tenha 2 argumentos, antes de acessar os índices.

- Executando o código sem adicionar argumentos.

Exemplo:

```
impl Config {
    fn new(args: &[String]) -> Config {
        if args.len() < 3 {
            panic!("not enough arguments");
        }
        // --snip--
```

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
       Finished dev [unoptimized + debuginfo] target(s) in 0.0s
       Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

# Retornando Result em vez de entrar em pânico

- Podemos mudar para retornar um valor Result.
- Vamos mudar o nome da função de new para build porque muitos programadores esperam que funções new nunca falhem.
- Retornar um valor Err, permite que a função main manipule o valor Result e encerre o processo de maneira mais limpa no caso de erro.

Exemplo:

```
impl Config {
    fn build(args: &[String]) ->
    Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        Ok(Config { query, file_path })
    }
}
```

# Retornando Result em vez de entrar em pânico

- unwrap_or_else permite definir um tratamento de error personalizado, sem recorrer a panic!
  - Se o valor for Ok, ele retorna o valor interno que o Ok está envolvendo.
  - Se o valor for um Err, este método chama o código na função de fechamento (closure).
- Adicionamos uma nova linha use para trazer process da biblioteca padrão para o escopo.

Exemplo:

```
use std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {err}");
        process::exit(1);
    });
    // --snip--
```

# Retornando Result em vez de entrar em pânico

- Executando o código, esperamos uma mensagem semelhante ao do panic!, porém sem as saídas adicionais, que não são interessantes para o usuário final.

- Com isso terminamos a refatoração a análise de configuração.

- No próximo slide, estaremos realizando mudanças na lógica do programa.

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
      Finished dev [unoptimized + debuginfo] target(s) in 0.48s
      Running `target/debug/minigrep`
Problem parsing arguments: not enough arguments
```

# Extraindo a lógica de main

- Vamos extrair uma função chamada run que conterá toda a lógica atualmente presente na função main que não está envolvida na configuração ou tratamento de erros.

- Essa função por enquanto permanece em src/main.rs.

Exemplo:

```
fn main() {
        // --snip--

        println!("Searching for {}", config.query);
        println!("In file {}", config.file_path);

        run(config);
}

fn run(config: Config) {
        let contents = fs::read_to_string(config.file_path)
        .expect("Should have been able to read the file");

        println!("With text:\n{contents}");
}
// --snip--
```

# Retornando erros da função run

- Em vez de permitir que o programa entre em pânico chamando expect, a função run retornará um Result.

- Permitindo criar um tratamento de erros na função main de uma maneira amigável para o usuário.

Exemplo:

```
use std::error::Error;
// --snip--

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    println!("With text:\n{contents}");

    Ok(())
}
```

# Retornando erros da função run

- Fizemos três alterações significativas aqui:
  - Alteramos o tipo de retorno da função run para Result<(), Box<dyn Error>>
  - Removemos a chamada para expect em favor do operador “?”
  - A função run agora retorna um valor Ok no caso de sucesso, Ok tem o tipo unit, ()

- Box<dyn Error> significa que a função retornará um tipo que implementa o traço Error, mas não precisamos especificar qual tipo particular o valor de retorno será.

- O operador “?” retornará o valor de erro da função atual para que o chamador possa manipular.

- Usar Ok(( )) dessa forma é a maneira idiomática de indicar que estamos chamando run apenas pelos seus efeitos colaterais; não retorna um valor que precisamos.

# Retornando erros da função run

- Se executarmos o código, teremos o warning abaixo.
- O warning avisa que nosso código ignorou o valor Result e que esse valor pode indicar que ocorreu um erro.
- O compilador nos lembra de que provavelmente deveria ter algum de tratamento de erro nessa parte.

```
warning: unused `Result` that must be used
  --> src/main.rs:19:5
   |
19 |     run(config);
   |     ^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
   = note: `#[warn(unused_must_use)]` on by default

warning: `minigrep` (bin "minigrep") generated 1 warning
```

# Tratamento de erros retornando por run em main

- Usamos if let em vez de unwrap_or_else para verificar se run retorna um valor Err e chamamos process::exit(1) se for o caso.

- Como run retorna ( ) no caso de sucesso, nos interessa apenas detectar um erro, então não precisamos que unwrap_or_else retorne o valor desembrulhado, que seria apenas ( )

- Os corpos do if let e das funções unwrap_or_else são iguais, imprimimos o erro e saimos.

