

Prof. esp. Thalles Canela

- **Graduado:** Sistemas de Informação - Wyden Facimp
- **Pós-graduado:** Segurança em redes de computadores - Wyden Facimp
- **Professor:** Todo núcleo de T.I. (Graduação e Pós) - Wyden Facimp
- **Diretor:** SCS
- **Gerente de Projetos:** Motoca Systems

Redes sociais:

- **Linkedin:** <https://www.linkedin.com/in/thalles-canela/>
- **YouTube:** <https://www.youtube.com/aXR6CyberSecurity>
- **Facebook:** <https://www.facebook.com/axr6PenTest>
- **Instagram:** https://www.instagram.com/thalles_canela
- **Github:** <https://github.com/ThallesCanela>
- **Github:** <https://github.com/aXR6>
- **Twitter:** <https://twitter.com/Axr6S>



Introdução aos Ponteiros na Linguagem C

Objetivos

- Compreender o conceito de ponteiros e sua importância na linguagem C.
- Aprender como declarar e inicializar ponteiros.
- Explorar operações básicas com ponteiros.

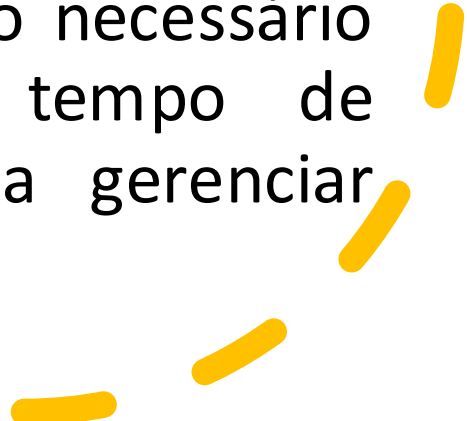


Conteúdo da Aula

- Alocação dinâmica de memória.



Alocação dinâmica de memória.

- A alocação dinâmica de memória é um conceito fundamental em programação que permite reservar espaço na memória do computador durante a execução de um programa.
 - Em linguagens como C, a alocação dinâmica é realizada por meio das funções **malloc()**, **calloc()** e **realloc()**.
 - Esse processo é especialmente útil quando você não sabe o tamanho exato necessário para armazenar dados em tempo de compilação ou quando precisa gerenciar eficientemente recursos.
- 

Usando malloc():

- A função malloc() (memory allocation) é usada para alocar uma quantidade específica de memória durante a execução do programa.
- Ela retorna um ponteiro para a primeira localização de memória alocada, ou retorna NULL se a alocação falhar.



Usando malloc():

c

Copy code

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

int main() {
    int *ptr; // Ponteiro para armazenar o endereço alocado
    ptr = (int *)malloc(5 * sizeof(int)); // Alocação de 5 inteiros

    if (ptr == NULL) {
        printf("A alocação de memória falhou.\n");
        return 1;
    }

    // Utilize ptr como um array normal de inteiros
    // ...

    // Libere a memória alocada quando não for mais necessária
    free(ptr);

    return 0;
}
```

Usando calloc():

- A função calloc() (contiguous allocation) é semelhante à malloc(), mas também inicializa os valores alocados com zero.

c

 Copy code

```
ptr = (int *)realloc(ptr, 10 * sizeof(int)); // Redimensionamento para 10
```

Usando calloc():

Usando
realloc():

A função realloc() (re-allocation) permite redimensionar a memória previamente alocada.

Ela é usada quando você precisa aumentar ou diminuir o tamanho de uma alocação existente.

c

 Copy code

