

# DevTitans

Alocação Dinâmica

# Alocação estática de memória

- Armazenamento de dados deve manter três propriedades: (i) **onde** a informação é armazenada; (ii) qual o **valor**; e (iii) qual o **tipo** de dado da informação

```
int v = 3;
```

- A declaração provê o **tipo** e um **nome** simbólico para o valor; e também faz com que o programa **aloque memória** para o valor e mantenha essa alocação internamente

# Alocação dinâmica de memória

A **alocação dinâmica** de memória se dá via **ponteiros**. Um ponteiro é uma variável que armazena o **endereço** (ou a **localização**) em vez do **valor** de uma variável.

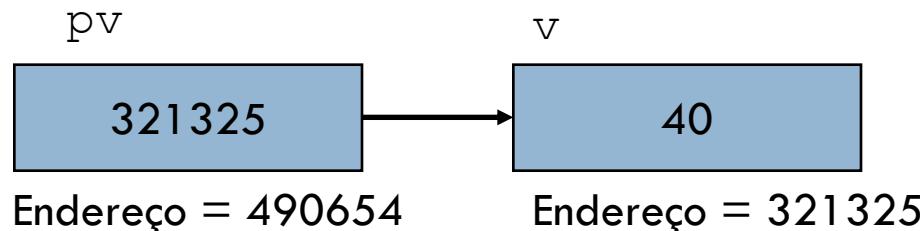
```
int main() {
    char *pc;
    double *pd;
    printf("Valores : pc=%p pd=%p\n", pc, pd);
    printf("Tamanhos: pc=%lu pd=%lu\n",
           sizeof(pc), sizeof(pd));
}
```

Valores : pc=0x108ec5025 pd=0x7ffee73828f8  
Tamanhos: pc=8 pd=8

# Operador de endereço (&)

Atribui o endereço da variável v a uma outra variável ponteiro pv

```
int v, *pv;  
v = 40;  
pv = &v;  
printf ("%d %p %p\n", v, pv, &pv);
```



Importante! A variável v armazena um valor; a variável ponteiro pv armazena um endereço de outra variável

# Operador de dereferenciação (\*)

Se **pv** for um ponteiro, a expressão **\*pv** representa o valor armazenado no endereço "apontado" por **pv**

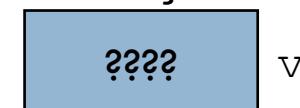
Qual seria o valor de **v**?

```
main() {  
    int u=3;  
    int v;  
    int *pu;  
    pu = &u;  
    v = *pu;  
    printf("v=%d\n", v);  
}
```

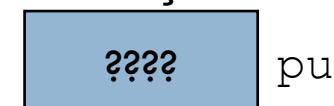
Endereço = 490654



Endereço = 490849



Endereço = 492251



# Operador de dereferenciação (\*)

Se **pv** for um ponteiro, a expressão **\*pv** representa o valor armazenado no endereço "apontado" por **pv**

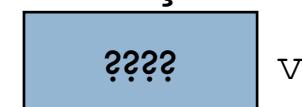
Qual seria o valor de **v**?

```
main() {  
    int u=3;  
    int v;  
    int *pu;  
    pu = &u;  
    v = *pu;  
    printf("v=%d\n", v);  
}
```

Endereço = 490654



Endereço = 490849



Endereço = 492251



# Operador de dereferenciação (\*)

Se **pv** for um ponteiro, a expressão **\*pv** representa o valor armazenado no endereço "apontado" por **pv**

Qual seria o valor de **v**?

```
main() {  
    int u=3;  
    int v;  
    int *pu;  
    pu = &u;  
    v = *pu;  
    printf("v=%d\n", v);  
}
```

Endereço = 490654

3

u

Endereço = 490849

3

v

Endereço = 492251

490654

pu

# Cuidado!

Analise o código abaixo

```
int *a;  
*a = 22;
```

Onde o valor 22 é colocado? **Não se sabe**, uma vez que a variável ponteiro **a** não foi inicializada (aponta para lixo)

