Roteiro de Estudo Aprofundado: A Alma de um Shell Interativo

Visão Geral

Nosso objetivo é construir o coração do Minishell: o loop que lê os comandos do usuário. Para fazer isso de forma profunda, não vamos apenas usar as funções, vamos primeiro entender *o problema* que elas resolvem. Depois, vamos dissecar cada função, entendendo seu contrato, suas responsabilidades e seus casos especiais, para só então integrá-las ao nosso projeto.

O que é um Loop de Shell? O coração de qualquer shell interativo é um ciclo infinito conhecido como REPL (Read-Evaluate-Print Loop - Ler, Avaliar, Imprimir, Repetir):

Read (Ler): Exibe um prompt (ex: minishell$) e aguarda o usuário digitar um comando.

Evaluate (Avaliar): Analisa e executa o comando digitado.

Print (Imprimir): Mostra o resultado do comando.

Loop (Repetir): Volta ao passo 1.

Módulo 1: O Problema - Por que Ler Comandos é um Desafio?

Objetivo: Compreender por que funções simples de leitura de entrada, como `scanf` ou `fgets`, são insuficientes para um shell de qualidade e por que uma biblioteca como a `readline` é necessária.

Exercício 1.1: A Experiência de Usuário Mínima

Antes de usar a ferramenta certa, vamos sentir na pele as limitações das ferramentas mais básicas.

Reflexão Teórica (não precisa codificar, apenas pensar):
Imagine que você construiu seu loop de shell usando `fgets` para ler a entrada do usuário. Pense em como você usa um terminal de verdade no seu dia a dia.
* O que acontece se você digita um comando e percebe um erro no meio dele? Como você volta para consertar? `fgets` permitiria isso?
* O que você faz quando quer executar o mesmo comando de três minutos atrás?
* Como as teclas de seta, `Home`, `End` e `Delete` funcionam no seu terminal?

Ponto de Consolidação:
Qual é a diferença fundamental entre simplesmente "ler uma linha de texto" e fornecer uma "experiência de edição de linha de comando"? O que você concluirá ser o principal problema que a biblioteca `readline` se propõe a resolver?

Módulo 2: A Ferramenta Principal - Dissecando a `readline`

Objetivo: Dominar o funcionamento da função `readline`, focando em seu "contrato": o que ela espera receber, o que ela promete devolver e quais responsabilidades ela transfere para você, o programador.

Exercício 2.1: O Contrato da `readline`

Vamos analisar a assinatura da função: `char *readline(const char *prompt);`

Perguntas-guia para seu estudo:

1. **O Argumento (`const char *prompt`):** O que é um "prompt"? Qual é o seu propósito em uma interface de linha de comando? Experimente passar strings diferentes (ex: `"minishell> "`, `"$ "`, ou até `NULL`) para a função em um programa de teste e observe o resultado.

O Retorno (`char *`): O `*` indica que a função retorna um ponteiro. Isso tem uma implicação direta sobre o gerenciamento de memória.

- Quem é o responsável por alocar a memória para a string que o usuário digita? É você ou é a função `readline`?

- Consequentemente, quem é o responsável por **liberar** essa memória depois que ela não for mais necessária? Qual função em C você **sempre** deve chamar em par com uma alocação de memória bem-sucedida de `readline`?

3. **O Caso Especial (Retorno `NULL`):** Um ponteiro pode ser `NULL`. Em que situação específica a `readline` retorna `NULL`? (Dica: pense em como o usuário sinaliza "fim da entrada" em um terminal). Como o seu loop principal deve interpretar um retorno `NULL`?

Ponto de Consolidação:
Descreva em suas palavras o "contrato" completo da função `readline`. Se você fosse

explicar para um colega o que ele precisa saber para usar `readline` corretamente e sem vazar memória, o que você diria?

Módulo 3: A Ferramenta Auxiliar - Dando Memória ao Shell

Objetivo: Entender o propósito da função `add_history` e, mais importante, como ela se integra de forma segura e lógica ao ciclo de vida da memória gerenciada pela `readline`.

Exercício 3.1: O Ciclo de Vida do Histórico

A função é `void add_history(const char *line);`.

Perguntas-guia para seu estudo:

1. **O Propósito:** Qual problema de experiência de usuário a `add_history` resolve? Como ela colabora com a `readline` para criar essa funcionalidade?

   A Ordem das Operações: Dentro do seu loop, você terá no mínimo três operações importantes:

   - `input = readline(...)`
   - `free(input)`
   - `add_history(input)`

   Qual é a única sequência lógica e correta para essas três operações? Pense no que aconteceria se você liberasse a memória de `input` *antes* de tentar adicioná-la ao histórico.

3. **O Filtro de Qualidade:** Seria útil para o usuário se o histórico ficasse cheio de linhas em branco (quando ele apenas pressiona Enter)? Que verificação simples você pode fazer na string `input` *antes* de chamar `add_history` para garantir que apenas comandos significativos sejam salvos?

Ponto de Consolidação:
Desenhe o fluxo lógico de uma única iteração do seu loop principal. Use caixas e setas se ajudar. Onde cada decisão (o usuário digitou `Ctrl+D`? a linha está em branco?) e cada ação (`readline`, `add_history`, `free`) se encaixam para formar um ciclo robusto e sem vazamentos de memória?