Strings em C++

Seção 1: Strings Estilo C (C-style)

As *strings* estilo C são o mecanismo herdado de C para manipulação de texto.

- **Definição:** São simplesmente *arrays* de caracteres (`char[]`) terminados por um caractere nulo especial, o **`\0`** (NULO).
- **Literal:** Um literal de *string* em C++ (ex: `"hello"`) ainda denota um *array* de caracteres terminado em NULO. O caractere nulo é automaticamente anexado ao final dos literais.

Exemplo de Declaração e Tamanho: Uma *string* de 4 caracteres, como `"ABCD"`, ocupa 5 *bytes* na memória, pois o `\0` requer um *byte* de armazenamento.

Manipulação e Funções Legadas

A manipulação dessas *strings* é realizada por funções da biblioteca C, acessíveis em C++ através do cabeçalho `<cstring>` (ou `string.h` em C).

Função C-style	Descrição	Observação
`strlen(str)`	Retorna o comprimento da *string*, **excluindo** o `\0`.	Complexidade de tempo linear, pois precisa percorrer todo o *array* até encontrar o `\0`.
`strcpy(dest, src)`	Copia a *string* de origem (`src`) para o destino (`dest`).	Não verifica o tamanho do *buffer* de destino, sendo insegura.
`strcat(dest, src)`	Concatena a *string* de origem ao final do destino.	Também não verifica limites e é insegura.
`strcmp(a, b)`	Compara duas *strings* lexicograficamente.	O retorno (positivo, negativo ou zero) não é intuitivo.

Riscos de Segurança

O uso de *strings* estilo C é propenso a erros, mesmo em C++.

- **Cópia Não Delimitada (*Unbounded Copy*):** Funções como `strcpy` não sabem o tamanho do *buffer* de destino, o que pode causar transbordamento de *buffer* (*buffer overflow*) e explorações.
- **Erros de Terminação Nula:** Falha em garantir que o *array* esteja corretamente terminado em `\0` faz com que as funções operem fora dos limites alocados.
- **Truncamento (*Truncation*):** Usar funções seguras de limite (como `strncpy` ou `snprintf`) mitiga o *buffer overflow*, mas pode resultar em perda ou truncamento de dados.

Seção 2: A Solução Moderna: Classe `std::string`

O `std::string` e suas Vantagens

- **Padrão C++:** `std::string` é o principal tipo de *string* na biblioteca padrão C++ desde o C++98.
- **Gerenciamento de Memória (RAII):** É uma classe complexa que usa o idioma RAII (*Resource Acquisition Is Initialization*) para gerenciar automaticamente a memória subjacente. Isso reduz o risco de acessos fora dos limites (*out-of-bounds accesses*).
- **Sintaxe Intuitiva:** Oferece sintaxe mais fácil e interfaces intuitivas, como o uso de operadores para comparação e concatenação.

Operações Fundamentais

O uso de `std::string` requer o cabeçalho `<string>`.

Operação	`std::string` (C++ Moderno)	C-style (Legado)
Atribuição/Cópia	Usa o operador `=`.	Requer `strcpy()`.
Concatenação	Usa os operadores `+` ou `+=`.	Requer `strcat()`.
Comprimento	Usa `size()` ou `length()`.	Requer `strlen()` (lenta).
Comparação	Usa operadores `==`, `<`, `>`.	Requer `strcmp()`.

O acesso a caracteres individuais é feito com o operador `[]` (sem verificação de limites) ou com o método `at()` (com verificação de limites, lançando exceção em caso de erro).

Interoperabilidade C: O Método `c_str()`

Para interagir com interfaces legadas ou APIs de sistemas operacionais (Windows e Linux) que esperam parâmetro do tipo `const char*`, o `std::string` fornece o método `c_str()`.

- **Função:** O método **`c_str()`** retorna um ponteiro constante para um *array* de caracteres terminado em NULO (`\0`), com dados equivalentes aos armazenados na *string*.
- **Imutabilidade:** O método `c_str()` é qualificado como `const`, o que significa que ele não altera o estado do objeto `std::string` no qual é chamado, sendo uma função somente leitura.
- **`data()` vs. `c_str()`:** Antes do C++11, `c_str()` era o único método que garantia que o *array* retornado fosse terminado em nulo. A partir do C++11, `data()` e `c_str()` fazem basicamente a mesma coisa.

