



# Estruturas de Dados 1

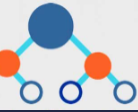
## 07 – Alocação de Memória

Antonio Angelo de Souza Tartaglia  
angelot@ifsp.edu.br

# Estrutura de Dados 1

## Alocação Dinâmica de Memória

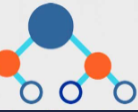
- Uma variável é uma posição de memória que armazena um dado que pode ser usado pelo programa.
- No entanto, por ser uma posição previamente reservada, uma variável deve ser declarada durante o desenvolvimento do programa;
- Infelizmente, nem sempre é possível saber o quanto de memória um programa vai precisar.



# Estrutura de Dados 1

## Alocação Dinâmica de Memória

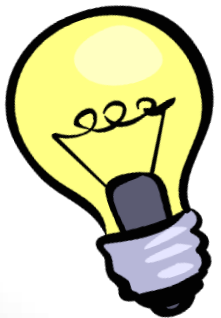
- Imagine que você está desenvolvendo para uma empresa, um programa que processe os valores dos salários de seus funcionários. Para resolver esse problema poderia ser declarado um vetor do tipo `float` bem grande, com por exemplo 1000 posições.
- Esse vetor parece ser a solução possível para o problema. Infelizmente, essa solução gera outros dois problemas:
- Se a empresa tiver menos de 1000 funcionários – será um desperdício de memória. Um vetor deste tamanho só deve ser declarado se realmente as 1000 posições forem utilizadas.
- Se a empresa possuir mais de 1000 funcionários – o vetor será insuficiente para lidar com os dados de todos os funcionários. Seu programa não atenderá as necessidades da empresa.



# Estrutura de Dados 1

## Alocação Dinâmica de Memória

- Devemos considerar de alguns fatos:
  - Quando declaramos um vetor, dizemos ao compilador para reservar uma certa quantidade de memória para o armazenamento dos seus elementos. Porém, nesse modo de declaração, a quantidade de memória será fixa, inalterável;
  - Vetores são agrupamentos **sequenciais** de dados de um mesmo tipo na memória;
  - Um ponteiro é uma variável que guarda um endereço de um dado na memória;
  - O nome do vetor declarado, é apenas um ponteiro que aponta para o primeiro elemento do vetor.



Posso solicitar um bloco de memória e colocar sua primeira posição em um ponteiro, e com esse ponteiro acessar as posições de memória como se fosse um vetor?

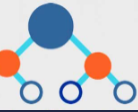


# Estrutura de Dados 1

## Alocação Dinâmica de Memória

- A linguagem C permite **alocar** (reservar) **dinamicamente** (em tempo de execução) blocos de memória utilizando ponteiros. A esse processo dá-se o nome ***alocação dinâmica***. A alocação dinâmica permite ao programador “criar” vetores ou *arrays* em tempo de execução, ou seja, alocar memória para novos *arrays* quando o programa está sendo executado, e não apenas quando se está escrevendo o programa.
- Essa estratégia é utilizada quando não se sabe ao certo quanto de memória será necessário para armazenar os dados com que se quer trabalhar. Desse modo, pode-se definir o tamanho do vetor ou *array* em tempo de execução, evitando assim o desperdício de memória.

Alocação dinâmica consiste em requisitar um espaço de memória ao sistema operacional, em tempo de execução, que então devolve para o seu programa o endereço do início desse espaço alocado. Este endereço devolvido pelo sistema operacional, por questões óbvias, deverá ser armazenado em um ponteiro.



