Roteiro de Desenvolvimento: Biblioteca de Strings Dinâmicas em C

Este documento apresenta um roteiro estruturado em 4 módulos e 10 exercícios para guiar a criação de uma biblioteca de strings dinâmicas em C, similar em conceito à biblioteca SDS (Simple Dynamic Strings). O objetivo é construir um sistema robusto que gerencia a memória automaticamente, oferecendo uma alternativa segura e eficiente às strings padrão de C.

Módulo 1: A Fundação - O Segredo por Trás do Ponteiro

Objetivo do Módulo: Compreender a arquitetura fundamental da biblioteca, onde metadados (como tamanho e capacidade) são armazenados em um cabeçalho contíguo e invisível ao usuário final, que manipula apenas um `char *`.

Exercício 1: A Estrutura de Controle (O Cabeçalho)

> **Objetivo da Estrutura:** Definir o "cérebro" da nossa string dinâmica. Esta estrutura interna armazenará os metadados essenciais que permitem o gerenciamento inteligente da memória e o acesso rápido às propriedades da string.
>
> **Tarefa:**
> Defina a `struct` que servirá como o cabeçalho de controle. Vamos chamá-la de `t_ds`. Ela deve conter no mínimo:
>
> - `len`: O comprimento atual da string (número de caracteres em uso).
> - `capacity`: O espaço total alocado para a string (quantos caracteres cabem no buffer).
> - `buf[]`: Um "Membro de Array Flexível" (`Flexible Array Member`) que marca o início dos dados da string.
>
> ```c

include // Necessário para o tipo size_t

typedef struct s_ds {
size_t len;
size_t capacity;
char buf[];
} t_ds;
```

> ***Ponto de Reflexão:*** *Como a técnica do Flexible Array Member permite que o cabeçalho e os dados da string sejam alocados em um único bloco de memória contíguo com* `malloc`*?*

## Exercício 2: A Mágica do Ponteiro (Indo e Voltando)

**Objetivo das Funções:** Criar as ferramentas internas e privadas (`static`) para navegar entre o ponteiro do usuário (`char *`) e o ponteiro do cabeçalho (`t_ds *`). Estas funções são a base para todas as outras operações da biblioteca.

**Tarefa:**
Crie duas funções `static inline` para a conversão de ponteiros:

1. **`ds_get_value`**: Recebe um ponteiro para o cabeçalho (`t_ds *`) e retorna o ponteiro para o início dos dados (`char *`), que é o que o usuário irá manipular.
2. **`ds_get_ds`**: Recebe o ponteiro do usuário (`char *`) e retorna o ponteiro para o cabeçalho (`t_ds *`) que o precede na memória.

> ***Ponto de Atenção:*** *A lógica central aqui é a aritmética de ponteiros. A operação para "voltar" do ponteiro do usuário para o cabeçalho deve subtrair exatamente o* `sizeof(t_ds)`*.*

---

# Módulo 2: O Ciclo de Vida - Criar, Consultar e Destruir
```

Objetivo do Módulo: Implementar as operações essenciais para criar uma nova string dinâmica, consultar suas propriedades básicas e liberá-la da memória de forma segura.

Exercício 3: O Nascimento da String (`ds_new`)

Objetivo da Função: Criar e inicializar uma nova string dinâmica a partir de uma string C padrão (`const char *`). Esta função é o ponto de entrada para o uso da biblioteca.

Roteiro de Implementação:

1. Receba uma string C (`const char *init`) como entrada.
2. Calcule o comprimento da string `init`.
3. Determine o tamanho total de memória a ser alocado: `sizeof(t_ds) + comprimento + 1` (o `+1` é para o terminador nulo).
4. Use `malloc` para alocar este bloco único. Verifique se a alocação foi bem-sucedida.
5. No cabeçalho, preencha os campos `len` e `capacity` com o comprimento da string inicial.
6. Copie o conteúdo de `init` para a área de dados (`buf`).
7. Garanta que a nova string seja terminada com .
8. Retorne o ponteiro para a área de dados (`buf`), escondendo o cabeçalho do usuário.

Exercício 4: O Fim da String (`ds_free`)

Objetivo da Função: Liberar de forma segura toda a memória ocupada por uma string dinâmica, incluindo seu cabeçalho e seu buffer de dados.

Tarefa:
Implemente a função `void ds_free(char *s)`.

1. Receba o ponteiro do usuário (`char *s`).
2. Se o ponteiro for `NULL`, não faça nada.

3. Use sua função auxiliar (`ds_get_ds`) para obter o ponteiro do início do bloco de memória (o endereço do cabeçalho).
4. Chame `free()` neste ponteiro do cabeçalho para liberar o bloco inteiro.

Exercício 5: A Vantagem O(1) (`ds_len`)

Objetivo da Função: Retornar o comprimento da string dinâmica de forma instantânea, sem a necessidade de percorrer os caracteres.

Tarefa:
Implemente a função `size_t ds_len(const char *s)`.

1. Receba o ponteiro do usuário (`const char *s`).
2. Obtenha o ponteiro para o cabeçalho.
3. Acesse e retorne o valor do campo `len` do cabeçalho.
4. Esta operação deve ter complexidade de tempo constante, `O(1)`.

> ***Ponto de Verificação:*** *Escreva um pequeno programa `main` que usa `ds_new`, `ds_len` e `ds_free`. Crie uma string, imprima seu conteúdo e seu comprimento, e depois libere a memória. Isso validará o trabalho feito até agora.*

Módulo 3: Crescimento Inteligente - Gerenciando a Capacidade

Objetivo do Módulo: Implementar a funcionalidade "dinâmica" da biblioteca, permitindo que a string cresça de forma eficiente para acomodar novas concatenações sem realocações de memória a cada passo.

