

**UNIVERSIDADE FEDERAL DO PARÁ  
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS  
FACULDADE DE COMPUTAÇÃO  
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**Disciplina: Projeto de Algoritmos I**

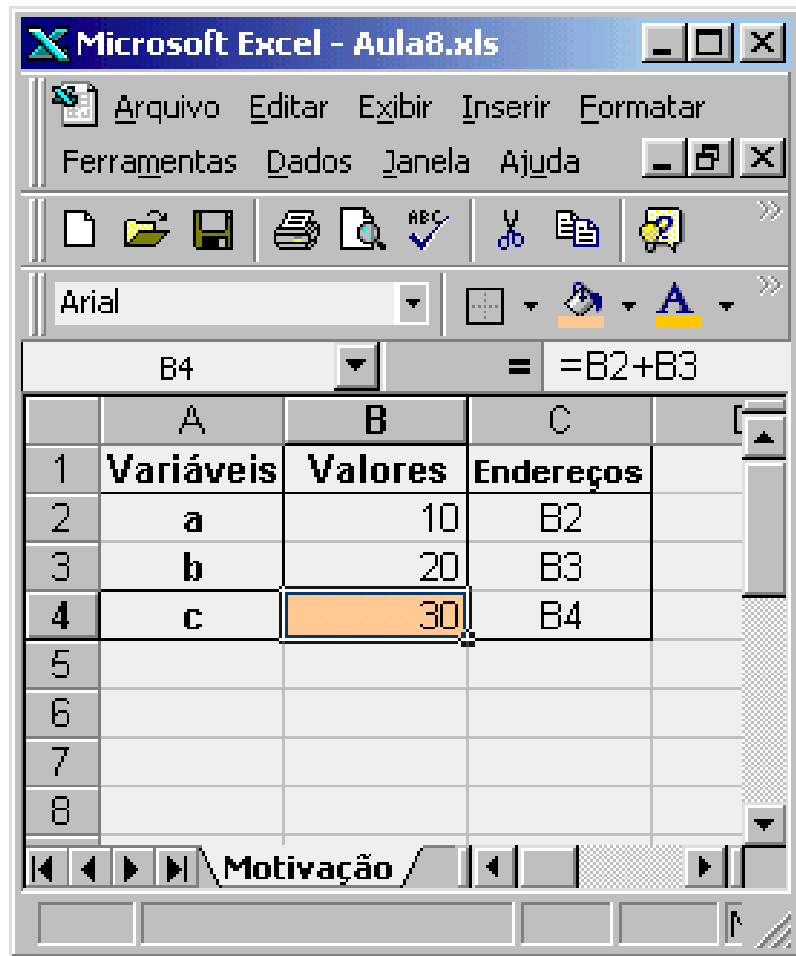
**Prof. Jefferson Moraes  
Email: [jmorais@ufpa.br](mailto:jmorais@ufpa.br)**



# Ponteiros

# Motivação

## ➤ Funcionamento do Excel



The screenshot shows the Microsoft Excel interface with the file 'Aula8.xls'. The spreadsheet has columns A, B, and C, and rows 1 through 8. The data is as follows:

	A	B	C
1	Variáveis	Valores	Endereços
2	a	10	B2
3	b	20	B3
4	c	30	B4
5			
6			
7			
8			

The formula bar shows the formula  $=B2+B3$  for cell B4.