```
fn main() {
        // --snip--

        println!("Searching for {}", config.query);
        println!("In file {}", config.file_path);

        if let Err(e) = run(config) {
                println!("Application error: {e}");
                process::exit(1);
        }
}
```

# Separando o código em crates

- Dividir o arquivo src/main.rs e colocar parte do código no arquivo src/lib.rs.
  - A definição da função run
  - As declarações use relevantes
  - A definição de Config
  - A definição da função Config::build

# Separando o código em crates

src/lib.rs

```rust
use std::error::Error;
use std::fs;

pub struct Config {
    pub query: String,
    pub file_path: String,
}

pub fn run(config: Config) ->
    Result<(), Box<dyn Error>> {
        let contents =
            fs::read_to_string(config.file_path)?;

        println!("With text:\n{contents}");

        Ok(())
}
```

```rust
impl Config {
    pub fn build(args: &[String]) ->
        Result<Config, &'static str> {
            if args.len() < 3 {
                return Err(
                    "not enough arguments"
                );
            }

            let query = args[1].clone();
            let file_path = args[2].clone();

            Ok(Config { query, file_path })
        }
}
```

# Separando o código em crates

- Adicionamos uma linha use minigrep::Config para trazer o tipo Config da crate.
- E prefixamos a função run com o nome de nossa crate.
- Agora será mais fácil adicionar testes a funcionalidade principal do programa.
- Link do código.

```
use std::env;
use std::process;
use minigrep::Config;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        println!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    println!("Searching for {}", config.query);
    println!("In file {}", config.file_path);

    if let Err(e) = minigrep::run(config) {
        println!("Application error: {e}");
        process::exit(1);
    }
}
```

# Desenvolvendo a Biblioteca de Funcionalidades com Desenvolvimento Guiado por Testes

- Vamos adicionar a lógica de pesquisa ao programa minigrep usando o processo de desenvolvimento orientado por teste (TDD) com as seguintes etapas:
  - Escrever um teste deve falhar e executá-lo para garantir que falhe pelo motivo esperado.
  - Escrever ou modificar apenas o suficiente de código para fazer o novo teste passar.
  - Refatorar o código que acabou de ser adicionado ou alterado e garantir que os testes continuem a passar.
  - Repetir a partir do passo 1!

- Escrever o teste antes de escrever o código que faz o teste passar ajuda a manter uma alta cobertura de testes ao longo do processo.

- Testaremos a implementação da funcionalidade que efetivamente fará a pesquisa pela string de consulta no conteúdo do arquivo e produzirá uma lista de linhas que correspondem à consulta. Adicionaremos essa funcionalidade em uma função chamada search.

# Escrevendo um teste de falha

- Em src/lib.rs, adicione um módulo de testes com uma função de teste.
- A função de teste especifica o comportamento que queremos que a função search tenha:
  - ela receberá uma consulta e o texto a ser pesquisado.
  - retornará apenas as linhas do texto que contém a consulta.
- O código do teste, ainda não será possível compilar

Exemplo:

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

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }
}
```

# Escrevendo um teste de falha

- Este teste procura pela string "duct".
- O texto que estamos pesquisando possui três linhas, apenas uma das quais contém "duct".
- Afirmamos que o valor retornado pela função search contém apenas a linha que esperamos.
- adicionaremos apenas código suficiente para fazer o teste ser executado, adicionando uma definição da função search que sempre retorna um vetor vazio.

Exemplo:

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

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }
}
```

# Escrevendo um teste de falha

- Adicionando a função search a src/lib.rs:

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

- Observe que precisamos definir uma vida útil explícita 'a na assinatura da função search e usar essa vida útil com o argumento contents e o valor de retorno.

- Os dados referenciados por uma fatia precisam ser válidos para que a referência seja válida; se o compilador presumir que estamos fazendo fatias de strings de query em vez de contents, ele fará suas verificações de segurança de forma incorreta.

# Escrevendo um teste de falha

- Se tentarmos executar os testes com “cargo test”, os testes irão falhar como o esperado.

```
   Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::one_result ... FAILED

failures:

---- tests::one_result stdout ----
thread 'tests::one_result' panicked at 'assertion failed: `(left == right)`
  left: `["safe, fast, productive."]`,
 right: `[]`', src/lib.rs:44:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace


failures:
    tests::one_result
```

# Escrevendo o código para passar o teste

- Para implementar a função search, nosso programa precisa seguir estes passos:
  - Iterar por cada linha do conteúdo.
  - Verificar se a linha contém a string que estamos procurando.
  - Se sim, adicioná-la à lista de valores que estamos retornando.
  - Se não, não fazer nada.
  - Retornar a lista de resultados correspondentes.

# Iterarando Através de Linhas com o Método lines

- O método lines é útil para lidar com a iteração linha a linha de string.

- O método lines retorna um iterador.

- lembre-se de que você viu essa forma de usar um iterador antes, no capítulo 3, onde usamos um loop for com um iterador para executar algum código em cada item em uma coleção.

Exemplo:

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```

# Pesquisando Cada Linha para a Consulta

- O código abaixo, adiciona a lógica que procura pela string.

- O código ainda não compila, pois não temos ainda nenhum retorno nessa função.

- As strings têm um método chamado contains, que verificar se a linha atual contém nossa string de consulta.

  Exemplo:

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

# Armazenando as linhas correspondentes

- Para finalizar esta função, precisamos de uma maneira de armazenar as linhas correspondentes que queremos retornar.
- Para isso, podemos criar um vetor mutável antes do loop for e chamar o método push para armazenar uma linha no vetor.
- Retornando o vetor no término do loop for. Link do código.

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

# Armazenando as linhas correspondentes

