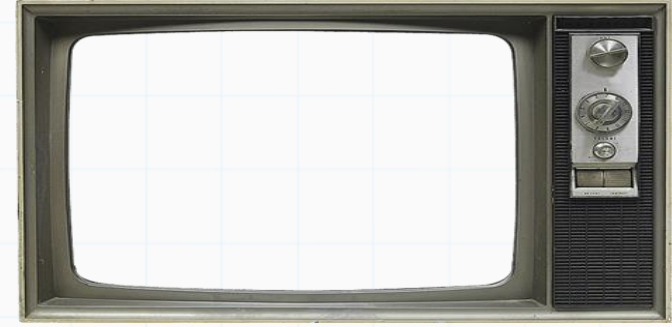


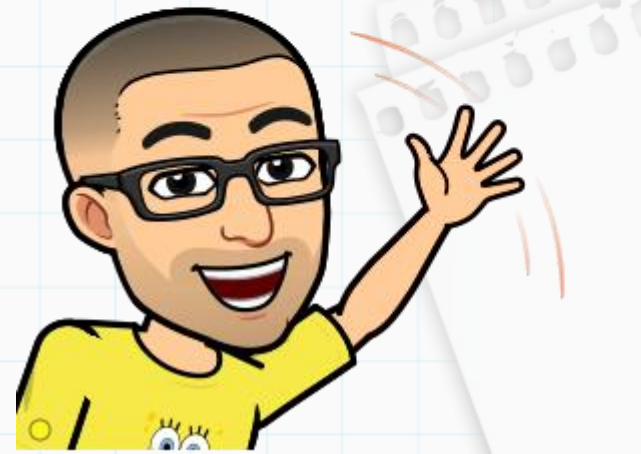
Programação Estruturada

Professor : Yuri Frota

yuri@ic.uff.br



HI

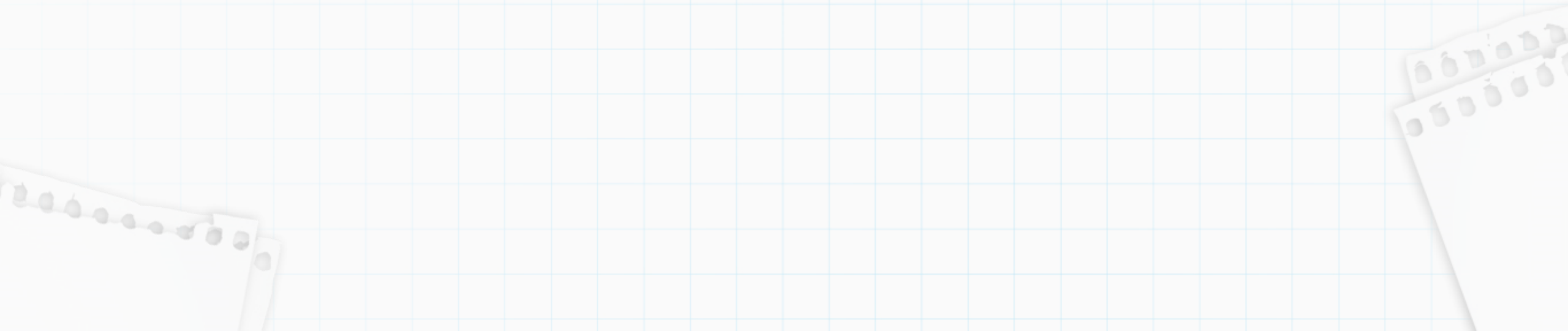
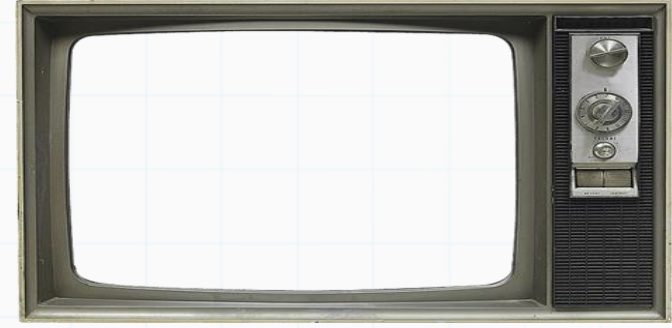
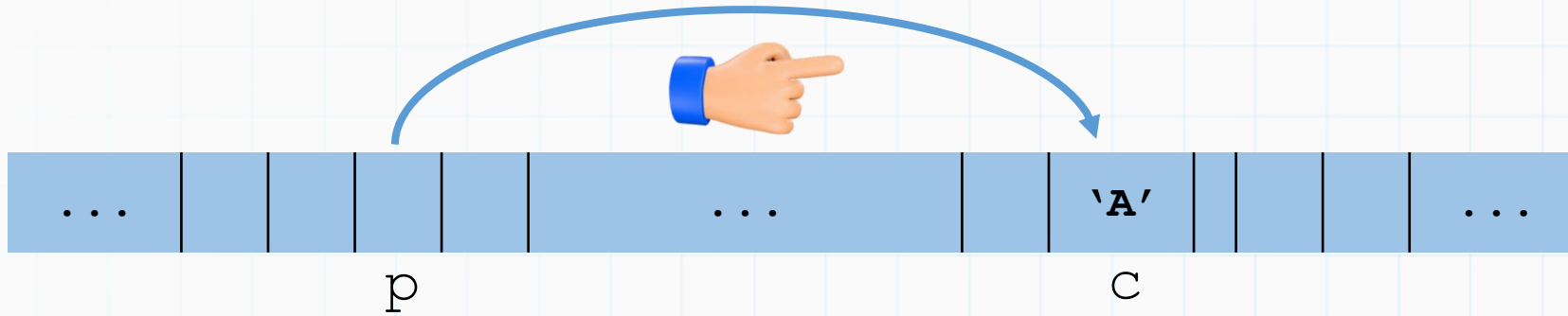


Ponteiros

Um ponteiro é uma variável que é capaz de guardar um endereço de memória

Exemplo:

Se `c` é um char e `p` um ponteiro que aponta para `c`, então `p` guarda o endereço da variável `c`.

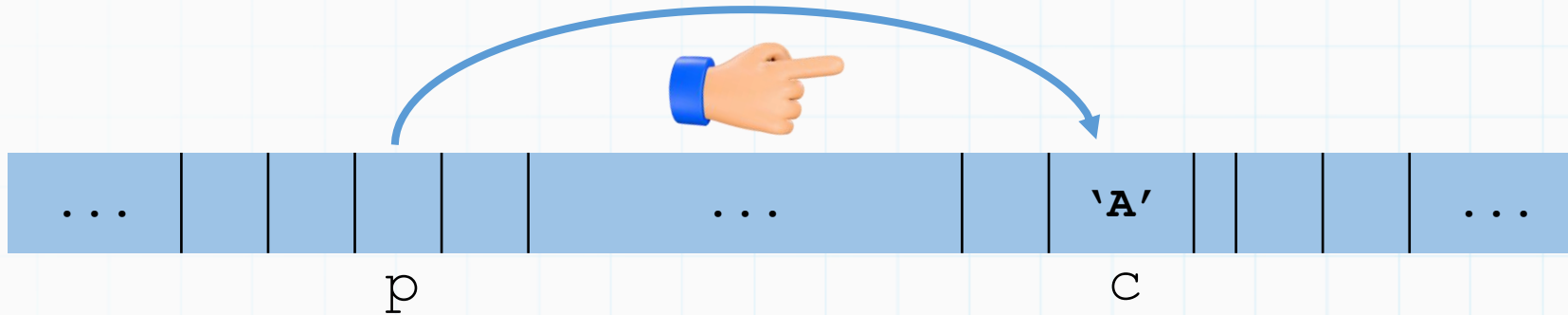


Ponteiros

Um ponteiro é uma variável que é capaz de guardar um endereço de memória

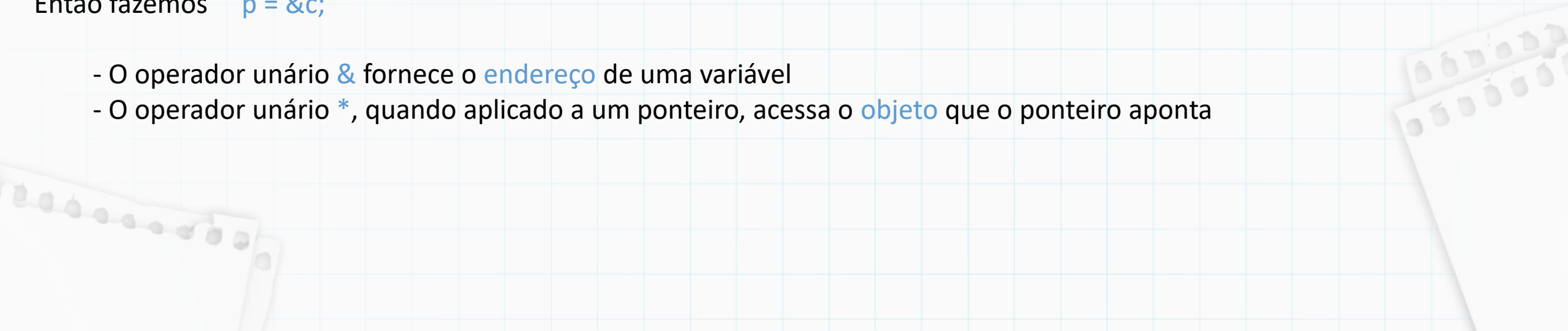
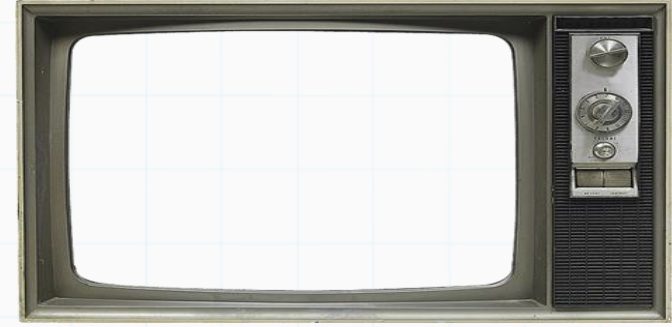
Exemplo:

Se `c` é um `char` e `p` um ponteiro que aponta para `c`, então `p` guarda o endereço da variável `c`.



