


COISAS ESTRANHAS



com
Ponteiros

Pedro Ferreira

01

O MEDO

PONTEIROS: são de outro mundo??

Desmitificando o medo de ponteiros

Seja bem-vindo, corajoso aventureiro, ao nosso clube de Dungeons & Dragons... quer dizer, de programação em C! Se você está aqui, provavelmente já ouviu histórias de arrepiar sobre uma criatura terrível que assombra os programadores iniciantes. O nome dela? Ponteiros. Dizem que é um monstro complexo, perigoso e que causa dores de cabeça. Mas, como os garotos de Hawkins nos ensinaram, os monstros mais assustadores são aqueles que a gente não entende.

A verdade é que ponteiros não são o Demogorgon. Pense neles mais como o mapa que o chefe Hopper usou para encontrar os portais em Hawkins. Um ponteiro não é uma coisa assustadora; ele é simplesmente uma **informação que aponta para um local**. É um endereço. Só isso! Quando você anota o endereço da casa da Joyce Byers, o papel com o endereço não é a casa, certo? Ele apenas diz *onde* encontrar a casa. Um ponteiro é exatamente a mesma coisa para as variáveis na memória do seu computador.

Então, por que todo esse alvoroço? Por que não podemos simplesmente ignorá-los? Bom, porque em C, os ponteiros são a nossa principal ferramenta para realizar missões de resgate no "Mundo Invertido" da memória. Eles nos dão superpoderes. Sem eles, a linguagem C perderia grande parte de sua força e flexibilidade. É como tentar lutar contra os monstros sem o estilingue do Lucas ou os poderes da Eleven.

Mas onde, no mundo real, usamos esses "mapas de memória"? Você ficaria surpreso!

- **No seu Sistema Operacional (Windows, Linux, macOS):** Como você acha que o sistema gerencia dezenas de programas abertos ao mesmo tempo? Ele usa ponteiros para organizar onde cada janela, cada arquivo e cada processo está na memória RAM. É uma verdadeira operação de busca e resgate em Hawkins, acontecendo o tempo todo!
- **Em Videogames:** Jogos com mundos gigantescos, como os de RPG que o Will e seus amigos tanto amam, não carregam o mapa inteiro de uma vez na memória. Isso seria impossível! Em vez disso, eles usam ponteiros para carregar dinamicamente apenas as partes do mundo que estão perto do jogador. O jogo "aponta" para os dados da floresta quando você está nela e depois aponta para os dados da cidade quando você se aproxima.

Ponteiros são sobre eficiência e controle. Em vez de mover uma "coisa" grande e pesada (como uma estrutura de dados gigante) de um lado para o outro, nós simplesmente passamos o endereço dela. É muito mais fácil entregar um bilhete com a localização do esconderijo do que carregar o esconderijo inteiro nas costas.

Então, respire fundo. Neste e-book, vamos ligar nossas lanternas, pegar nossos walkie-talkies e explorar esse mundo juntos. Você vai ver que ponteiros não são o monstro, mas sim a ferramenta que vai nos ajudar a derrotá-lo. Agora que o medo inicial passou, vamos ao próximo passo: aprender a criar e usar nosso primeiro mapa.

