

# Roadmap de Estudo: A Integração de Sinais e `readline`

Este roteiro guiará você pela jornada de descoberta para construir a solução de manipulação de `Ctrl+C` para o seu projeto. O objetivo não é seguir uma receita, mas entender os problemas em cada etapa, explorar as ferramentas e, através de provocações, chegar à arquitetura elegante e funcional que você mesmo desenvolveu.

---

## Módulo 1: Fundamentos Isolados - As Ferramentas

**Objetivo:** Compreender o comportamento básico de `readline` e `sigaction` de forma independente, antes de tentar combiná-los.

### Exercício 1.1: O Loop `readline` Básico

- **Foco:** O contrato de `readline`.

**Tarefa:** Crie um programa com um loop `while(1)` que chama `input = readline(...)`. Implemente a lógica correta para:

1. Detectar `Ctrl+D` (quando `readline` retorna `NULL`).
2. Liberar a memória de `input` com `free()` a cada iteração bem-sucedida.

- **Provocação:** Qual a diferença entre o que `readline` retorna quando o usuário digita "enter" (linha vazia) e quando ele pressiona `Ctrl+D`? Por que a gestão de memória é crucial aqui?

### Exercício 1.2: A Captura de Sinal Básica

- **Foco:** O conceito de `sigaction`.
  - **Tarefa:** Crie um programa que não usa `readline`, apenas um loop `while(1) { sleep(1); }`. Use `sigaction` para registrar um handler para `SIGINT`. O handler deve apenas usar `write` para imprimir uma mensagem ("SIGINT capturado!").
  - **Provocação:** O que acontece se você pressionar `Ctrl+C`? O programa termina? Por que registrar um handler, mesmo que vazio, já altera o comportamento padrão do sinal?
-

## Módulo 2: O Problema da Comunicação Entre Arquivos

**Objetivo:** Entender por que e como compartilhar o estado de um sinal entre `signal.c` e `main.c`, explorando as abordagens erradas para solidificar o conceito da correta.

### Exercício 2.1: A Tentativa com `static` no Header (O Erro)

- **Foco:** O escopo da palavra-chave `static`.
- **Tarefa:** Estructure seu código em três arquivos (`main.c`, `signal.c`, `signal.h`). Declare sua variável global em `signal.h` usando `static volatile sig_atomic_t g_signal_status = 0;`. Faça o handler em `signal.c` modificar essa variável. Faça o `main.c` ler essa variável.
- **Provocação:** O `main` consegue ver a mudança que o handler fez? Se não, por que não? O que a palavra-chave `static` faz quando usada em um arquivo `.h` que é incluído por múltiplos arquivos `.c`? Você concluirá que esta abordagem cria cópias privadas e independentes da variável.

### Exercício 2.2: A Solução Encapsulada (O Acerto)

- **Foco:** Encapsulamento e interface pública.

**Tarefa:** Corrija o problema anterior sem usar `extern`.

1. Mova a definição da variável global para `signal.c` e a declare como `static` para torná-la privada àquele arquivo.
  2. Em `signal.c`, crie duas novas funções: `int get_g_signal_status(void)` que retorna o valor da variável, e `void reset_g_signal_status(void)` que a redefine para 0.
  3. Em `signal.h`, declare apenas os protótipos dessas duas novas funções.
- **Provocação:** Como essa abordagem resolve o problema de comunicação sem expor a variável global diretamente? Qual a vantagem de forçar o `main.c` a interagir com o estado do sinal através de uma interface controlada (getters/setters)?
-

## Módulo 3: A Primeira Tentativa de Integração (E Por Que Falha)

**Objetivo:** Tentar a abordagem mais "óbvia" para integrar os dois sistemas e entender, na prática, por que ela não funciona, revelando a natureza bloqueante da `readline`.