```
ptr = (int *)realloc(ptr, 10 * sizeof(int)); // Redimensionamento para 10
```

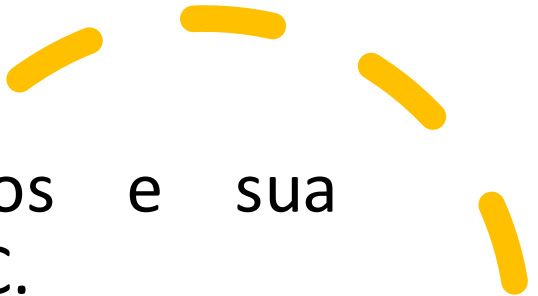
Usando realloc():

free()

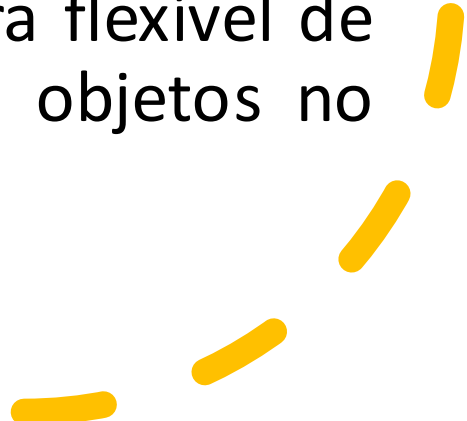
- É importante lembrar de sempre liberar a memória alocada quando ela não for mais necessária, usando a função `free()`. A não liberação de memória pode levar a vazamentos de memória (memory leaks), onde o programa consome mais e mais memória à medida que é executado.



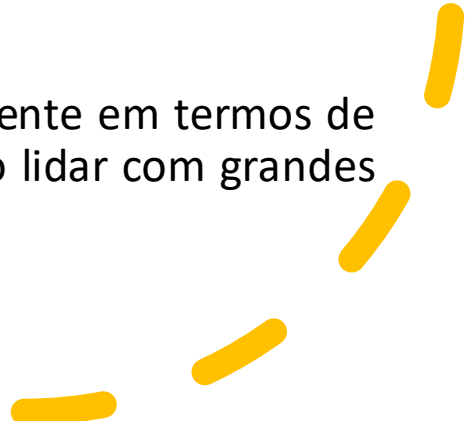
Introdução aos Ponteiros

- 
- Definição de ponteiros e sua utilidade na linguagem C.
 - Como os ponteiros lidam com endereços de memória.

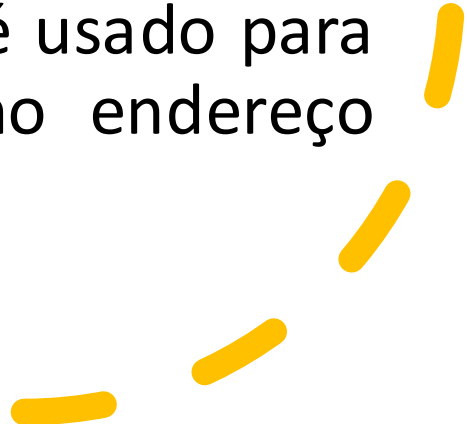
Definição de Ponteiros e Sua Utilidade na Linguagem C

- Ponteiros são variáveis especiais em linguagens de programação, como C, que armazenam endereços de memória como seu valor.
 - Eles são utilizados para acessar diretamente os dados armazenados na memória, permitindo operações eficientes e manipulação avançada de dados.
 - Ponteiros fornecem uma maneira flexível de trabalhar com a memória e os objetos no programa.
- 

Utilidade dos Ponteiros na Linguagem C:


- **Acesso Direto à Memória:** Ponteiros permitem acessar e modificar os valores armazenados em endereços específicos da memória, o que é essencial para a eficiência e flexibilidade do código.
 - **Passagem de Parâmetros por Referência:** Ao passar um ponteiro como parâmetro para uma função, você pode modificar os valores do argumento original diretamente na memória, em vez de criar uma cópia local.
 - **Gerenciamento de Recursos:** Ponteiros são amplamente usados para gerenciar recursos como alocação dinâmica de memória. Eles permitem a criação de estruturas de dados dinâmicas que podem crescer e encolher conforme necessário.
 - **Manipulação de Strings e Arrays:** Arrays são tratados como ponteiros em C, o que facilita a manipulação de strings e outras estruturas de dados complexas.
 - **Eficiência:** O uso de ponteiros pode ser mais eficiente em termos de memória e tempo de execução, especialmente ao lidar com grandes volumes de dados.
- 

Como os Ponteiros Lidam com Endereços de Memória:

- Os ponteiros armazenam endereços de memória, permitindo o acesso aos dados localizados nesses endereços.
 - Ao declarar um ponteiro, ele não contém o valor real dos dados, mas sim o endereço de onde esses dados estão armazenados.
 - O operador de referência & é usado para obter o endereço de uma variável.
 - O operador de desreferência * é usado para acessar o valor armazenado no endereço apontado por um ponteiro.
- 

Exemplo:

c

 Copy code