02

O MUNDO INVERTIDO

Ligando a Lanterna e o Walkie-Talkie

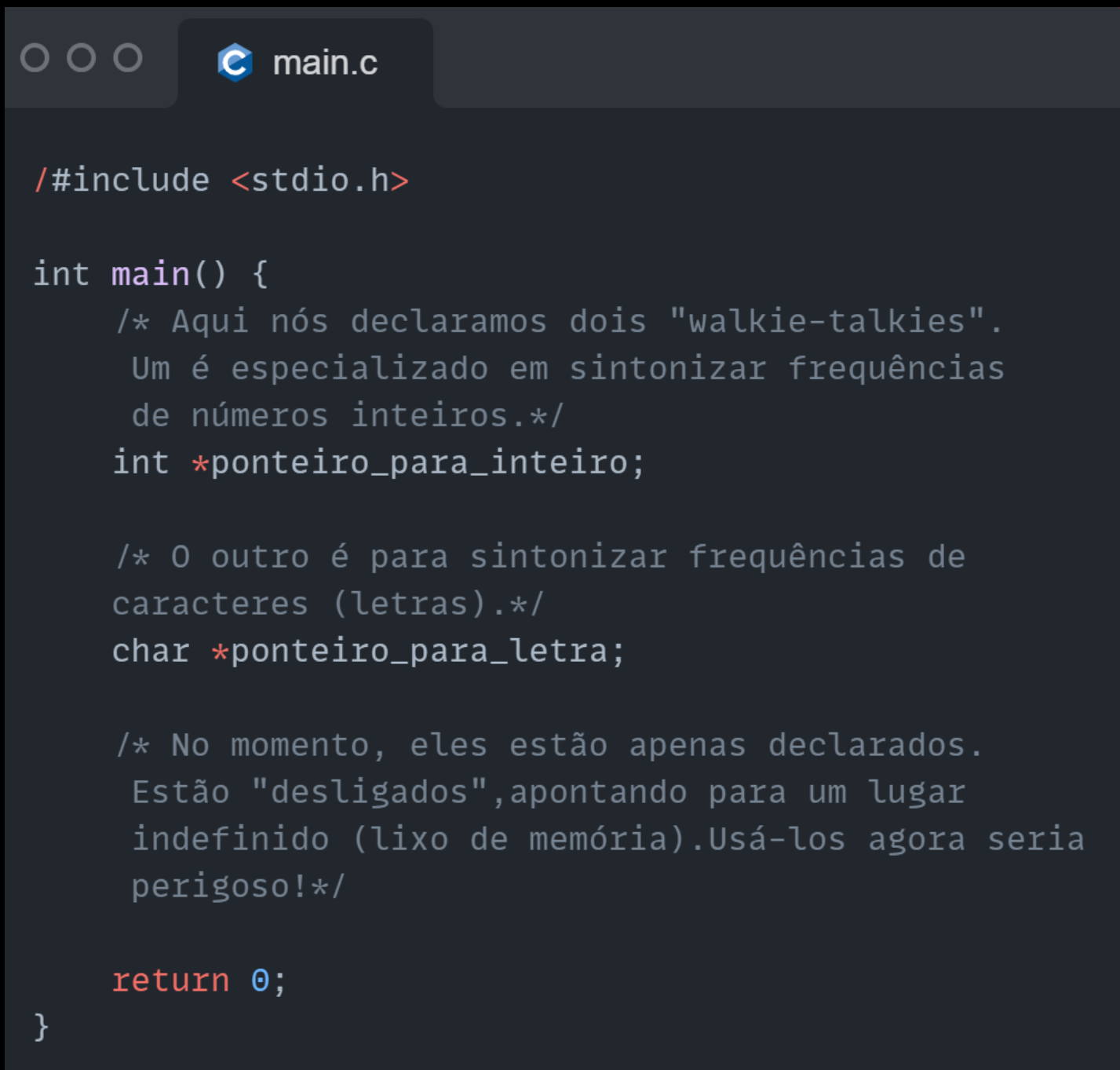
A Mecânica Fundamental: conceitos e sintaxe de ponteiros

No último capítulo, nós abrimos a porta e espiamos o Mundo Invertido, descobrindo que ele não é tão assustador assim. Agora, vamos pegar nosso equipamento. Para sobreviver e navegar por lá, você precisa basicamente de duas coisas: uma lanterna para iluminar os locais (ver os endereços) e um walkie-talkie para se comunicar com o que quer que esteja lá (acessar os valores). Vamos aprender a usá-los.

