

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
GRADUAÇÃO EM TECNOLOGIA DA INFORMAÇÃO

RELATÓRIO TÉCNICO: Projeto de Introdução às Técnicas de Programação -
Unidade 2

ANA CLARA LIMA DA SILVA

NATAL, RIO GRANDE DO NORTE
2025

Departamento de Informática e Matemática Aplicada
Introdução às Técnicas de Programação — IMD0012

NOME DO PROJETO: Codificador/Decodificador Simples de textos/*strings*.

OBJETIVO: O projeto visa realizar a codificação de letras em textos/*strings*.

PROBLEMA SOLUCIONADO: Necessidade de codificar textos.

JUSTIFICATIVA: Este projeto foi escolhido visando facilitar a codificação de textos, sem que o usuário possua a necessidade de fazer autonomamente.

METODOLOGIA

Compilador utilizado: GCC versão 15.2.0

Editor utilizado: Visual Studio Code

APLICAÇÃO DOS CONCEITOS DA UNIDADE 2

Conceitos fundamentais da Unidade 2 aplicados: Vetores, *Strings*, Repetições aninhadas, Matrizes, Ponteiros e Alocação.

Como foram utilizadas as Variáveis:

As variáveis foram utilizadas para armazenar dados temporários, controlar o fluxo dos *loops* e armazenar o tamanho das *strings*.

- **Variáveis de Controle de Loop:** Variáveis como *i* (*size_t*) e *j* (*int*) foram usadas para indexar *strings* e matrizes, percorrendo cada caractere e cada posição da chave de substituição.
- **Variáveis de Armazenamento Temporário:** *char letra* foi usada dentro das funções cifrar e decifrar para armazenar o caractere atual da *string* após ser convertido para maiúscula (*toupper*), facilitando a busca.
- **Variáveis de Tamanho:** *size_t tamanho* (em todas as funções) armazenou o comprimento da *string* lida (*strlen*), essencial para a alocação dinâmica de memória precisa.

Como foram utilizadas as Estruturas Condicionais:

As estruturas condicionais (*if* e *else*) foram essenciais para garantir o fluxo de controle seguro e a lógica da cifra.

- **Validação de Alocação:** *if (entrada_dinamica == NULL)*: Essencial após cada chamada a *malloc* para verificar se a memória foi alocada com sucesso, prevenindo falhas no programa.
- **Validação da Leitura:** *if (fgets(...) == NULL)*: Verifica se a leitura do console (*ler_entrada*) ocorreu sem erros.
- **Lógica de Cifragem:** *if (letra >= 'A' && letra <= 'Z')*: Determina se o caractere atual é uma letra alfabética (e, portanto, deve ser cifrada) ou se deve ser copiado diretamente (como espaços, números e pontuação).

- **Remoção de Nova Linha:** `if (len > 0 && buffer[len - 1] == '\n')`: Condição usada para remover o `\n` da *string* lida pelo `fgets`.

Como foram utilizadas as Funções:

A função `ler_entrada()` é responsável por ler a *string* do usuário e retornar um ponteiro para uma área de memória alocada dinamicamente. Já a função `cifrar(const char *texto)` é responsável por executar o algoritmo de substituição, criando um novo *buffer* de memória com a mensagem cifrada.

Ademais, a função `decifrar(const char *texto_cifrado)` é responsável por executar o algoritmo de substituição inversa, restaurando a mensagem original em um novo *buffer*. Já a função principal `main()` é responsável por coordenar o fluxo do programa (chamar as funções em ordem) e, o mais importante, gerenciar a liberação da memória alocada por todas as outras funções (`free`).

Como foram utilizados os Vetores/Arrays:

Os *arrays* (vetores) foram utilizados para armazenar blocos de dados próximos de maneira temporária. `char buffer[MAX_INPUT_BUFFER]` é um *array* estático temporário na função `ler_entrada` usado como *buffer* de leitura para armazenar a entrada do usuário antes que o tamanho exato fosse conhecido e a memória dinâmica fosse alocada.

Como foram utilizadas as Strings:

As *strings* são o tipo de dado central do projeto e são tratadas como vetores de caracteres (`char*`) terminados pelo caractere nulo (`\0`).

- **Manipulação:** Funções como `strlen` (para obter o tamanho exato da alocação), `strcpy` (para copiar o conteúdo para a memória alocada) e `fgets` (para leitura) foram essenciais.
- **Strings Dinâmicas:** Os textos de entrada, cifrado e decifrado são tratados como *strings* dinâmicas (`char *`), permitindo que seus tamanhos variem em tempo de execução.

Como foram utilizadas as Repetições Aninhadas:

As repetições aninhadas garantem que o mapeamento da cifra seja realizado corretamente. O Loop Externo (for i) percorre sequencialmente cada caractere da *string* de texto (original ou cifrado).

Já o Loop Interno (for j): Para cada caractere do texto, percorre as 26 posições da Matriz de Mapeamento (mapa_cifra) para encontrar o índice correspondente, realizando a busca e a substituição. O *break* é usado para otimizar, interrompendo a busca assim que o mapeamento é encontrado.

Como foram utilizadas as Matrizes:

A matriz é a principal estrutura de dados para implementar a chave criptográfica. `char mapa_cifra[2][26]` é a Matriz bidimensional de char que armazena a regra de transformação:

- Linha 0: O conjunto de caracteres a ser substituído (o alfabeto original).
- Linha 1: O conjunto de caracteres substitutos (a chave cifrada).