## ➤ No Excel

**As células são identificadas por coordenadas coluna/linha.**

**Exemplo:**

Célula **B2** possui o valor 10

Célula **B3** possui o valor 20

## ➤ Em linguagem C

**Os identificadores são compostos por nomes definidos pelo usuário.**

**Exemplo:**

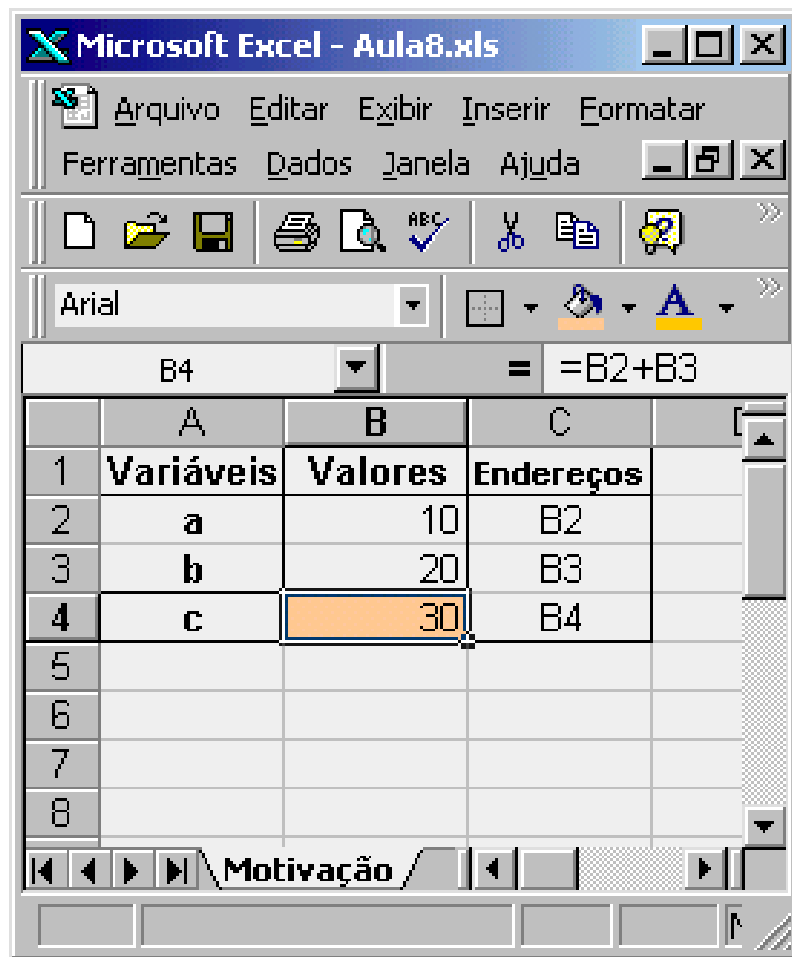
```
int a = 10;
```



**Mas o identificador 'a' possui um endereço na memória?**

# Motivação

## ➤ Endereço de uma variável



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - Aula8.xls". The menu bar includes "Arquivo", "Editar", "Exibir", "Inserir", "Formatar", "Ferramentas", "Dados", "Janela", and "Ajuda". The toolbar contains various icons for file operations and editing. The font is set to "Arial". The active cell is B4, and the formula bar shows "=B2+B3". The spreadsheet contains the following data:

	A	B	C
1	Variáveis	Valores	Endereços
2	a	10	B2
3	b	20	B3
4	c	30	B4
5			
6			
7			
8			

## ➤ Suponha as variáveis em C

### 3 variáveis:

```
int a, b, c;
```

```
a = 10;
```

```
b = 20;
```

```
c = a + b; //c = 30;
```

## ➤ Endereço das variáveis

Fazendo um paralelo com o Excel, a formação de endereços segue a notação Coluna/Linha:

```
End(a) = B2;
```

```
End(b) = B3;
```

```
End(c) = B4;
```

## ➤ Manipulação de endereços

```
B4 = B2 + B3; // B4 = 30
```

# ENDEREÇOS DE MEMÓRIA

## ➤ Endereços em C

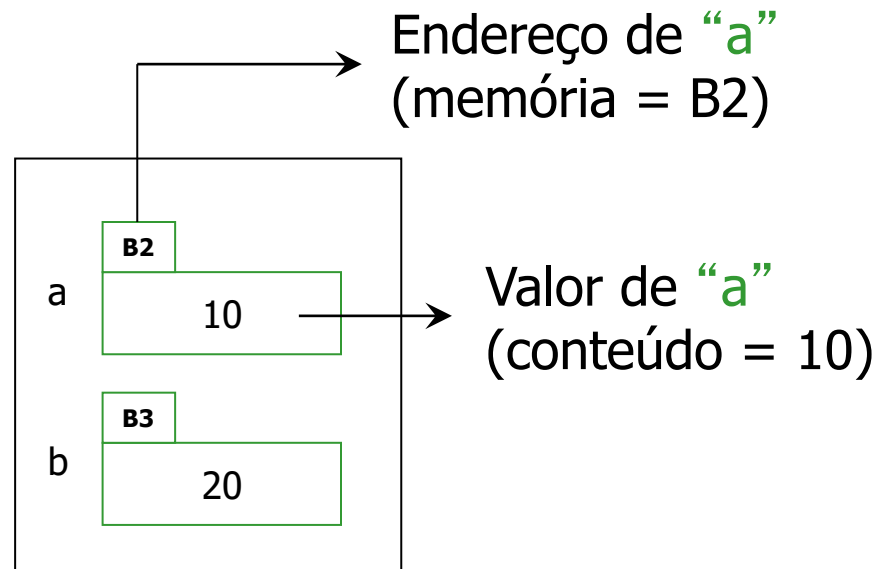
**Em linguagem C, todo identificador possui um endereço de memória. Identificadores são usados para representar variáveis, constantes e funções.**