Declarando um Ponteiro (Pegando seu Walkie-Talkie):

Antes de mais nada, você precisa da ferramenta. Em C, um ponteiro é uma variável especial, projetada para uma única tarefa: guardar o endereço de memória de outra variável. Pense nele como um walkie-talkie novinho em folha. Ele ainda não está sintonizado em nenhuma frequência, mas está pronto para ser usado.

A sintaxe para declarar um ponteiro é: *tipo *nome_do_ponteiro;*. O tipo é super importante, pois ele diz qual "linguagem" o seu walkie-talkie fala. Um ponteiro para *int* só pode se comunicar com variáveis do tipo *int*. Um ponteiro para *char* só se comunica com *char*, e assim por diante.



```
main.c

#include <stdio.h>

int main() {
    /* Aqui nós declaramos dois "walkie-talkies".
       Um é especializado em sintonizar frequências
       de números inteiros.*/
    int *ponteiro_para_inteiro;

    /* O outro é para sintonizar frequências de
       caracteres (letras).*/
    char *ponteiro_para_letra;

    /* No momento, eles estão apenas declarados.
       Estão "desligados", apontando para um lugar
       indefinido (lixo de memória). Usá-los agora seria
       perigoso!*/

    return 0;
}
```


O Operador "&" (Encontrando as Coordenadas com a Lanterna):

Ok, você tem o walkie-talkie. E agora? Você precisa de uma frequência para sintonizar! É aqui que entra nossa lanterna, o operador & (conhecido como "endereço de"). Quando você aponta o & para uma variável comum, ele não te mostra o valor dela; ele revela o endereço de memória onde aquele valor está guardado.

Vamos usar o & para descobrir a "localização" do poder da Eleven e sintonizar nosso ponteiro nela.

```
main.c

#include <stdio.h>

int main() {
    /* A Eleven começa com um nível de poder.
       O compilador guarda esse '11' em algum lugar
       da memória.*/
    int poder_da_eleven = 11;

    // Nosso walkie-talkie para inteiros.
    int *ponteiro_para_o_poder;

    /* Agora a mágica acontece!
       Usamos a lanterna '&' para pegar o endereço
       de 'poder_da_eleven' e o guardamos em nosso
       ponteiro.*/
    ponteiro_para_o_poder = &poder_da_eleven;
```

Código continua na próxima página.

Continuação


```
/* Missão cumprida! O ponteiro agora sabe
   EXATAMENTE onde encontrar o poder da Eleven.
   Ele não contém o número 11, mas sim a
   coordenada para chegar até ele.*/
printf("
O endereço de memória do poder da Eleven é: %p\n"
, ponteiro_para_o_poder);
/* A saída será um número hexadecimal esquisito,
   como 0x7ffc1234abcd. Essa é a coordenada!*/

return 0;
}
```

O Operador "*" (Falando pelo Walkie-Talkie):

Sintonia feita! Nosso ponteiro está travado na frequência certa. Como fazemos para **ouvir o que está lá** ou **enviar uma nova mensagem**? Apertamos o botão para falar, e esse botão é o operador "*" (conhecido como "no endereço"). Quando usamos o "*" em um ponteiro já sintonizado, nós acessamos o **valor** que está no endereço para o qual ele aponta. Cuidado para não confundir: o "*" na declaração (`int *p;`) cria o ponteiro. O "*" no uso (`*p = 10;`) acessa o valor no endereço.



 main.c

```
#include <stdio.h>
int main() {
    int poder_da_eleven = 11;
    int *ponteiro_para_o_poder = &poder_da_eleven;

    /* Câmbio, qual o nível de poder?
    Usamos o '*' para LER o valor que está no endereço
    apontado.*/

    printf("
    Pelo walkie-talkie, ouvimos que o poder é: %d\n"
    , *ponteiro_para_o_poder);

    /* Usamos o '*' para MODIFICAR o valor
    no endereço apontado.*/

    *ponteiro_para_o_poder = 99;

    /* Vamos checar a variável original. Ela foi alterada
    à distância!*/
    printf("
    O poder da Eleven, na variável original, agora é: %d\n"
    , poder_da_eleven);

    // Imprime 99. Nós mudamos a variável sem tocar nela
    diretamente
    return 0;
}
```