Como foram utilizados os Ponteiros e Alocação:

O projeto utiliza o gerenciamento explícito de memória para manipular *strings* de tamanho variável, recorrendo às duas principais áreas de memória do sistema: a Stack e o Heap.

1. Alocação no Heap (malloc)

A estratégia central é a Alocação Dinâmica, que utiliza a função `malloc()` para solicitar memória na área do Heap (Montículo) em tempo de execução. Isso é essencial porque o tamanho do texto do usuário é desconhecido inicialmente.

- **Necessidade:** Três blocos de memória no Heap são alocados (`texto_entrada`, `texto_cifrado`, `texto_decifrado`) para garantir a flexibilidade e a separação dos *buffers* de dados.
- **Precisão:** A alocação é precisa, utilizando `strlen` para determinar o tamanho exato da *string* (tamanho + 1 byte para o terminador nulo).

2. Função dos Ponteiros (char *)

As variáveis Ponteiro (char *), como texto_cifrado, são variáveis armazenadas na Stack, mas cujo valor é o endereço de memória inicial do bloco de dados alocado no Heap.

3. Gerenciamento e Limpeza (free)

O gerenciamento de memória é manual e crucial para evitar *memory leaks* (vazamentos).

- **Regra de Desalocação:** A função main() implementa a regra um free() para cada malloc().
- **Finalidade:** Os comandos free(ponteiro) são chamados explicitamente para devolver cada um dos três blocos de memória alocados no Heap ao sistema operacional, garantindo que o programa finalize de forma limpa.

COMO FORAM IMPLEMENTADAS AS ESTRUTURAS DE DADOS COMPLEXAS

A estrutura de dados mais complexa utilizada foi a Matriz (*Array Bidimensional*), que serviu como a chave de mapeamento da cifra de substituição.

- **Matriz de Caracteres (char mapa_cifra[2][26]):** Esta matriz foi implementada como uma estrutura estática global para armazenar a regra de transformação:
 - **Linha 0:** Guarda o alfabeto original (A a Z).
 - **Linha 1:** Guarda o alfabeto cifrado (a chave, ex: Q a M).
- **Strings como Arrays de Ponteiros:** As *strings* de texto (entrada, cifrado e decifrado) são manipuladas como arrays de caracteres usando ponteiros (char *). O uso de char * permitiu que as funções recebessem o endereço de memória da *string*, trabalhando diretamente no conteúdo sem copiar grandes volumes de dados.

ESTRATÉGIA PARA GERENCIAMENTO DE MEMÓRIA

A estratégia central foi o Gerenciamento de Memória Baseado em Responsabilidade (*Ownership*), utilizando alocação dinâmica para todos os dados cujo tamanho não era conhecido em tempo de compilação.

- **Alocação na Fonte:** Toda *string* gerada com tamanho variável (*texto_entrada*, *texto_cifrado*, *texto_decifrado*) foi alocada usando a função *malloc* dentro da função responsável por sua criação (*ler_entrada*, *cifrar*, *decifrar*).
- **Transferência de Responsabilidade:** As funções retornam o ponteiro (*char **) para a *string* alocada, transferindo a responsabilidade de liberação de volta para a função chamadora, que é a *main()*.

GARANTINDO QUE NÃO HÁ VAZAMENTOS DE MEMÓRIA(MEMORY LEAKS)

A garantia de que não há *memory leaks* está na implementação rigorosa da regra de um *free()* para cada *malloc()* dentro da função principal (*main*). Essa abordagem sequencial e explícita no *main* assegura que todos os blocos de memória alocados dinamicamente sejam liberados antes que o programa finalize.

VANTAGENS EM UTILIZAR ALOCAÇÃO DINÂMICA NO PROJETO

- **Flexibilidade no Tamanho da Entrada:** O programa pode lidar com qualquer tamanho de texto que o usuário digitar (limitado apenas pela memória disponível do sistema), sem a necessidade de definir *buffers* estáticos arbitrários e restritivos (como *char texto[1000]*).
- **Eficiência no Uso da Memória:** O uso de *malloc(len + 1)* garante que apenas a quantidade de memória exatamente necessária (o tamanho do texto lido + 1 para o *\0*) seja alocada. Evitando o desperdício de memória que ocorreria ao se usar grandes *arrays* estáticos predefinidos.
- **Desenvolvimento Modular:** As funções (*cifrar*, *decifrar*) podem operar de forma independente, criando e gerenciando seus próprios *buffers* de saída, tornando o código mais limpo, reutilizável e fácil de manter.

DIFICULDADES ENCONTRADAS

O VS Code apresentou algumas falhas no terminal.

SOLUÇÕES IMPLEMENTADAS

Provisoriamente as falhas foram corrigidas e depois retornaram, então não foram encontradas soluções adequadas.

ORGANIZAÇÃO DO CÓDIGO

O código foi organizado com funções, além da *main* que são responsáveis por etapas específicas do fluxo de trabalho do programa. A função `char* ler_entrada()` lida com a entrada do usuário e aloca dinamicamente a memória necessária para armazená-la. A função `char* cifrar(const char *texto)` converte o texto original em um texto codificado, usando a Cifra de Substituição. A função `char* decifrar(const char *texto_cifrado)` reverte o processo de cifragem e transforma a mensagem codificada de volta ao seu formato original, em caixa alta (letras maiúsculas).

CONCLUSÃO

A maneira a qual foi feito o código está apropriada para realizar codificações de textos/*strings*, porém, há a necessidade de serem implementadas codificações sem a necessidade de transformar todas as letras minúsculas em letras maiúsculas para trabalhar somente com elas, também poderiam ser implementadas codificações dos demais caracteres, além de somente em letras.