Então fazemos `p = &c;`

- O operador unário `&` fornece o **endereço** de uma variável
- O operador unário `*`, quando aplicado a um ponteiro, acessa o **objeto** que o ponteiro aponta



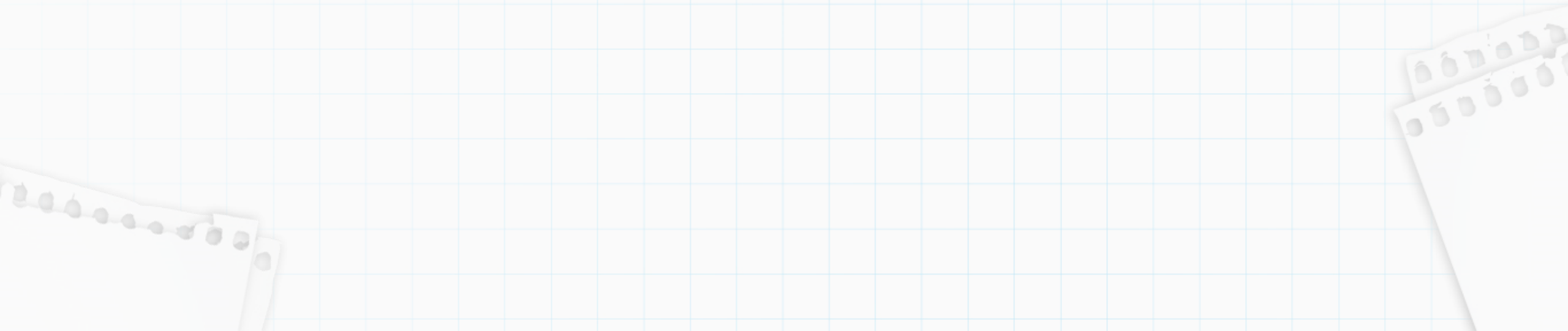
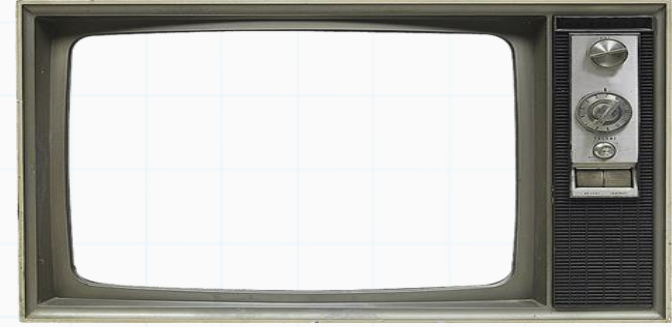
Ponteiros

Exemplo:



```
int x=1, y=2;  
int *ap;
```

nomes	conteúdo	endereços
ap		2001
		...
y		1002
x		1001



Ponteiros

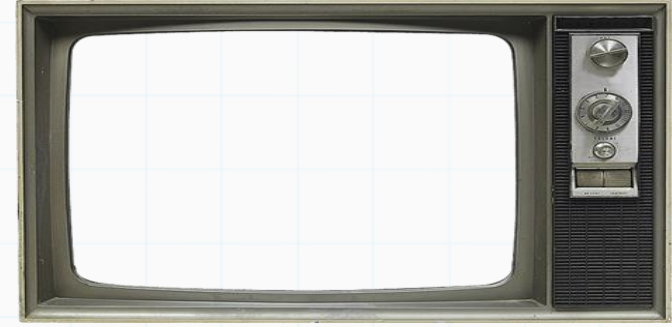
Exemplo:

```
int x=1, y=2;
```



```
int *ap;
```

nomes	conteúdo	endereços
ap		2001
		...
y	2	1002
x	1	1001



Ponteiros

Exemplo:

```
int x=1, y=2;
```

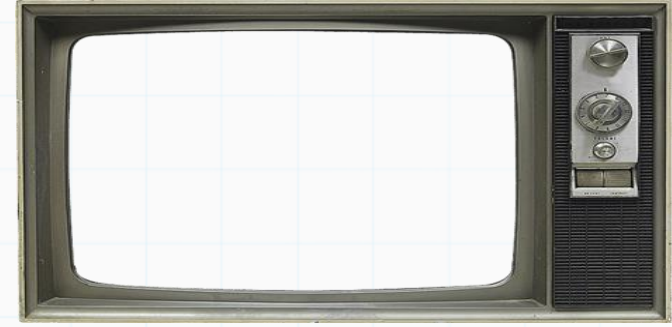
```
int *ap;
```

```
...
```

→ `ap = &x;`

ap recebe o
endereço de x

nomes	conteúdo	endereços
ap	1001	2001
		...
y	2	1002
x	1	1001



Ponteiros

Exemplo:

```
int x=1, y=2;
```

```
int *ap;
```

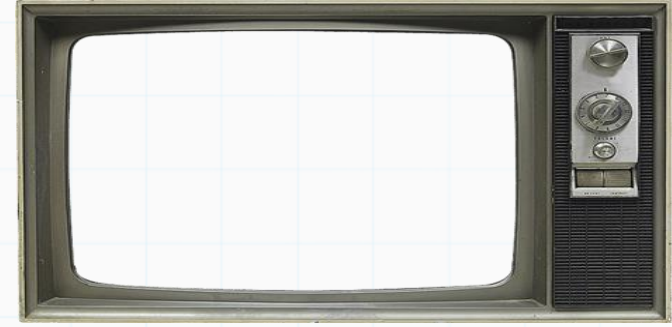
```
...
```

```
ap = &x;
```

```
→ y = *ap;
```

y recebe o
conteúdo do
endereço de ap

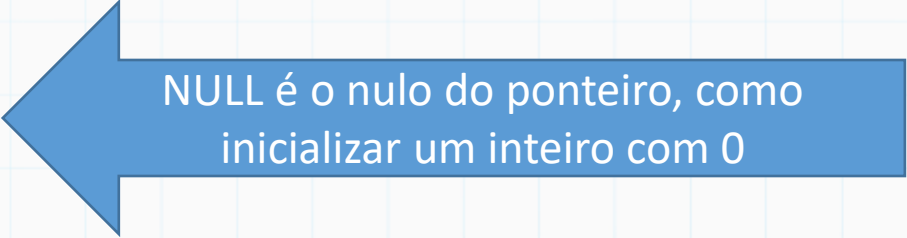
nomes	conteúdo	endereços
ap	1001	2001
		...
y	2 1	1002
x	1	1001



Ponteiros

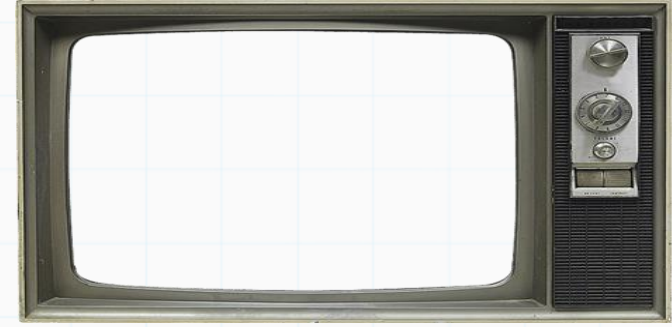
Se ponteiro `ap` aponta para um inteiro, então `*ap` pode fazer qualquer coisa que `x` poderia.

```
int x=1;  
int *ap = NULL;
```



NULL é o nulo do ponteiro, como
inicializar um inteiro com 0

NULL é da biblioteca `<stddef.h>`
Se não quiser usar o NULL, pode
usar 0 também



Ponteiros

Se ponteiro `ap` aponta para um inteiro, então `*ap` pode fazer qualquer coisa que `x` poderia.

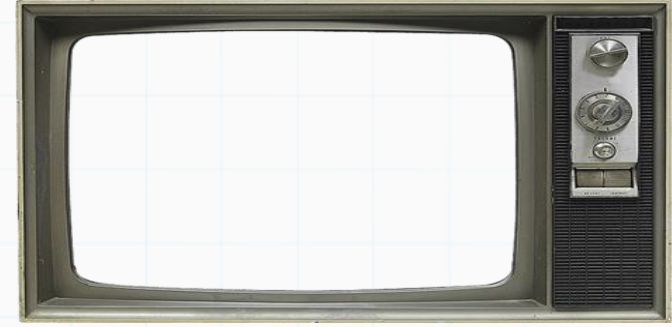
```
int x=1;  
int *ap = NULL;
```

NULL é o nulo do ponteiro, como
inicializar um inteiro com 0

Então:

```
*ap = &x  
*ap = *ap + 10; → equivale a x = x + 10  
(*ap)++; → equivale a x++  
*ap++; → incrementa o conteúdo de ap (i.e. incrementa o endereço do ponteiro)
```

NULL é da biblioteca `<stddef.h>`
Se não quiser usar o NULL, pode
usar 0 também



Ponteiros

Se ponteiro `ap` aponta para um inteiro, então `*ap` pode fazer qualquer coisa que `x` poderia.

```
int x=1;  
int *ap = NULL;
```

NULL é o nulo do ponteiro, como
inicializar um inteiro com 0

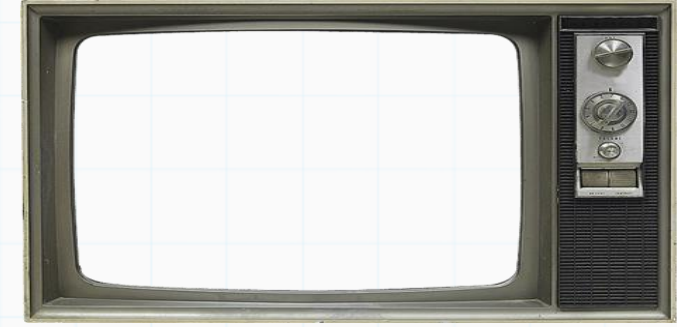
Então:

```
*ap = &x  
*ap = *ap + 10; → equivale a x = x + 10  
(*ap)++; → equivale a x++  
*ap++; → incrementa o conteúdo de ap (i.e. incrementa o endereço do ponteiro)
```

```
print("ponteiro = %p", ap) → imprime endereço do ponteiro ap
```

```
ponteiro = 000000000061FE08
```

