Aula 09:
- Ponteiros (parte 2)

Prof. João Henrique Kleinschmidt
Material elaborado pelo prof. Jesús P. Mena-Chalco

3Q-2018

Sobre funções (“uma ideia”)

Qual função é mais "eficiente"?

```c
int F1(int a, int b) {
    int i, t1, t2;

    t1 = a;
    t2 = b;

    a = t2;
    b = t1;

    for (i=a; i<b; i++)
        // ...
```

```c
int F2(int a, int b) {
    int i, t;

    t = a;

    a = b;
    b = t;

    for (i=a; i<b; i++)
        // ...
```

1995

2016

Qual função é mais "eficiente"?

```
int f1(int a, int k) {
    if (k==1)
        return a;
    else
        return a*f1(a, k1);
}
```

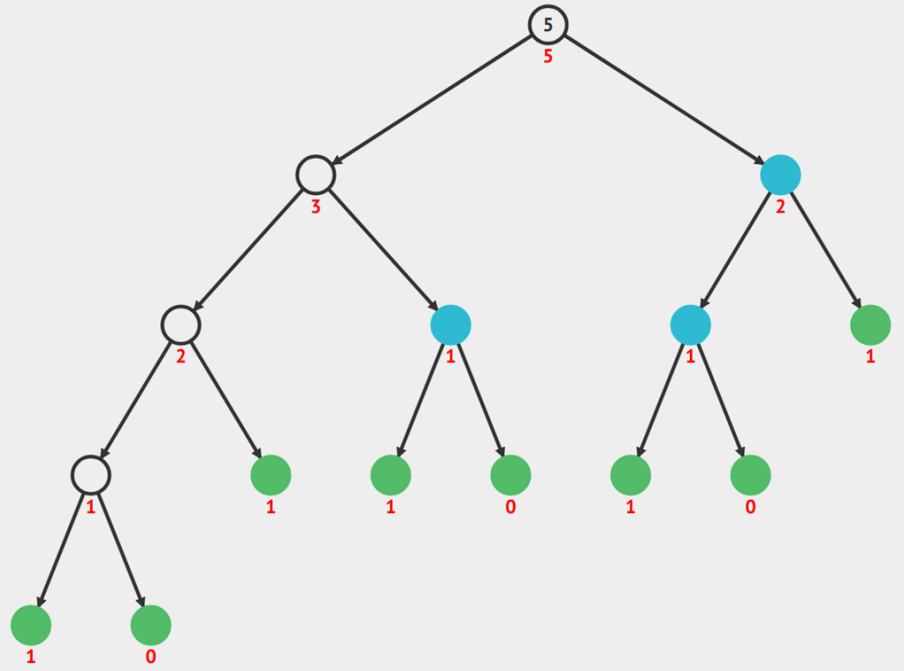
Número de multiplicações?
Proporcional a k

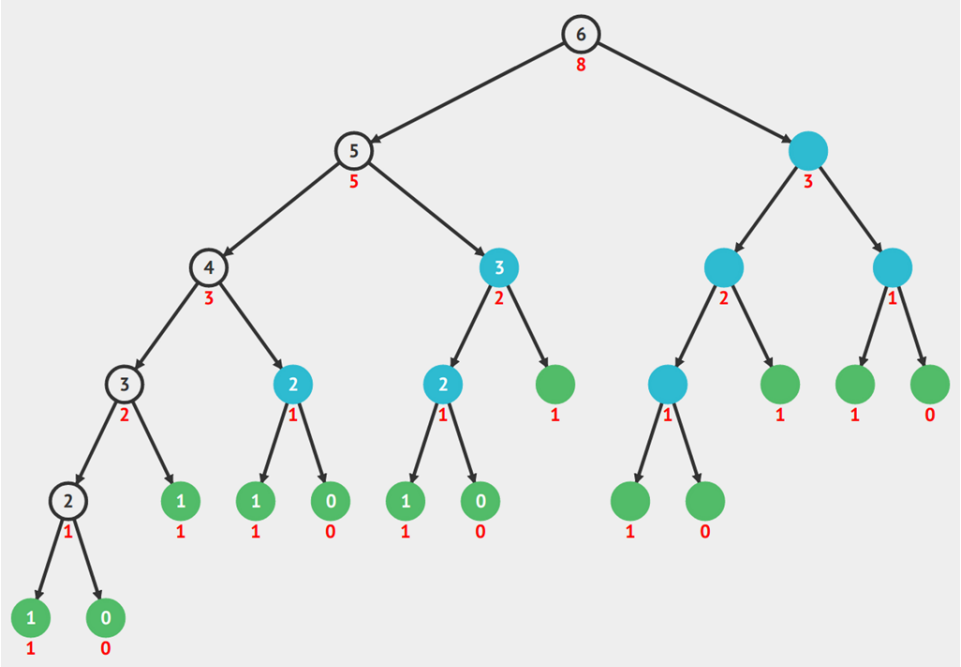
```
int f2(int a, int k) {
    int x;

    if (k==1)
        return a;
    else {
        x = f2(a, k/2);
        if (k%2==0)
            return x*x;
        else
            return x*x*a;
    }
}
```