O Ponteiro NULL (Um Walkie-Talkie Desligado):

E se tivermos um walkie-talkie, mas nenhuma frequência segura para sintonizar? Deixá-lo ligado numa frequência aleatória é perigoso; podemos ouvir estática ou, pior, a voz do Devorador de Mentes (causar um crash no programa). A solução segura é mantê-lo desligado. Em C, o "desligado" dos ponteiros é o NULL.

NULL é um valor especial que significa "este ponteiro não aponta para lugar nenhum". É uma ótima prática de segurança inicializar seus ponteiros com NULL até que você tenha um endereço válido para eles.

```
main.c

#include <stdio.h>

int main() {
    // Declaramos um ponteiro para um portal, mas ainda não o encontramos.
    // Para garantir a segurança, o inicializamos como NULL.
    char *portal_encontrado = NULL;

    // ... algum tempo depois, no código ...

    // O programa pode verificar se o portal já foi encontrado antes de usá-lo.
    if (portal_encontrado == NULL) {
        printf("Ainda não encontramos nenhum portal. Continuem procurando!\n");
    } else {
        printf("Portal encontrado! Coordenadas: %p\n", portal_encontrado);
    }

    return 0;
}
```

Ponteiros, Arrays e Aritmética (Navegando pelas Luzes de Natal):

Lembra das luzes de Natal da Joyce? Aquilo é um array! Em C, ponteiros e arrays são melhores amigos, como Mike e Will. O nome de um array funciona como um ponteiro constante para o seu primeiríssimo elemento. Isso torna a navegação por arrays incrivelmente fácil.

Se temos um ponteiro para o começo do array, podemos simplesmente "somar 1" a ele para pular para o próximo elemento. Isso é chamado de **aritmética de ponteiros**. O compilador é inteligente: se o ponteiro é para `int`, "somar 1" avança 4 bytes (ou o tamanho de um `int`); se for para `char`, avança 1 byte. Ele sabe o tamanho de cada "luz de Natal".



main.c

```
#include <stdio.h>

int main() {
    // O código secreto para abrir o cofre do laboratório.
    int codigo_secreto[] = {8, 0, 8, 1, 9, 8, 6}; // 80's, 1986 ;)

    // Criamos um ponteiro e o apontamos para o INÍCIO do array.
    // Note que não usamos '&' no nome do array. Ele já é um endereço!
    int *ponteiro_para_codigo = codigo_secreto;

    // Lendo o primeiro número usando o ponteiro.
    printf("Primeiro número: %d\n", *ponteiro_para_codigo); // Saída: 8
```

Código continua na próxima página.

Continuação

```
// Vamos para o próximo número. Isso é aritmética de ponteiros!
ponteiro_para_codigo++;

// Lendo o segundo número.
printf("Segundo número: %d\n", *ponteiro_para_codigo); // Saída: 0

// Podemos pular direto para o quarto número (índice 3) a partir do início.
printf("O quarto número é: %d\n", *(codigo_secreto + 3)); // Saída: 1

return 0;
}
```

E aí está! Você agora tem seu kit de sobrevivência completo. Você sabe como pegar um walkie-talkie (* na declaração), sintonizá-lo em uma frequência (&), comunicar-se com o que está lá (* no uso), desligá-lo por segurança (NULL) e andar de uma frequência para outra (aritmética).

Agora que você já domina as ferramentas, está pronto para a verdadeira missão no próximo capítulo: usar esses poderes para realizar tarefas complexas, como abrir e fechar portais por conta própria.

Câmbio, desligo. Prepare-se para a Batalha de Starcourt.

