Exercícios

1. Movendo e Atualizando Dados em Structs

Crie uma `struct` chamada `ContaBancaria` com os campos `nome_titular`, `saldo`, e `numero_conta`. Instancie duas contas bancárias, uma para 'Alice' e outra para 'Bob'. Simule uma transferência de saldo movendo a posse dos valores entre as contas. Utilize a sintaxe de atualização de structs para transferir o saldo de uma conta para outra.

2. Gerenciamento de Empréstimos Mutáveis em Struct

Crie uma `struct` chamada `Livro` com os campos `titulo`, `autor` e `quantidade_paginas`. Implemente um método `adicionar_paginas` que aceita uma referência mutável ao livro e adiciona páginas ao campo `quantidade_paginas`. O método deve garantir que não se possa adicionar páginas de maneira inválida (como números negativos). No programa principal, faça diversas alterações na struct e garanta que o valor seja atualizado corretamente.

3. Enum Complexo com Parâmetros

Crie um `enum` chamado `EventoSistema` que tenha as variantes `Conexao(String)`, `Erro(u32, String)`, e `Desconexao(String, u32)`. No programa principal, use `match` para lidar com cada evento, exibindo mensagens de status diferentes com base no evento. Cada variante deve ter um comportamento diferenciado.

4. Função com Retorno de Enum `Option`

Implemente uma função chamada `encontrar_numero` que recebe um vetor de inteiros e um número alvo. A função deve retornar um `Option<usize>`, indicando a posição do número no vetor. Utilize `match` para lidar com o caso em que o número é encontrado ou não, imprimindo uma mensagem apropriada.

5. Referências Imutáveis e Mutáveis Simultâneas

Crie uma `struct` chamada `Retangulo` com os campos `largura` e `altura`. Escreva um programa que simultaneamente passa uma referência imutável da `largura` e uma referência mutável da `altura` para diferentes funções. A função que recebe a referência mutável deve alterar o valor da altura. Mostre como o Rust previne a violação das regras de empréstimo com referências simultâneas.

6. Transferência de Ownership com Vetores

Implemente um programa que cria dois vetores de strings representando listas de compras de duas pessoas. A primeira pessoa decide mover todos os seus itens para a lista da segunda pessoa. Após o movimento, o primeiro vetor deve estar vazio. Utilize as regras de *ownership* para garantir que a operação seja segura e eficiente.

7. Enumeração com Métodos e Atualizações

Crie um `enum` chamado `Jogo` que tenha as variantes `Iniciado(u32)`, `EmProgresso(u32, u32)` e `Finalizado(u32)`, representando o status de um jogo com pontuações. Implemente métodos para iniciar o jogo, atualizar a pontuação e finalizar o jogo. Utilize `match` no programa principal para gerenciar a lógica do jogo.

8. Função que Aceita e Retorna `Structs` com Empréstimo

Implemente uma função `juntar_nomes` que recebe referências imutáveis para duas `structs` chamadas `Pessoa`, cada uma contendo o campo `nome`. A função deve retornar uma nova `Pessoa` com o nome concatenado. O programa principal deve testar a função sem mover as `structs` originais.

9. Ciclo de Empréstimo e Match com Enum

Crie um `enum` chamado `OperacaoBancaria` que tenha as variantes `Deposito(f64)`, `Saque(f64)` e `ConsultaSaldo`. Crie uma função que recebe uma referência mutável para o saldo de uma conta e um `OperacaoBancaria`, e realiza a operação correta, utilizando `match`. Garanta que o saldo não possa ficar negativo e imprima mensagens apropriadas para cada operação.

10. Implementando um Sistema de Pedidos com Enum e Struct

Crie uma `struct` chamada `Pedido` que tenha os campos `numero_pedido`, `cliente`, e `itens` (um vetor de strings). Em seguida, crie um `enum` chamado `StatusPedido` com as variantes `Processando`, `Enviado(u32)`, e `Concluido`. Escreva um programa que atualiza o status de um pedido usando `match` e imprime uma mensagem apropriada para cada estado.