Número de multiplicações?
Proporcional a log2(k)

Exercício de Fibonacci

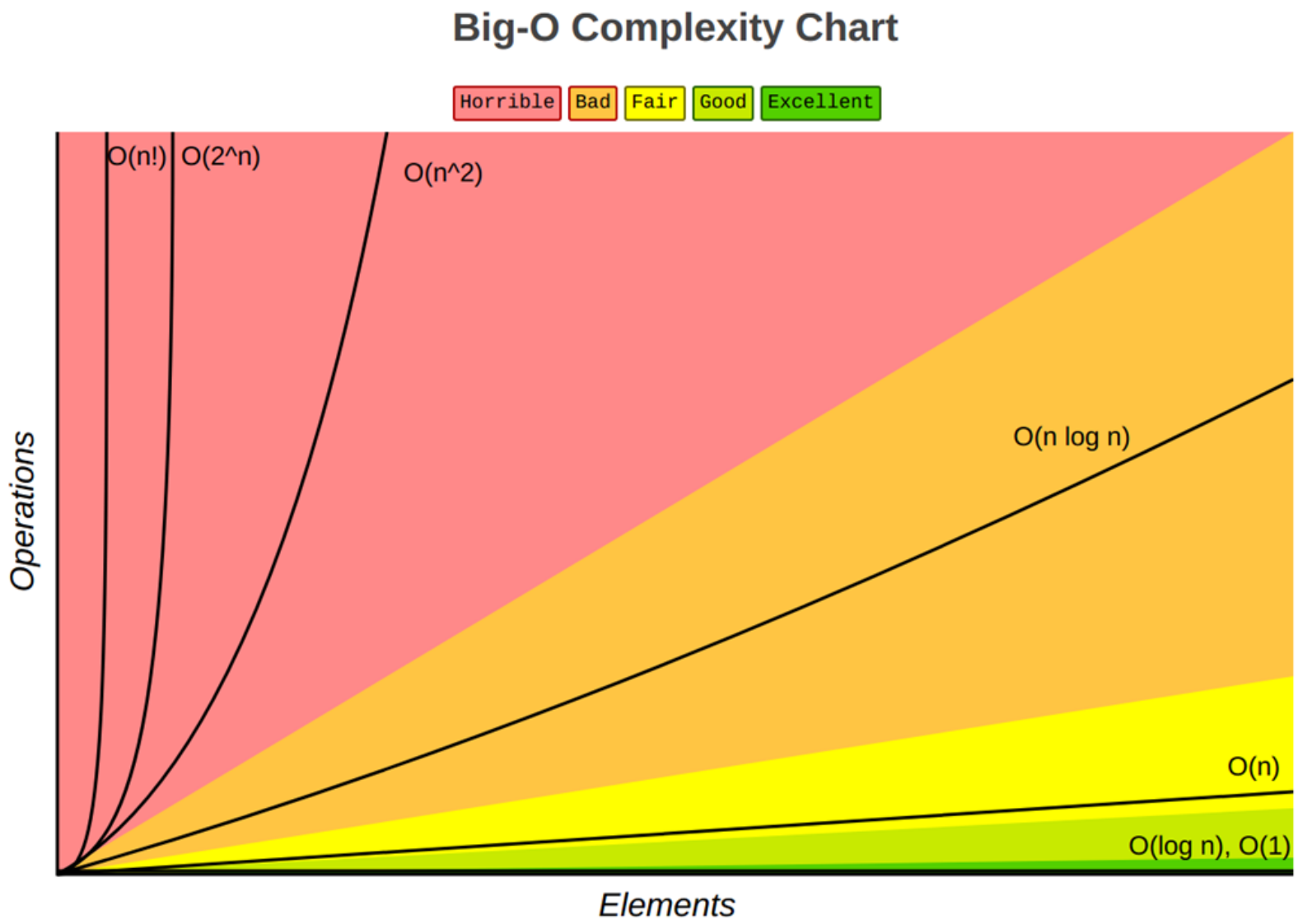




Número de chamados?
Proporcional a 2^n

https://visualgo.net/en/recursion

Qual função é mais "eficiente"?



http://bigocheatsheet.com/?utm_content=buffer0b573