O correto seria

```
int *a, b;  
a = &b;  
*a = 22;
```

Agora o valor **22** é acessado tanto por **b** quanto por **\*a**

# Alocando memória com malloc

## :: Exemplo 1

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *pi;
    pi=(int *) malloc(sizeof(int));
    *pi = 1001;
    printf("valor = %d\nendereco = %p\n
        tamanho: %lu\n", *pi, pi, sizeof(*pi));
}
```

```
valor = 1001
endereco = 0x7fd91bd05870
tamanho: 4
```

# Alocando memória com malloc

## :: Exemplo 2

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    double * pd;
    pd=(double *) malloc(sizeof(double));
    *pd = 1001.1001;
    printf("valor = %f\nendereco = %p\n
        tamanho: %lu\n", *pd, pd, sizeof(*pd));
}
```

```
valor = 1001.100100
endereco = 0x7fce37c05930
tamanho: 8
```

# Array Dinâmico

# Criando um array dinâmico

Basta dizer ao **malloc()** o tipo e o número de elementos que se deseja. Para gerar um array de 10 ints:

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

**malloc()** retorna o endereço do primeiro elemento do bloco de memória.

Para liberar após o uso, deve-se usar a função **free**

```
free(pvet);
```

# Criando um array dinâmico

```
int main() {  
    int *pvet, *aux;  
    pvet=(int *)malloc(10*sizeof(int));  
    aux = pvet;  
    for (int i=10; i>0;i--) {  
        *aux = i; aux++;  
    }  
    aux = pvet;  
    for (int i=0; i<10;i++) {  
        printf("%d ", *aux); aux++;  
    }  
    printf("\n"); free(pvet);  
}
```

10 9 8 7 6 5 4 3 2 1

# Acessando um array

Há diversas formas de acesso a valores de um array

```
main() {  
    int x[5] = {10,11,12,13,14};  
    int i;  
    for (i=0; i<5; i++)  
        printf("i=%d  x[i]=%d  *(x+i)=%d  &x[i]=%p  
               x+i=%p\n", i, x[i], *(x+i), &x[i], x+i);  
}
```

i=0	x[i]=10	* (x+i)=10	&x[i]=0xa8c0	x+i=0xa8c0
i=1	x[i]=11	* (x+i)=11	&x[i]=0xa8c4	x+i=0xa8c4
i=2	x[i]=12	* (x+i)=12	&x[i]=0xa8c8	x+i=0xa8c8
i=3	x[i]=13	* (x+i)=13	&x[i]=0xa8cc	x+i=0xa8cc
i=4	x[i]=14	* (x+i)=14	&x[i]=0xa8d0	x+i=0xa8d0

# Indireção Múltipla

Um ponteiro pode armazenar o endereço de outro ponteiro (indireção múltipla)

```
int ** ptr2;
int * ptr1;
int i = 10;

ptr2 = &ptr1;
ptr1 = &i;

*ptr1 = 30;
**ptr2 = 50;
```