```
#include <stdio.h>

int main() {
    int x = 10;           // Variável inteira
    int *ptr;             // Ponteiro para inteiro

    ptr = &x;             // Ponteiro ptr armazena o endereço de x
    printf("Valor de x: %d\n", *ptr); // Imprime o valor de x usando o pont

    return 0;
}
```



Explicação

- Neste exemplo, ptr armazena o endereço de memória de x.
- Ao usar *ptr, estamos acessando o valor de x através do ponteiro.
- Isso demonstra como os ponteiros lidam diretamente com endereços de memória para acessar os dados correspondentes.

Declaração e Inicialização de Ponteiros

- Sintaxe para declarar ponteiros.
- Exemplos de inicialização de ponteiros.



Sintaxe para Declarar Ponteiros

- A sintaxe para declarar um ponteiro em C envolve o uso do operador asterisco (*) antes do nome da variável.
- O tipo do ponteiro deve coincidir com o tipo de dado que ele irá apontar.
- Aqui está a forma básica da declaração de um ponteiro:



C

```
tipo_de_dado *nome_do_ponteiro;
```

Sintaxe para Declarar Ponteiros

- `tipo_de_dado`: Indica o tipo de dado que o ponteiro irá apontar (int, char, float, etc.).
- `nome_do_ponteiro`: É o nome da variável que representa o ponteiro.

```
int x = 10;  
int *ptr;           // Declaração do ponteiro  
ptr = &x;           // Inicialização com o endereço de x
```

Exemplos de Inicialização de Ponteiros

- Inicialização com um endereço válido:

c

```
int *ptr = NULL;    // Inicialização com um po
```

Exemplos de Inicialização de Ponteiros

- Inicialização com NULL (ponteiro nulo):

c

```
int *dyn_ptr;  
dyn_ptr = (int *)malloc(sizeof(int)); // Alocação dinâmica
```

Inicialização com
resultado de
alocação dinâmica
de memória:

c

```
const int *const_ptr; // Ponteiro para uma con
```

Inicialização com um
ponteiro constante:

c