Common Data Structure Operations

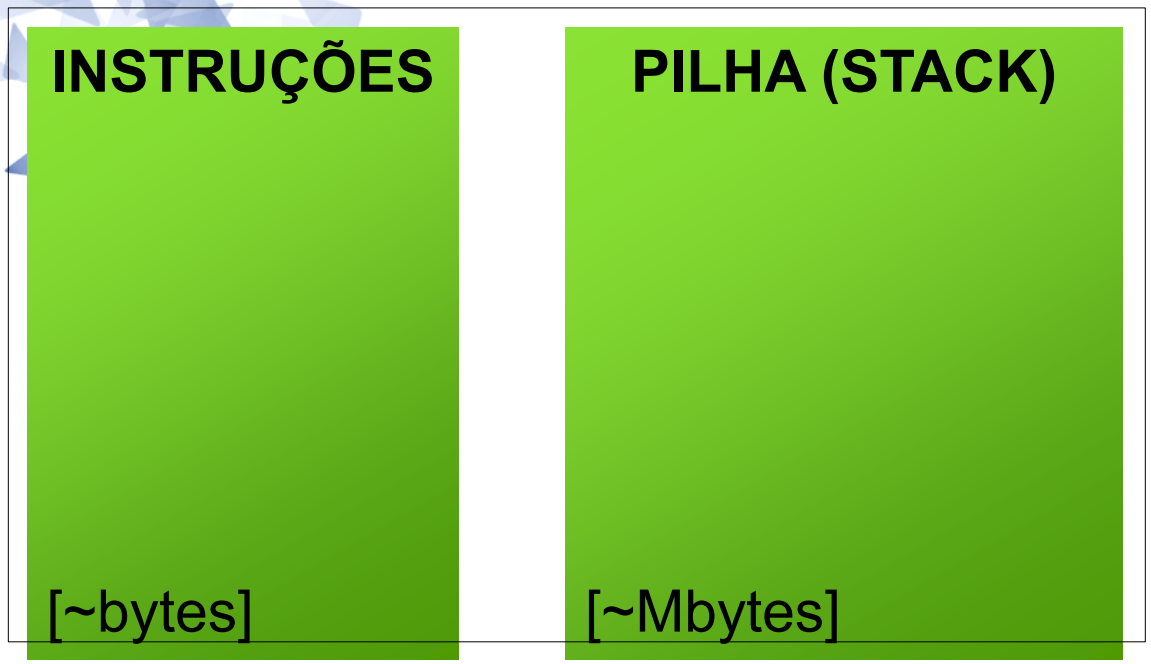
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Melhores momentos da aula anterior

Processo na memória



Alocação estática

HEAP

Espaço de memória principal gerenciado pelo SO.