# Indireção Múltipla

```
int ** ptr2;  
int * ptr1;  
int i = 10;
```

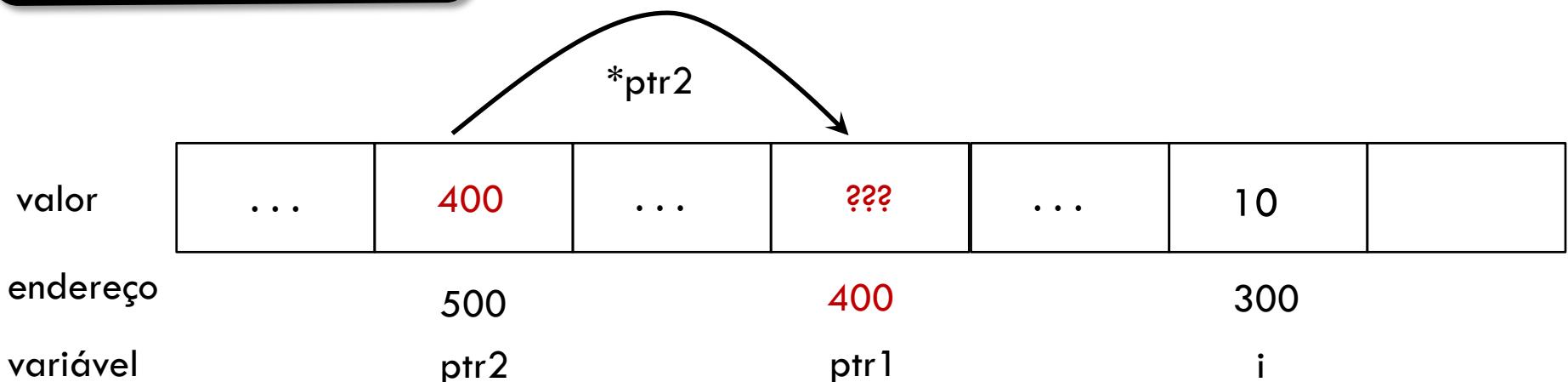
i=10

valor	...	???	...	???	...	10	
endereço		500		400		300	
variável		ptr2		ptr1		i	

# Indireção Múltipla

```
int ** ptr2;  
int * ptr1;  
int i = 10;  
  
ptr2 = &ptr1;
```

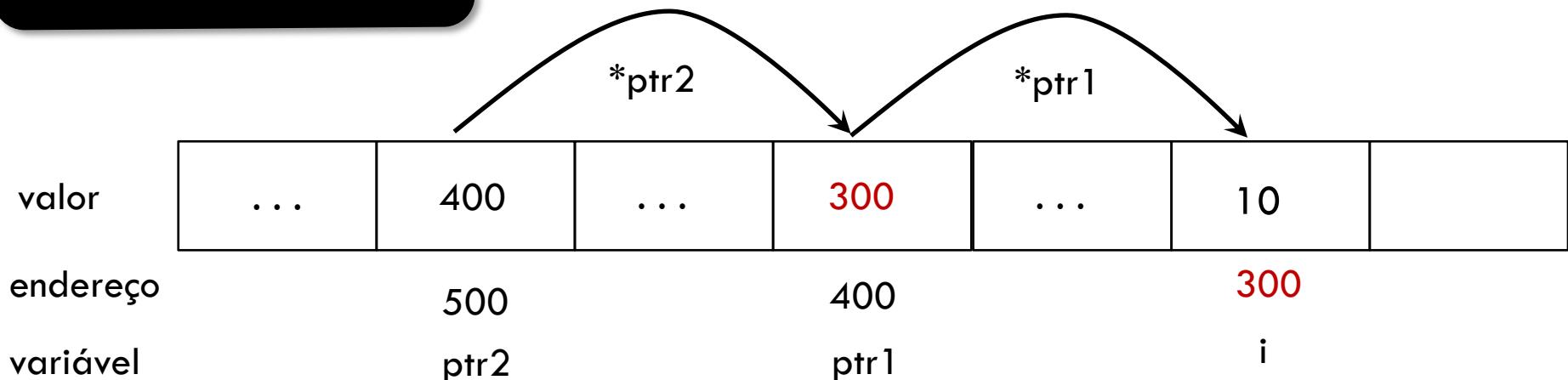
i=10



# Indireção Múltipla

```
