Strings em C: Ponteiros em Ação

O segredo por trás do texto no seu terminal

O Mistério das Strings (A Ancoragem)

Em muitas linguagens, você tem um tipo `string`. Em C, não. Então, o que é isso?

```c
char nome[] = "Jett";
```

Será que é um vetor? Será que é um ponteiro? Por que usamos `%s` no `printf`?

Na verdade, uma string em C é simplesmente um VETOR de caracteres terminado por um caractere especial invisível.

A Revelação: O `\0` (Caractere Nulo)

Toda string em C é, por baixo dos panos, um vetor de `char` que **obrigatoriamente** termina com um caractere especial chamado `\0` (lê-se "barra zero" ou "caractere nulo").

É assim que as funções como `printf` sabem onde o texto **termina**.

A string `"Jett"` na memória é, na verdade, o vetor:
`['J', 'e', 't', 't', '\0']`

Ela não tem 4, mas sim **5 caracteres** de tamanho!

Declarando e Inicializando Strings

Existem duas formas principais de colocar uma string na memória.

Forma 1: O Vetor de `char` (Mais comum)
Abre espaço na memória para guardar a string.

```
// O compilador calcula o tamanho (4 + 1 = 5) e já inclui o \0.
char nome_personagem[20] = "Sova";
```

Aqui, `nome_personagem` é um vetor.

Forma 2: O Ponteiro para `char`

Aponta para um texto que já existe em uma área somente leitura da memória.

```
// 'ptr_nome' aponta para a primeira letra de "Killjoy".
const char* ptr_nome = "Killjoy";
```

Aqui, `ptr_nome` é um ponteiro. ***Você não pode alterar essa string!***

O Grande Perigo: Buffer Overflow

O que acontece se você tentar colocar um texto maior do que o espaço que você reservou?

```
char nome[5] = "Jett"; // Espaço para 4 letras + \0. OK!

// O que acontece se tentarmos fazer isso?
strcpy(nome, "Phoenix"); // "Phoenix" tem 7 letras!
```

O programa vai tentar escrever `['P','h','o','e','n','i','x','\0']` na memória, invadindo o espaço de outras variáveis. Isso causa bugs, falhas de segurança e comportamentos inesperados. É o famoso **Buffer Overflow**.

A Caixa de Ferramentas: `<string.h>`

Nunca manipule strings na mão (letra por letra). Use a biblioteca padrão `<string.h>`!

Funções Essenciais:

- `strcpy(destino, origem)`: **Copia** uma string para outra.
- `strlen(string)`: Retorna o **tamanho** da string (sem contar o `\0`).
- `strcmp(str1, str2)`: **Compara** duas strings. Retorna `0` se forem iguais.
- `strcat(destino, origem)`: **Concatena** (junta) a string de origem no final da de destino.

Mão na Massa (Live Coding)

```c
int main() {
    char nome_completo[50] = "Brimstone";
    char sobrenome[] = " da Silva";

    printf("Nome original: %s\n", nome_completo);
    printf("Tamanho do nome: %zu\n\n", strlen(nome_completo));

    // Juntando o sobrenome ao nome original
    strcat(nome_completo, sobrenome);

    printf("Nome completo: %s\n", nome_completo);
    printf("Tamanho novo: %zu\n", strlen(nome_completo));

    // Comparando com outra string
    if(strcmp(nome_completo, "Brimstone da Silva") == 0) {
        printf("\nO nome está correto!\n");
    }
    return 0;
}
```

Resumo da Ópera

- Uma **String** em C é um **vetor de `char` que termina em `\0`**.
- O `\0` é crucial! É ele quem diz onde a string acaba.
- Cuidado com o **tamanho do vetor** para não causar **Buffer Overflow**.
- **SEMPRE** use as funções da biblioteca `<string.h>` para copiar, comparar e manipular strings. Elas são seguras e eficientes.

Fim da Trilogia Introdutória!

- **Structs:** Organizaram nossos dados.
- **Ponteiros:** Nos deram poder e eficiência para manipular a memória.
- **Strings:** Vimos a aplicação prática de ponteiros e vetores.

Com essa base, vocês estão prontos para mergulhar nas estruturas de dados dinâmicas (Pilhas, Filas, Listas...), que é o coração da nossa matéria!