[~Toda a memória RAM]

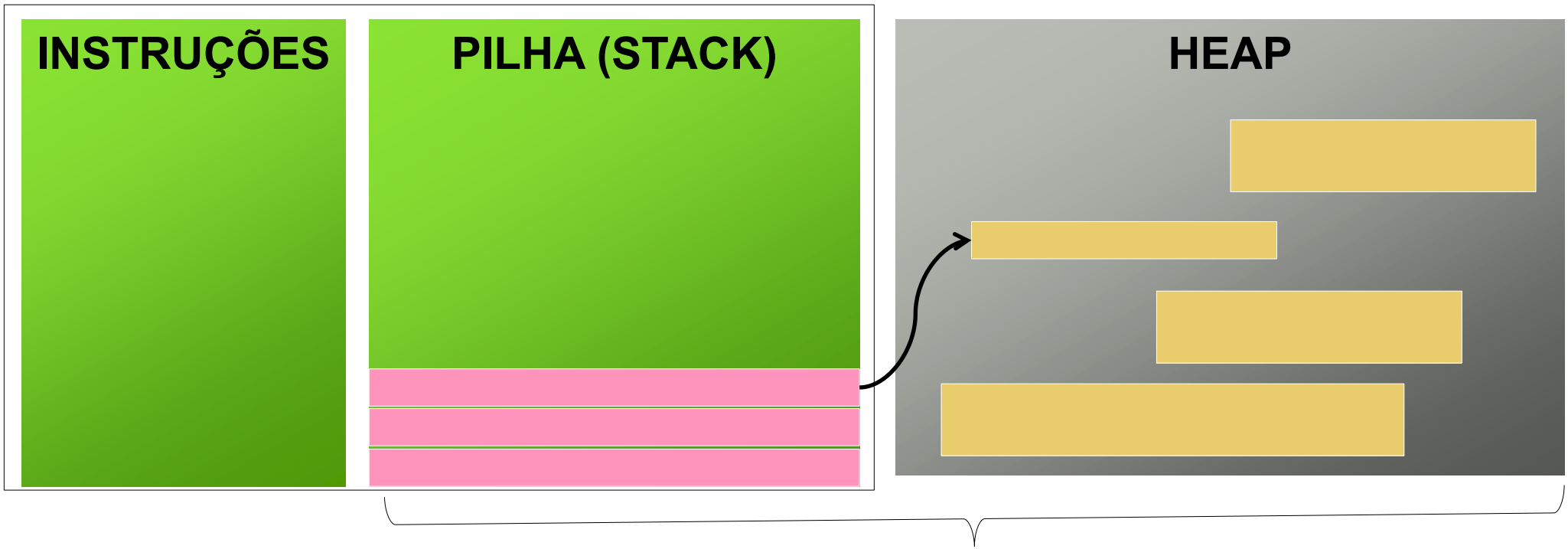
Alocação dinâmica

```
int x;
double M[10][20];
char *c;
```

```
double M = malloc(...);
```

Ponteiros?