```
int *ptr1, *ptr2;    // Declaração de múltiplos ponteiros
```

Inicialização de
múltiplos ponteiros:

Sintaxe para Declarar Ponteiros

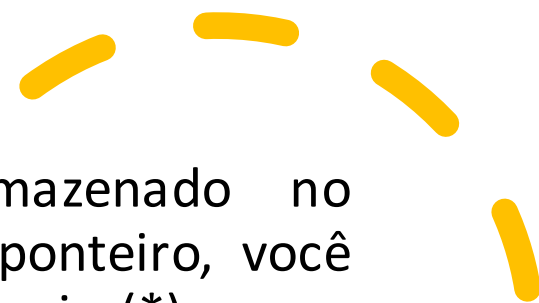
- Lembre-se de que a declaração de um ponteiro não aloca automaticamente memória para o objeto apontado.
- A inicialização de um ponteiro com um endereço válido permite que ele aponte para uma variável já existente na memória.
- Ao trabalhar com alocação dinâmica de memória, é importante liberar a memória alocada quando não for mais necessária usando a função `free()`.



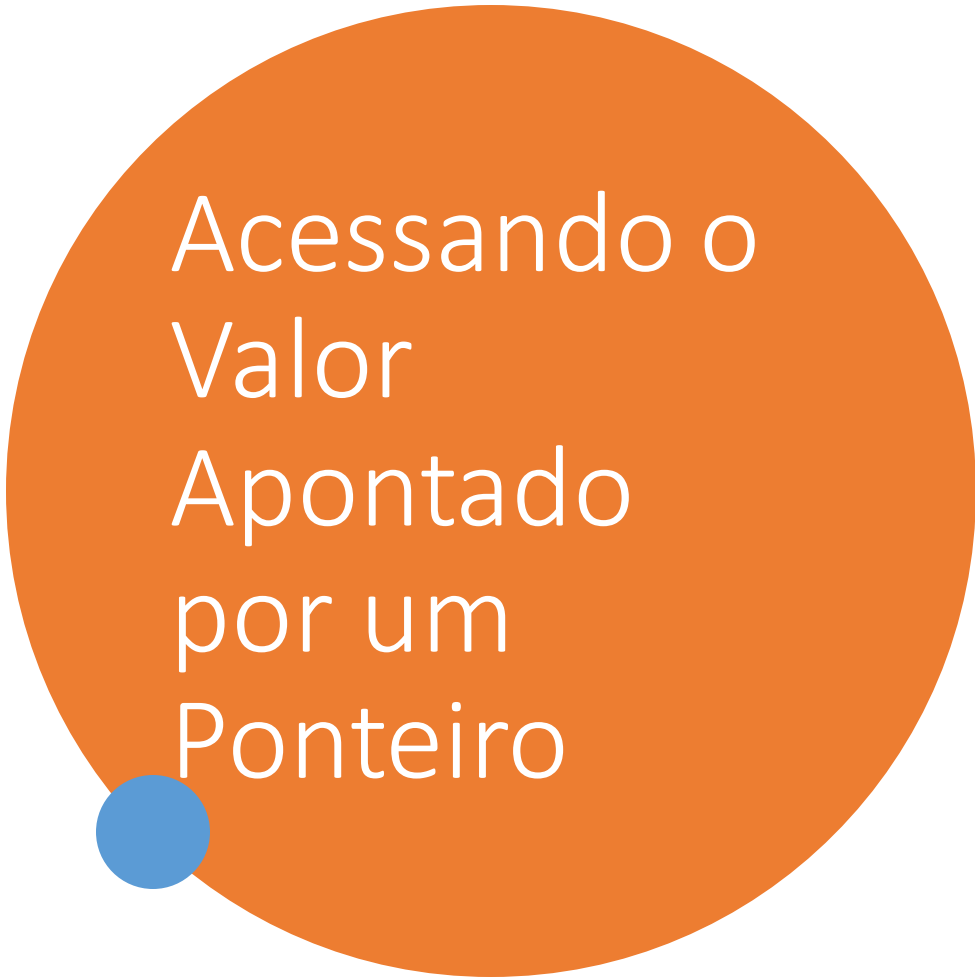
Operações com Ponteiros

- Acessando o valor apontado por um ponteiro.
- Utilização do operador de referência (&) e do operador de desreferência (*).





Acessando o Valor Apontado por um Ponteiro



- Para acessar o valor armazenado no endereço apontado por um ponteiro, você utiliza o operador de desreferência (*).
- O operador de desreferência é colocado antes do nome do ponteiro para obter o valor no endereço que o ponteiro está apontando.
- Isso é especialmente útil quando você quer trabalhar diretamente com os dados referenciados pela variável apontada pelo ponteiro.

Acessando o Valor Apontado por um Ponteiro

```
c Copy code  
  
#include <stdio.h>  
  
int main() {  
    int x = 42;  
    int *ptr = &x; // Ponteiro ptr aponta para o endereço de x  
  
    printf("Valor de x: %d\n", *ptr); // Usando o operador * para acessar o valor  
  
    return 0;  
}
```


- Nesse exemplo, `*ptr` é usado para acessar o valor armazenado no endereço apontado por `ptr`, ou seja, o valor de `x` (que é 42) é impresso na tela.

c

 Copy code