int ** ptr2;  
int * ptr1;  
int i = 10;  
  
ptr2 = &ptr1;  
ptr1 = &i;
```

i=10

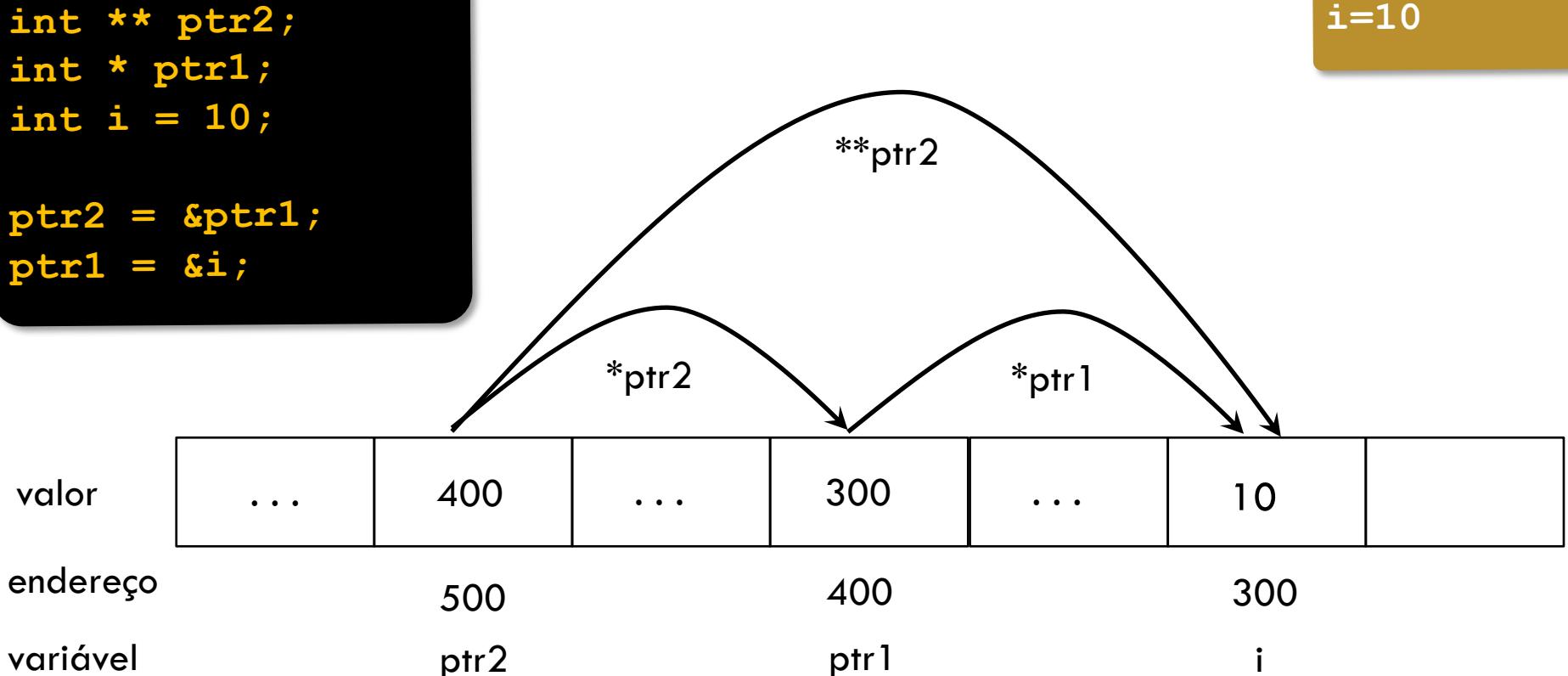


# Indireção Múltipla

```
int ** ptr2;
int * ptr1;
int i = 10;

ptr2 = &ptr1;
ptr1 = &i;
```

i=10



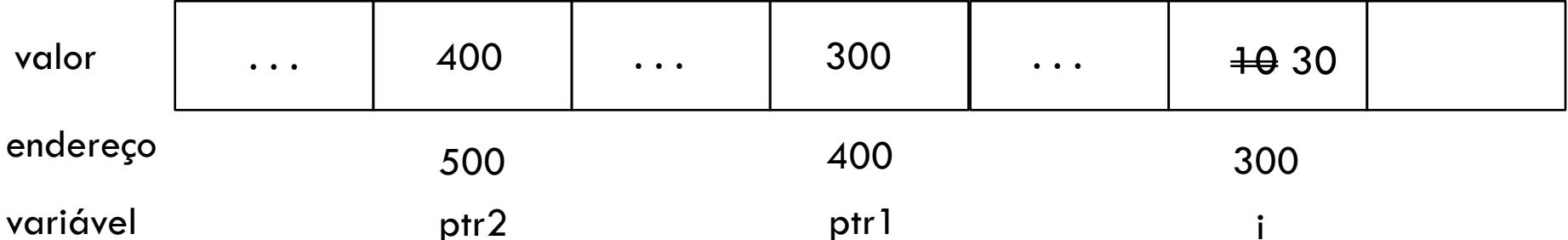
# Indireção Múltipla

```
int ** ptr2;
int * ptr1;
int i = 10;

ptr2 = &ptr1;
ptr1 = &i;

*ptr1 = 30;
```