```
running 1 test
test tests::one_result ... ok

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

     Running unittests src/main.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

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

   Doc-tests minigrep

running 0 tests

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

# Utilizando a função search na função run

- Agora que a função search está funcionando e está testada, podemos utilizar ela na nossa função run.

- Precisamos passar o valor config.query e o conteúdo que run lê do arquivo para a função search.

- Em seguida, run deve imprimir cada linha retornada pela função search.

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    for line in search(&config.query, &contents) {
        println!("{line}");
    }

    Ok(())
}
```

# Utilizando a função search na função run

```
$ cargo run -- frog poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.38s
     Running `target/debug/minigrep frog poem.txt`
How public, like a frog

$ cargo run -- body poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
       Finished dev [unoptimized + debuginfo] target(s) in 0.0s
       Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!

$ cargo run -- monomorphization poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep monomorphization poem.txt`
```

# Trabalhando com variáveis de ambiente

- Opção para pesquisa sem diferenciação entre maiúsculas e minúsculas que o usuário pode ativar e desativar por meio de uma variável de ambiente.

- Uma nova função search_case_insensitive que será chamada quando a variável de ambiente tiver um valor.

- Adicionaremos um novo teste para a nova função search_case_insensitive e mudaremos o nome de nosso teste antigo de one_result para case_sensitive.

- Também modificamos o teste antigo para garantir que não quebremos acidentalmente a funcionalidade de pesquisa sensível a maiúsculas que já implementamos.

# Trabalhando com variáveis de ambiente

Exemplo:

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

    #[test]
    fn case_sensitive() {
    let query = "duct";
    let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

    assert_eq!(
        vec!["safe, fast, productive."],
        search(query, contents)
    );
    }
```

```
#[test]
fn case_insensitive() {
    let query = "rUsT";
    let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

    assert_eq!(
        vec!["Rust:", "Trust me."],
        search_case_insensitive(
            query,
            contents
        )
    );
    }
}
```

# Trabalhando com variáveis de ambiente

- Opção para pesquisa sem diferenciação entre maiúsculas e minúsculas que o usuário pode ativar e desativar por meio de uma variável de ambiente.

- Uma nova função search_case_insensitive que será chamada quando a variável de ambiente tiver um valor.

- Adicionaremos um novo teste para a nova função search_case_insensitive e mudaremos o nome de nosso teste antigo de one_result para case_sensitive.

- Também modificamos o teste antigo para garantir que não quebremos acidentalmente a funcionalidade de pesquisa sensível a maiúsculas que já implementamos.

# Implementando a função search_case_insensitive

- Na função search_case_insensitive que estamos prestes a adicionar, a consulta "rUsT" deve corresponder à linha contendo "Rust:" e à linha "Trust me."

- Devemos definir a função search_case_insensitive.

- Link do código

```
pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

# Implementando a função search_case_insensitive

- Primeiro, transformamos a string da consulta em minúsculas e a armazenamos em uma variável sombreada com o mesmo nome.
- to_lowercase lida somente com Unicode básico.
- a consulta agora é uma String em vez de uma fatia de string, porque chamar to_lowercase cria novos dados em vez de referenciar dados existentes.
- Também utilizamos to_lowercase para todas as linhas de contents.

```
pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

# Implementando a função search_case_insensitive

- Executando os testes, eles devem passar.

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 1.33s
     Running unittests src/lib.rs
(target/debug/deps/minigrep-9cd200e5fac0fc94)

running 2 tests
test tests::case_insensitive ... ok
test tests::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

# Implementando a função search_case_insensitive

- Vamos chamar a nova função search_case_insensitive na função run
- Primeiro, vamos adicionar uma opção de configuração à estrutura Config para alternar entre essas duas configurações de pesquisa.
- A função run precisa verificar o valor do campo ignore_case, para decidir se deve chamar a função search ou a função search_case_insensitive.

```
pub struct Config {
    pub query: String,
    pub file_path: String,
    pub ignore_case: bool,
}
```