03

BATALHA NO STARCOURT

Abrindo e Fechando Portais

O Poder e a Responsabilidade: utilidades dos ponteiros

Você sobreviveu. Você aprendeu a usar a lanterna (&) e o walkie-talkie (*). Você não tem mais medo do escuro. Agora, é hora de parar de apenas se esconder e começar a lutar. Neste capítulo, vamos usar nossos poderes para manipular o Mundo Invertido a nosso favor: alterar coisas à distância, criar portais de memória do nada e, o mais importante, aprender a fechá-los para que os monstros não escapem.

Ponteiros em Funções (Enviando o Hopper em uma Missão):

Imagine que você está na base (função `main`) e precisa que uma tarefa seja feita lá no Laboratório de Hawkins (outra função). Se você apenas gritar no rádio "aumentem a segurança!", o pessoal do laboratório ouve, mas nada muda na sua base. Isso é passar por **valor**: a função recebe apenas uma cópia da informação.

Mas e se, em vez de gritar, você enviar o Chefe Hopper com um mapa exato do local (ponteiro)? Ele pode ir até lá e executar a ordem, alterando o mundo real. Isso é passar por **referência** (usando um ponteiro), e é um dos superpoderes mais úteis de C.

```
main.c

#include <stdio.h>

// Esta função recebe um ponteiro para um inteiro.
// Ela recebe o MAPA para a variável de coragem.
void missao_para_aumentar_a_coragem(int *ponteiro_para_a_coragem) {

    printf("Hopper chegou ao local. Coragem atual: %d\n",
    *ponteiro_para_a_coragem);

    // Usando o mapa (*), ele acessa o local e aumenta a coragem.
    *ponteiro_para_a_coragem = *ponteiro_para_a_coragem + 75;

    printf("Missão cumprida. Hopper alterou a coragem para: %d\n",
    *ponteiro_para_a_coragem);
}

int main() {
    int coragem_do_steve = 20;
    printf("Steve (o babá) começa com %d de coragem.\n",
    coragem_do_steve);

    // Não passamos a coragem (20), passamos o ENDEREÇO de onde ela
    está.
    // Enviamos o Hopper com o mapa!
    missao_para_aumentar_a_coragem(&coragem_do_steve);

    // A variável original na função main foi alterada!
    printf("De volta à base, a coragem do Steve agora é: %d\n",
    coragem_do_steve);

    return 0;
}
```

Alocação Dinâmica (Abrindo o Portal com malloc):

Até agora, todas as nossas variáveis tinham um tamanho fixo, decidido antes de o programa começar. Mas e se a gente precisar de mais espaço *durante* a execução? E se encontrarmos um grupo de sobreviventes e precisarmos de uma lista para anotar o nome de todos, mas não sabemos quantos são?

É para isso que serve a alocação dinâmica. A função `malloc` (memory allocation) é a nossa máquina russa: ela abre um portal para a memória sob demanda. Você pede um pedaço de memória de um certo tamanho, e `malloc` te devolve um ponteiro para esse novo espaço.

```
main.c

#include <stdio.h>
#include <stdlib.h> // As funções malloc e free vivem aqui!

int main() {
    int num_demogorgons;
    printf("URGENTE! Quantos Demogorgons foram avistados perto de Starcourt? ");
    scanf("%d", &num_demogorgons);

    // Precisamos de um array para guardar a energia de cada um,
    // mas só agora sabemos o tamanho necessário.

    // ABRINDO O PORTAL: Pedimos memória para 'num_demogorgons'
    // inteiros.
    int *niveis_de_energia = (int*) malloc(sizeof(int) *
num_demogorgons);

    // ABRINDO O PORTAL: Pedimos memória para 'num_demogorgons'
    // inteiros.
```

Código continua na próxima página.

