Engenharia de Computação

# Fundamentos de Programação

## Aula 20 – Alocação Dinâmica

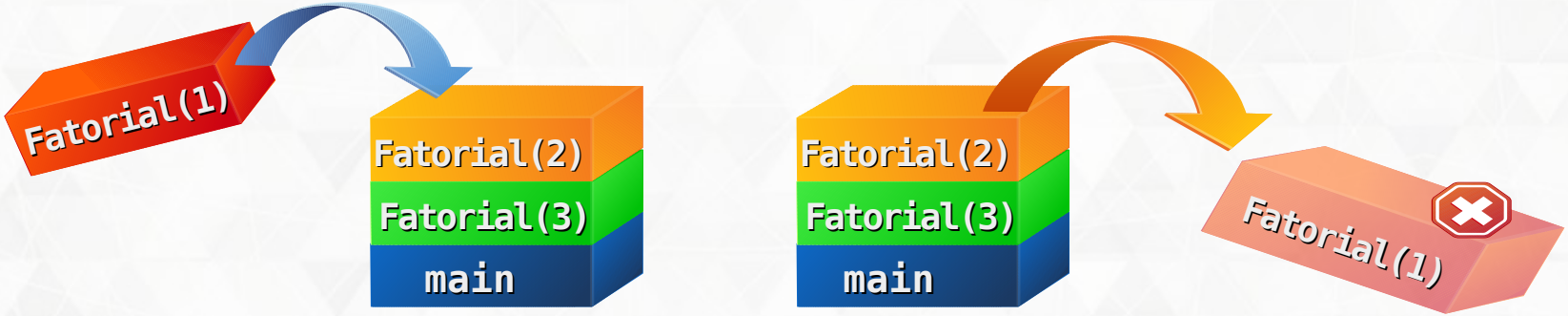
**Prof. Muriel de Souza Godoi**
muriel@utfpr.edu.br

# Como seu programa usa a memória?

- Sempre que criamos variáveis e estruturas locais, essas variáveis eram colocadas em uma pilha
- Mas o que é uma pilha?
  - Os novos elementos só podem ser **colocados no topo** da pilha;
  - Os elementos só podem ser **retirados do topo** da pilha;
  - Também chamada de **LIFO** (**Last IN**, **First OUT**);

# Pilha (Stack)

- Durante a execução de um programa:
  - Quando uma função a vai ser executada (incluindo main()), as variáveis declaradas nessa função (locais) são temporariamente armazenadas na memória em uma estrutura de pilha;
  - Assim que a função finaliza a execução, suas variáveis são retiradas da pilha;



# Heap

- O Heap é uma outra região de memória geralmente maior que a Stack
  - Essa região **não** é automaticamente gerenciada;
  - Para usá-la o programador deve realizar **ALOCAÇÃO DINÂMICA**;
  - Não existe restrição de tamanho de variáveis (exceto pelo hardware);
  - Pode-se criar uma estrutura que pode ir crescendo de acordo com a necessidade do programa;
  - Geralmente, o acesso ao Heap é “mais lento” do que a Stack
  - As variáveis alocadas na Heap são acessíveis para qualquer função, em qualquer parte do programa via ponteiros.

# Stack ou Heap?

- Quando usar **stack**:
  - Quando se está usando variáveis relativamente pequenas;
  - Quando as variáveis devem existir apenas enquanto a função onde elas foram definidas está ativa;
- Quando usar **heap**:
  - Quando se deseja manipular variáveis extremamente grandes;
  - Quando se deseja que as variáveis durem um longo período (e não estejam ligadas a vida de uma função);
  - Se as variáveis precisarem aumentar ou diminuir de tamanho
    - **Por exemplo:** em vetores e matrizes