# Implementando a função search_case_insensitive

- Link do código

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.file_path)?;

    let results = if config.ignore_case {
        search_case_insensitive(&config.query, &contents)
    } else {
        search(&config.query, &contents)
    };

    for line in results {
        println!("{line}");
    }

    Ok(())
}
```

# Implementando a função search_case_insensitive

- Finalmente, precisamos verificar a variável de ambiente.
- As funções para trabalhar com variáveis de ambiente estão no módulo env na biblioteca padrão, então trazemos esse módulo para o escopo no início de src/lib.rs.
- Em seguida, usaremos a função var do módulo env para verificar se algum valor foi definido para uma variável de ambiente chamada IGNORE_CASE.

# Implementando a função search_case_insensitive

Exemplo:

```
use std::env;
// --snip--
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();
        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

# Implementando a função search_case_insensitive

- Para definir de ignore_case, chamamos a função env::var e passamos a ela o nome da variável de ambiente IGNORE_CASE.

  ```
  let ignore_case = env::var("IGNORE_CASE").is_ok();
  ```

- A função env::var retorna um Result que será do tipo Ok, contendo o valor da variável de ambiente, se a variável de ambiente não estiver definida retorna Err.

- Estamos usando o método is_ok no Result para verificar se a variável de ambiente está definida, para o programa realizar uma pesquisa insensível.

- Não estamos nos importamos com o valor da variável de ambiente, apenas se ela está definida ou não, então estamos verificando is_ok em vez de usar unwrap, expect ou qualquer outro método que vimos em Result.

# Implementando a função search_case_insensitive

```
$ cargo run -- to poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
       Finished dev [unoptimized + debuginfo] target(s) in 0.0s
       Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!


$ IGNORE_CASE=1 cargo run -- to poem.txt
       Finished dev [unoptimized + debuginfo] target(s) in 0.00s
       Running `target/debug/minigrep to poem.txt`
Searching for to
In file poem.txt
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

# Implementando a função search_case_insensitive

- Caso esteja utilizando PowerShell, o comando seria:

  PS> $Env:IGNORE_CASE=1; cargo run -- to poem.txt

- Isso deixará a variável IGNORE_CASE definida até o encerrar o PowerShell.

- Para remover a variável de ambiente:

  PS> Remove-Item Env:IGNORE_CASE

# Escrevendo Mensagens de Erro para Erro Padrão em Vez de Saída Padrão

- Atualmente, estamos escrevendo todas as nossas saídas no terminal usando a macro println!.

- Na maioria dos terminais, existem dois tipos de saída: saída padrão (stdout) para informações gerais e saída de erro padrão (stderr) para mensagens de erro.

- Essa distinção permite que os usuários escolham direcionar a saída bem-sucedida de um programa para um arquivo, mas ainda imprimem mensagens de erro na tela.

- A macro println! é capaz apenas de imprimir na saída padrão.

- A biblioteca padrão fornece a macro eprintln! que imprime erro padrão

# Verificando onde os erros são escritos

- Faremos um redirecionamento da corrente de saída padrão para um arquivo, enquanto causamos intencionalmente um erro.

- Em programas de linha de comando é esperado que ele envie mensagens de erro para a saída de erro padrão, para que possamos ver as mensagens de erro na tela e no arquivo que redirecionamos a saída padrão para um arquivo.

- Nosso programa atualmente salva a saída de mensagens de erro em um arquivo.

# Verificando onde os erros são escritos

- Podemos utilizar o comando abaixo, para verificar isso. O “>” indica para o Rust salvar a saída padrão em um arquivo.

```
$ cargo run > output.txt
```

- Poderemos perceber que o erro não é imprimido na tela.
- Abrindo o arquivo, podemos ver a mensagem de erro salvo nele.

```
Problem parsing arguments: not enough arguments
```

- Precisamos mudar a mensagem de erro que atualmente é uma saída padrão para um erro padrão.

# Imprimindo mensagens de erro padrão

- Devido à refatoração que fizemos anteriormente neste capítulo, todo o código que imprime mensagens de erro está na função main.

- A biblioteca padrão fornece a macro eprintln! que imprime na corrente de erro padrão. Então vamos substituir os println! por eprintln!.

Exemplo:

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {e}");
        process::exit(1);
    }
}
```

# Imprimindo mensagens de erro padrão

- Agora o programa deve imprimir o erro na tela e o arquivo deve estar vazio.

```
$ cargo run > output.txt
Problem parsing arguments: not enough arguments
```

- Se utilizarmos o comando “$ cargo run -- to poem.txt > output.txt”, esperamos que o resultado da pesquisa esteja salvo no arquivo.

```
Are you nobody, too?
How dreary to be somebody!
```

- Isso demonstra que estamos utilizando saídas padrões e erros padrões de forma apropriada.

# Referências

Slides Baseados no Capítulo 12 do Rust Book
https://doc.rust-lang.org/book/ch12-00-an-io-project.html