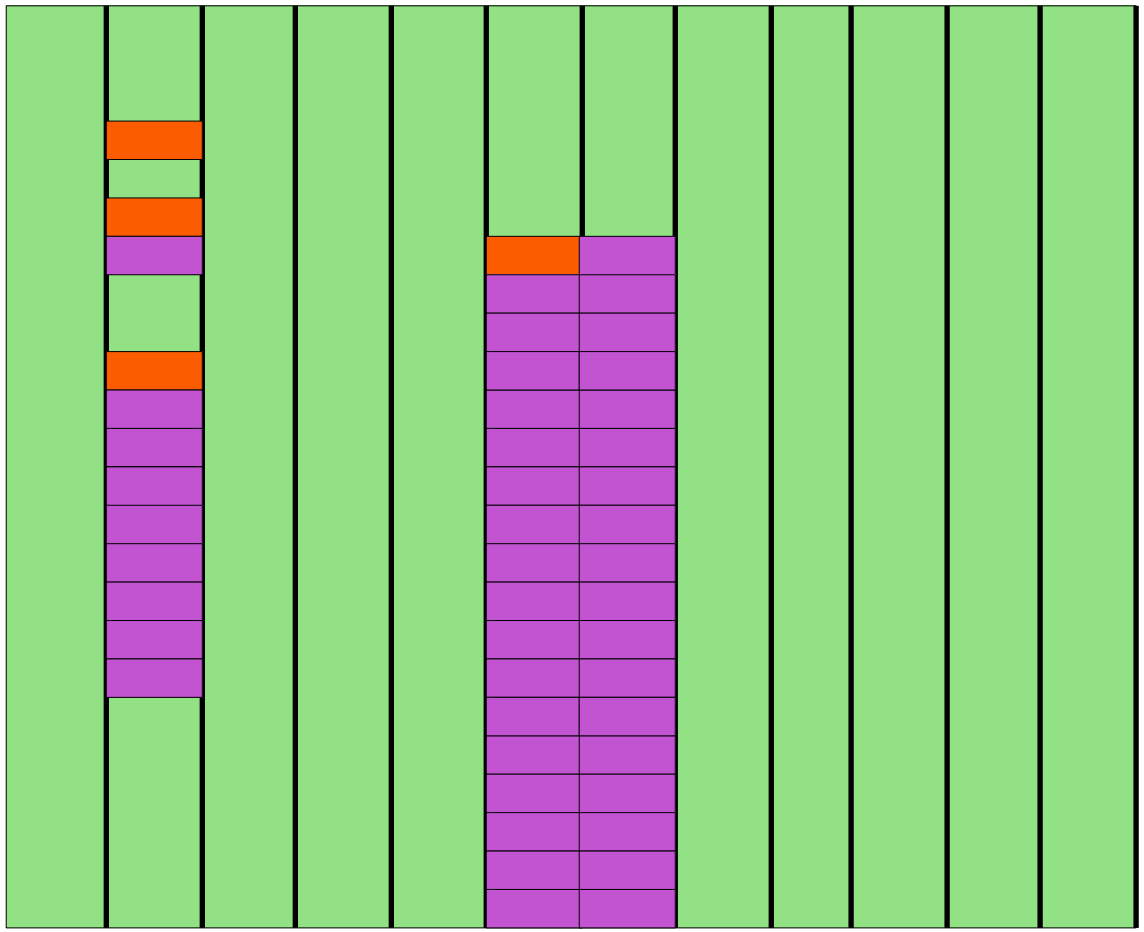
Processo na memória



Em Java e Python o uso é transparente. Não precisa se preocupar de alocar e liberar memória

Endereços

	...
37FD00	01010111
37FD01	11000011
37FD02	01100100
37FD03	11100010
	...



Geralmente o endereço do objeto é o endereço do 1ro byte.

Endereços

Há vários tipos de ponteiros:

- P. para caracteres
- P. para inteiros
- P. para registros
- P. para ponteiros para inteiros
- P. para função

```
int* p ;
```
← Um tipo de dado novo int* (conceitualmente correto)

```
int *p;
```
← O "*" modifica a variável e não o int (mais aceito)

```
int * p;
```

O compilador C aceita qualquer das formas.

exemploPonteiro.c

```c
#include<stdio.h>

int main() {
    int x;
    int i = 100;

    int *p;          /* p é um ponteiro para um inteiro */
    p = &i;          /* p aponta para i*/

    x = *p+900;      /* o mesmo que x = i+900 */

    printf("O valor de i   : %d\n", i);
    printf("O endereco de i: %p\n", &i);
    printf("O valor de p   : %p\n", p);
    printf("O valor de x   : %d\n", x);

}
```

```
O valor de i   : 100
O endereco de i: 0x7ffd987e5930
O valor de p   : 0x7ffd987e5930
O valor de x   : 1000
```