**Suponha, por enquanto, que **endereços** em linguagem C sejam iguais a representação de **coordenadas** das células de uma planilha Excel (B1, B2, etc.)**

## ➤ Exemplo

```
int a, b;  
a = 10;  
b = 20;
```

## ➤ Ilustração:



# ENDEREÇOS DE MEMÓRIA

## ➤ Endereços em C

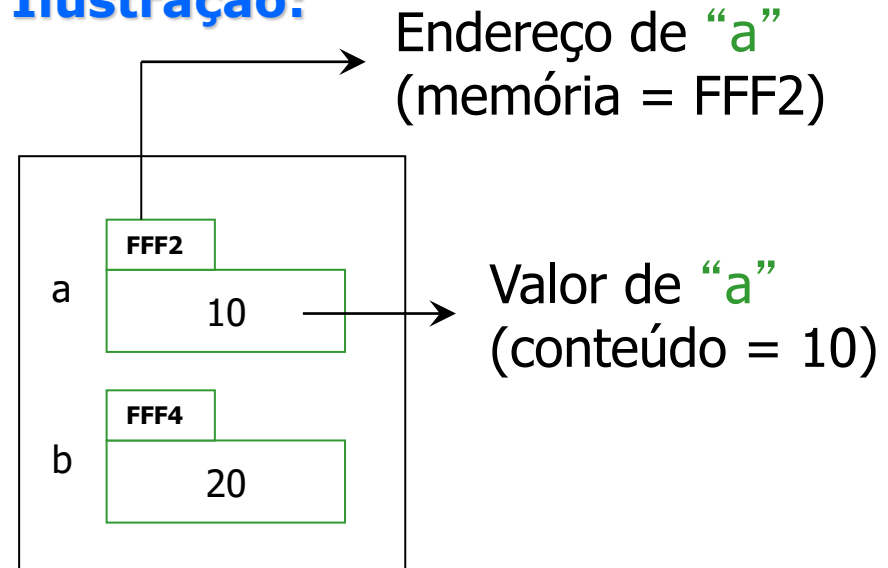
Os endereços são representados por números em base **hexadecimal**.

Vamos fazer a mesma analogia de endereços das células de uma planilha Excel (B1, B2, etc.) mas, agora, com valores em **hexadecimal**

## ➤ Exemplo

```
int a, b;  
a = 10;  
b = 20;
```

## ➤ Ilustração:



# ENDEREÇOS DE MEMÓRIA

## ➤ Operador &

**Em linguagem C, o operador & permite determinar o endereço de uma variável.**

## ➤ Exemplo

```
int a = 10;
cout << a << endl;
cout << &a;
```

## ➤ Exemplo:

```
1  #include <iostream>
2
3  using namespace std;
4  int main() {
5      int a = 10;
6
7      /*0 conteúdo da variavel a.*/
8      cout << a << endl;
9
10     /*0 endereço da variavel a (hexa).*/
11     cout << &a;
12 }
```



&a retorna algo como FFF4

# ENDEREÇOS DE MEMÓRIA

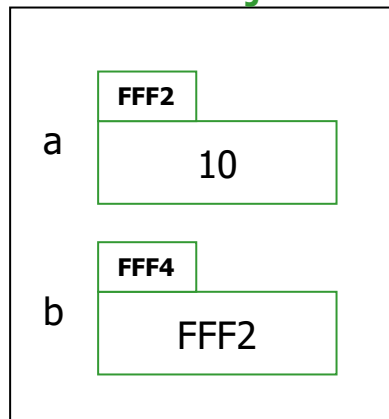
## ➤ Manipulação de Endereços

Em linguagem C, uma variável do tipo `int` pode armazenar endereços de memória (porém, alguns compiladores apontarão um **warning**).

