

Estrutura de Dados 1

Aula 06 – Alocação Dinâmica de Memória

Antonio Angelo de Souza Tartaglia

angelot@ifsp.edu.br

Estrutura de Dados 1

Alocação Dinâmica de Memória

- ▶ Nesta aula veremos como:
 - Reconhecer as funções utilizadas na alocação dinâmica;
 - Descobrir o tamanho de um tipo de dado;
 - Realizar a alocação dinâmica de um vetor;
 - Realizar a alocação dinâmica de uma matriz;
 - Liberar a memória alocada.

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.

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.

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?

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.
- ▶ Ela 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 computador, em tempo de execução, o qual, usando um ponteiro, devolve para o programa o endereço do início desse espaço alocado.

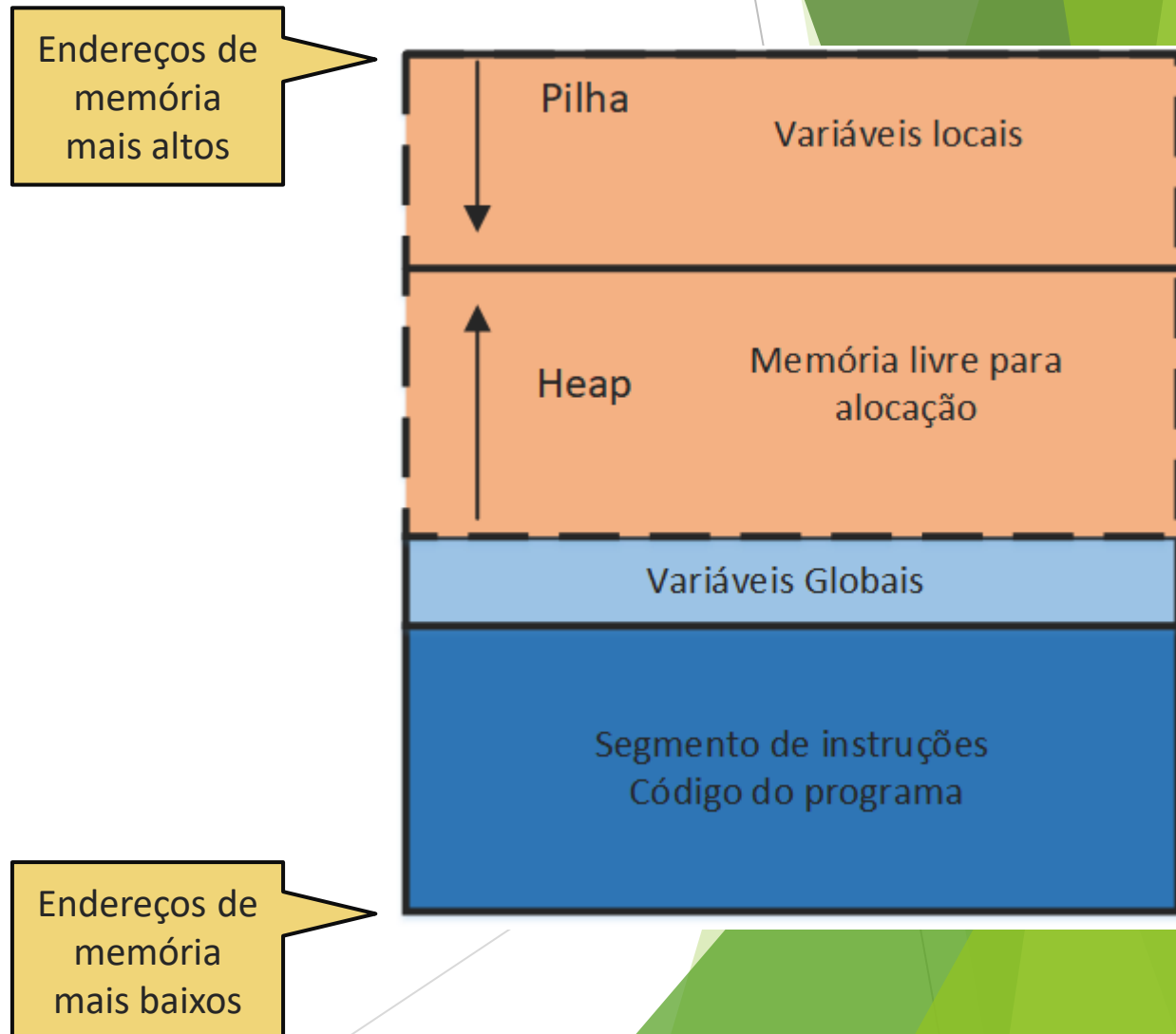
Estrutura de Dados 1

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

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

Estrutura de Dados 1

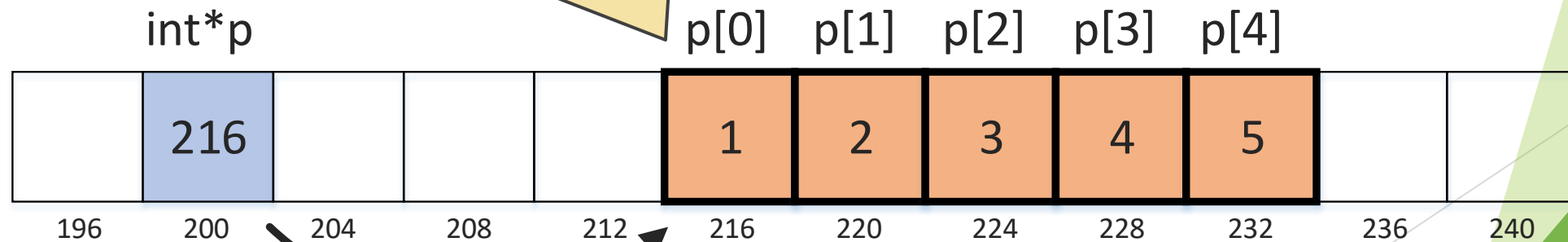
Alocação Dinâmica de Memória

int*p



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

int*p



Alocação Dinâmica de Memória

- ▶ A biblioteca **stdlib.h** possui 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 passa para as funções o tamanho do **tipo base** a 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
char	1 byte
int	4 bytes
float	4 bytes
double	8 bytes
struct	?? Bytes

► Como fazer isso?

Estrutura de Dados 1

Alocação Dinâmica de Memória

- ▶ O operador `sizeof()` retorna o número de bytes de um determinado tipo de dado;

- ▶ Sintaxe:

`sizeof(nome_do _tipo)`

Neste caso específico, o número de bytes necessários para alocar um único elemento, o **tipo base**.

- ▶ Exemplo:

```
int x = sizeof(int);  
printf("x = %d \n", x); //imprimirá o valor 4
```

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

Estrutura de Dados 1

Alocação Dinâmica de Memória

- Mais exemplos de `sizeof()`:

```
struct ponto{  
    int x;  
    int y;  
};
```

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

Crie um programa, inclua nele estes exemplos e verifique o resultado.

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...

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:

50 * 4 = 200

```
//criando um vetor de 50 inteiros (200 bytes)  
int *v = malloc(200);
```

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

200 * 1 = 200

```
//criando um vetor de 200 caracteres (200 bytes)  
char *c = malloc(200);
```

