



# TÉCNICAS DE PROGRAMAÇÃO

Aula 07 – Alocação estática e dinâmica de memória



# Objetivos de Aprendizagem

1. Diferenciar a alocação de memória estática da alocação de memória dinâmica;
2. Desenvolver programas com alocação de memória dinâmica com uso de ponteiros;
3. Manipular programas com nível de apontamento;
4. Desenvolver programas com alocação de memória dinâmica com uso de ponteiros para ponteiros.



# Gerenciamento de memória

- A compreensão da memória é um aspecto importante da programação em C.
- Quando você declara uma variável usando um tipo de dados básico, C aloca automaticamente espaço para a variável em uma área da memória chamada **heap** (pilha, do inglês).





# Gerenciamento de memória

- Uma variável **int**, por exemplo, geralmente recebe 4 bytes quando declarada. Sabemos disso usando o operador **sizeof**:

```
int x;  
printf("%d", sizeof(x)); /* saída: 4 */
```





# Gerenciamento de memória


- Como outro exemplo, um vetor com um tamanho especificado recebe blocos contínuos de memória com cada bloco do tamanho de um elemento:

```
int arr[10];  
printf("%d", sizeof(arr)); /* saída: 40 */
```





# Gerenciamento de memória

- Desde que o seu programa declare explicitamente um tipo de dado básico ou o tamanho de um array (vetor ou matriz) de forma estática, ou seja, antecipada, a memória é gerenciada automaticamente.
  - No entanto, você provavelmente já desejou implementar um programa em que o tamanho da matriz ou do vetor é indeciso até o tempo de execução.
- 




# Alocação Dinâmica de Memória

- A alocação dinâmica de memória é o processo de alocar e liberar memória, conforme necessário.
- Agora você pode solicitar em tempo de execução o número de elementos do vetor e, em seguida, criar um vetor com número de elementos que desejar.






# Alocação Dinâmica de Memória

- A memória dinâmica é gerenciada com ponteiros que apontam para blocos de memória recém-alocados em uma área chamada **heap** (pilha, em inglês).
  - A biblioteca **stdlib.h** inclui funções de gerenciamento de memória.
- 






# Alocação Dinâmica de Memória

- A instrução **#include <stdlib.h>** na parte superior do seu programa fornece acesso ao seguinte:
    - **malloc(bytes)** retorna um ponteiro para um bloco contíguo de memória com tamanho de bytes.
    - **calloc(num\_items, item\_tam)** retorna um ponteiro para um bloco contínuo de memória que possui **num\_items** itens, cada um com tamanho **item\_tam** bytes.
      - Normalmente usado para arrays, estruturas e outros tipos de dados derivados.
      - A memória alocada é inicializada em 0.
- 



# Alocação Dinâmica de Memória

- **realloc(ptr, bytes)** redimensiona a memória apontada por **ptr** para o tamanho de bytes.
    - A memória alocada recentemente não é inicializada.
  - **free(ptr)** libera o bloco de memória apontado por **ptr**.
    - Quando você não precisar mais de um bloco de memória alocada, use a função **free()** para tornar o bloco disponível para ser alocado novamente.
- 

# A função malloc

- A função **malloc()** aloca um número especificado de bytes contínuos na memória.

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

```
int *ptr;
```

```
/* um bloco de memória de 10 inteiros */
```

```
ptr = malloc(10 * sizeof(*ptr));
```

```
if (ptr != NULL) {
```

```
    *(ptr + 2) = 50;
```

```
/* atribui 50 ao terceiro inteiro */
```

```
}
```




# A função malloc

- **malloc** retorna um ponteiro para a memória alocada.
- Observe que **sizeof** foi aplicado a **\* ptr** em vez de **int**, tornando o código mais robusto caso a declaração **\*ptr** seja alterada para um tipo de dados diferente posteriormente.






# A função malloc

- A memória alocada é contínua e pode ser tratada como um vetor.
  - Em vez de usar colchetes [] para se referir a elementos, a aritmética do ponteiro é usada para percorrer um vetor.
  - É recomendável usar + para se referir aos elementos do vetor.
  - Usar ++ ou + = altera o endereço armazenado pelo ponteiro.
- 



# A função malloc

- Se a alocação não tiver êxito, será retornado **NULL**. Por isso, você deve incluir código para verificar se há um ponteiro **NULL**.
  - Uma vetor bidimensional simples requer (linhas \* colunas) \* tamanho dos (tipos de dados) bytes de memória.
- 



# A função free


- A função **free()** é uma função de gerenciamento de memória chamada para liberar memória.
- Ao liberar memória, você disponibiliza mais para uso posteriormente no seu programa.





# Exemplo com free


```
int* ptr = malloc(10 * sizeof(*ptr));  
if (ptr != NULL)  
    *(ptr + 2) = 50;  
/* atribui 50 ao terceiro inteiro */  
printf("%d\n", *(ptr + 2));  
  
free(ptr);
```







# A função calloc

- A função **calloc()** aloca memória com base no tamanho de um item específico, como uma estrutura.
  - O programa a seguir usa **calloc** para alocar memória para uma estrutura e **malloc** para alocar memória para a cadeia de caracteres na estrutura.
- 



# Exemplo com calloc (I)

```
typedef struct {  
    int num;  
    char *info;  
} registro;
```

```
registro *regs;
```

```
int num_regs = 2;
```

```
int k;
```


```
char str[ ] = "Isto é uma informação";
```






## Exemplo com calloc (II)

```
regs = calloc(num_recs, sizeof(registro));  
if (regs != NULL) {  
    for (k = 0; k < num_recs; k++) {  
        (regs+k)->num = k;  
        (regs+k)->info = malloc(sizeof(str));  
        strcpy((regs+k)->info, str);  
    }  
}
```






# A função calloc

- **calloc** aloca blocos de memória dentro de um bloco contínuo de memória para uma matriz de elementos de estrutura.
  - Você pode navegar de uma estrutura para a próxima com aritmética de ponteiro.
  - Depois de alocar espaço para uma estrutura, a memória deve ser alocada para a sequência dentro da estrutura.
- 



# A função calloc

- O uso de um ponteiro para o membro **info** permite que uma string de qualquer comprimento seja armazenada.
  - Estruturas alocadas dinamicamente são a base de listas vinculadas e árvores binárias, além de outras estruturas de dados.
- 



# A função realloc

- A função **realloc()** expande um bloco atual para incluir memória adicional.
- O **realloc** deixa o conteúdo original na memória e expande o bloco para permitir mais armazenamento.





# Exemplo com realloc

```
int *ptr;
ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL) {
    *(ptr + 2) = 50;
    /* atribui 50 ao terceiro inteiro */
}
ptr = realloc(ptr, 100 * sizeof(*ptr));

*(ptr + 30) = 75;
```