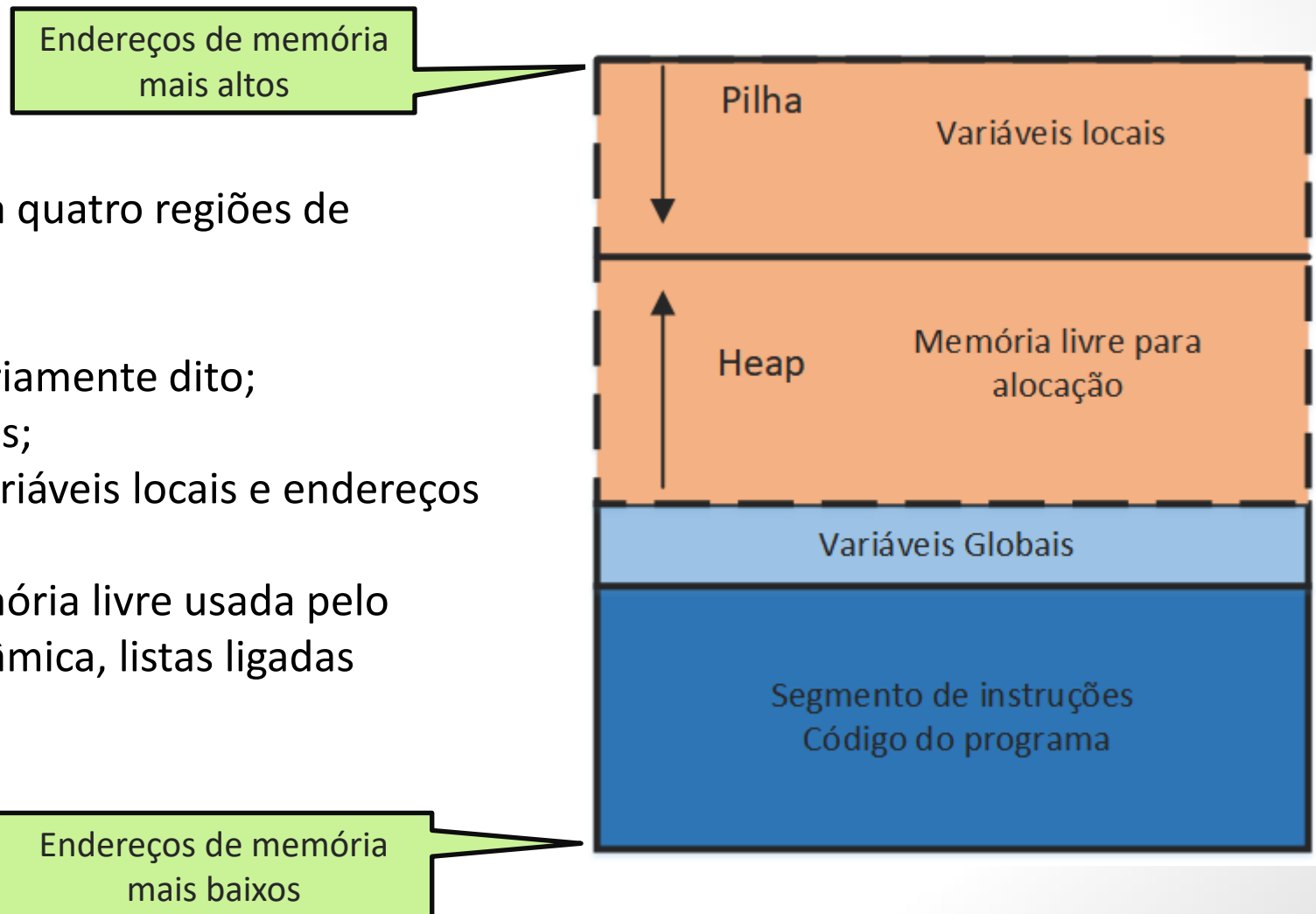
# Estrutura de Dados 1

## Alocação Dinâmica de Memória

- Mapa de memória de C

Um programa C compilado cria e usa quatro regiões de memória:

- O Segmento do código propriamente dito;
- A região das Variáveis Globais;
- A Pilha onde são alocadas variáveis locais e endereços de funções;
- A *Heap* que é região de memória livre usada pelo programa para alocação dinâmica, listas ligadas (encadeadas) e árvores.



# Estrutura de Dados 1

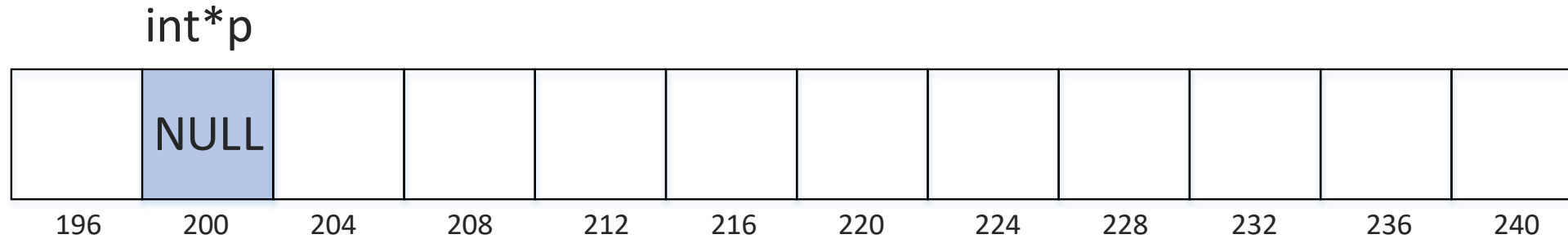
## Alocação Dinâmica de Memória

- Então a alocação dinâmica de memória:
  - Reserva um bloco consecutivo de bytes na memória e retorna o endereço inicial deste bloco;
  - Permite escrever programas mais flexíveis;
  - Poupa memória ao evitar a alocação de grandes espaços de memória que só serão liberados quando o programa terminar.

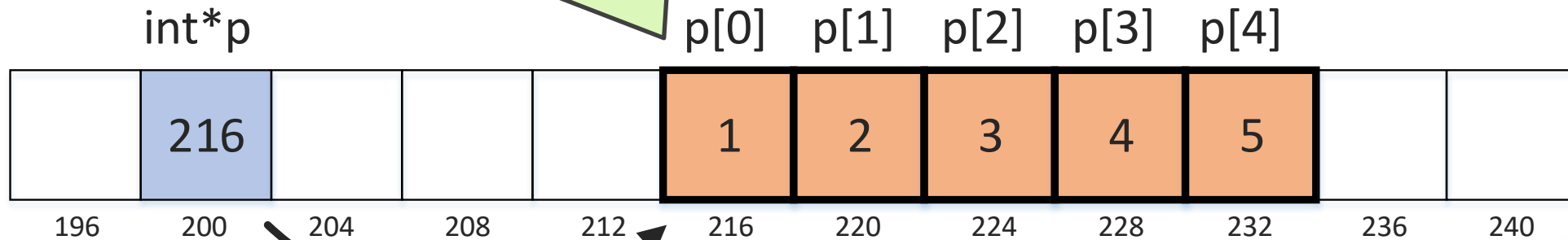


# Estrutura de Dados 1

## Alocação Dinâmica de Memória



Alocando um bloco de memória com 20 Bytes, que comporta 5 posições de memória do tipo `int`





# Estrutura de Dados 1

## Alocação Dinâmica de Memória

- A biblioteca `stdlib.h` possui as três funções que permitem a alocação de blocos de memória em tempo de execução e também uma função que torna possível a liberação dessa memória alocada quando esta não é mais necessária;
  - `malloc()`;
  - `calloc()`;
  - `realloc()`;
  - `free()`;
- Em conjunto com as três funções de alocação utilizamos também o operador `sizeof()`, que retorna o tamanho do **tipo base** para ser alocado.