Estrutura de Dados 1

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(50); // 50 posições de char
```

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

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

Convertendo o ponteiro genérico retornado por `malloc()`

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

Estrutura de Dados 1

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

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

Trabalha como um vetor

A função `exit()` da biblioteca `stdlib` 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.

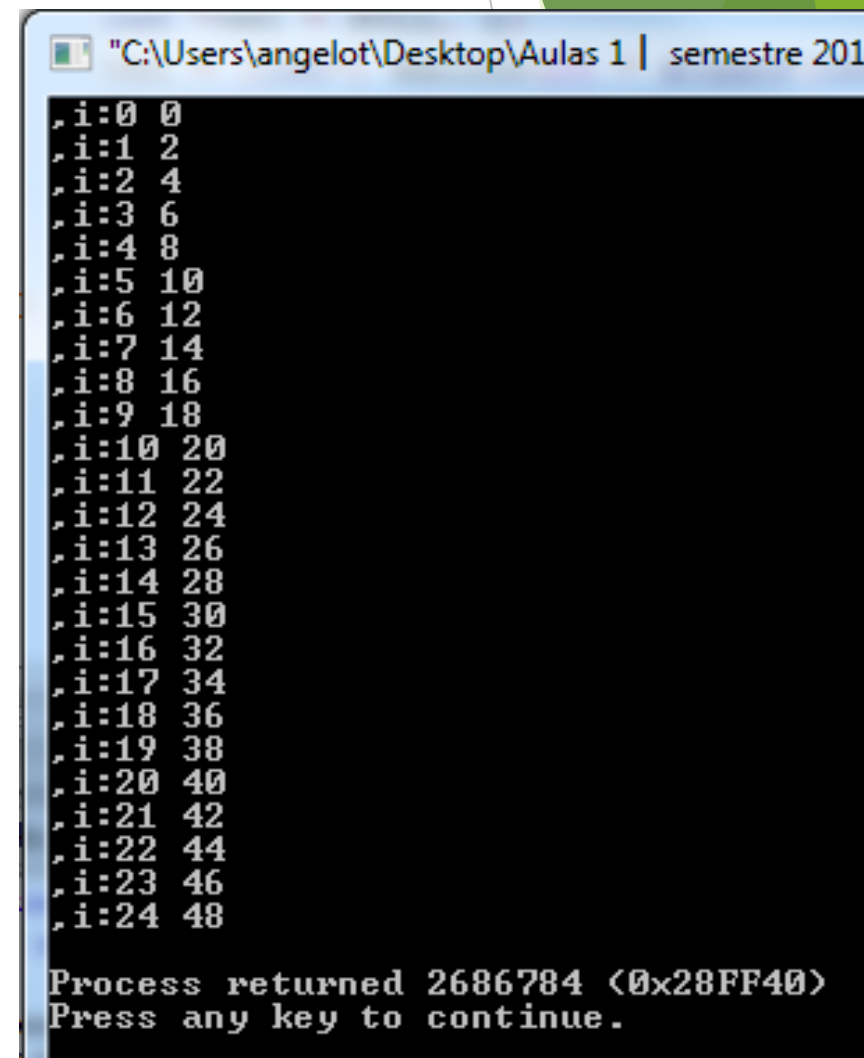
Estrutura de Dados 1

Função malloc

► Exemplo:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4      int *vet = NULL, i;
5      vet = (int*) malloc(25 * sizeof(int));
6      for(i=0; i < 25; i++){
7          vet[i]=i*2;
8          printf(",i:%d %d\n",i,vet[i]);
9      }
10 }
```

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



```
"C:\Users\angelot\Desktop\Aulas 1 | semestre 201"
,i:0 0
,i:1 2
,i:2 4
,i:3 6
,i:4 8
,i:5 10
,i:6 12
,i:7 14
,i:8 16
,i:9 18
,i:10 20
,i:11 22
,i:12 24
,i:13 26
,i:14 28
,i:15 30
,i:16 32
,i:17 34
,i:18 36
,i:19 38
,i:20 40
,i:21 42
,i:22 44
,i:23 46
,i:24 48

Process returned 2686784 (0x28FF40)
Press any key to continue.
```

Estrutura de Dados 1

Função `malloc`

► Exemplo 2:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4      char *str;
5      str = (char*) malloc(25 * sizeof(char));
6      if(str == NULL) { //testa se foi possível alocar
7          printf("Espaço insuficiente");
8          exit(1);
9      } else { //memória alocada
10         str = "teste";
11         printf("Palavra: %s", str);
12     }
13 }
```

Liberando 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); // termina o programa  
}  
  
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 utilizada, assim outros processos podem utilizá-la. Desta forma garantimos que nunca teremos memória “presa”.

Alocando memória – Função `calloc()`

- ▶ Serve para alocar memória durante a execução do programa;
- ▶ Esta função 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 bytes que se quer alocar – quantas posições de memória se quer para o vetor

Qual será o tamanho de cada posição de memória do vetor

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.

```
//criando um vetor de 50 inteiros (200 bytes)  
int *v = (int*) calloc(50, 4);
```

```
//criando um vetor de 200 caracteres (200 bytes)  
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(50, 1); // 50 posições de char
```

- Usando o operador `sizeof()`:

```
int *v = (int*) calloc(50, sizeof(int));  
char *c = (char*) calloc(50, 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); // termina o programa  
}  
  