Operadores unários

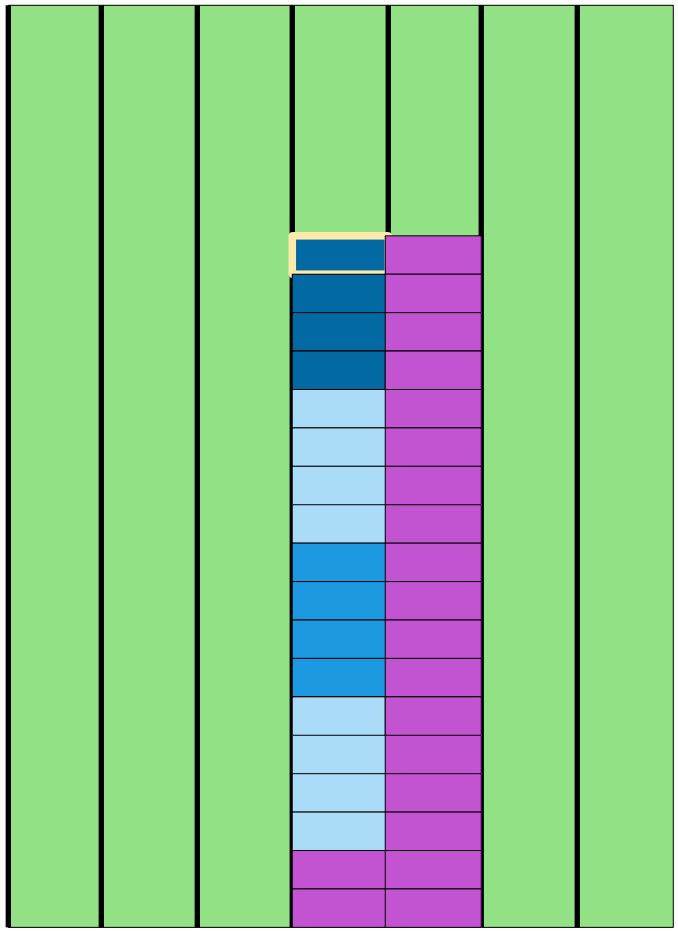
& → Referência: na frente de **uma variável**:
Devolve o endereço de memória onde a variável está armazenada

* → Derreferência: na frente de **variável ou expressão:**
Devolve o valor ou conteúdo do endereço de memória apontada pela variável ou expressão

Vetores e endereços

Vetores

Os elementos de um vetor são alocados consecutivamente na memória do computador.

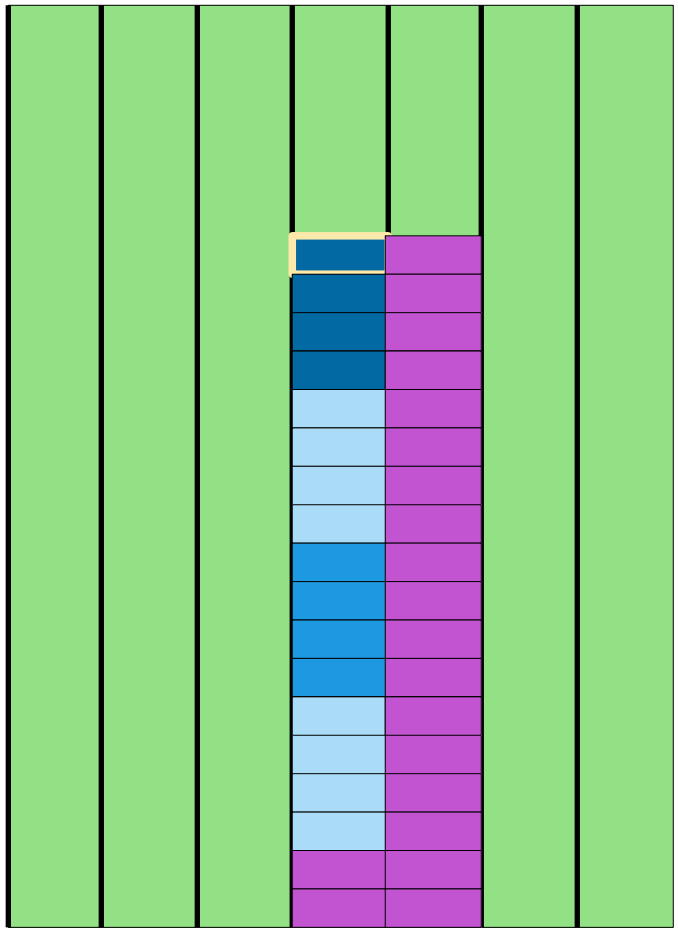


Se cada elemento ocupa **b** bytes, a diferença entre os endereços de dois elementos consecutivos será de **b**.