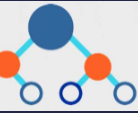
# Estrutura de Dados 1

## Alocação Dinâmica de Memória

- `sizeof()`:
  - 
  - Alocar memória do tipo `int` é diferente de alocar memória do tipo `char`;
  - Tipos diferentes podem ter tamanhos diferentes na memória:

Tipo	Tamanho
<code>char</code>	1 byte
<code>int</code>	4 bytes
<code>float</code>	4 bytes
<code>double</code>	8 bytes
<code>struct</code>	?? Bytes

- Como fazer isso?



# Estrutura de Dados 1

## Alocação Dinâmica de Memória

- O operador `sizeof()` retorna o número de *bytes* ocupados ou utilizados em memória por um determinado tipo de dado;
- Sintaxe:

`sizeof(nome_do_tipo)`

Passamos o tipo e ele retorna o número de *bytes* necessários para armazenar para 1 elemento deste tipo de dado.

- Exemplo:

```
int x = sizeof(int);  
//comando abaixo imprimirá o valor 4 - tamanho de um inteiro  
printf("Tamanho de um inteiro em bytes = %d", sizeof(int));
```



# Estrutura de Dados 1

## Alocação Dinâmica de Memória

- Mais exemplos de `sizeof()`:

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

```
struct ponto{
    int x;
    int y;
    char nome[30];
};
```

Crie um programa com estes exemplos e verifique o resultado.

```
int main() {
    printf("tamanho do float: %d bytes\n", sizeof(float));
    printf("tamanho do char: %d bytes\n", sizeof(char));
    printf("tamanho do double: %d bytes", sizeof(double));
    printf("tamanho da struct: %d bytes\n", sizeof(struct ponto));
}
```



# Estrutura de Dados 1

## Alocando memória – Função malloc()

- Aloca ou reserva, um bloco de memória durante a execução do programa.
- Esta função faz o pedido de memória ao sistema operacional e retorna um ponteiro **genérico** com o endereço do início do espaço de memória alocado.
- Protótipo na biblioteca `stdlib.h`:

```
void * malloc(unsigned int num);
```

A função retorna um ponteiro genérico

Recebe como parâmetro um inteiro sem sinal, somente valores positivos afinal não existem posições de memória negativas, com a quantidade de bytes de memória a alocar.

Protótipo: é como a função está definida na biblioteca a qual pertence





## Alocando memória – Função malloc()

- A função `malloc()`, recebe como parâmetro a quantidade de bytes a ser alocada e retorna:
  - Um ponteiro para a primeira posição de memória do bloco alocado, ou;
  - NULL no caso de erro de alocação, por exemplo falta de memória disponível.
  - Exemplo:

`//alocando um bloco de 200 bytes, onde cabem 50 inteiros:`

$50 * 4 = 200$

`int *v = malloc(200);`

`v[0], v[1], v[2], ..., v[49]`

`//para o mesmo bloco de 200 bytes, cabem 200 caracteres:`

$200 * 1 = 200$

`char *c = malloc(200);`

`c[0], c[1], c[2], ..., c[199]`



## Alocando memória – Função malloc()

- Na alocação de memória deve-se levar em conta o tipo de dado:

```
int *v = malloc(200); // 50 posições de int
char *c = malloc(200); // 200 posições de char
```

- Facilitando a alocação com o uso do operador sizeof():

```
int *v = (int*) malloc( 50 * sizeof(int));
char *c = (char*) malloc(200 * sizeof(char));
```

Convertendo o ponteiro genérico  
retornado por malloc()

Sempre devemos trabalhar com  
a definição do tipo do sistema,  
nunca usar valores absolutos.



## Alocando memória – Função malloc()

- Se não houver memória suficiente para alocar a quantidade requisitada, a função malloc(), retorna NULL no lugar de um endereço válido:

```
int *p;  
p = (int*) malloc(5 * sizeof(int));  
  
if(p == NULL) {  
    printf("ERRO: Sem memória! \n");  
    exit(1); // encerra a execução  
}  
  
int i;  
for(i = 0; i < 5; i++){  
    printf("Digite p[%d]: ", i);  
    scanf("%d", &p[i]);  
}
```

p agora trabalha como um vetor

Feita a alocação e se **p** não era nulo, ou seja o **if** não foi executado, não precisamos mais nos lembrar que **p** é um ponteiro

A função `exit()` da biblioteca `stdlib.h` interrompe a execução do programa e fecha todos os arquivos que o programa tenha porventura aberto. Se o argumento da função for 0, o sistema operacional é informado de que o programa terminou com sucesso; caso contrário, o sistema operacional é informado de que o programa terminou de maneira excepcional.