int i;  
for(i = 0; i < 5; i++){  
    printf("Digite p[%d]: ", i);  
    scanf("%d", &p[i]);  
}
```

If captura se a alocação foi malsucedida.

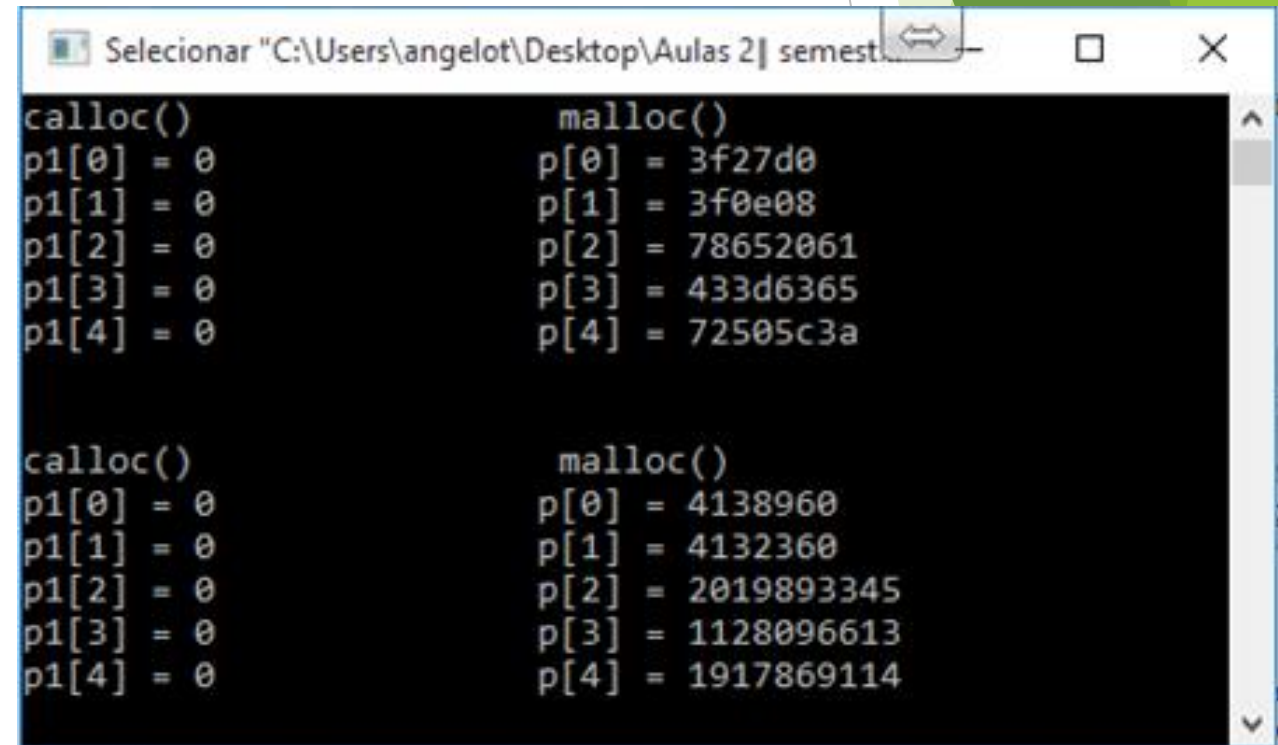
Novamente, tratamos `p` como um vetor

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).

```
int *p, *p1;
p = (int*) malloc(5 * sizeof(int));
p1 = (int*) calloc(5, sizeof(int));
printf("calloc() \t\t malloc() \n");
for(i = 0; i < 5; i++){
    printf("p1[%d] = %d \t\t", i, p1[i]);
    printf("p[%d] = %x \n", i, p[i]);
}
printf("\n\n");
printf("calloc() \t\t malloc() \n");
for(i = 0; i < 5; i++){
    printf("p1[%d] = %d \t\t", i, p1[i]);
    printf("p[%d] = %d \n", i, p[i]);
}
```



calloc()		malloc()	
p1[0] = 0		p[0] = 3f27d0	
p1[1] = 0		p[1] = 3f0e08	
p1[2] = 0		p[2] = 78652061	
p1[3] = 0		p[3] = 433d6365	
p1[4] = 0		p[4] = 72505c3a	
calloc()		malloc()	
p1[0] = 0		p[0] = 4138960	
p1[1] = 0		p[1] = 4132360	
p1[2] = 0		p[2] = 2019893345	
p1[3] = 0		p[3] = 1128096613	
p1[4] = 0		p[4] = 1917869114	

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); // termina o programa
}

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

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

Estrutura de Dados 1

Alocando memória – Função `realloc()`

- ▶ Serve para alocar ou realocar 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 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.

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

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

```
//realocando:
```

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

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);
```

Estrutura de Dados 1

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 *p1 = (int*) realloc(p, 15 * sizeof(int));
```

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

p permanece intacto, pois o novo bloco de memória é alocado em p1

Se a alocação deu certo, `realloc()` por padrão, já copia os dados de p para p1, automaticamente. Por tanto este trecho de código é desnecessário.

Estrutura de Dados 1

Função `realloc`

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