# Definição

- **Alocação dinâmica** permite ao programador criar “variáveis” em tempo de execução, ou seja, alocar memória na **heap** para novas variáveis enquanto o programa está sendo executado, e não apenas quando se está escrevendo o programa.
  - Quantidade de memória é alocada sob demanda, ou seja, quando o programa precisa
  - Menos desperdício de memória
    - Espaço é reservado até liberação explícita
    - Depois de liberado, estará disponibilizado para outros usos e não pode mais ser acessado
    - Espaço alocado e não liberado explicitamente é automaticamente liberado ao final da execução

# Alocação Dinâmica

- A linguagem C padrão usa apenas 4 funções para o sistema de alocação dinâmica
- Elas estão disponíveis na biblioteca `stdlib.h`:
  - `malloc`
  - `calloc`
  - `realloc`
  - `free`

# Alocação Dinâmica - malloc

- **malloc**
  - A função malloc() serve para alocar memória e tem o seguinte protótipo:

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

- Funcionalidade
  - Dado o número de bytes que queremos alocar (num), ela aloca na memória e retorna um ponteiro **void*** para o primeiro byte alocado.

# Alocação Dinâmica - malloc

- O ponteiro **void*** pode ser atribuído a qualquer tipo de ponteiro via *type cast* (conversão de tipo).

  - Alocar 1000 bytes de memória livre.

```
char *ptr;
ptr = (char*) malloc(1000);
```

  - Alocar espaço para 50 inteiros:

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

# Alocação Dinâmica - malloc

- Operador **`sizeof()`**
  - Retorna o número de bytes de um dado tipo de dado.
    - **Ex.:** int, float, char, struct...

```c
typedef struct {
    int x, y;
}Ponto;

int main(){

    printf("Caracter: %ld\n", sizeof(char));// 1
    printf("Inteiro: %ld\n", sizeof(int));  // 4
    printf("Float: %ld\n", sizeof(float));  // 4
    printf("Ponto: %ld\n", sizeof(Ponto));  // 8

    return 0;
}// main
```

# Alocação Dinâmica - malloc

- Operador **`sizeof()`**
  - No exemplo anterior:

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

  - **`sizeof`**`(`**`int`**`)` retorna `4`
  - número de bytes do tipo **int** na memória
  - Portanto, são alocados 200 bytes (50 * 4)
  - `200 bytes = 50` posições do tipo **int** na memória

# Alocação Dinâmica - malloc

- Se não houver memória suficiente para alocar a memória requisitada, a função **`malloc()`** retorna um ponteiro nulo

```c
int *ptr;
ptr = (int*) malloc(5 * sizeof(int));

if(ptr == NULL){
        printf("Erro: Memória Insuficiente!\n");
        exit(1);
}//if
```

# Alocação Dinâmica - calloc

- A função calloc() também serve para alocar memória, mas possui um protótipo um pouco diferente:

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

- Funcionalidade
  - Faz o mesmo que a função malloc(), mas recebe a quantidade de posições a serem alocadas e o tamanho do tipo de dado alocado separadamente.
- Alocando com **calloc:**

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

if(ptr == NULL){
        printf("Erro: Memória Insuficiente!\n");
        exit(1);
}//if
```

# Alocação Dinâmica - realloc

- **realloc**
  - A função realloc() serve para realocar memória e tem o seguinte protótipo:

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

- Funcionalidade
  - A função modifica o tamanho da memória previamente alocada e apontada por ***ptr** para aquele especificado por num.
  - O valor de **num** pode ser maior ou menor que o original.
  - Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

# Alocação Dinâmica - realloc

- **realloc**
  - Um ponteiro para o bloco é devolvido porque realloc() pode precisar mover o bloco para aumentar seu tamanho.
  - Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco, e nenhuma informação é perdida.

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

for (int i = 0; i < 5; i++){
  p[i] = i+1;
  printf("%d ", p[i]);
}// for
printf("\n");

//Diminui o tamanho do array
p = realloc(p, 3*sizeof(int));
for (int i = 0; i < 3; i++){
  printf("%d ", p[i]);
}// for
printf("\n");

//Aumenta o tamanho do array
p = realloc(p, 10*sizeof(int));
for (int i = 0; i < 10; i++){
  printf("%d ", p[i]);
}// for
printf("\n");
```