O argumento da função é tipicamente a constante `EXIT_FAILURE`, que vale 1, ou a constante `EXIT_SUCCESS`, que vale 0.



# Estrutura de Dados 1

## Alocando memória – Função malloc()

- Exemplo:

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

int main(){
    int *vet = NULL, i;
    vet = (int*) malloc(25 * sizeof(int));
    for(i=0; i < 15; i++){
        vet[i] = i * 2;
        printf("i: %2d * 2 = %2d\n", i, vet[i]);
    }
}
```

- preenche o vetor com o dobro do índice i, em seguida imprime o resultado.

```
"C:\Users\angelot\Documents\Aulas\GRUEDA1\
i:  0 * 2 =  0
i:  1 * 2 =  2
i:  2 * 2 =  4
i:  3 * 2 =  6
i:  4 * 2 =  8
i:  5 * 2 = 10
i:  6 * 2 = 12
i:  7 * 2 = 14
i:  8 * 2 = 16
i:  9 * 2 = 18
i: 10 * 2 = 20
i: 11 * 2 = 22
i: 12 * 2 = 24
i: 13 * 2 = 26
i: 14 * 2 = 28

Process returned 0 (0x0)
Press any key to continue.
```



# Estrutura de Dados 1

## Alocando memória – Função malloc()

- Exemplo 2:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char *str;
    char *so_ponteiro;
    str = (char*) malloc(25 * sizeof(char));
    if(str == NULL) {
        printf("Espaço insuficiente");
        exit(1);
    } else { //memória alocada
        str = "teste";
        so_ponteiro = "Novo teste com ponteiro!";
        printf("String no vetor alocado: %s \n\n", str);
        printf("String direto no ponteiro: %s \n\n", so_ponteiro);
    }
}
```

```
"C:\Users\angelot\Documents\Aulas\GRUEDA1\Aulas\Aula 07 - Alocação de Memória\material de ap
String no vetor alocado: teste

String direto no ponteiro: Novo teste com ponteiro!

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Neste caso, o *array* de caracteres terá o tamanho exato da *string* atribuída





## Librando a memória – Função free()

- Sempre que alocamos memória é necessário liberá-la quando esta não for mais necessária:

```
int *p;
p = (int*) malloc(5 * sizeof(int));

if(p == NULL) {
    printf("ERRO: Sem memória! \n");
    exit(1); // encerra a execução
}

int i;
for(i = 0; i < 5; i++){
    printf("Digite p[%d]: ", i);
    scanf("%d", &p[i]);
}

free(p); // libera a memória alocada
```

Ao término da utilização temos que liberar a memória alocada, assim outros processos podem utilizá-la. Desta forma garantimos que nunca teremos memória “presa”.

# Estrutura de Dados 1

## Librando a memória – Função free()

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    int *p, i;
    p = (int *) malloc(10*sizeof(int));
    for(i=0; i<10; i++){
        p[i] = i;
    }
    for(i=0; i<10; i++){
        printf("%d\n", p[i]);
    }
    free(p);
    printf("\n\n");
    for(i=0; i<10; i++){
        printf("%d\n", p[i]);
    }
}
```

```
"C:\Users\angelot\Documents\Aulas\GRUEDA1\Aulas\Aula 07 - Alocaçã
2
3
4
5
6
7
8
9
0
7536976
7567184
0
4
5
6
7
8
9
Process returned 2 (0x2)   execution
```

Observe que após a liberação da memória, algumas posições já estão sendo utilizadas por outros processos...





## Alocando memória – Função `calloc()`

- Aloca memória durante a execução do programa;
- Faz o pedido de memória ao sistema operacional e retorna um ponteiro com o endereço do espaço de memória alocado;
- Protótipo na biblioteca `stdlib.h`:

```
void * calloc(unsigned int num, unsigned int size);
```

A função retorna um ponteiro genérico.

Quantidade de elementos que se quer armazenar no bloco alocado.

Qual será o tamanho, em *bytes*, que cada elemento ocupará na memória.



## Alocando memória – Função calloc()

- A função `calloc()` recebe por parâmetro:
  - Número de elementos no vetor a ser alocado;
  - Tamanho de cada elemento do vetor.
- E retorna:
  - Ponteiro para a primeira posição de memória do vetor, ou;
  - `NULL`, no caso de erro de alocação.

```
//alocando um bloco de 200 bytes, onde cabem 50 inteiros:
```

```
int *v = (int*) calloc(50, 4);
```

```
//para o mesmo bloco de 200 bytes, cabem 200 caracteres:
```

```
char *c = (char*) calloc(200, 1);
```



## Alocando memória – Função calloc()

- Na alocação de memória com calloc(), assim como em malloc(), deve-se levar em conta o tamanho do tipo de dado:

```
int  *v = (int*)  calloc(50, 4); // 50 posições de int
char *c = (char*) calloc(200, 1); // 50 posições de char
```

- Usando o operador sizeof():

```
int  *v = (int*)  calloc(50, sizeof(int));
char *c = (char*) calloc(200, sizeof(char));
```

Desta forma, não é necessário lembrar o tamanho de cada tipo, principalmente se for uma estrutura.

# Estrutura de Dados 1

## Alocando memória – Função calloc()

- Se não houver memória suficiente para alocar a quantidade requisitada, a função calloc() retorna NULL:

```
int *p;  
p = (int*) calloc(5, sizeof(int));  
  
if(p == NULL) {  
    printf("ERRO: Sem memória! \n");  
    exit(1); // encerra a execução  
}  
  
int i;  
for(i = 0; i < 5; i++) {  
    printf("Digite p[%d]: ", i);  
    scanf("%d", &p[i]);  
}
```

if captura se a alocação foi malsucedida.