```
int x = 10;  
int *ptr = &x;    // Ponteiro ptr aponta para o endereço de x usando &
```


Utilização do Operador de Referência (&) e do Operador de Desreferência (*)

- Operador de Referência (&):
 - O operador de referência (&) é usado para obter o endereço de memória de uma variável. Ele fornece o endereço no qual a variável está armazenada.
- 

```
int x = 42;  
int *ptr = &x;  
int valor = *ptr;    // Usando * para acessar o valor apontado por ptr
```

Utilização do Operador de Referência (&) e do Operador de Desreferência (*)

- Operador de Desreferência (*):
- O operador de desreferência (*) é usado para acessar o valor armazenado no endereço apontado por um ponteiro. Ele é colocado antes do nome do ponteiro.



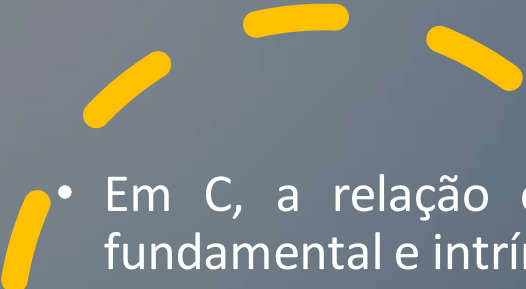
Utilização do Operador de Referência (&) e do Operador de Desreferência (*)


- Em resumo, o operador de desreferência * é usado para acessar o valor de uma variável apontada por um ponteiro.
- O operador de referência & é usado para obter o endereço de memória de uma variável.
- Esses operadores são fundamentais quando se trabalha com ponteiros, pois permitem acessar e modificar os dados referenciados de forma direta.



Ponteiros e Vetores

- Introdução à relação entre ponteiros e vetores.
- Como os nomes de vetores atuam como ponteiros.

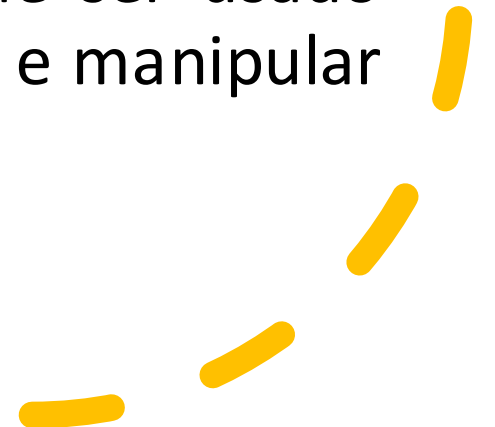
- 
- Em C, a relação entre ponteiros e vetores é fundamental e intrínseca.
 - Na verdade, os vetores são tratados como ponteiros em muitas situações, e entender essa relação é crucial para uma compreensão mais profunda da linguagem.
 - Vetores são apenas uma forma de gerenciar sequências de dados em que os elementos são armazenados de forma contígua na memória, e os ponteiros são utilizados para acessar esses elementos de maneira eficiente.



Introdução à Relação entre Ponteiros e Vetores


Como os Nomes de Vetores Atuam como Ponteiros

- Quando você declara um vetor em C, o nome do vetor atua como um ponteiro constante para o primeiro elemento do vetor.
- Isso significa que o nome do vetor armazena o endereço de memória do primeiro elemento do vetor.
- Portanto, o nome do vetor pode ser usado como um ponteiro para acessar e manipular os elementos do vetor.



Como os
Nomes de
Vetores
Atuam como
Ponteiros

c

 Copy code

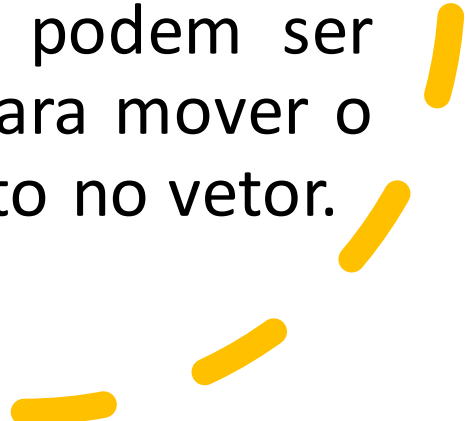
```
#include <stdio.h>