# Alocação Dinâmica - free

- Libera uma região de memória que foi alocada dinamicamente
  - A função tem o seguinte protótipo:

```
void free(void *ptr);
```

- Porque precisamos liberar?
  - Diferente das variáveis definidas durante a escrita do programa, as variáveis alocadas dinamicamente não **são liberadas automaticamente** pelo programa.
  - Quando alocamos memória dinamicamente é **necessário que nós a liberemos** quando ela não for mais necessária.

# Alocação Dinâmica - free

- Como o programa sabe quantos bytes devem ser liberados?
  - Quando se aloca a memória, o programa guarda o número de bytes alocados numa "tabela de alocação" interna.
- Exemplo da função **`free()`**

```
//Aloca memória para o vetor
int *p = malloc(50*sizeof(int));

for (int i = 0; i < 50; i++){
  p[i] = i+1;
}// for

for (int i = 0; i < 50; i++){
     printf("%d ", p[i]);
}// for
printf("\n");

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

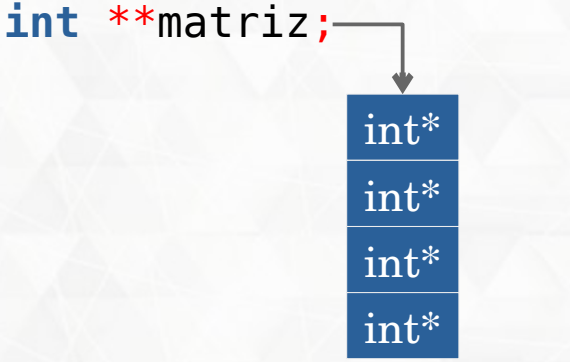
# Alocação de matrizes

- Para alocar matrizes, utilizamos ponteiros para ponteiros
  - Cada nível do ponteiro permite criar uma nova dimensão.

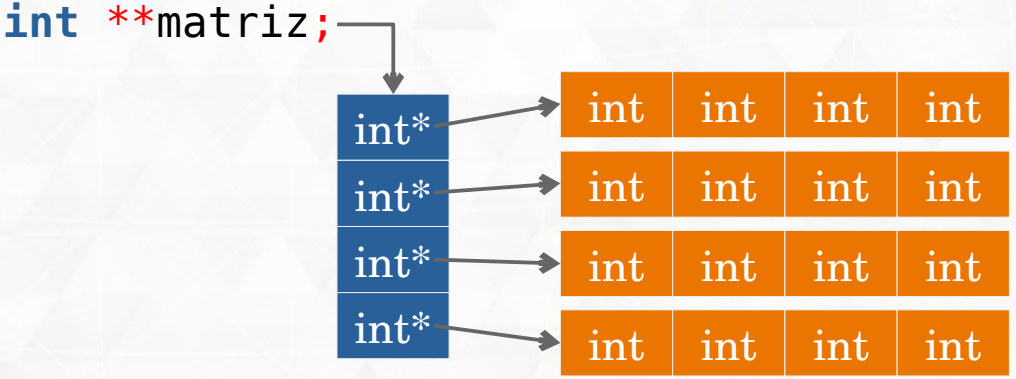
```
int **matriz = (int **) malloc(4 * sizeof(int*));  1º