Continuação

```
// ABERTURA DE PORTAL PODE FALHAR! Se não houver memória,
malloc retorna NULL.
// Sempre temos que checar!
if (niveis_de_energia == NULL) {
    printf("FALHA CRÍTICA! Não há memória para rastrear os
monstros. Abortar!\n");
    return 1; // Termina o programa com um código de erro.
}

printf("Portal de memória aberto com sucesso! Espaço para %d
leituras criado.\n", num_demogorgons);
// Agora podemos usar 'niveis_de_energia' como se fosse um
array normal.
// niveis_de_energia[0] = 100; ...

// ... código para usar a memória ...

// ATENÇÃO: Se abrimos, temos que fechar! (próximo tópico)
free(niveis_de_energia); // Não se esqueça disso!

return 0;
}
```

Liberando Memória (Fechando o Portal com free):

Este é o momento de maior responsabilidade. Se a máquina russa abre um portal com `malloc` e nós simplesmente vamos embora, o portal fica aberto. Do Mundo Invertido da memória, começam a vazar "monstros" chamados **memory leaks** (vazamentos de memória). Cada vazamento consome um pouco dos recursos do computador, e com o tempo, eles podem deixar seu programa lento ou até mesmo travá-lo completamente.

A nossa Eleven, capaz de fechar o portal, é a função `free()`. Para cada `malloc` que você chama, você **deve**, em algum momento, chamar um `free()` correspondente, passando o mesmo ponteiro. Isso devolve a memória para o sistema, fechando o portal e mantendo tudo seguro.

```
● ● ● main.c

#include <stdio.h>
#include <stdlib.h>

int main() {
    int num_demogorgons = 3;

    // 1. ABRIR O PORTAL
    int *niveis_de_energia = (int*) malloc(sizeof(int) *
num_demogorgons);

    if (niveis_de_energia == NULL) {
        return 1; // Abortar
    }

    // 2. USAR A MEMÓRIA
    printf("Rastreando a energia dos Demogorgons:\n");
    for(int i = 0; i < num_demogorgons; i++) {
        niveis_de_energia[i] = 50 + i * 25; // Energia aumenta a
cada um
        printf("  - Demogorgon %d: Nível de energia %d\n", i + 1,
niveis_de_energia[i]);
    }
}
```

Código continua na próxima página.

Continuação

```
// 3. A BATALHA ACABOU. FECHAR O PORTAL!
printf("\nPerigo neutralizado. Fechando o portal de
memória...\n");
free(niveis_de_energia);

// Boa prática: Apontar o ponteiro para NULL depois de liberar,
// para evitar usá-lo acidentalmente (ponteiro selvagem).
niveis_de_energia = NULL;

printf("Portal fechado. O sistema está seguro.\n");

return 0;
}
```

Ponteiros para Ponteiros (O Mapa que Leva a Outro Mapa):

Parece assustador, mas a ideia é simples. E se o Hopper encontrasse um cofre (ponteiro) e, dentro dele, não estivesse o tesouro, mas sim outro mapa (outro ponteiro) que leva ao tesouro final? Isso é um ponteiro para ponteiro (**). É uma variável que guarda o endereço de outra variável de ponteiro.

O uso mais comum é para criar uma lista de nomes (um array de strings). Como cada "string" em C já é um ponteiro (char *), uma lista delas se torna um array de ponteiros (char *lista[]), e um ponteiro para essa lista é, portanto, um char **.


```
#include <stdio.h>

int main() {
    // Cada nome é um ponteiro para o primeiro caractere da string.

    char *membro1 = "Eddie";
    char *membro2 = "Dustin";
    char *membro3 = "Mike";
    char *membro4 = "Lucas";

    // Criamos uma lista de membros do Hellfire Club.
    // Esta lista é um array de ponteiros de char.
    char *clube_do_inferno[] = {membro1, membro2, membro3, membro4,
NULL}; // NULL para marcar o fim.

    // Criamos o "mapa para o mapa".
    // Este ponteiro aponta para o início da lista de nomes.
    char **mestre_da_masmorra = clube_do_inferno;

    printf("Membros do Hellfire Club, reportem-se:\n");

    // Navegamos pela lista de mapas.
    while(*mestre_da_masmorra != NULL) {
        // *mestre_da_masmorra nos dá o mapa (o 'char *', ex:
membro1)
        // O printf sabe como seguir esse mapa para imprimir a
string.
        printf(" - %s está presente!\n", *mestre_da_masmorra);

        // Pulamos para o próximo mapa na lista.
        mestre_da_masmorra++;
    }

    return 0;
}
```


