A20.Reestruturação Corporativa

10 Pontos Possíveis

03/11/2025

Tentativa 1

Em progresso

PRÓXIMO: Enviar tarefa

Adicionar comentário

Tentativas ilimitadas permitidas

03/11/2025

Detalhes

Atividade Prática: Reestruturação Corporativa com `Rc`, `RefCell` e `Weak`

Objetivo: Diagnosticar e corrigir um vazamento de memória (*memory leak*) em uma estrutura de dados cíclica, utilizando **Smart Pointers**. O foco é a aplicação prática de `std::rc::Weak` para quebrar ciclos de referência criados com `std::rc::Rc` e `std::cell::RefCell`. Esta atividade reforça a diferença crucial entre posse compartilhada (forte) e referências de observação (fraca).

Contexto: Você é um engenheiro de software sênior encarregado de manter um sistema de RH. Foi descoberto um bug crítico: quando um departamento da empresa é dissolvido, o sistema falha em liberar a memória associada a ele e a seus funcionários. Estes registros permanecem como "departamentos fantasmas", consumindo recursos preciosos. Sua tarefa é investigar a estrutura de dados, identificar o ciclo de referência que causa o problema e implementar a solução usando `Weak`.

Esta atividade parte de um código inicial com o bug já presente, permitindo que você se concentre na análise e na solução do problema. Você não precisa de um template, apenas de um projeto Rust novo (`cargo new reestrutura_corp`).

Requisitos Funcionais:

1. Estrutura de Dados Inicial:

- As estruturas de dados que modelam o sistema são:
  - `Funcionario`: Contém o nome do funcionário e uma referência para o `Departamento` ao qual pertence.
  - `Departamento`: Contém o nome do departamento e uma lista de todos os funcionários que são seus membros.
- **O Problema:** No código inicial, ambas as referências (do departamento para seus funcionários e dos funcionários de volta para seu departamento) são implementadas com `Rc<T>`, criando um ciclo de referências fortes.

2. Diagnóstico do Vazamento de Memória:

- O código inicial deve conter implementações do trait `Drop` para ambas as structs (`Funcionario` e `Departamento`).
- Cada implementação de `drop` imprime uma mensagem no console (e.g., "Dissolvendo departamento..." ou "Notificação de desligamento para...").
- Ao executar o programa, você observará que essas mensagens **não são exibidas**, mesmo quando as variáveis que contêm o departamento saem de escopo. Isso prova visualmente o vazamento de memória.

3. A Missão: Corrigindo o Ciclo com `std::rc::Weak`:

- Sua tarefa principal é modificar a estrutura de dados para quebrar o ciclo de referência.
- A relação de posse define que um `Departamento` "possui" seus `Funcionarios`. Portanto, a referência do funcionário de volta para o departamento **não deve ser de posse**.

4. Verificação da Solução:

- Após aplicar a correção, execute o programa novamente.
- O resultado esperado é que as mensagens da trait `Drop` agora **sejam exibidas** no console, na ordem correta: primeiro o departamento é dissolvido, e em seguida, os funcionários são desligados.
- Isso confirma que o ciclo foi quebrado e que o gerenciamento de memória do Rust está funcionando como esperado.

Código Inicial (Com o Bug):

Use este código como ponto de partida para a sua atividade.

```
use std::rc::Rc;
use std::cell::RefCell;

// Um funcionário pertence a um departamento.
// A referência ao departamento é forte! Este é o problema.
struct Funcionario {
```

```
    nome: String,
    departamento: Rc<Departamento>,
}

impl Drop for Funcionario {
    fn drop(&mut self) {
        println!("Notificação de desligamento para {}.", self.nome);
    }
}

// Um departamento tem uma lista de seus funcionários (membros).
struct Departamento {
    nome: String,
    membros: RefCell<Vec<Rc<Funcionario>>>,
}

impl Drop for Departamento {
    fn drop(&mut self) {
        println!("Dissolvendo o departamento: {}!", self.nome);
    }
}

fn main() {
    println!("--- Início da Simulação ---");

    let depto_inovacao = Rc::new(Departamento {
        nome: "Inovação".to_string(),
        membros: RefCell::new(vec![]),
    });

    let alice = Rc::new(Funcionario {
        nome: "Alice".to_string(),
        departamento: Rc::clone(&depto_inovacao),
    });

    let bob = Rc::new(Funcionario {
        nome: "Bob".to_string(),
        departamento: Rc::clone(&depto_inovacao),
    });

    depto_inovacao.membros.borrow_mut().push(Rc::clone(&alice));
    depto_inovacao.membros.borrow_mut().push(Rc::clone(&bob));

    println!("Contagem de referências para o depto: {}", Rc::strong_count(&depto_inovacao));
    println!("--- Fim da Simulação ---");
    // A variável `depto_inovacao` sai de escopo, mas a memória não é liberada.
}
```

Dicas e Considerações Técnicas:

- **Dependências:** Para esta atividade específica, você não precisará de nenhuma crate externa em seu `Cargo.toml`.
- `Drop` **como Ferramenta de Diagnóstico:** Entenda que o trait `Drop` é uma ferramenta poderosa para depurar o ciclo de vida de seus dados. Se a mensagem em `drop` não aparece, é um sinal claro de que seu objeto nunca foi liberado.
- **Acessando Dados Através de** `Weak`**:** Depois de corrigir o bug, como um `Funcionario` poderia acessar o nome de seu departamento? A referência `Weak` não permite acesso direto. É necessário "promovê-la" temporariamente para uma referência forte usando o método `upgrade()`, que retorna um `Option<Rc<Departamento>>`.

```
// Dentro de um método de `Funcionario`:
if let Some(depto_forte) = self.departamento.upgrade() {
    println!("Meu departamento é: {}", depto_forte.nome);
} else {
    println!("Meu departamento já foi dissolvido.");
}
```

Lembrete Importante sobre Propriedade e Referência

A escolha entre `Rc` e `Weak` é uma decisão sobre o **modelo de propriedade** dos seus dados. Use `Rc` quando uma entidade precisa compartilhar a *posse* e garantir que outra entidade permaneça viva. Use `Weak` quando uma entidade precisa apenas *observar* ou referenciar outra, sem ter qualquer poder sobre seu ciclo de vida. Em uma hierarquia (como um departamento e seus membros), a relação pai-filho é tipicamente forte (`Rc`), enquanto a referência de volta do filho para o pai deve ser fraca (`Weak`) para evitar ciclos.

Instruções para Entrega

- Certifique-se de que seu código corrigido compila sem erros (`cargo build`) e que, ao executar (`cargo run`), ele exibe as mensagens do `Drop`, confirmando que o vazamento de memória foi solucionado.
- Antes de compactar o projeto, execute o comando `cargo clean` no terminal, dentro do diretório do seu projeto. Isso remove os arquivos de compilação temporários e reduz significativamente o tamanho final do arquivo.
- Compacte o diretório completo do projeto (que deve incluir a pasta `src/` e os arquivos `Cargo.toml` e `Cargo.lock`) em um único arquivo `.zip`.
- Nomeie o arquivo de forma clara (e.g., `seu_nome-atividade_weak.zip`) e submeta-o através da plataforma de ensino, conforme as orientações da atividade.

Visualizar rubrica

Atividade Prática: Reestruturação Corporativa com Weak						
Critérios	**Avaliações**					**Pontos**
Correção da Estrutura de Dados com `Weak`	**A struct `Funcionario` foi corretamente alterada para usar `Weak<Departame quebrando o ciclo de referência de forma ideal e limpa.** 4 pts	**A alteração para `Weak` foi feita, mas com complexidade desnecessária (e.g., `Option<Weak<T>>` quando não era preciso).** 3 pts	**Houve uma tentativa de alterar a struct, mas o tipo utilizado está incorreto ou a lógica para quebrar o ciclo não funciona.** 2 pts	**A alteração foi apenas parcial e contém erros significativos de compilação ou de lógica.** 1 pts	**A struct `Funcionario` não foi alterada, mantendo o ciclo de referência original.** 0 pts	/4 pts
Criação da Referência Fraca com `Rc::downgrade`	**O método `Rc::downgrade()` foi aplicado corretamente na criação de cada `Funcionario` para gerar a referência `Weak`.** 4 pts	**O uso de `Rc::downgrade()` está funcional, mas aplicado de forma subótima ou em um local pouco intuitivo.** 3 pts	**Houve uma tentativa de usar `Rc::downgrade()` mas foi aplicada no local errado, causando erro de compilação ou falha lógica.** 2 pts	**O conceito de `downgrade` foi mal interpretado, levando a uma implementação incorreta.** 1 pts	**O código não foi ajustado para usar `Rc::downgrade()`, impossibilitando a criação da referência fraca.** 0 pts	/4 pts
Diagnóstico com o Trait `Drop`	**O trait `Drop` foi corretamente implementado em ambas as structs (`Funcionario` e `Departamento`) com mensagens claras para o diagnóstico.** 4 pts		**O trait `Drop` foi implementado em apenas uma das structs, ou as mensagens são ambíguas, dificultando a verificação.** 2 pts		**O trait `Drop` não foi implementado, impossibilitando a verificação visual da correção do vazamento de memória.** 0 pts	/4 pts
Resultado Final e Funcionalidade	**O programa corrigido compila e executa perfeitamente, exibindo as mensagens de `Drop` para o departamento e todos os funcionários.** 4 pts		**O programa compila, mas não resolve completamente o vazamento de memória (as mensagens de `Drop` não aparecem ou aparecem incorretamente).** 2 pts		**O programa final não compila devido a erros na implementação ou não foi alterado.** 0 pts	/4 pts
Qualidade e Organização do Código	**O código está bem formatado (compatível com `cargo fmt`), os nomes de variáveis são claros e a lógica é fácil de seguir.** 4 pts	**O código funciona, mas a formatação é inconsistente ou os nomes são pouco claros.** 3 pts	**O código é funcional, mas está desorganizado, tornando a leitura e a compreensão difíceis.** 2 pts		**O código não apresenta um padrão mínimo de organização e clareza.** 0 pts	/4 pts

Escolher um tipo de envio

Carregar | Office 365 | Mais

Arraste um arquivo aqui, ou

Selecione um arquivo para carregar

Arquivo permitido: ZIP

ou

Arquivos do Canvas

Enviar tarefa