## ➤ Exemplo

```
int a, b;  
a = 10;  
b = &a;
```

## Ilustração



## ➤ Exemplo:

Aula06\_01

```
#include <stdio.h>  
  
int main()  
{  
    int a = 10;  
    int b;  
  
    b = &a;  
  
    /* O conteúdo da variavel b. */  
    cout << b << endl;  
  
    /* O endereço da variavel b (hexa). */  
    cout << &b << endl;  
  
    return (0);  
}
```



# PONTEIROS

## ➤ Declaração: Operador \*

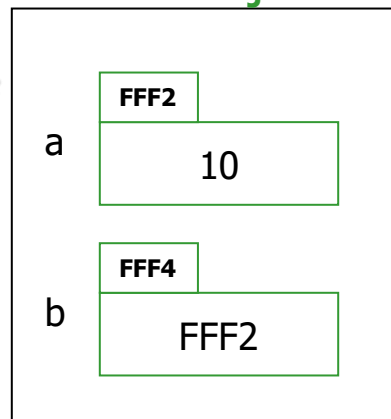
**Tipo\*** Nome\_Ponteiro;

**Em linguagem C, para poder armazenar endereços de memória, recomenda-se utilizar ponteiros.**

## ➤ Exemplo

```
int a = 10;  
int* b;  
b = &a;
```

## Ilustração



## ➤ Exemplo:

Aula06\_01

```
#include <stdio.h>  
  
int main()  
{  
    int a = 10;  
    int* c;  
  
    c = &a;  
  
    /* O conteúdo da variavel b. */  
    cout << c << endl;  
  
    /* O endereço da variavel b (hexa). */  
    cout << &c << endl;  
  
    return (0);  
}
```

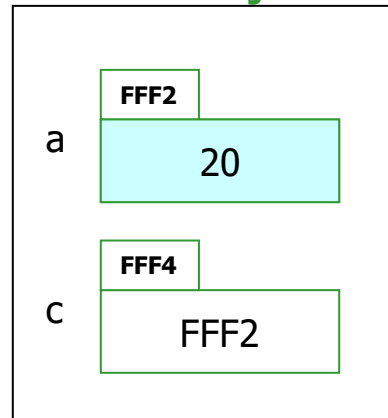
**Ponteiros são variáveis que armazenam em seu conteúdo apenas endereços (Hexa).**

# PONTEIROS

## ➤ Acesso: Operador \*

**Pode-se acessar o conteúdo que um ponteiro possui também através do operador \***

### Ilustração



## ➤ Exemplo

```
int a = 10;
int* c;
c = &a;
(*c) = 20;
```

## ➤ Paralelo com o Excel

**a = 10;**  
**End(c) = End(a);**  
**c = 20; //a = 20**



## ➤ Exemplo:

Aula06\_01

```
#include <stdio.h>

int main()
{
    int a = 10;
    int* c;
    /* O conteúdo da variavel a. */
    cout << a << endl;

    c = &a;
    (*c) = 20;
    /* O conteúdo da variavel a. */
    cout << a << endl;

    return (0);
}
```

**int\* c** armazena algo como **FFF4**  
**(\*c)** acessa o cont. do endereço



# **Alocação Dinâmica**



# Alocação Dinâmica de Memória

- Matrizes e vetores possuem tamanhos pré-definidos, ocupando a memória como outra variável qualquer. Isso gera o incômodo de definirmos um tamanho muito maior que aquele que normalmente será usado em tempo de execução.
- Mas como definir um tamanho dinamicamente, ou seja em tempo de execução, conforme a necessidade do usuário, desde que haja memória disponível?



# Alocação Dinâmica de Memória

- Isso é feito utilizando a chamada memória dinâmica, oferecida pelo sistema operacional, durante a execução do programa.
- As variáveis globais e locais são armazenadas na Memória Estática. A Memória Dinâmica é utilizada para o armazenamento de dados, “desafogando” a memória estática, mas não de variáveis.

# Memória Estática x Dinâmica

- Para podermos usar a memória dinâmica, precisamos saber o endereço de memória em que os dados serão armazenados.
- É como se uma variável, em vez de armazenar os dados propriamente ditos, possui apenas o endereço da localização dos dados (uma caixa postal para as correspondências, por exemplo).