Enfrentando o Vecna (hora da prática):

É de extrema importância que após aprender algo teoricamente, que coloque seus novos conhecimentos a prova. Dito isso, trago dos cantos mais profundos do mundo invertido, alguns exercícios de ponteiro para que então domine esse grande poder.

1) Crie a função ordena que receba como parâmetro três variáveis (passagem por referência) e ordene os valores destas variáveis em ordem crescente, respeitando a ordem da passagem dos parâmetros. Por exemplo, considerando as seguintes variáveis com os valores: $a=7$, $b=3$ e $c=5$, se as mesmas forem passadas por referência, nesta ordem, para a função pedida, o valor das mesmas após o retorno da função deve ser $a=3$, $b=5$ e $c=7$.

2) Crie uma matriz 5x5 de ponteiros pra inteiros que represente uma matriz identidade. Cada valor 0 na matriz deve ser uma referência para a mesma região na memória, assim como cada valor 1 desta matriz. Em seguida, crie uma função que receba esta matriz e altere o valor da diagonal principal para n (outro parâmetro da função). Escreva também uma função que receba a matriz como parâmetro e a escreva na tela.

3) Crie uma função chamada `criaMatriz` que receba como parâmetro a altura e a largura de uma matriz e retorne a memória alocada para uma matriz do tipo `float`. Dica: a função deverá retornar um ponteiro para ponteiro de `float`. Em seguida, preencha a matriz retornada com valores de sua preferência e a escreva na tela para testar.

4) Faça um programa que leia números inteiros digitados pelo usuário e os armazene em um vetor. O programa deve ler os números digitados até que o usuário digite um número negativo (e este não deve ser armazenado no vetor). A memória utilizada no programa deve ser alocada dinamicamente. Ao iniciar, o tamanho do vetor deve ser 5. Antes de armazenar um novo valor ao vetor, o programa deve verificar se o tamanho do vetor não será excedido e, caso isso vá ocorrer, o tamanho do vetor deve ser aumentado para que este permita o armazenamento de duas vezes mais elementos do que o permitido no momento (ou seja, na primeira vez que o tamanho do vetor é aumentado, este passa a possuir tamanho 10, na segunda vez, tamanho 20, e assim por diante). Utilize neste programa as funções `malloc` e `free` (não esqueça de liberar memória não utilizada, caso seja necessário).

E com isso, a nossa aventura chega ao fim. Você enfrentou o medo, aprendeu a manusear as ferramentas e, finalmente, usou o poder dos ponteiros para controlar a própria estrutura da memória. Você viu que eles não são o monstro, mas sim a arma secreta que dá a C sua velocidade e flexibilidade.

Você abriu portais e, de forma responsável, os fechou. Você venceu. O poder agora é seu.

Câmbio final, desligo.

AGRADECIMENTOS

Obrigado por ler até aqui!!

Este e-book foi gerado por IA's (ChatGPT e Google Gemini) e supervisionado por uma pessoa responsável (eu mesmo)

A criação deste material foi feita com objetivo de pôr em prática os meus conhecimentos adquiridos durante o bootcamp da DIO sobre IA e Java.

O e-book leva uma carga educacional voltada para ponteiros e que, apesar da atmosfera lúdica, teve suas informações verificadas e aprovadas. Entretanto, este e-book não substitui um tutor, recomendo que leia-o e qualquer dúvida no conteúdo ou nos exercícios, consulte um professor.

https://github.com/ferreiraPepe/e-bookIA_DIO

Acima está um link para o repositório em que este arquivo estará, junto com os prompts utilizados para gera-lo.