Seção 3: Otimização de Desempenho: Movimentação e Views

Otimização de Strings Curtas (SSO)

- **Propósito:** O SSO (*Small String Optimization*) é uma técnica de otimização implementada pela maioria das bibliotecas que faz a classe `std::string` guardar textos curtos dentro dela mesma, em vez de pedir memória emprestada na área dinâmica (a heap).
- **Funcionamento:** Imagine que o objeto std::string é como uma carteira. Essa carteira tem um espaço interno (um buffer embutido) que está sempre lá.
  1. Se a string for pequena (por exemplo, "Fácil" ou "marte"): Em vez de passar pelo processo lento de alocar memória dinâmica (`new[]` ou `malloc`), o texto é guardado diretamente no espaço interno da “carteira”. Para a maioria das implementações modernas de C++, esse espaço interno costuma ser suficiente para guardar entre 15 e 22 caracteres, dependendo da arquitetura.
  2. Se a string for grande ("O Exterminador do Futuro"): Aí sim, o objeto percebe que o texto não cabe, e ele volta para o método tradicional, alocando memória na heap
- **Benefícios:** O SSO traz grandes vantagens para o desempenho da sua aplicação, especialmente em programas que usam muitas strings curtas
  1. **Economia de Tempo e Recursos (Sem Burocracia):** Evitar a alocação na *heap* (memória dinâmica) significa que você ignora todo o *overhead* (o custo e a burocracia) associado a pedir e liberar memória. Para *strings* curtas, o armazenamento ocorre na *stack* (memória local, que é muito mais rápida).
  2. **Melhor "Organização" de Memória (*Cache Efficiency*):** Guardar o texto dentro do próprio objeto `std::string` (em vez de em um bloco de memória distante, referenciado por um ponteiro) melhora a **localidade de cache** (cache locality). Isso significa que a CPU acessa os dados da *string* de forma mais rápida e contínua.

Semântica de Movimentação

A Semântica de Movimentação (*Move Semantics*, C++11) é uma **otimização de desempenho** para objetos grandes como `std::string`.

Em vez de realizar uma cópia profunda e lenta, essa técnica permite **transferir a propriedade dos recursos internos** (como memória alocada dinamicamente) de um objeto para outro.

A movimentação é muito mais rápida porque envolve apenas a cópia de ponteiros internos. Essa transferência ocorre de forma segura com objetos temporários, deixando o objeto original (fonte) sem os recursos, geralmente com seu ponteiro interno definido para nulo (`nullptr`).

A função `std::move()` não move nada, mas sim *habilita* o compilador a realizar essa otimização.

Visualizações de String (`std::string_view` - C++17)

O `std::string_view` (C++17) é um objeto **leve de somente leitura** que permite "olhar" para uma sequência de caracteres existente.

- **O que é:** É um par que armazena apenas o **ponteiro** para os dados e o seu **tamanho**.

- **Não é Proprietário:** Ele não gerencia, aloca ou libera a memória da *string* que ele visualiza.
- **Vantagem Principal:** Evita a **cópia de dados** quando você passa *strings* longas para funções que só precisam lê-las (seja `std::string` ou *C-style string*), melhorando drasticamente o desempenho.
- **Risco de *Dangling*:** O programador deve garantir que a *string* original (a dona da memória) não seja destruída enquanto o `std::string_view` estiver sendo usado

Seção 4: C++ Moderno (C++20/23): Busca e Formatação

O C++ Moderno introduziu funcionalidades mais intuitivas para buscar e verificar conteúdo em `std::string` e `std::string_view`.

- **`starts_with()` e `ends_with()` (C++20):** Verificam de forma direta se uma *string* começa ou termina com um prefixo ou sufixo específico.
- **`contains()` (C++23):** Simplifica a verificação de existência de uma sub-*string* ou caractere, substituindo a necessidade da sintaxe `str.find(sub) != std::string::npos`.

Formatação Segura com `std::format` (C++20)

- **Função:** `std::format` é uma nova função introduzida no C++20 que oferece uma maneira de formatar *strings* substituindo *placeholders* (`{}`) dentro de uma *string* de formato.
- **Sintaxe:**
  - Exemplo simples: `std::format("My name is {} and my favorite number is {}", name, num)`.
  - **Argumentos Posicionais:** Permite especificar a ordem dos argumentos, usando índices, como `{1}` e `{0}`.
  - **Opções de Formatação:** Suporta opções detalhadas dentro dos *placeholders* (ex: `:.2f` para formatar um *double* com duas casas decimais).
- **Vantagem:** É geralmente mais eficiente do que opções mais antigas, como `sprintf` ou `printf`. A sintaxe usa `std::string_view` para a *string* de formato.