i=10  
i=30



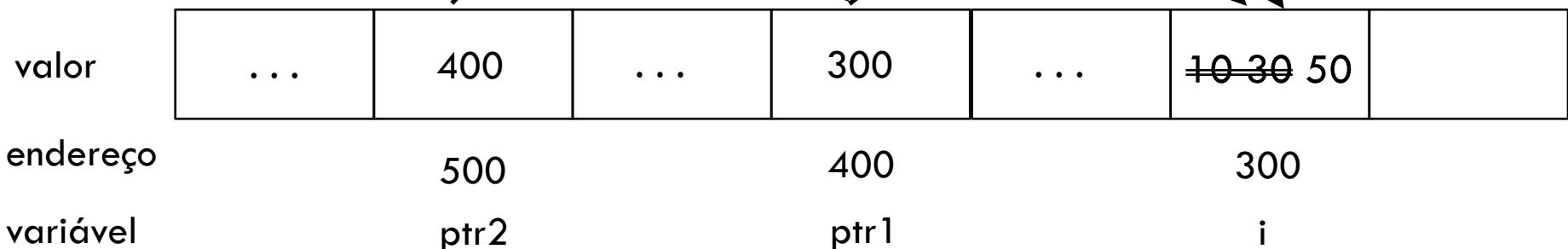
# Indireção Múltipla

```
int ** ptr2;
int * ptr1;
int i = 10;

ptr2 = &ptr1;
ptr1 = &i;

*ptr1 = 30;
**ptr2 = 50;
```