Novamente, tratamos **p** como um vetor





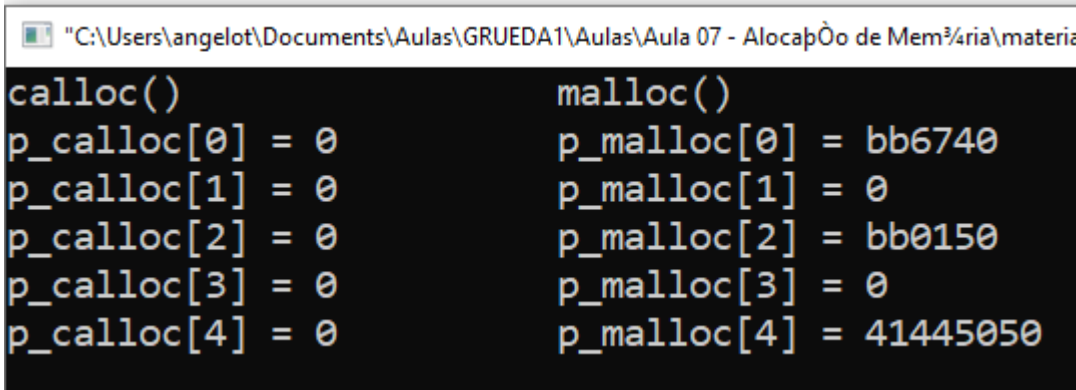
# Estrutura de Dados 1

## Alocando memória – malloc() versus calloc()

- Ambas servem para alocar memória, mas calloc() inicializa todos os bits do espaço alocado com zeros (0).

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

```
void main() {
    int i;
    int *p_malloc, *p_calloc;
    p_malloc = (int*) malloc(5 * sizeof(int));
    p_calloc = (int*) calloc(5, sizeof(int));
    printf("calloc() \t\t malloc() \n");
    for(i = 0; i < 5; i++) {
        printf("p_calloc[%d] = %d \t ", i, p_calloc[i]);
        printf("p_malloc[%d] = %x \n", i, p_malloc[i]);
    }
    printf("\n\n");
}
```



calloc()	malloc()
p_calloc[0] = 0	p_malloc[0] = bb6740
p_calloc[1] = 0	p_malloc[1] = 0
p_calloc[2] = 0	p_malloc[2] = bb0150
p_calloc[3] = 0	p_malloc[3] = 0
p_calloc[4] = 0	p_malloc[4] = 41445050



# Estrutura de Dados 1

## Alocando memória – Função free()

- Também nesta função temos que cuidar para não deixarmos memória alocada que não será mais utilizada:

```
int *p;
p = (int*) calloc(5, sizeof(int));

if(p == NULL) {
    printf("ERRO: Sem memória! \n");
    exit(1); // encerra a execução
}

int i;
for(i = 0; i < 5; i++){
    printf("Digite p[%d]: ", i);
    scanf("%d", &p[i]);
}

free(p); // libera a memória alocada
```





## Alocando memória – Função `realloc()`

- Aloca ou realoca memória durante a execução do programa.
- Esta função faz o pedido ao sistema operacional e retorna um ponteiro com o endereço do início do bloco de memória que foi alocado.
- Protótipo da função na biblioteca `stdlib.h`:

```
void * realloc(void *ptr, unsigned int num);
```

A função retorna um ponteiro genérico

Recebe um ponteiro de qualquer tipo (genérico), para onde previamente havia sido alocada a memória

Novo tamanho para o bloco de memória alocado



## Alocando memória – Função realloc()

- A função `realloc()`, recebe por parâmetro:
  - Um ponteiro para um bloco de memória já alocado;
  - A nova quantidade de bytes a ser alocada.
- E retorna:
  - Um ponteiro para a primeira posição do vetor, ou
  - NULL, se houver erro de alocação.

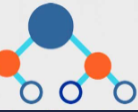
```
int *v = (int*) malloc(50 * sizeof(int));
```

```
//realocando:
```

```
v = (int*) realloc(v, 100 * sizeof(int));
```

Alocação inicial com `malloc()`, que se tornou insuficiente

Realocação para novo tamanho final de 100 elementos. Os 50 elementos anteriores estão contidos nas novas 100 posições.



## Alocando memória – Função `realloc()`

- Se o ponteiro para o bloco de memória previamente alocado for `NULL`, a função `realloc()`, irá alocar memória da mesma forma que a função `malloc()`:

```
int *p;  
p = (int*) realloc(NULL, 50 * sizeof(int));
```

//o comando acima equivale a:

```
p = (int*) malloc(50 * sizeof(int));
```



## Alocando memória – Função `realloc()`

- Se o tamanho da memória solicitado for igual a zero (0), `realloc()` irá liberar a memória alocada da mesma forma que a função `free()`:

```
int *p;  
p = (int*) malloc(50 * sizeof(int));
```

```
//realocando com tamanho 0
```

```
p = (int*) realloc(p, 0);
```

Equivale a liberar o vetor

```
//o comando acima equivale a:
```

```
free(p);
```



## Alocando memória – Função realloc()

- Cuidado: Se não houver memória suficiente para alocar a quantidade requisitada, a função realloc(), retorna NULL:

```
int *p = (int*) malloc(5 * sizeof(int));  
int *p_aux = (int*) realloc(p, 15 * sizeof(int));
```

```
if (p1 != NULL) {  
    p = p_aux;  
}
```

p permanece intacto, pois o novo bloco de memória é alocado em p\_aux

Se a alocação deu certo, realloc() por padrão, já copia os dados de p para p\_aux, automaticamente.