O endereço é um número
hexadecimal



NULL é da biblioteca `<stddef.h>`
Se não quiser usar o NULL, pode
usar 0 também

Ponteiros

Se ponteiro `ap` aponta para um inteiro, então `*ap` pode fazer qualquer coisa que `x` poderia.

```
int x=1;  
int *ap = NULL;
```

NULL é o nulo do ponteiro, como
inicializar um inteiro com 0

Então:

```
*ap = &x  
*ap = *ap + 10; → equivale a x = x + 10  
(*ap)++; → equivale a x++  
*ap++; → incrementa o conteúdo de ap (i.e. incrementa o endereço do ponteiro)
```

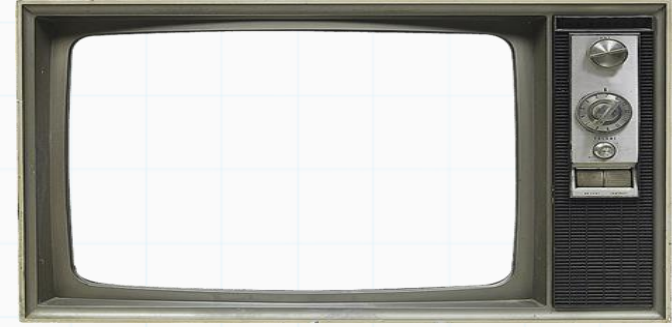
```
print("ponteiro = %p", ap) → imprime endereço do ponteiro ap
```

```
ponteiro = 000000000061FE08
```

O endereço é um número
hexadecimal

Se lembra da função `scanf()` ? Estávamos usando “&” para obter o endereço de memória para atribuir o valor a variável.

```
int n;  
scanf("%d", &n);
```



NULL é da biblioteca `<stddef.h>`
Se não quiser usar o NULL, pode
usar 0 também

Ponteiros

Se ponteiro `ap` aponta para um inteiro, então `*ap` pode fazer qualquer coisa que `x` poderia.

```
int x=1;  
int *ap = NULL;
```

NULL é o nulo do ponteiro, como
inicializar um inteiro com 0

Então:

```
*ap = &x  
*ap = *ap + 10; → equivale a x = x + 10  
(*ap)++; → equivale a x++  
*ap++; → incrementa o conteúdo de ap (i.e. incrementa o endereço do ponteiro)
```

```
print("ponteiro = %p", ap) → imprime endereço do ponteiro ap
```

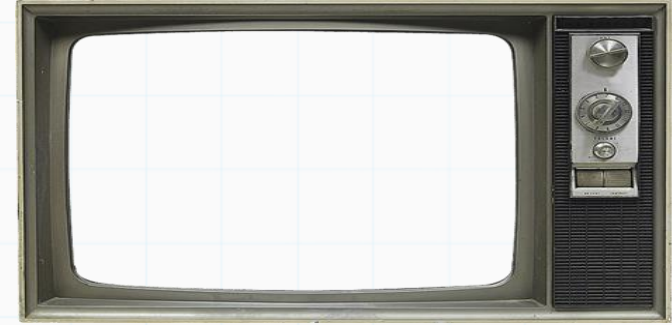
```
ponteiro = 000000000061FE08
```

O endereço é um número
hexadecimal

Se lembra da função `scanf()` ? Estávamos usando “&” para obter o endereço de memória para atribuir o valor a variável. Podemos então passar o ponteiro que funciona da mesma forma.

```
int n;  
scanf("%d", &n);
```

```
int n;  
int *p = &n;  
scanf("%d", *p);
```



NULL é da biblioteca `<stddef.h>`
Se não quiser usar o NULL, pode
usar 0 também

Ponteiros

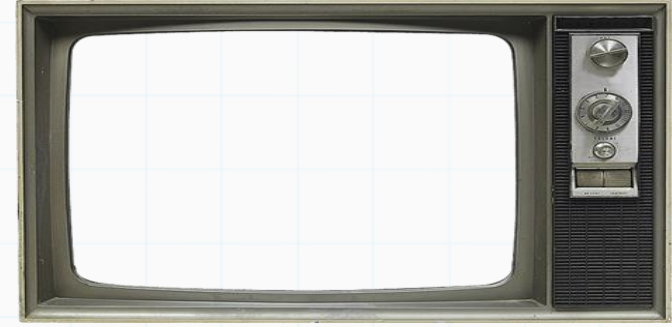
Ponteiros imutáveis: São ponteiros onde não podemos mudar o conteúdo do valor para onde ele aponta. Usamos o comando `const`.

```
int x=9999;  
const int *ap = &x;
```

Conteúdo de x não pode ser
mudado pelo ponteiro

```
*ap = 8888;
```

ERRO



Ponteiros

Ponteiros imutáveis: São ponteiros onde não podemos mudar o conteúdo do valor para onde ele aponta. Usamos o comando `const`.

```
int x=9999;  
const int *ap = &x;
```

Conteúdo de x não pode ser
mudado pelo ponteiro

```
*ap = 8888;
```

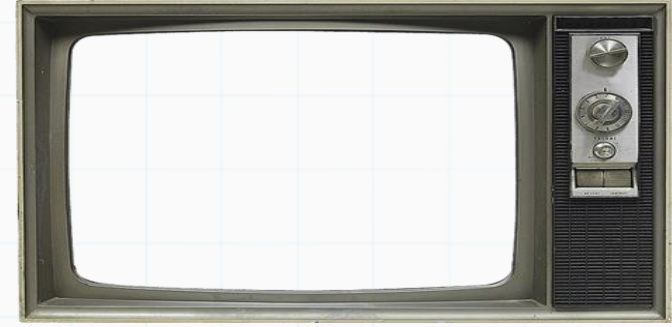
ERRO

```
x = 8888;
```

Ainda podemos mudar o
valor, mas não pelo
ponteiro

```
int x2=7777;  
ap = &x2;
```

E podemos mudar para
onde ele aponta



Ponteiros

Ponteiros imutáveis: São ponteiros onde não podemos mudar o conteúdo do valor para onde ele aponta. Usamos o comando `const`.

```
int x=9999;  
const int *ap = &x;
```

Conteúdo de x não pode ser mudado pelo ponteiro

```
*ap = 8888;
```

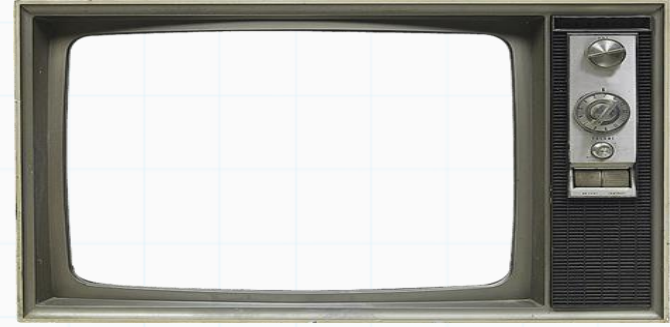
ERRO

```
x = 8888;
```

Ainda podemos mudar o valor, mas não pelo ponteiro

```
int x2=7777;  
ap = &x2;
```

E podemos mudar para onde ele aponta



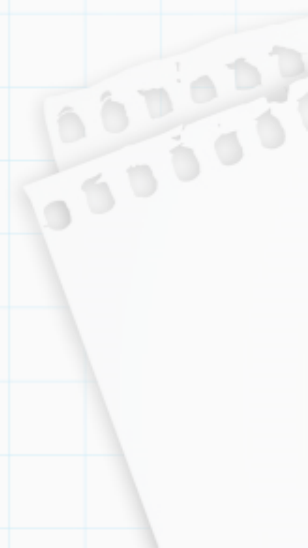
Se quisermos um ponteiro que não pode mudar para onde ele aponta:

```
int x=9999;  
int *const ap = &x;
```

```
int x2 = 7777;
```

```
ap = &x2;
```

ERRO



Ponteiros

Ponteiros imutáveis: São ponteiros onde não podemos mudar o conteúdo do valor para onde ele aponta. Usamos o comando `const`.

```
int x=9999;  
const int *ap = &x;
```

Conteúdo de x não pode ser mudado pelo ponteiro

```
*ap = 8888;
```

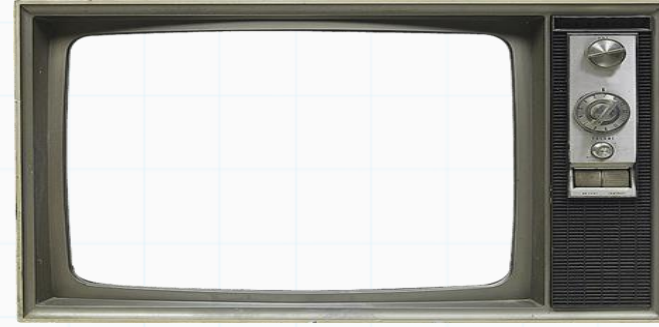
ERRO

```
x = 8888;
```

Ainda podemos mudar o valor, mas não pelo ponteiro

```
int x2=7777;  
ap = &x2;
```

E podemos mudar para onde ele aponta



Se quisermos um ponteiro que não pode mudar para onde ele aponta:

```
int x=9999;  
int *const ap = &x;
```

```
int x2 = 7777;
```

```
ap = &x2;
```