for (int l = 0; l < 4; l++){
    matriz[l] = (int *) malloc(4 * sizeof(int));  2º
}// for
```

1º **malloc:** cria vetor de ponteiros para as linhas




2º **malloc:** cria um vetor para cada uma das linhas

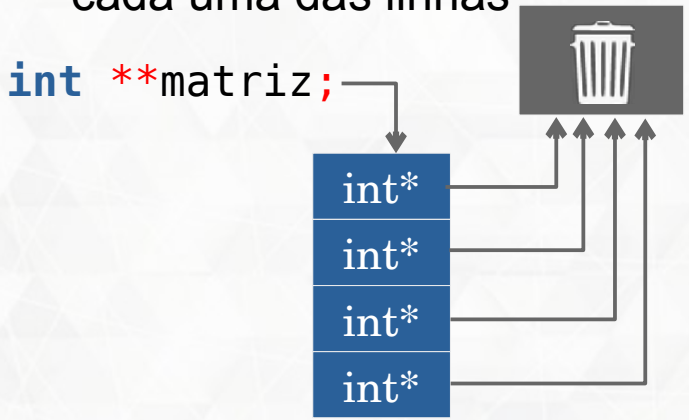


# Liberação de matrizes

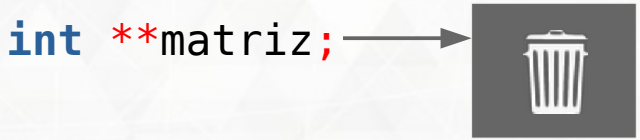
- Para liberar uma matriz é preciso liberar a memória alocada em cada uma de suas dimensões na **ordem inversa** da que foi alocada

```
for (int l = 0; l < 4; l++){
    free(matriz[l]); 1º
}// for
free(matriz); 2º
```

1º **free:** libera a memória de cada uma das linhas




2º **free:** libera o a memória do vetor de ponteiros para as linhas



# Alocação de struct

- Para alocar uma única estrutura
  - Um ponteiro para struct receberá o **malloc()**
  - Utilizamos o operador seta **->** para acessar os membros
  - Usamos **free()** para liberar a memória alocada

```
typedef struct{
    char nome[50];
    int idade;
} Pessoa;

int main(){

  Pessoa* cadastro = (Pessoa*) malloc (sizeof(Pessoa));

  strcpy(cadastro->nome,"Rosinosvalda");
  cadastro->idade = 22;

  free(cadastro);
  return 0;
}// main
```

# Alocação de struct

- Para alocar um vetor de estruturas
  - Um ponteiro para struct receberá o **`malloc()`**
  - Utilizamos colchetes `[ ]` para acessar os elementos do vetor
  - Usamos **`free()`** para liberar a memória alocada

```
Pessoa* vetCadastro = (Pessoa*) malloc (3 * sizeof(Pessoa));

strcpy(vetCadastro[0].nome,"Rosinosvalda");
vetCadastro[0].idade = 22;

strcpy(vetCadastro[1].nome,"Rosinosberto");
vetCadastro[1].idade = 28;

strcpy(vetCadastro[2].nome,"Rosinoscleia");
vetCadastro[2].idade = 35;

free(vetCadastro);
```

# Exercícios

- **1)** Elabore um programa que leia do usuário o tamanho de um vetor a ser lido. Em seguida, faça a alocação dinâmica desse vetor. Por fim, leia o vetor do usuário e o imprima.
- **2)** Escreva uma função que receba um valor inteiro positivo N por parâmetro e retorne o ponteiro para um vetor de tamanho N alocado dinamicamente. Se N for negativo ou igual a zero, um ponteiro nulo deverá ser retornado.
- **3)** Escreva uma função que receba como parâmetro dois vetores via referência( A e B) e o tamanho N. A função deve retornar o ponteiro para um vetor C de tamanho N alocado dinamicamente, em que:

```
C[i] = A[i] * B[i]
```

- **4)** Escreva uma função que receba como parâmetro um valor L e um valor C e retorne o ponteiro para uma matriz alocada dinamicamente contendo L linhas e C colunas. Essa matriz deve ser inicializada com o valor 0 em todas as suas posições.

  **Obs:** Ao final de cada programa, não se esqueça de liberar a memória alocada dinamicamente