int main() {
    int numeros[5] = {10, 20, 30, 40, 50};

    printf("Primeiro elemento: %d\n", numeros[0]); // Acesso usando índice
    printf("Usando nome do vetor: %d\n", *numeros); // Acesso usando o nome do


    return 0;
}
```

Como os Nomes de Vetores Atuam como Ponteiros

- Nesse exemplo, `numeros` é um vetor de inteiros.
 - O nome `numeros` atua como um ponteiro para o primeiro elemento do vetor.
 - Você pode acessar o primeiro elemento usando `numeros[0]` ou usando `*numeros`, ambos resultam no mesmo valor.
 - Além disso, os operadores de incremento e decremento (`++` e `--`) também podem ser usados com o nome do vetor para mover o ponteiro para o próximo elemento no vetor.
- 

Como os
Nomes de
Vetores
Atuam como
Ponteiros

c

 Copy code

```
#include <stdio.h>

int main() {
    int numeros[3] = {10, 20, 30};

    int *ptr = numeros; // Atribuindo o nome do vetor a um ponteiro

    printf("%d\n", *ptr); // Imprime o primeiro elemento (10)
    ptr++;                // Move o ponteiro para o próximo elemento
    printf("%d\n", *ptr); // Imprime o segundo elemento (20)

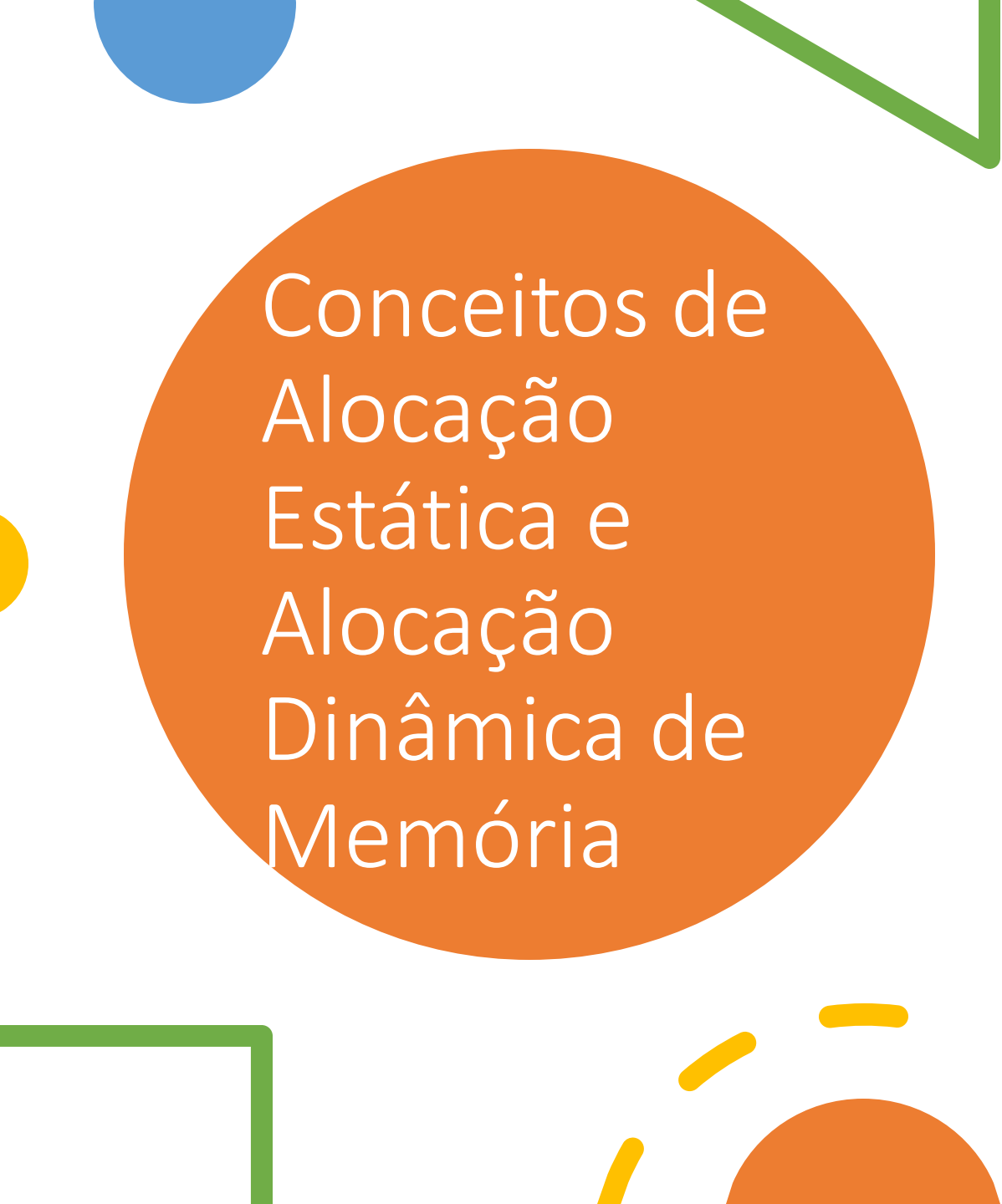
    return 0;
}
```

Recapitulação e Tarefa de Casa

- Resumir os pontos-chave da aula.
- Atribuir uma tarefa de casa relacionada à manipulação de ponteiros em arrays.

Utilização de Ponteiros para Alocação Dinâmica de Memória

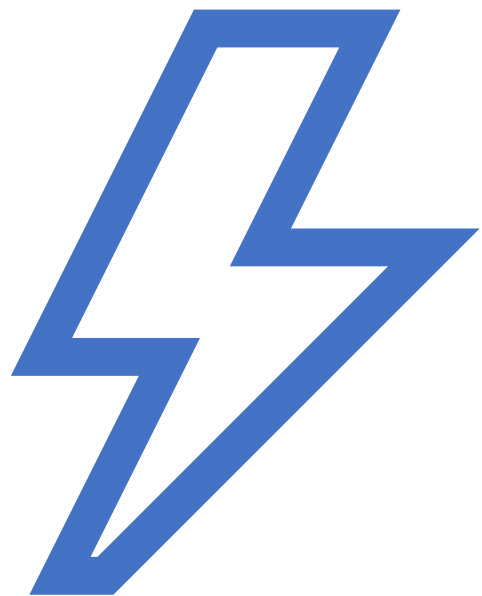
