Rust

Linguagem

Cargo & Bibliotecas

MELHOR 100

BY KANDO TOMOYA

Index

Chapter 1 Introduction

1. Purpose

Caro leitor,

É com grande satisfação que apresentamos um recurso indispensável para todos aqueles in teressados em aprofundar seus conhecimentos na linguagem de programação Rust.

Este guia prático foi desenvolvido para ajudar você a dominar as ferramentas e bibliotecas mais utilizadas em Rust, através de uma abordagem direta e eficiente. Em cada capítulo, pr opomos desafios que simulam situações reais, acompanhados de soluções detalhadas que não só resolvem os problemas propostos, mas também explicam os conceitos fundamentais por trás de cada decisão.

Acreditamos que, ao final deste livro, você terá adquirido uma compreensão sólida dos com ponentes mais importantes do ecossistema Rust, preparando-o para enfrentar projetos mais complexos com confiança e habilidade.

Desejamos a você uma jornada enriquecedora e produtiva.

Boa leitura!

Chapter 2 standard library

1. std::vec::Vec

O Vec<T> é uma estrutura de dados que representa uma lista ou array que pode crescer di namicamente em Rust. É uma das estruturas mais usadas quando se trata de armazename nto de uma coleção de valores do mesmo tipo.

Ex:std::vec::Vec

```
fn main() {
    let mut numeros = Vec::new(); // Cria um vetor vazio
    numeros.push(1); // Adiciona o valor 1 ao vetor
    numeros.push(2); // Adiciona o valor 2 ao vetor
    println!("Números: {:?}", numeros);
}
```

```
Números: [1, 2]
```

O exemplo acima demonstra como criar e manipular um vetor em Rust usando Vec<T>. Inic ializamos um vetor vazio com Vec::new(), o qual não contém elementos inicialmente. Atravé s do método push, adicionamos valores ao vetor. O método push adiciona um elemento ao f inal do vetor e aumenta seu tamanho quando necessário. A macro println! é utilizada para i mprimir o vetor na tela, onde {:?} é um marcador que permite a visualização de estruturas d e dados como vetores para fins de depuração.

2. std::collections::HashMap

HashMap<K, V> é uma estrutura de dados que implementa um mapa de hash, útil para arm azenar e acessar dados através de chaves únicas. É amplamente utilizado para operações de busca rápida onde a ordem dos elementos não é importante.

Ex:std::collections::HashMap

```
use std::collections::HashMap;
fn main() {
    let mut pontuacoes = HashMap::new();
    pontuacoes.insert("Alice", 50);
    pontuacoes.insert("Bob", 60);
    println!("Pontuações: {:?}", pontuacoes);
}
```

```
Pontuações: {"Bob": 60, "Alice": 50}
```

No exemplo acima, criamos um HashMap para armazenar as pontuações de jogadores em um jogo. O método insert é usado para adicionar um par chave-valor ao mapa, onde "Alice" e "Bob" são as chaves e 50 e 60 são seus respectivos valores. O HashMap é particularment e útil em casos onde a velocidade de acesso aos dados é crítica, pois permite um acesso d e tempo quase constante às suas entradas. A ordem de impressão das chaves no HashMap pode variar, pois a estrutura não garante a ordem dos elementos.4

3. std::slice

Permite manipular fatias de arrays de forma segura e eficiente, sem ter que manipular a me mória diretamente.

Ex:std::slice

```
let arr = [1, 2, 3, 4, 5];
let fatia = &arr[1..4]; // Obtém uma fatia do array
for i in fatia {
    println!("{}", i);
}
```

```
2
3
4
```

No exemplo acima, arr é um array de inteiros. A sintaxe &arr[1..4] é usada para criar uma fat ia que inclui os elementos do índice 1 ao 3 (o índice 4 não é incluído). As fatias em Rust são seguras porque o compilador garante que os índices usados estão dentro dos limites do arr ay original, evitando acessos a memória não alocada. Além disso, usar fatias ajuda a escrev er código mais limpo e eficiente, pois você não precisa criar cópias dos dados que já estão na memória.

4. std::ptr

Fornece funções para manipulação manual de ponteiros na memória, permitindo operações de baixo nível.

Ex:std::ptr

```
let mut x = 10;
let p = &mut x as *mut i32; // Converte a referência mutável em um ponteiro mutável
unsafe {
    *p = 20; // Acesso direto à memória para modificar o valor
    println!("Valor de x: {}", x);
}
```

```
Valor de x: 20
```

Neste código, x é uma variável inteira comum. Usamos &mut x as *mut i32 para converter u ma referência mutável em um ponteiro mutável que aponta para x. Esta conversão é comu m quando precisamos realizar operações que não são cobertas pelo sistema de segurança de tipos de Rust. A operação de dereferenciação *p = 20 dentro do bloco unsafe permite mo dificar diretamente o valor de x através do ponteiro. Usar blocos unsafe é necessário para o perações de ponteiros, pois elas podem violar as garantias de segurança de Rust, como ac essos concorrentes ou violações de memória.4

5. std::mem

Esta biblioteca fornece funções para manipular a memória de forma segura e eficiente em R ust, como trocar valores entre variáveis sem usar uma variável temporária adicional.

Ex:std::mem

```
fn main() {
    let mut x = 5;
    let mut y = 8;
    // Usando std::mem::swap para trocar os valores de x e y
    std::mem::swap(&mut x, &mut y);
    println!("x: {}, y: {}", x, y);
}
```

```
x: 8, y: 5
```

No código acima, utilizamos a função swap do módulo std::mem para trocar os valores das variáveis x e y. Esta função é especialmente útil quando queremos trocar valores sem criar uma variável temporária adicional, o que economiza recursos e mantém o código mais limp o. A função swap toma dois argumentos por referência mutável (&mut) para os valores que serão trocados. Isso significa que ambos os valores devem ser mutáveis, ou seja, devem pe rmitir alterações durante a execução do programa.