# ALOCAÇÃO DINÂMICA DE MEMÓRIA

## ➤ Os 3 Mandamentos da ADM:

- 1: “Não esquecerás de liberar a memória alocada por ti”
- 2: “Não acessarás um ponteiro não inicializado”
- 3: “Aprenderás e respeitarás os dois primeiros mandamentos em prol da integridade dos sistemas rodando em memória, seja em Windows ou Linux”

# Tamanho do Tipo de Dado

- Para realizarmos a alocação dinâmica, precisamos saber quantos bytes um tipo de dado ocupa na memória. Assim podemos solicitar ao sistema operacional o tamanho total em bytes que desejamos.
- Para isso, vamos fazer uso do operador `sizeof()`, que retorna o tamanho em bytes de um tipo de dado:
  - `int sizeof(tipo_de_dado);`
  - Ex:



# Biblioteca stdlib.h – Alocação 1

- Contém as funções para alocação e liberação dinâmica de espaços de memória

- Alocar memória

- `void *malloc(int tamanho_em_bytes);`

- `void *` significa que o retorno é um endereço de memória genérico, não importante o tipo de dado.

```
int *x; // vetor criado dinamicamente
x = (int *) malloc(sizeof(int) * 10000);
x[9876] = 1234;
```

# Biblioteca stdlib.h – Alocação 1

- Contém as funções para alocação e liberação dinâmica de espaços de memória

- Alocar memória

- `void *malloc(int tamanho_em_bytes);`

- `void *` significa que o retorno é um endereço de memória genérico, não importante o tipo de dado.

```
int *x; // vetor criado dinamicamente
      // Mesmo tipo de dado
x = (int *) malloc(sizeof(int) * 10000);
x[9876] = 1234;
```

Tamanho

# Biblioteca stdlib.h – Alocação 1

- Contém as funções para alocação e liberação dinâmica de espaços de memória
- Alocar memória
  - `void *malloc(int tamanho_em_bytes);`
    - `void *` significa que o retorno é um endereço de memória genérico, não importante o tipo de dado.

```
int *x; // vetor criado dinamicamente
Obrigatório
x = (int *) malloc(sizeof(int) * 10000);
x[9876] = 1234;
...
```

# Biblioteca stdlib.h – Alocação 2

- Alocar memória e zerar o conteúdo alocado:
  - `void *calloc(int num_elem, int tamanho_elem);`
    - `num_elem` é o numero de elementos que serão criados.
    - `tamanho_elem` é o tamanho de cada elemento que será criado.

```
int *x; // vetor criado dinamicamente
x = (int *) calloc(10000, sizeof(int));
x[9876] = 1234;
...
```

# Biblioteca stdlib.h – Alocação 3

## ■ Realocar memória (para mais ou para menos):

□ `void *realloc((void *) apontador, int novo_tam);`

- `apontador` é a variável `apontador` que será alterada. Note que `(void*)` deve ser usado obrigatoriamente, independente do tipo de dado.
- `novo_tam` é o novo tamanho alocado, em bytes. Se menor, o restante será perdido.

```
int *x; // vetor criado dinamicamente
x = (int *) calloc(10000, sizeof(int));
x[9876] = 1234;
x = (int *) realloc((void *) x, 1000 * sizeof(int));
...
```

# Biblioteca stdlib.h – Liberação

## ■ Liberar memória alocada:

- `void free(void * apontador);`

- `apontador` é a variável `apontador` que contém o endereço para ser liberado.

- É muito importante liberar a memória que não será mais usada. Isso evita o esgotamento da memória dinâmica oferecida pelo sistema operacional.

```
int *x; // vetor criado dinamicamente
x = (int *) calloc(10000, sizeof(int));
x[9876] = 1234;
...
free(x); // libera todo espaço alocado
```



# Resumo

- Definição de ponteiros para alocação dinâmica de memória.
- Funções da `stdlib.h` para alocação dinâmica de memória: `malloc`, `calloc`, `realloc` e `free`.
- Não se pode usar um ponteiro sem primeiro alocar memória para ele.
- Nunca esquecer de liberar a memória que não será mais usada, pois uma hora, ela pode acabar