ERRO

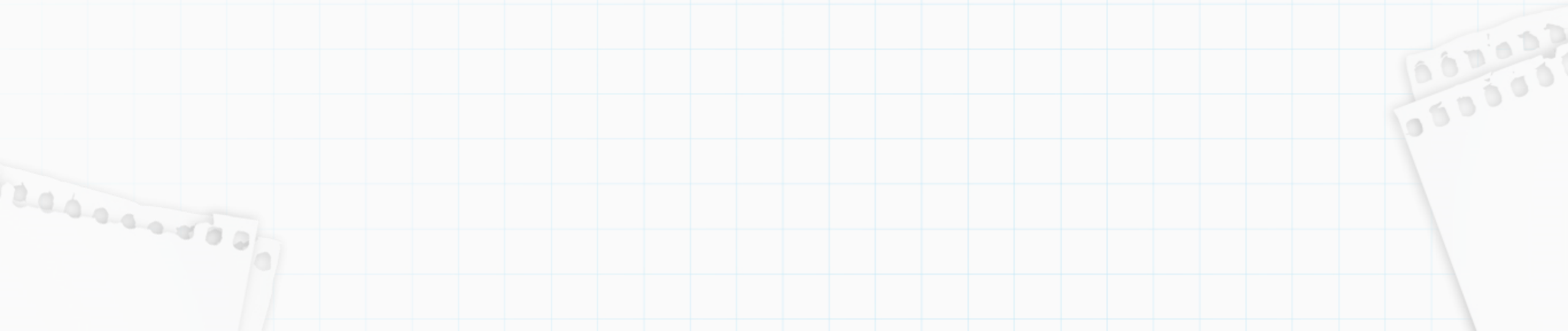
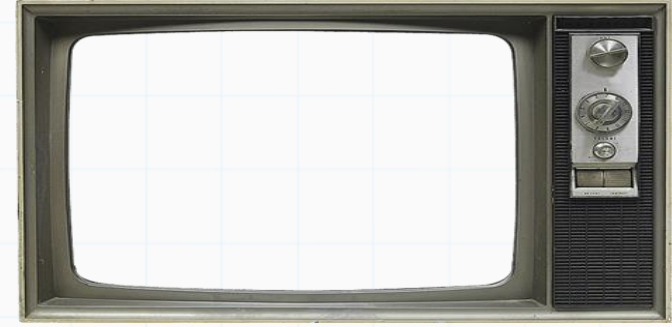
Podemos deixar tudo constante, o valor apontado e o endereço

```
int x=9999;  
const int *const ap = &x;
```


Ponteiros

Ponteiros sem tipo: Podemos definir um ponteiro sem tipo (`void`).

- Podem apontar para qualquer tipo 😊
- Porém, para ser referenciado, precisa ser convertido para o tipo do valor de que está apontando 😞



Ponteiros

Ponteiros sem tipo: Podemos definir um ponteiro sem tipo (`void`).

- Podem apontar para qualquer tipo 😊
- Porém, para ser referenciado, precisa ser convertido para o tipo do valor de que está apontando 😞

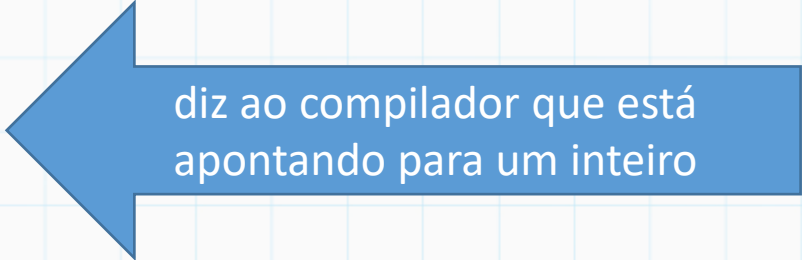
```
int i=10;  
float f=2.34;  
char c = 'k';
```

```
void * pt;
```

```
pt = &i;  
printf("%d \n", * (int*) pt);
```

```
pt = &f;  
printf("%.2f \n", * (float*) pt);
```

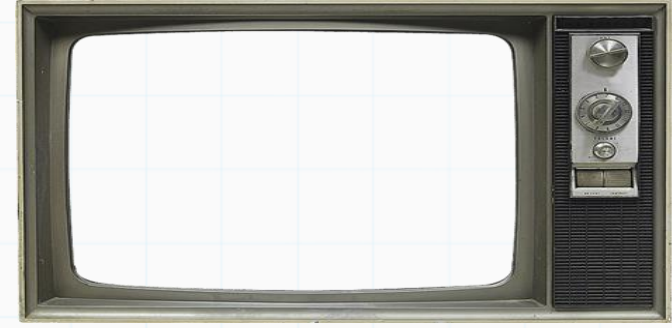
```
pt = &c;  
printf("%c \n", * (char*) pt);
```



diz ao compilador que está apontando para um inteiro

Vantagens: utilizar o mesmo ponteiro para várias variáveis

Desvantagens: tem sempre que saber para que tipo ele aponta



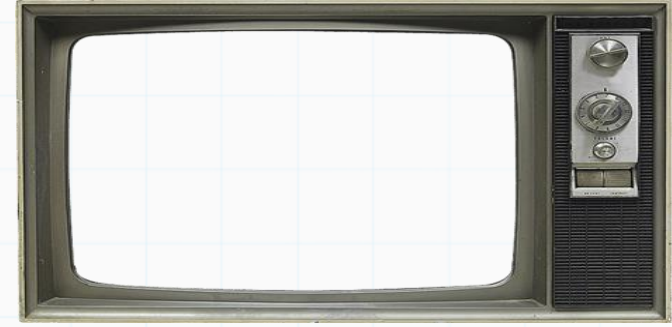
Ponteiros

Até agora usamos apenas a **passagem de parâmetros por valor**

```
int teste(int param1, float param2, ...)
```

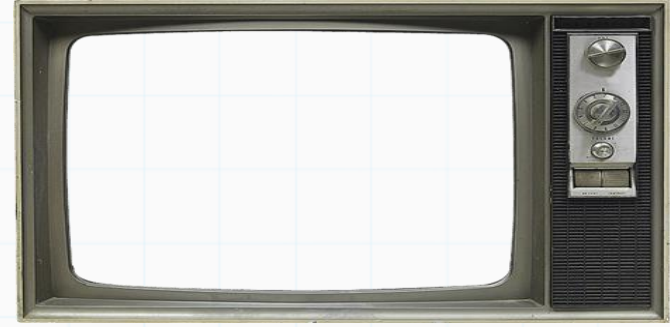
- A função chamadora passa uma cópia dos valores das variáveis para a função chamada
- Logo, os valores que são manipulados/alterados dentro da função chamada são apenas cópias
- Os valores das variáveis não são alterados na função chamadora

exceto vetores e matrizes pois estamos passando cópias para ponteiros da primeira posição das estruturas



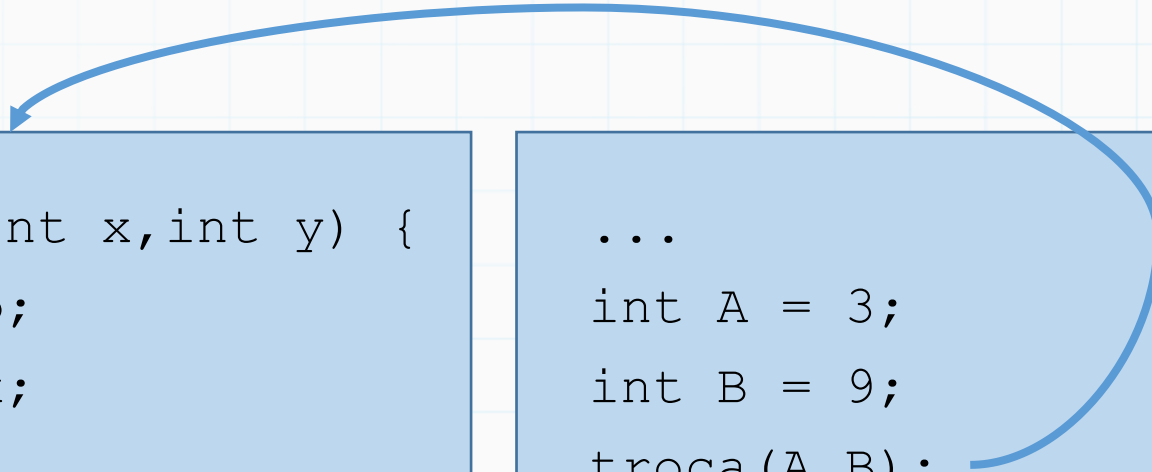
Ponteiros

Exemplo:



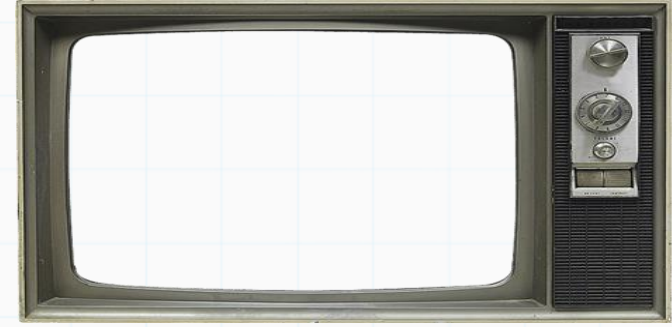
```
void troca(int x,int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
...  
int A = 3;  
int B = 9;  
troca(A,B);  
printf("%d %d",A,B);  
...
```