- Entender os conceitos de alocação estática e alocação dinâmica de memória.
- Aprender a utilizar a alocação dinâmica de memória com ponteiros na linguagem C.



Conceitos de
Alocação
Estática e
Alocação
Dinâmica de
Memória

Alocação Estática de Memória:

- Na alocação estática de memória, o espaço para variáveis é reservado durante a compilação e permanece constante durante a execução do programa.
- Isso significa que você precisa saber o tamanho necessário para as variáveis em tempo de compilação.
- Variáveis globais e variáveis locais em funções que não usam alocação dinâmica são exemplos de alocação estática.



Vantagens:

- Simplicidade.
- Tempo de acesso rápido.
- Não há preocupação com vazamento de memória.



Desvantagens:

- Tamanho fixo.
- Não é adequado para estruturas de dados que precisam crescer ou diminuir dinamicamente.

Alocação Dinâmica de Memória:

- A alocação dinâmica de memória ocorre em tempo de execução e permite que você aloque e libere memória conforme necessário.
- Isso é especialmente útil quando você não sabe o tamanho necessário antecipadamente ou quando precisa criar estruturas de dados flexíveis.



Vantagens:

- Flexibilidade no uso de memória.
- Acomoda estruturas de dados dinâmicas.
- Redimensionamento de memória possível.



Desvantagens:

- Requer gerenciamento cuidadoso para evitar vazamentos de memória.
- Utilização da Alocação Dinâmica de Memória com Ponteiros em C