```
"C:\Users\angelot\Documents\Aulas 1\ ...
posicao p[0] 10
posicao p[1] 11
posicao p[2] 12
posicao p[3] 13
posicao p[4] 14
posicao p[5] 15
posicao p[6] 16
posicao p[7] 17
posicao p[8] 18
posicao p[9] 19
posicao p[10] -860450119
posicao p[11] 40369
posicao p[12] 45562376
posicao p[13] 45553024
posicao p[14] 1176515638
posicao p[15] 1818848609
posicao p[16] 540418169
posicao p[17] 1701080909
posicao p[18] 943136876
posicao p[19] 1702122272
**** Agora com novo espaco alocado ****

posicao p[10] 110
posicao p[11] 111
posicao p[12] 112
posicao p[13] 113
posicao p[14] 114
posicao p[15] 115
posicao p[16] 116
posicao p[17] 117
posicao p[18] 118
posicao p[19] 119
```


Estrutura de Dados 1

Alocando memória – Função `free()`

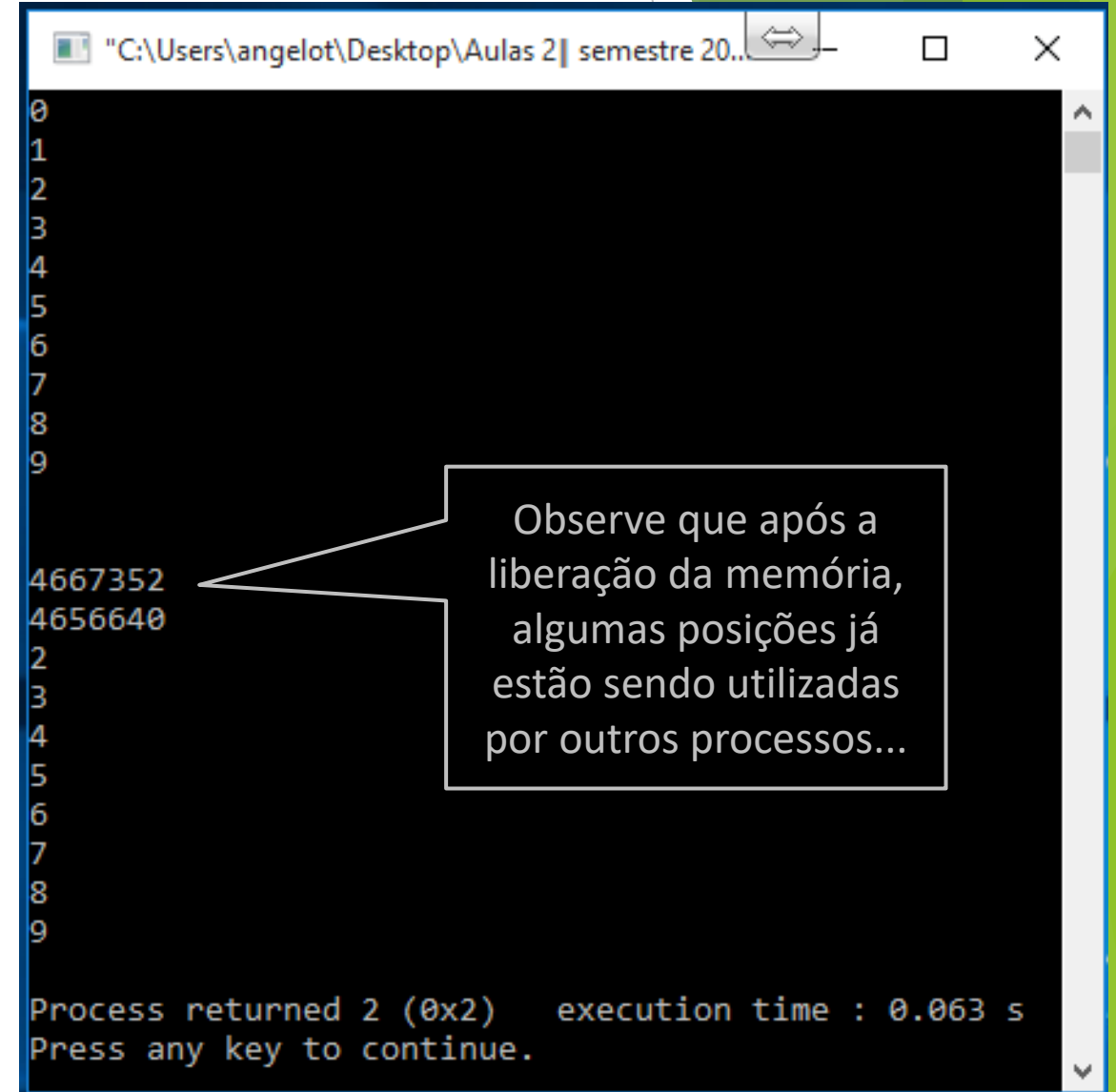
- Também nesta função é necessária a liberação de memória quando esta não for mais necessária:

```
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
```

Estrutura de Dados 1

Alocando memória – Função `free()`

```
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\Desktop\Aulas 2\ semestre 20..."
0
1
2
3
4
5
6
7
8
9
4667352
4656640
2
3
4
5
6
7
8
9

Process returned 2 (0x2)   execution time : 0.063 s
Press any key to continue.
```

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

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
```

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

Permite criar um ponteiro que aponta p/ uma matriz bidimensional

```
//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
int i, j, N = 2;
```

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

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

Cria um array de ponteiros

Estrutura de Dados 1

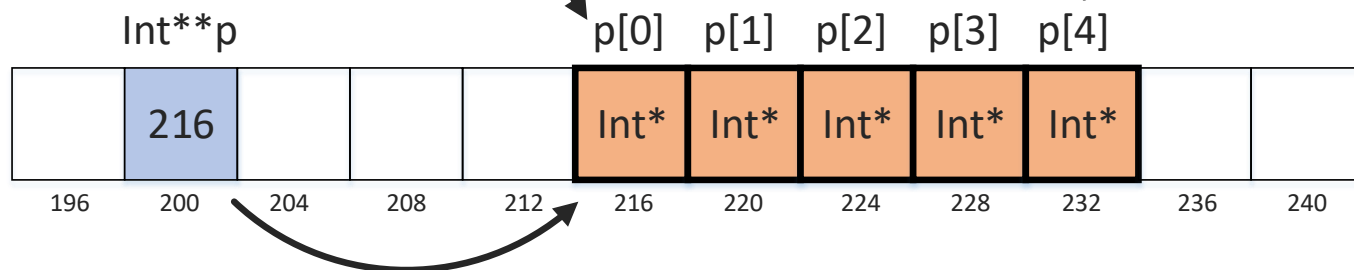
Alocação de matrizes multidimensionais

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

```
//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!



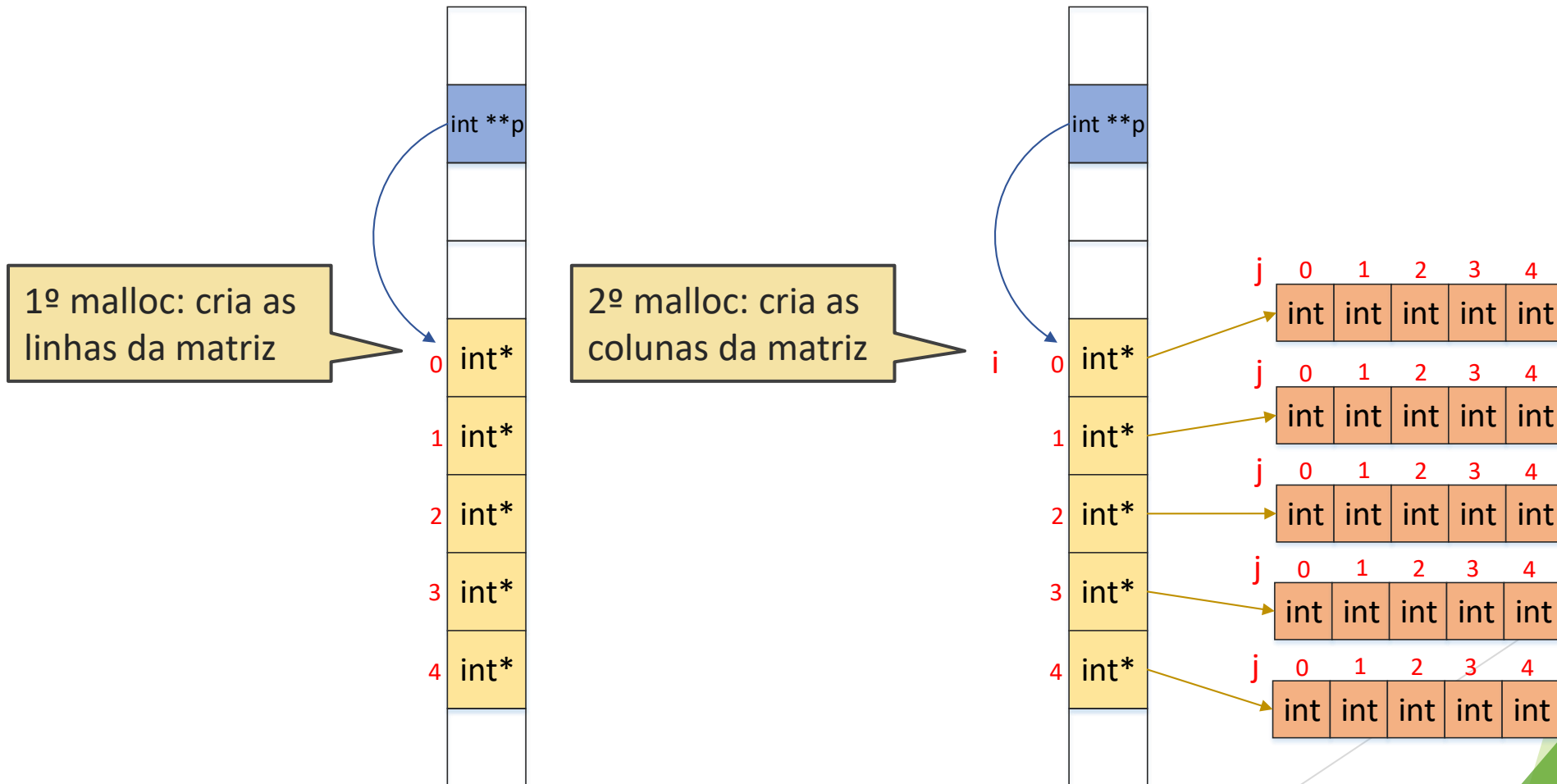
Alocação de matrizes multidimensionais

- 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*));
for(i = 0; i < N; i++){
    //cria um array de int
    p[i] = (int*) malloc(N * sizeof(int));
    for(j = 0; j < N; j++){
        //lê a matriz de inteiros
        scanf("%d", &p[i][j]);
    }
}
```

Estrutura de Dados 1

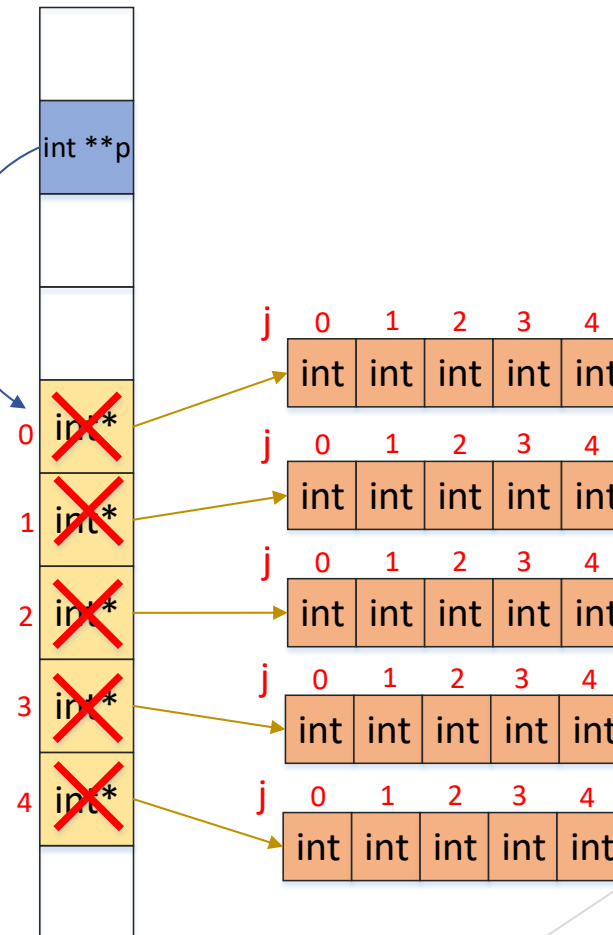
Alocação de matrizes multidimensionais



Liberação de memória em matrizes multidimensionais

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

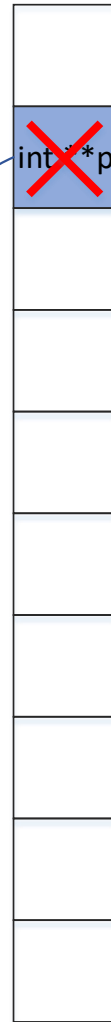
```
for(i = 0; i < N; i++){  
    free(p[i]);  
}  
free(p);
```