Ponteiros

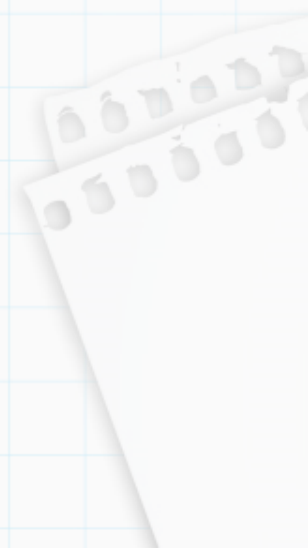
Exemplo:



```
void troca(int x,int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

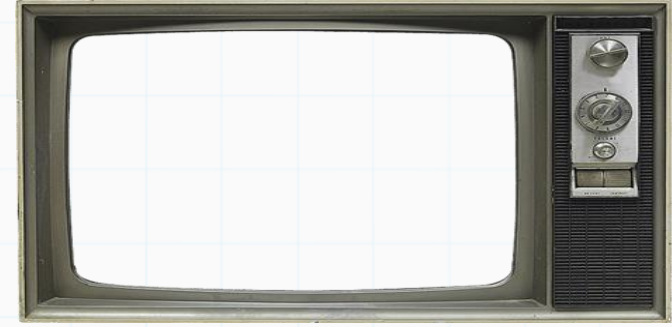
```
...  
int A = 3;  
int B = 9;  
troca(A,B);  
printf("%d %d",A,B);  
...
```

A	B	x	y	temp
3	9			



Ponteiros

Exemplo:



```
void troca(int x,int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

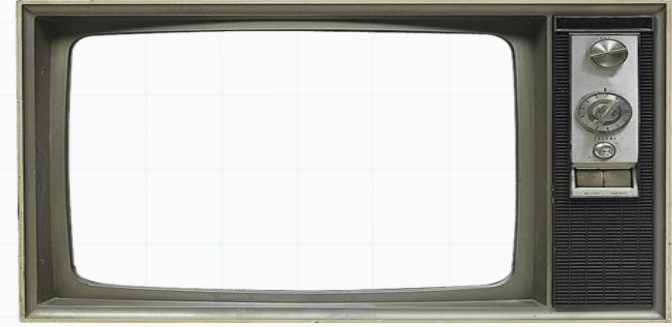
```
...  
int A = 3;  
int B = 9;  
troca(A,B);  
printf("%d %d",A,B);  
...
```

A	B	x	y	temp
3	9	3	9	



Ponteiros

Exemplo:



```
void troca(int x,int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

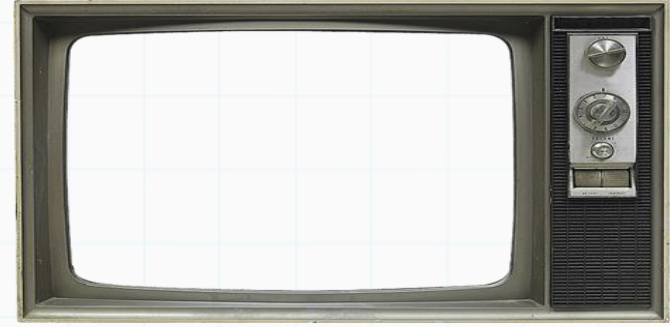
```
...  
int A = 3;  
int B = 9;  
troca(A,B);  
printf("%d %d",A,B);  
...
```

A	B	x	y	temp
3	9	3	9	3



Ponteiros

Exemplo:



```
void troca(int x,int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

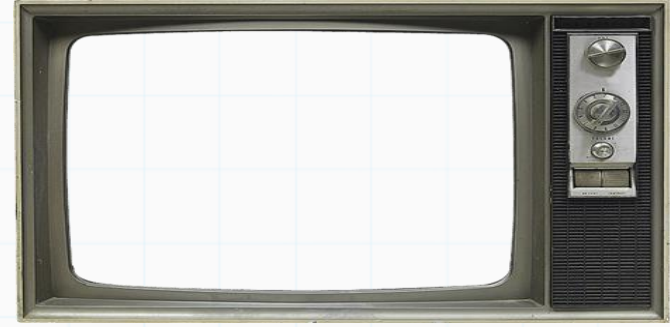
```
...  
int A = 3;  
int B = 9;  
troca(A,B);  
printf("%d %d",A,B);  
...
```

A	B	x	y	temp
3	9	3	9	3
		9		



Ponteiros

Exemplo:



```
void troca(int x,int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

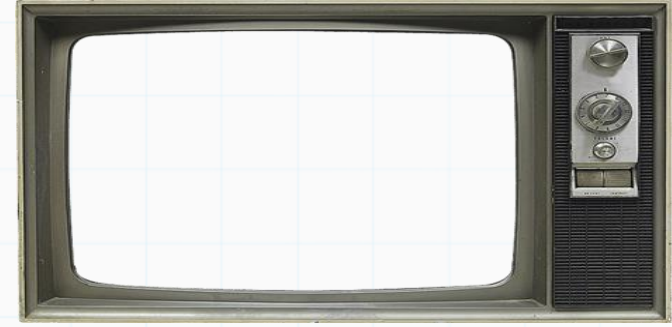
```
...  
int A = 3;  
int B = 9;  
troca(A,B);  
printf("%d %d",A,B);  
...
```

A	B	x	y	temp
3	9	3	9	3
		9	3	



Ponteiros

Exemplo:



```
void troca(int x,int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
...  
int A = 3;  
int B = 9;  
troca(A,B);  
printf("%d %d",A,B);  
...
```

A	B	x	y	temp
3	9	3	9	3
		9	3	

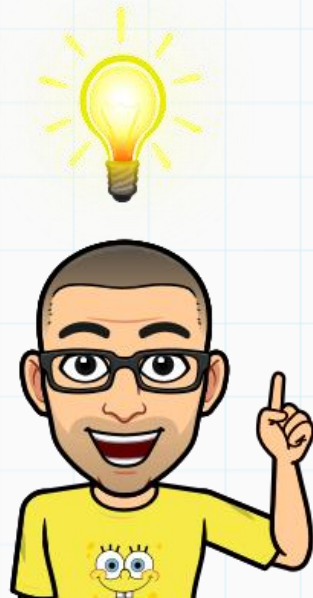
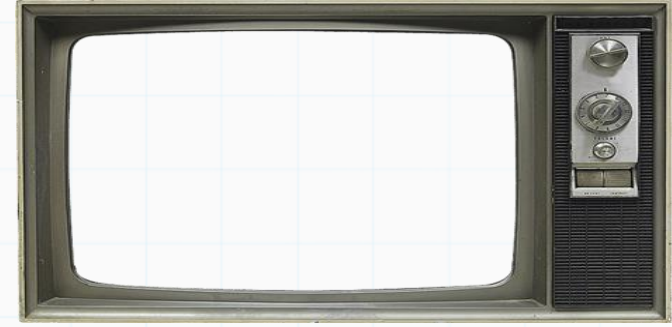
Ponteiros

O resultado da função troca não é o desejado!

Não podemos usar a **passagem de parâmetros por VALOR**.
Temos que usar a **passagem por REFERÊNCIA (ponteiro)**.

Assim, ao invés de enviarmos cópias dos valores para a função chamada, temos que enviar o endereço das variáveis.

Desta forma, as alterações feitas dentro da função chamada se refletirão na função chamadora.



Ponteiros

Exemplo:

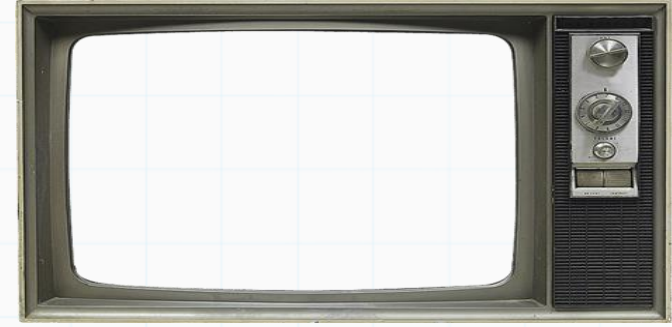
```
void troca(int *x,int *y){  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

```
...  
int A = 3;  
int B = 9;  
troca(&A,&B);  
printf("%d %d",A,B);  
...
```

A *x	B *y	temp
3	9	

&A e &B são
endereços de memória
de onde estão
armazenados os
valores das variáveis

*x e *y são variáveis
que armazenam
endereços (ponteiros)
para as variáveis A e B



Ponteiros

O que será impresso:



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

```
int main ( ){  
    int i = 10 , j = 20;  
    int temp ;  
    int *p1 , *p2 ;  
    p1 = &i ;  
    p2 = &j;  
    temp = *p1;  
    *p1 = *p2 ;  
    *p2 = temp;  
    printf ( " %i %i\n " , i , j);  
    return 0;  
}
```

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

```
void func(int *px, int *py) {  
    px = py;  
    *py = (*py) * (*px);  
    *px = *px + 2;  
}  
  
void main() {  
    int x = 3, y = 4;  
    func(&x,&y);  
    printf("x = %i, y = %i", x, y);  
}
```

Alocação Dinâmica

- Sempre que definimos uma variável, o compilador aloca espaço na memória para ela

```
int vet[100];
```

- este espaço de memória fica sendo **usado durante todo o programa**, mesmo se você não for mais utilizar as variáveis (com exceção das variáveis locais de funções auxiliares)
- Seria interessante ter o **poder de alocar quando usar uma variável/estrutura e desalocar quando não precisar mais**, assim otimizamos o uso da memória.



*Com grandes
poderes
vêm grandes
responsabilidades*



Alocação Dinâmica

- Alocação de Memória: Comando `malloc`
 - Aloca um espaço contínuo de memória.