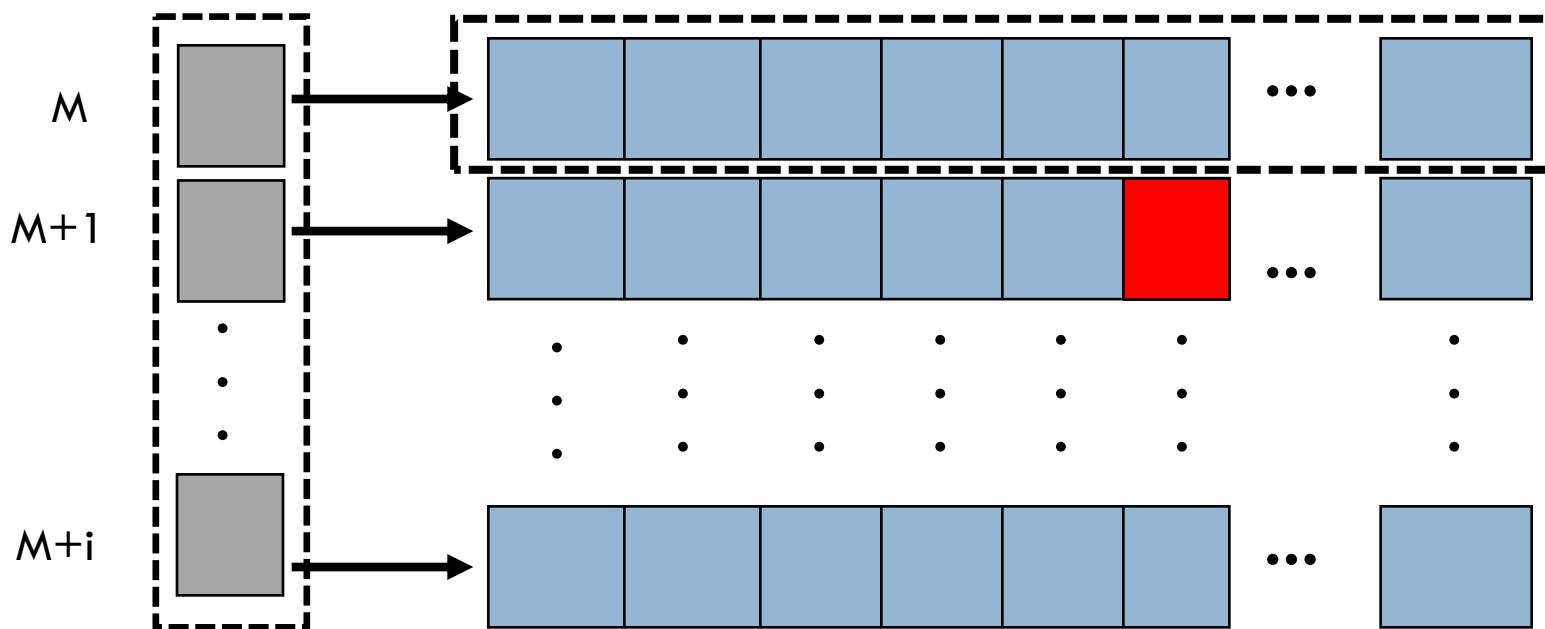
i=10  
i=30  
i=50



# Ponteiros e matrizes multidimensionais

```
#define LIN 3
#define COL 4
int main(void) {
    int **M, i, j;
    M = (int**) malloc (sizeof(int*) * LIN);
    for (i=0; i<COL; i++)
        M[i] = (int*) malloc (sizeof(int) * COL);
    for (i=0; i<LIN; i++)
        for (j=0; j<COL; j++) {
            printf("M[%d] [%d] = ", i, j);
            scanf("%d", &M[i][j]);
        }
    for (i=0; i<LIN; i++) {
        for (j=0; j<COL; j++)
            printf("%d\t", M[i][j]);
        printf("\n");
    }
    for (i=0; i<LIN; i++) free (M[i]);
    free (M);
}
```

# Ponteiros e matrizes multidimensionais



**Este elemento pode ser acessado por:  $*(*(\text{M+1}) + 5)$**

# Ponteiros e matrizes multidimensionais

```
M[0][0] = 1
M[0][1] = 2
M[0][2] = 3
M[0][3] = 4
M[1][0] = 5
M[1][1] = 6
M[1][2] = 7
M[1][3] = 8
M[2][0] = 9
M[2][1] = 10
M[2][2] = 11
M[2][3] = 12
1 2 3 4
5 6 7 8
9 10 11 12
```

# Ponteiros e matrizes multidimensionais

```
int main(void) {
    int LIN, COL;
    printf("Entre com o tamanho da matriz: ");
    scanf("%d %d", &LIN, &COL);
    int M[LIN][COL], i, j;
    for (i=0; i<LIN; i++)
        for (j=0; j<COL; j++) {
            printf("M[%d] [%d] = ", i, j);
            scanf("%d", &M[i][j]);
        }
    for (i=0; i<LIN; i++) {
        for (j=0; j<COL; j++)
            printf("%d\t", M[i][j]);
        printf("\n");
    }
}
```

# Ponteiros e matrizes multidimensionais

```
M[0][0] = 1  
M[0][1] = 2  
M[0][2] = 3  
M[0][3] = 4  
M[1][0] = 5  
M[1][1] = 6  
M[1][2] = 7  
M[1][3] = 8  
M[2][0] = 9  
M[2][1] = 10  
M[2][2] = 11  
M[2][3] = 12  
1 2 3 4  
5 6 7 8  
9 10 11 12
```

O resultado é o mesmo. A diferença é que neste último não é possível desalocar a memória previamente alocada