Estrutura de Dados 1

Liberação de memória em matrizes multidimensionais

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



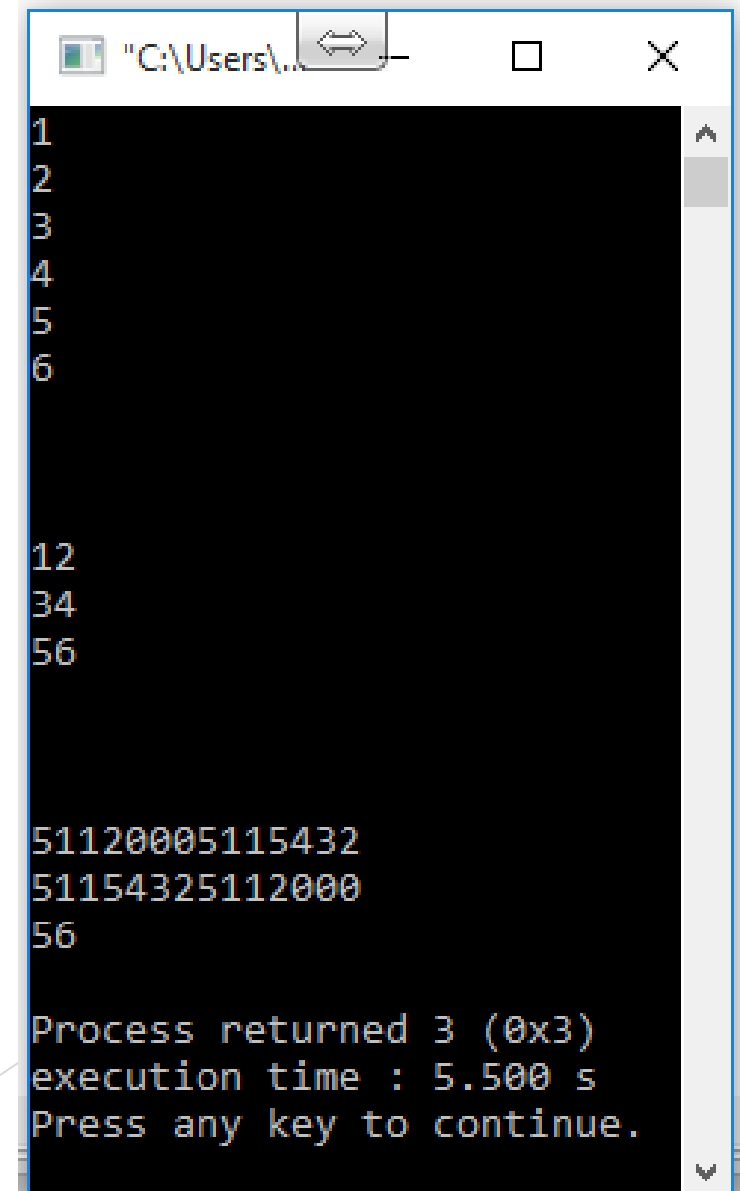
Estrutura de Dados 1

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\..."
1
2
3
4
5
6

12
34
56

51120005115432
51154325112000
56

Process returned 3 (0x3)
execution time : 5.500 s
Press any key to continue.
```

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.

Estrutura de Dados 1

Atividade 4

- Explique os seguintes códigos e entregue no Moodle:

```
1 void funcao_troca_inteiros(int *a, int *b){
2     int *aux;
3     aux = (int*) malloc(sizeof(int));
4     if (aux == NULL){
5         printf("memoria insuficiente\n");
6         exit(1);
7     }else{
8         *aux = *a;
9         *a = *b;
10        *b = *aux;
11        free(aux);
12    }
13 }
```

```
1 void main() {
2     char *p;
3     p = (char*)malloc(4*sizeof(char));
4     strcpy(p,"IFSP");
5     p = (char*)realloc(p,15*sizeof(char));
6     strcat(p,"-Guarulhos");
7     printf("%s",p);
8     free(p);
9 }
```