Exercício 6: Planejando o Futuro (`ds_make_room`)

Objetivo da Função: Garantir que uma string dinâmica tenha espaço livre suficiente para uma futura adição de dados. Se não houver espaço, esta função irá realocar a memória de forma inteligente (pré-alocação).

Lógica de Implementação:

1. Receba a string (`char *s`) e o comprimento adicional necessário (`size_t addlen`).
2. Obtenha o cabeçalho para acessar `len` e `capacity`.
3. Verifique se a capacidade atual (`capacity`) é suficiente para o comprimento atual mais o adicional (`len + addlen`). Se sim, retorne a string `s` sem modificações.
4. Se não, calcule uma nova capacidade. **Estratégia de pré-alocação:** uma boa política é dobrar o tamanho total necessário (`new_capacity = (len + addlen) * 2`).
5. Use `realloc` (ou sua implementação) para redimensionar o bloco de memória inteiro, usando o ponteiro do cabeçalho.
6. **Ponto Crítico:** Se a realocação for bem-sucedida, atribua o novo ponteiro a uma variável, atualize o campo `capacity` no novo cabeçalho e retorne o novo ponteiro de dados. Se falhar, retorne o ponteiro original para evitar perda de dados.

Exercício 7: O Bloco de Construção (`ds_cat_len`)

Objetivo da Função: Anexar um bloco de memória de tamanho conhecido (que pode conter quaisquer dados, incluindo) ao final de uma string dinâmica. Esta é a função de concatenação mais fundamental e segura.

Passos de Implementação:

1. Receba a string de destino (`char *s`), um ponteiro para os dados a serem anexados (`const void *t`) e o comprimento desses dados (`size_t len`).
2. Chame `s = ds_make_room(s, len);` para garantir que há espaço. **Lembre-se de reatribuir `s`**, pois seu endereço pode ter mudado.
3. Obtenha o cabeçalho do `s` (potencialmente novo).
4. Use `memcpy` para copiar `len` bytes de `t` para o final da string `s` (na posição `s + ds->len`).
5. Atualize o `len` no cabeçalho: `ds->len += len;`.
6. Adicione o terminador nulo na nova posição final.
7. Retorne o ponteiro `s` atualizado.

Exercício 8: A Interface Amigável (`ds_cat`)

Objetivo da Função: Fornecer uma interface de conveniência para o caso de uso mais comum: concatenar uma string C padrão (terminada em nulo) a uma string dinâmica.

Tarefa:
Implemente `char *ds_cat(char *s, const char *t)`.

1. Esta função é um simples "wrapper" (invólucro).
2. Dentro dela, calcule o comprimento de `t` usando `strlen(t)`.
3. Chame e retorne o resultado de `ds_cat_len`, passando `s`, `t`, e o comprimento que você acabou de calcular.

Módulo 4: Boas Práticas e Prova de Fogo

Objetivo do Módulo: Organizar o código de forma profissional em um módulo reutilizável e testá-lo rigorosamente para garantir que não há vazamentos de memória ou outros comportamentos inesperados.

Exercício 9: Organizando o Código (Criando um Módulo)

Objetivo: Estruturar o código seguindo as boas práticas de C, separando a interface pública da implementação interna. Isso torna a biblioteca fácil de usar e manter.

Tarefa:

1. **Arquivo de Cabeçalho (`ds.h`):** Crie este arquivo e coloque nele as **declarações** das funções que o usuário final irá chamar (`ds_new`, `ds_free`, `ds_len`, `ds_cat`, etc.). Use *include guards* (`#ifndef`/`#define`/`#endif`) para evitar inclusões múltiplas.
2. **Arquivo de Implementação (`ds.c`):** Crie este arquivo para conter a **implementação** de todas as funções. A definição da `struct t_ds` e as funções auxiliares (`ds_get_ds`, `ds_make_room`) devem estar aqui e, idealmente, marcadas como `static` para serem privadas ao módulo.

Exercício 10: O Teste de Estresse (Validação e Robustez)

Objetivo: Escrever um programa de teste que submeta a biblioteca a cenários rigorosos para verificar sua correção, eficiência e, principalmente, a ausência de erros de gerenciamento de memória.

Cenários de Teste:
Crie um arquivo `test.c` que inclua `ds.h` e realize os seguintes testes:

1. **Teste de Crescimento:** Crie uma string vazia e, dentro de um loop (ex: 1000 iterações), use `ds_cat` para concatenar uma pequena palavra. A cada iteração, verifique com `ds_len` se o comprimento está correto.

2. **Teste de Vazamentos de Memória:** Compile seu programa de teste com a flag `-g` (para informações de depuração). Execute-o com a ferramenta Valgrind:
`bash valgrind --leak-check=full ./seu_programa_de_teste`
O Valgrind deve reportar "0 errors" e "0 bytes in 0 blocks are definitely lost", indicando que não há vazamentos de memória.

3. **Demonstração do Bug Clássico:** Escreva um pequeno trecho de código que force uma realocação (concatenando uma string grande) mas **esqueça de reatribuir o resultado** (`ds_cat(s, ...);` em vez de `s = ds_cat(s, ...);`). Tente imprimir ou usar a string `s` original depois disso. Observe o comportamento inesperado ou a falha do programa. Este teste serve para entender a importância da API de reatribuição.