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

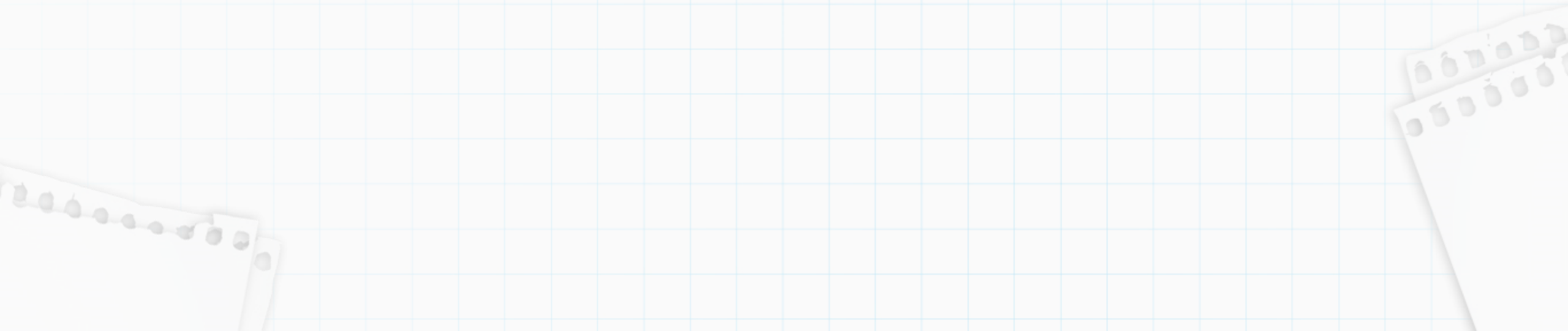
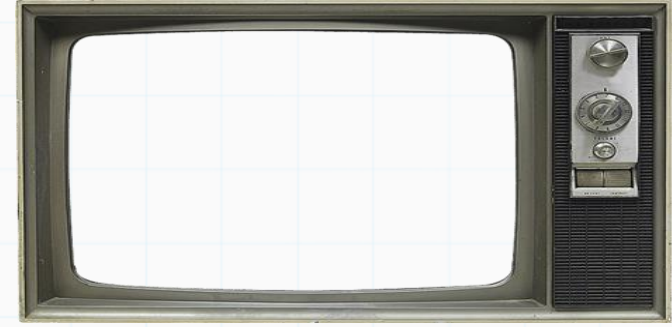
```
int *vet;
```

```
// aloca vetor de 10 inteiros
```

```
vet = (int*) malloc( 10 * sizeof(int));
```

`sizeof()` retorna o tamanho em bytes de um determinado tipo

o retorno de `malloc` é sem tipo (`void`) então temos que converter para o tipo específico



Alocação Dinâmica

- Alocação de Memória: Comando `malloc`
 - Aloca um espaço contínuo de memória.

```
#include<stdlib.h>

int *vet;

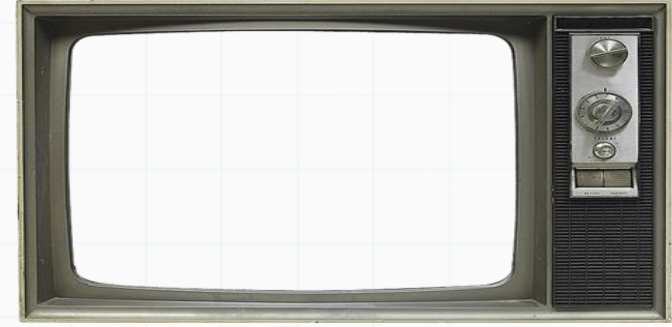
// aloca vetor de 10 inteiros
vet = (int*) malloc( 10 * sizeof(int));

if (vet == NULL)
    printf(" Não tem espaço" );
```

`sizeof()` retorna o tamanho em bytes de um determinado tipo

o retorno de `malloc` é sem tipo (`void`) então temos que converter para o tipo específico

caso não exista espaço em memória, `malloc` retorna `NULL`



Alocação Dinâmica

- Alocação de Memória: Comando `malloc`
 - Aloca um espaço contínuo de memória.

```
#include<stdlib.h>

int *vet;

// aloca vetor de 10 inteiros
vet = (int*) malloc( 10 * sizeof(int));

if (vet == NULL)
    printf(" Não tem espaço" );

....

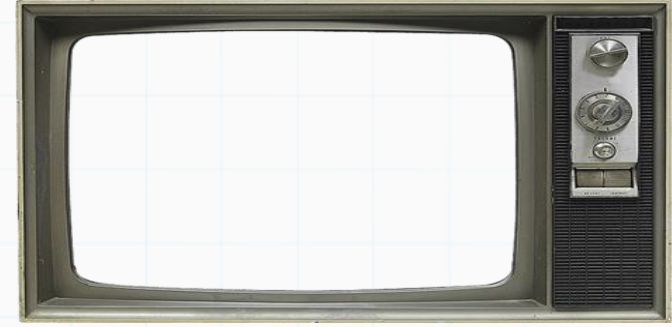
free(vet);
```

`sizeof()` retorna o tamanho em bytes de um determinado tipo

o retorno de `malloc` é sem tipo (`void`) então temos que converter para o tipo específico

caso não exista espaço em memória, `malloc` retorna `NULL`

após o uso, devemos SEMPRE liberar memória com o comando `free()`



Alocação Dinâmica

- Alocação de Memória: Comando `malloc`
 - Aloca um espaço contínuo de memória.

```
#include<stdlib.h>

int *vet;

// aloca vetor de 10 inteiros
vet = (int*) malloc( 10 * sizeof(int));

if (vet == NULL)
    printf(" Não tem espaço" );

....

free(vet);
```

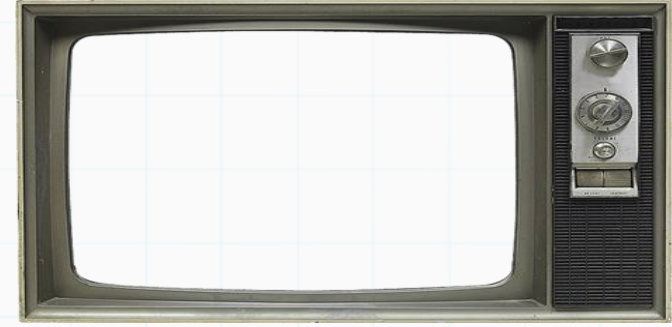
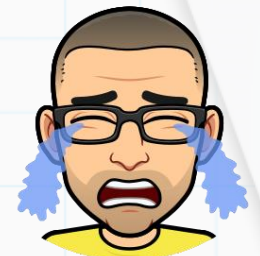
`sizeof()` retorna o tamanho em bytes de um determinado tipo

o retorno de `malloc` é sem tipo (`void`) então temos que converter para o tipo específico

caso não exista espaço em memória, `malloc` retorna `NULL`

após o uso, devemos SEMPRE liberar memória com o comando `free()`

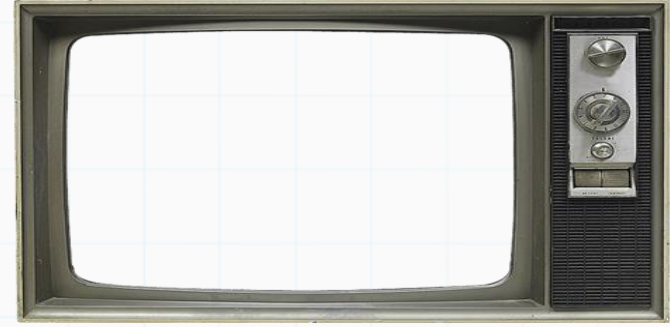
Vazamento de memória: ocorre quando não liberamos a memória (`free`) e perdemos o ponteiro para aquela memória alocada (fazemos o ponteiro apontar para outra coisa). Fica um LIXO na memória que se acumulado vai gerar dados corrompidos.



Ponteiros

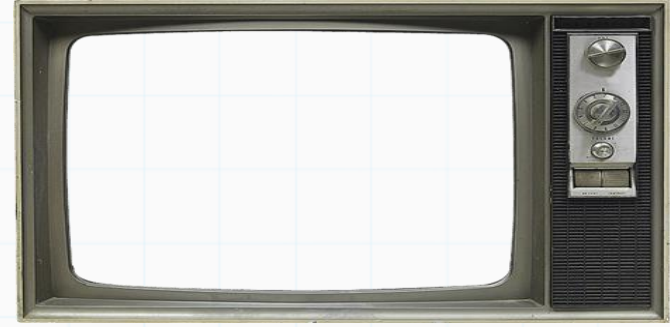
- Passagem: Quando o vetor é alocado dinamicamente, podemos passar para a função apenas seus ponteiros.

```
int main(){  
    int *x, *y;  
    x = (int*) malloc(10* sizeof(int));  
    y = (int*) malloc(10* sizeof(int));  
    troca(x,y,10);  
    ...  
}
```



Ponteiros

- Passagem: Quando o vetor é alocado dinamicamente, podemos passar para a função apenas seus ponteiros.



```
int main(){
    int *x, *y;
    x = (int*) malloc(10* sizeof(int));
    y = (int*) malloc(10* sizeof(int));
    troca(x,y,10);
    ...
}
```

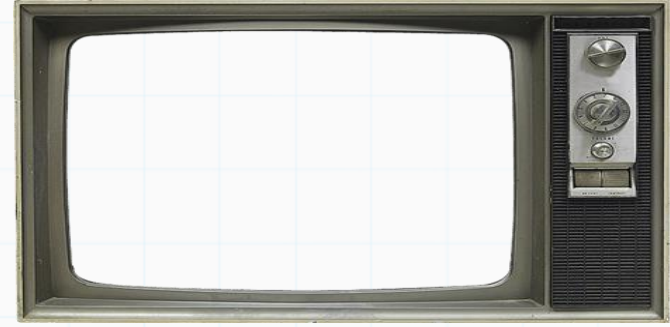
```
void troca(int *x, int *y, int n){
    for (int temp,i=0; i<n; i++)
    {
        temp = x[i];
        x[i] = y[i];
        y[i] = temp;
    }
}
```

troca 2 vetores

passando apenas o ponteiro não precisamos definir seu tamanho

Ponteiros

- Passagem: Reparem que a passagem de vetores e a passagem por referencia é igual, pois na verdade estamos passando um ponteiro para a primeira posição do vetor. O que é diferente é o conteúdo (um é um vetor e o outro é um inteiro)



```
void troca(int *x, int *y, int n){  
    for (int temp,i=0; i<n; i++)  
    {  
        temp = x[i];  
        x[i] = y[i];  
        y[i] = temp;  
    }  
}
```

troca 2 vetores

```
void troca(int *x,int *y){  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

troca 2 números

tem que saber na origem o que você está passando

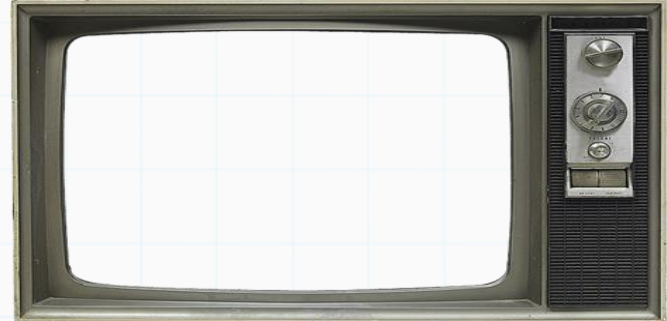


Alocação Dinâmica

Exemplo: Média de elementos do vetor