### Exercício 3.1: O Loop `main` "Inteligente"

- **Foco:** A natureza bloqueante de `readline`.
- **Tarefa:** Integre os conceitos. Em `main.c`, dentro do seu loop `while(1)`, chame `readline` e, *imediatamente após*, verifique o estado do sinal com `if (get_g_signal_status() == SIGINT)`.
- **Provocação:** Execute o programa e pressione `Ctrl+C`. O seu `if` é acionado instantaneamente? Ou apenas depois que você pressiona Enter? O que isso prova sobre a função `readline`? Ela devolve o controle para o seu loop quando um sinal é recebido, ou ela continua bloqueada esperando por entrada de texto?

---

## Módulo 4: A Revelação - A Ferramenta Correta para o Trabalho

**Objetivo:** Descobrir e dominar a ferramenta `rl_event_hook` como a solução para a falha observada no módulo anterior.

### Exercício 4.1: O Gancho de Eventos (`rl_event_hook`)

- **Foco:** Executando seu código *dentro* do loop da `readline`.
- **Tarefa:** Crie uma nova função `int ft_event_hook(void)`. Na `main`, antes do loop, atribua o endereço desta função à variável global da `readline`: `rl_event_hook = ft_event_hook;`. Dentro do seu hook, adicione um `printf` para ver com que frequência ele é chamado.
- **Provocação:** O seu `printf` é chamado continuamente enquanto `readline` está esperando por entrada? O que isso significa? Você acabou de encontrar uma "janela" para executar seu próprio código enquanto `readline` está bloqueada.

### Exercício 4.2: Forçando o Retorno com `rl_done`

- **Foco:** Tomando o controle do loop da `readline`.
- **Tarefa:** Modifique seu `ft_event_hook`. Faça-o verificar `if (get_g_signal_status() != 0)`. Se a condição for verdadeira, atribua `rl_done = 1;`. A variável `rl_done` é uma flag global da própria `readline` que, quando definida como 1, a instrui a parar de esperar e retornar imediatamente.
- **Provocação:** Pressione `Ctrl+C`. O `readline` agora retorna instantaneamente? O que acontece com o valor de `input` no `main`? Ele é `NULL` ou é um ponteiro para a linha que estava sendo digitada? (Resposta: é a linha parcial, que precisa ser liberada).

---

## Módulo 5: A Síntese Final - A Arquitetura Elegante

**Objetivo:** Refinar a lógica dentro do `event_hook` e do `main` para chegar à sua solução final, limpa e funcional.

### Exercício 5.1: Centralizando a Reação

- **Foco:** Onde a lógica de reação ao sinal deve morar?

**Tarefa:** Mova a lógica para dentro do `ft_event_hook`. Além de definir `rl_done = 1`, faça-o também:

1. Usar `write(1, "^C", 2);` para fornecer o feedback visual que você queria.
2. Chamar `reset_g_signal_status();` para limpar a flag imediatamente.

- **Provocação:** O que acontece com o seu `main` agora? Ele ainda precisa de um `if` para checar o sinal? Ou suas responsabilidades foram simplificadas para apenas chamar `readline`, checar `Ctrl+D` e liberar a memória? Você concluirá que o `main` se torna muito mais limpo.

### Exercício 5.2 (Final): A Gestão de Memória Correta

- **Foco:** Prevenir vazamentos de memória.
- **Tarefa:** Analise seu loop `main` final. A função `readline`, mesmo quando interrompida pelo `event_hook`, retorna um ponteiro para a linha parcialmente digitada (que foi alocada com `malloc`).

- **Provocação:** Em quais cenários a variável `input` contém um ponteiro que precisa ser liberado com `free()`? Apenas no caso de entrada bem-sucedida? Ou também no caso de interrupção por `Ctrl+C`? Como seu loop `main` deve ser estruturado para garantir que `free(input)` seja chamado em todos os casos necessários, mas evitado quando `input` for `NULL` (caso `Ctrl+D`)? A sua solução final de `main.c` já faz isso corretamente. Analise-a para confirmar.