# Estrutura de Dados 1

## Alocando memória – Função realloc()



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

int main(){
    int *p, i;
    p = (int*) malloc(5 * sizeof(int));
    for(i = 0; i < 5; i++){
        p[i] = i + 10;
    }
    for(i = 0; i < 10; i++){
        printf("posicao p[%d] %d\n", i, p[i]);
    }
    //Realocando para 10 posições
    p = (int*) realloc(p, 10 * sizeof(int));
    //preenchendo a partir da 6ª posição
    for(i = 5; i < 20 ; i++){
        p[i] = i + 100;
    }
    printf("\t**** Agora com novo espaco alocado ****\n");
    for(i = 0; i < 10; i++){
        printf("posicao p[%d] %d\n", i, p[i]);
    }
    system("pause");
}
```

"C:\Users\angelot\Documents\Aulas\GRUEDA1\Aulas\Aula 07 - Alocação de Memória\mate

```
posicao p[0] 10
posicao p[1] 11
posicao p[2] 12
posicao p[3] 13
posicao p[4] 14
posicao p[5] 0
posicao p[6] 0
posicao p[7] 0
posicao p[8] -1522785049
posicao p[9] -1522785049
entrou **** Agora com novo espaco alocado ****
posicao p[0] 10
posicao p[1] 11
posicao p[2] 12
posicao p[3] 13
posicao p[4] 14
posicao p[5] 105
posicao p[6] 106
posicao p[7] 107
posicao p[8] 108
posicao p[9] 109
Pressione qualquer tecla para continuar. . .
```



# Estrutura de Dados 1

## Alocando memória – Função `free()`

- Também nesta função é necessária a liberação de memória quando esta não for mais necessária:

```
int *p = (int*) malloc(5 * sizeof(int));  
p = (int*) realloc(p, 15 * sizeof(int));  
  
if(p == NULL) {  
    printf("ERRO: Sem memória! \n");  
    exit(1);  
}  
  
free(p); //libera a memória alocada
```





## Alocação de matrizes multidimensionais

- Para alocar uma matriz com mais de uma dimensão, precisamos utilizar o conceito de ponteiro para ponteiro:

Permite criar um vetor

```
//ponteiro (1) nível: cria um vetor
```

Permite criar um ponteiro que aponta p/ uma matriz bidimensional

```
int *p = (int*) malloc(5 * sizeof(int));
```

```
//ponteiro para ponteiro (2 níveis):  
//permite criar uma matriz bidimensional
```

```
int **m;
```

Permite criar uma matriz tridimensional, um cubo de memória

```
//ponteiro para ponteiro para ponteiro  
//(3 níveis): permite criar uma matriz tridimensional
```

```
int ***d;
```

Lembre-se:  
Não existem limites para os níveis!!



## Alocação de matrizes multidimensionais

- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão da matriz:

- `int*` - Permite criar uma matriz, *array* ou vetor de `int`;
- `int**` - Permite criar uma matriz, *array* ou vetor de `int*`;

Uma matriz de ponteiros!!

```
int **p; // 2 "*" = 2 níveis = 2 dimensões
```

```
//criando um array de ponteiros (int*)
```

```
p = (int**) malloc(N * sizeof(int*));
```

Cria um *array* de ponteiros

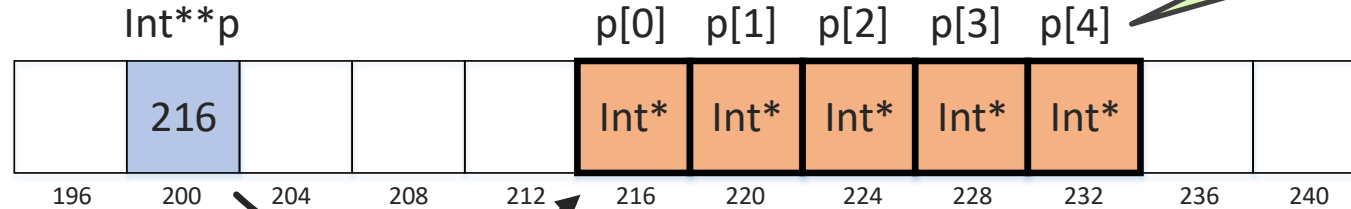
# Estrutura de Dados 1

## Alocação de matrizes multidimensionais

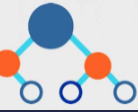
```
int **p; // 2 "*" = 2 níveis = 2 dimensões
```

```
//criando um array de ponteiros (int*)
```

```
p = (int**) malloc(N * sizeof(int*));
```



Isso significa que cada posição do *array* pode apontar para outro *array* ou vetor!



# Estrutura de Dados 1

## Alocação de matrizes multidimensionais

- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no *array*:

```
int **p; // 2 "*" = 2 níveis = 2 dimensões
int i, j, N = 5;