```
int main ( ) {  
  
    int i, n;  
    float *vetor;  
  
    /* leitura do número de valores */  
    printf("Tamanho do vetor:");  
    scanf("%d", &n);  
  
    /* alocação dinâmica */  
    vetor = (float*) malloc(n*sizeof(float));  
  
    /* leitura dos valores */  
    for (i = 0; i < n; i++)  
        scanf("%f", &vetor[i]);  
  
    printf("media = %f",media(vetor, n));  
  
    /* libera memória */  
    free(vetor);  
    return 0;  
}
```

```
float media(float *v, int l){  
    float m=0;  
    for (int i = 0; i < l; i++)  
        m += v[i];  
    m = m/l;  
}
```



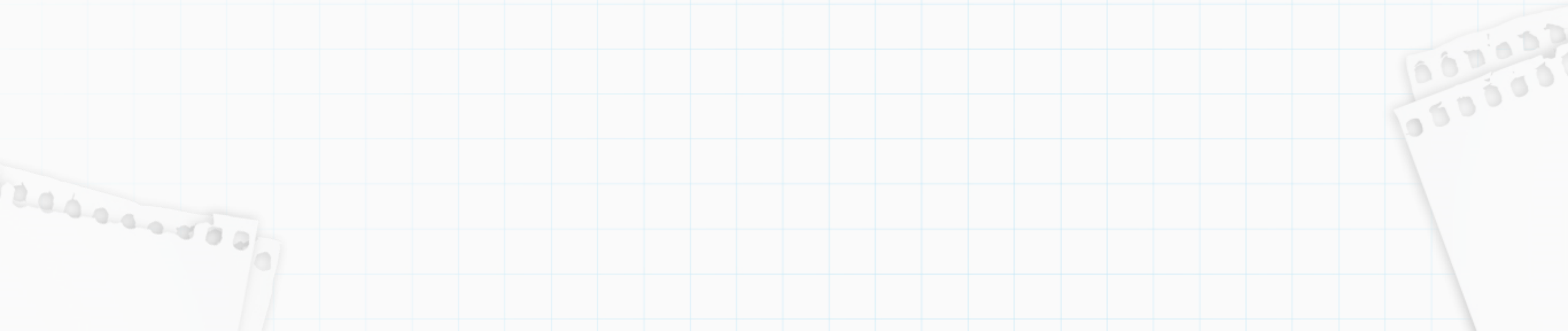
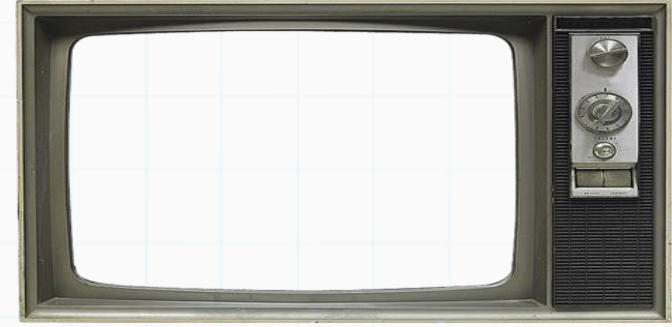
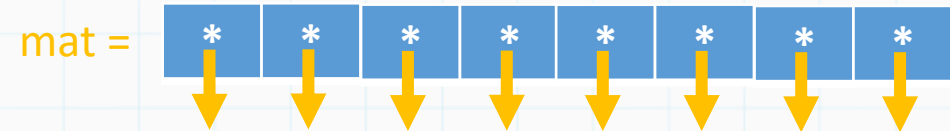
Matrizes Dinâmica

```
int l=2,c=3;
```

```
float **mat;      ponteiro para ponteiro de float
```

```
mat = (float**) malloc(l*sizeof(float*));
```

Aloca vetor de
ponteiros para float
de tamanho l



Matrizes Dinâmica

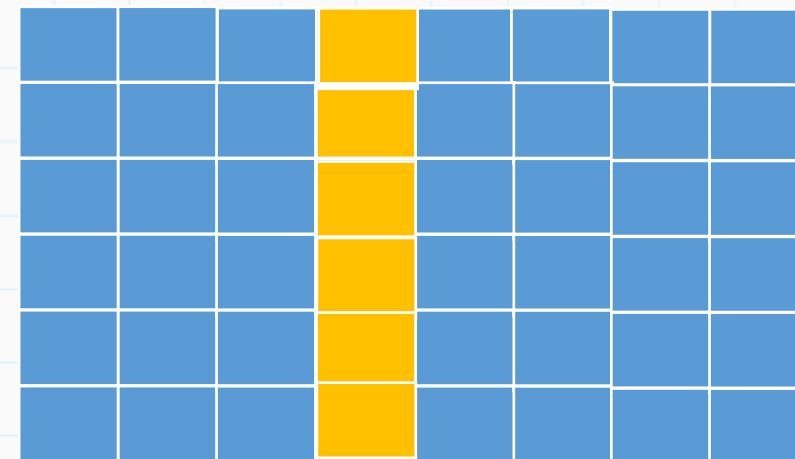
```
int l=2,c=3;  
float **mat;    ponteiro para ponteiro de float
```

```
mat = (float**) malloc(l*sizeof(float*));
```

```
for (int i=0; i<l; i++)  
    mat[i] = (float*) malloc(c*sizeof(float));
```

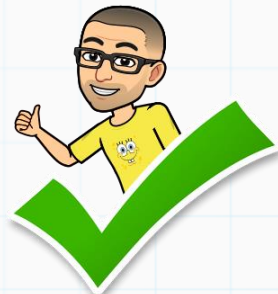
Para cada posição do vetor, aloca um vetor de c floats

mat[i]

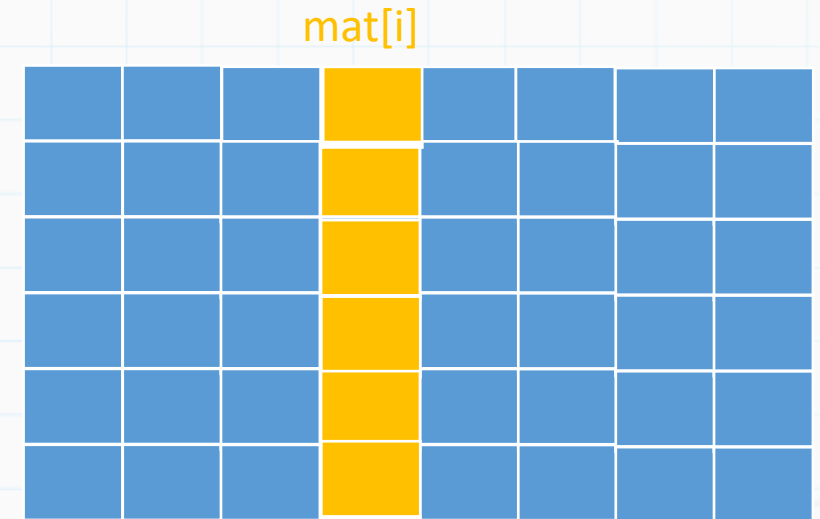
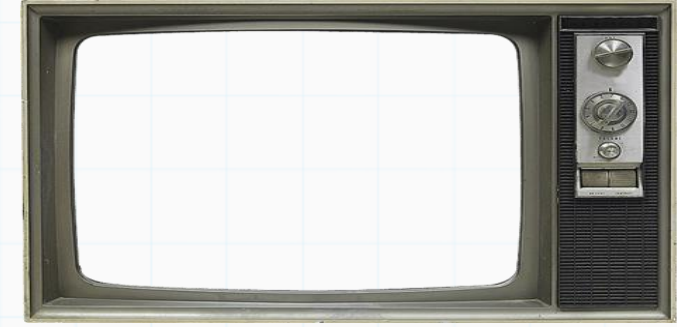


Matrizes Dinâmica

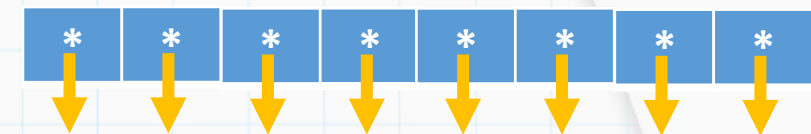
```
int l=2,c=3;  
float **mat;    ponteiro para ponteiro de float  
  
mat = (float**) malloc(l*sizeof(float*));  
  
for (int i=0; i<l; i++)  
    mat[i] = (float*) malloc(c*sizeof(float));  
  
....  
  
....  
  
....  
  
for (int i=0; i<l; i++)  
    free(mat[i]);  
free(mat);
```



**Para liberar a
memoria da matriz,
liberamos cada um
dos vetores mat[i]
para só depois
liberarmos o vetor
mat**



mat =



Matrizes Dinâmica



```
float **mat;

mat = (float**) malloc(l*sizeof(float*));

for (int i=0; i<l; i++)
    mat[i] = (float*) malloc(c*sizeof(float));

imprime_mat(mat, l, c);
```

Mas como passar matrizes para funções ?

passando apenas o
ponteiro não precisamos
definir seu tamanho

```
void imprime_mat(float **mat, int n, int m)
{
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<m; j++)
            printf("%.2f ", mat[i][j]);
        printf("\n");
    }
}
```

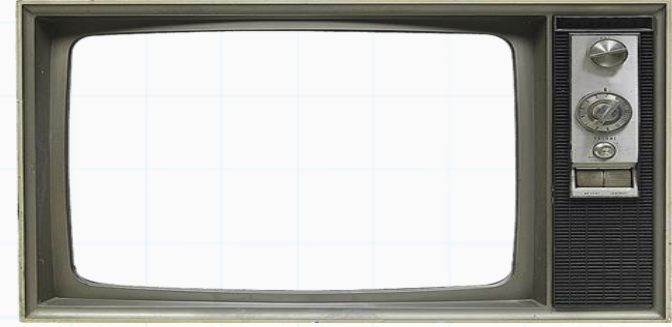


Matrizes Dinâmica

Podemos também alocar dentro de funções e retornar seus ponteiros

```
int main () {  
    float *p;  
    int a;  
    p = alocar_vetor_real (10);  
    return 0;  
}
```

```
float * alocar_vetor_real (int n) {  
    float *v;  
    v = (float *) malloc (n*sizeof(float));  
    return v;  
}
```



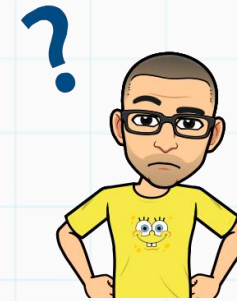
Exercicios

1) Maior elemento: Dado uma vetor de tamanho n, encontre e imprima o seu maior elemento.



OBS

- Alocar o vetor de forma dinâmica.
- Faça uso de uma função recursiva para encontrar o máximo do vetor:
- Não esqueça de liberar a memória



Fazer programa principal e função (por partes)

```
int *vet;
```

```
// aloca vetor de 10 inteiros  
vet = (int*) malloc( 10 * sizeof(int));
```

```
int main()
{
    int i, n;
    int * vetor;

    printf("n:");
    scanf("%d", &n);

    vetor = (int*) malloc( n * sizeof(int));

    for(i=0; i<n; i++)
    {
        printf("v[%d]=", i);
        scanf("%d", &vetor[i]);
    }
    printf("maior elemento = %d", maximo(vetor, 0, n));

    free(vetor);
    return 0;
}
```

```
int maximo(int *vetor, int i, int n)
{
    //caso base
    if (i==n-1)
        return vetor[i];

    int maior = maximo(vetor, i+1, n);
    if (vetor[i] > maior)
        return vetor[i];
    else
        return maior;
}
```

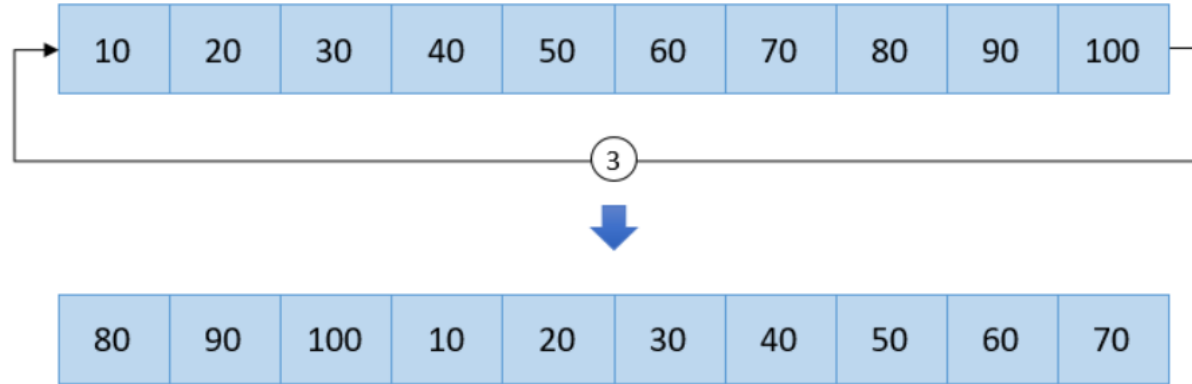
Largest Element in Array



Exercícios

2) Rotacionar vetor: Dado um vetor de inteiros de tamanho n , receba valor inteiro k rotacione k vezes para direita o vetor.

Exemplo: $n=10$, $k=3$



OBS

- Você tem que alocar o vetor de forma dinâmica, e só pode alocar um vetor (nada de vetor auxiliar)
- Implemente e faça uso da função:

```
void rota_1_direita(int* vetor, int n)
```

que vai rotacionar 1 elemento do vetor para direita (logo você vai chamar ela k vezes na função main)

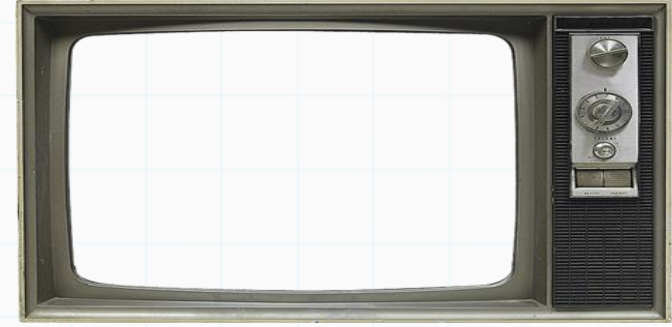
- não esqueça de liberar a memória

lembrando

```
int *vet;
```

```
// aloca vetor de 10 inteiros  
vet = (int*) malloc( 10 * sizeof(int));
```

Fazer programa principal e função (por partes)



```

int main()
{
    int i, n, k;
    int * vetor;

    printf("n:");
    scanf("%d", &n);
    printf("k:");
    scanf("%d", &k);

    vetor = (int*) malloc( n * sizeof(int));

    for(i=0; i<n; i++)
    {
        printf("v[%d]= ", i);
        scanf("%d", &vetor[i]);
    }

    for(i=1; i<=k; i++)
        rota_1_direita(vetor, n);

    for(i=0; i<n; i++)
        printf("%d, ", vetor[i]);
    free(vetor);
    return 0;
}

```



```

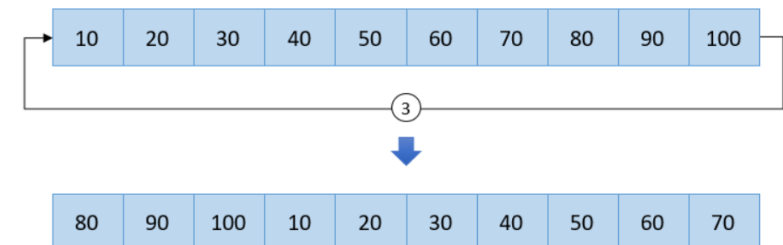
void rota_1_direita(int *vetor, int n)
{
    int i, ultimo;

    ultimo = vetor[n-1];

    for(i=n-1; i>0; i--)
        vetor[i] = vetor[i - 1];

    vetor[0] = ultimo;
}

```



Exercícios

3) Matrizes Simétricas: Dado uma matriz M de inteiros de tamanho n,m; diga se ela é simétrica ou não.

Uma matriz é simétrica quando ela é igual a sua transposta, isto é, o elemento que está na linha l coluna c, tem que ser igual ao elemento que esta na linha c coluna l.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix}^T$$

Fazer programa principal e função (por partes)

OBS

- Alocar a matriz de forma dinâmica.
- Faça uso de uma função:

```
int simetrica(int** mat, int n, int m)
```

para saber se é simétrica ou não

- Não esqueça de liberar a memoria

lembrando

```
float **mat;
```

```
mat = (float**) malloc(l*sizeof(float*));
```

```
for (int i=0; i<l; i++)
```

```
mat[i] = (float*) malloc(c*sizeof(float));
```



```

int main()
{
    int i, j, n, m;

    printf("n:");
    scanf("%d", &n);
    printf("m:");
    scanf("%d", &m);
    if (n != m)
    {
        printf("nao eh simetrica");
        return 0;
    }

    float **mat;
    mat = (float**) malloc(n*sizeof(float*));
    for (i=0; i<n; i++)
        mat[i] = (float*) malloc(m*sizeof(float));

    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
        {
            printf("m[%d][%d]=", i, j);
            scanf("%d", &mat[i][j]);
        }

    if (simetrica(mat, n, m) == 1)
        printf("eh simetrica");
    else
        printf("nao eh simetrica");
}

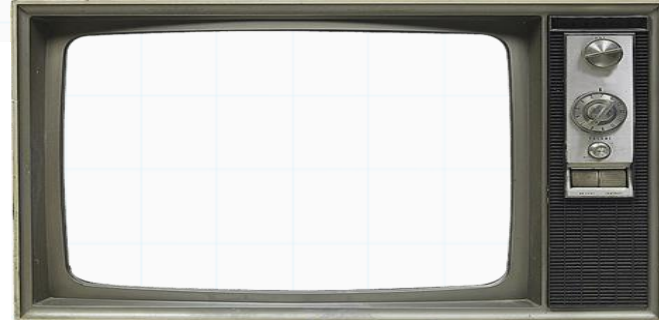
```

```

for (int i=0; i<n; i++)
    free(mat[i]);
free(mat);

return 0;
}

```



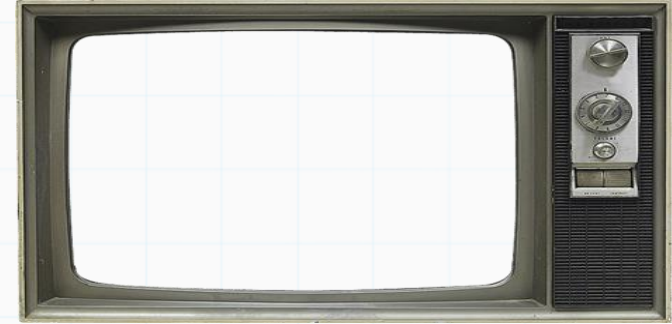
```

int simetrica(int ** mat, int n, int m)
{
    int verdade = 1;
    for(int i=0; i<n && verdade; i++)
        for(int j=0; j<m; j++)
            if(mat[i][j] != mat[j][i])
            {
                verdade = 0;
                break;
            }
    return verdade;
}

```

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 8 \end{bmatrix}^T$$

Até a próxima



Slides baseados no curso de Aline Nascimento