//criando um array de ponteiros (int*)
p = (int**) malloc(N * sizeof(int*));

//cria N arrays de int
for(i = 0; i < N; i++){
    p[i] = (int*) malloc(N * sizeof(int));
}
```



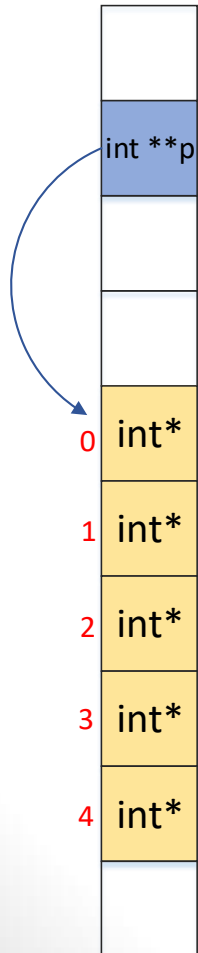
# Estrutura de Dados 1

## Alocação de matrizes multidimensionais

```
int **p; // 2 "*" = 2 níveis = 2 dimensões
int i, j, N = 5;
```

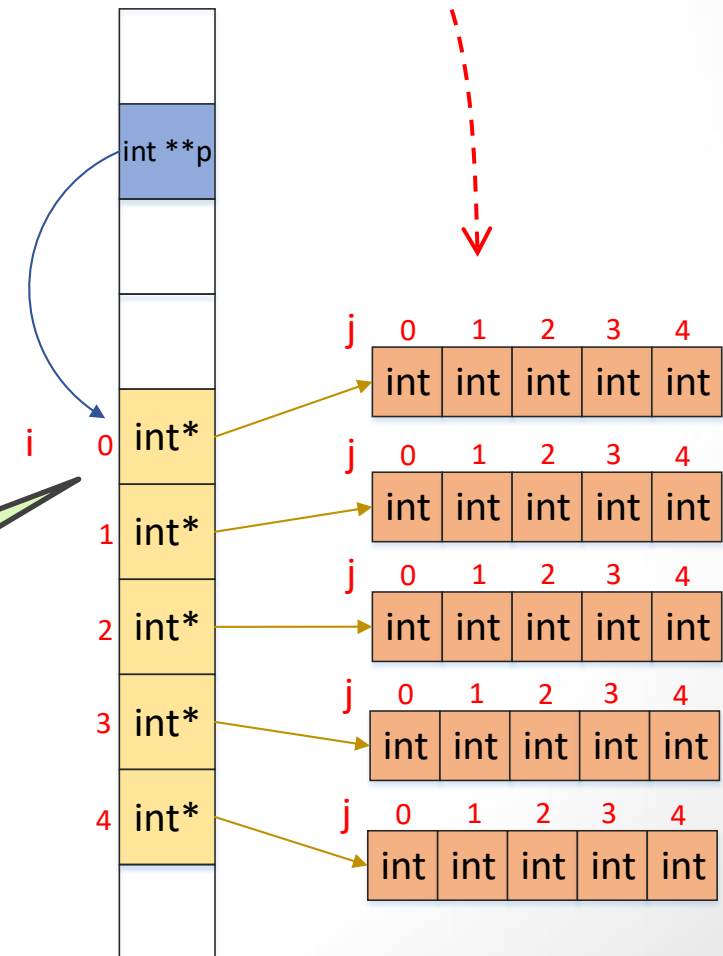
```
//criando um array de ponteiros (int*)
p = (int**) malloc(N * sizeof(int*));

//cria N arrays de int
for(i = 0; i < N; i++){
    p[i] = (int*) malloc(N * sizeof(int));
}
```



1º malloc: cria as linhas da matriz

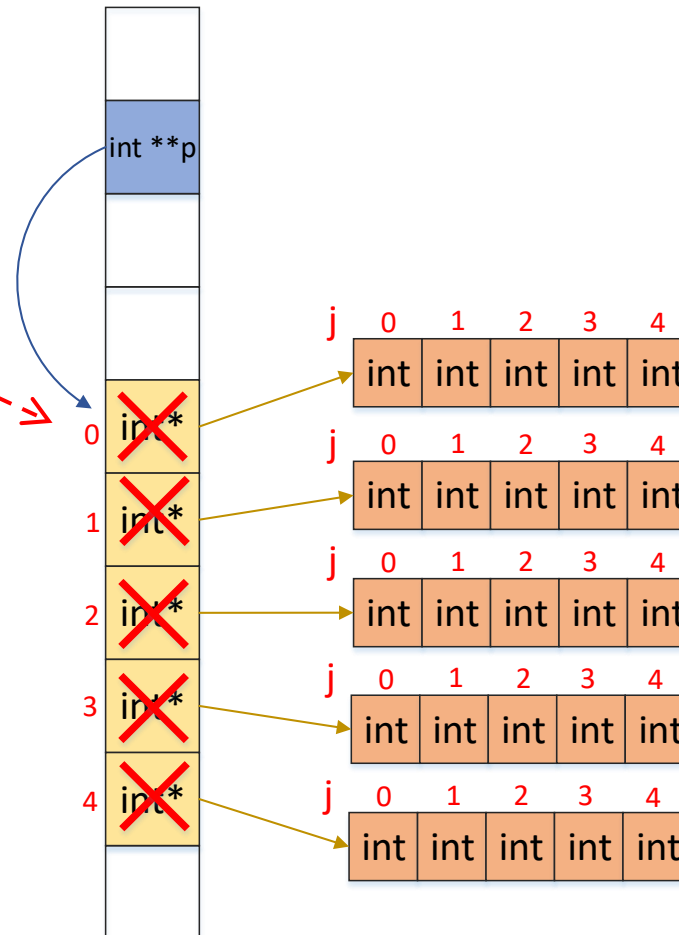
2º malloc: cria as colunas da matriz, para cada ponteiro do array linha



## Liberação de memória em matrizes multidimensionais

- Em uma matriz com mais de uma dimensão, a memória é liberada na ordem inversa a de alocação:

```
for(i = 0; i < N; i++){  
    free(p[i]);  
}  
free(p);
```



# Estrutura de Dados 1

## Liberação de memória em matrizes multidimensionais

```
for(i = 0; i < N; i++){  
    free(p[i]);  
}  
free(p);
```



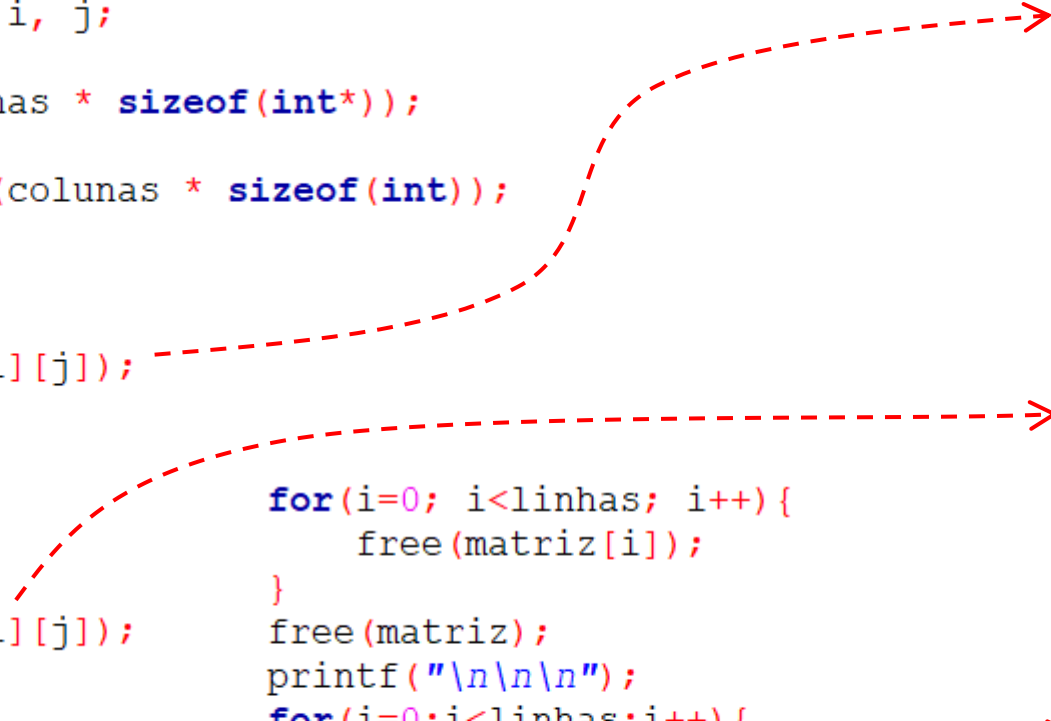


# Estrutura de Dados 1

## Função free()

```
#include <stdio.h>
#include <stdlib.h>
void main(void){
    int linhas = 3, colunas = 2, i, j;
    int **matriz;
    matriz = (int **) malloc(linhas * sizeof(int*));
    for(i=0;i<linhas;i++){
        matriz[i]= (int*) malloc(colunas * sizeof(int));
    }
    for(i=0;i<linhas;i++){
        for(j=0;j<colunas;j++){
            scanf("%d", &matriz[i][j]);
        }
    }
    printf("\n\n\n");
    for(i=0;i<linhas;i++){
        for(j=0;j<colunas;j++){
            printf("%d", matriz[i][j]);
        }
        printf("\n");
    }

    for(i=0; i<linhas; i++){
        free(matriz[i]);
    }
    free(matriz);
    printf("\n\n\n");
    for(i=0;i<linhas;i++){
        for(j=0;j<colunas;j++){
            printf("%d", matriz[i][j]);
        }
        printf("\n");
    }
}
```



"C:\Users\angelot\Documents\Ar"

1  
2  
3  
4  
5  
6

12  
34  
56

111061280  
110808640  
56

Process returned 3  
Press any key to co



# Estrutura de Dados 1

## Atividade 1

- Faça um programa que aloque memória para um vetor dinâmico com  $n$  números inteiros ímpares maiores que 0, em seguida imprima o vetor. Entregue no Moodle.

## Atividade 2

- Escreva um programa que solicita ao usuário a quantidade de alunos de uma turma aloque um vetor dinamicamente com esta quantidade e armazene as notas dos alunos. Depois de coletar do teclado, armazenar no vetor dinâmico e imprimir as notas de todos os alunos, imprime também a média aritmética de toda a turma. Entregue no Moodle.

## Atividade 3

- Elabore um programa que calcule a soma de duas matrizes ( $M \times N$ ) dinâmicas de números inteiros. Deve-se considerar as dimensões fornecidas pelo usuário. Entregue no Moodle.