(ex. inteiros ocupam 4 bytes, em uma plataforma de 64 bits)

Vetores

Os elementos de um vetor são alocados consecutivamente na memória do computador.



Se cada elemento ocupa **b** bytes, a diferença entre os endereços de dois elementos consecutivos será de **b**.

O compilador C cria a ilusão de que b vale 1 qualquer que seja o tipo dos elementos do vetor.

Alocação de memória

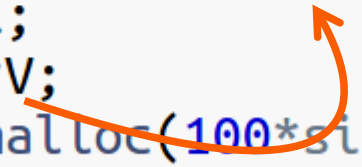
```c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int i;
    int *V;
    V = malloc(100*sizeof(int));

    V[0] = 10;
    V[1] = 20;
    V[2] = 30;

    for(i=0; i<5; i++)
        printf("%d\n", V+i);

}
```



Alocação de memória

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

int main() {
    int i;
    int *V;
    V = malloc(100*sizeof(int));

    V[0] = 10;
    V[1] = 20;
    V[2] = 30;

    for(i=0; i<5; i++)
        printf("%d\n", V+i);

}
```

Ideia
V+i*sizeof(int)

```
0x17af010
0x17af014
0x17af018
0x17af01c
0x17af020
```

Alocação memória

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

int main() {
    int i;
    int *V;
    V = malloc(100*sizeof(int));

    V[0] = 10;
    V[1] = 20;
    V[2] = 30;

    for(i=0; i<5; i++)
        printf("%d\n", *(V+i));   //  V[i]

}
```

```
10
20
30
0
0
```

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

int main() {

    int *V = malloc(100*sizeof(int));

    if(V==NULL) {
        printf("out of memory\n");
        return 1;
    }

    *(V+0) = 10;
    *(V+1) = 20;
    *(V+99) = 30;

    printf("%d %d %ld\n", V[0], V[99], &V[99]-V+1);

}
```

Quando não for possível
Separar memória suficiente
Um ponteiro NULO é devolvido

```
10 30 100
```

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

int main() {

    int *V = malloc(100*sizeof(int));

    if(V==NULL) {
        printf("out of memory\n");
        return 1;
    }

    *(V+0) = 10;
    *(V+1) = 20;
    *(V+99) = 30;

    printf("%d %d %ld\n", V[0], V[99], &V[99]-V+1);

}
```

```
10 30 100
```

Quando não for possível
Separar memoria suficiente
Um ponteiro nulo é devolvido

A diferença de ponteiros
Devolve um long e é
Permitida se os dois forem do
Mesmo tipo

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

int main() {

    int *V = malloc(100*sizeof(int));

    if(V==NULL) {
        printf("out of memory\n");
        return 1;
    }

    *(V+0) = 10;
    *(V+1) = 20;
    *(V+99) = 30;

    printf("%d %d %ld\n", V[0], V[99], &V[99]-V+1);

}
```

10 20 100

Quando não for possível Separar memoria suficiente Um ponteiro nulo é devolvido

Subtração de ponteiros: Retorna o número de elementos entre os dois ponteiros

A diferença de ponteiros Devolve um long e é Permitida se os dois forem do Mesmo tipo

Os ponteiros facilitam a **alocação dinâmica de memória**

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

int main() {
    int *v;
    int n, i;

    scanf( "%d", &n);

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

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

    for (i=n; i>0; i--)
        printf( "%d ", v[i-1]);

    free(v);
}
```

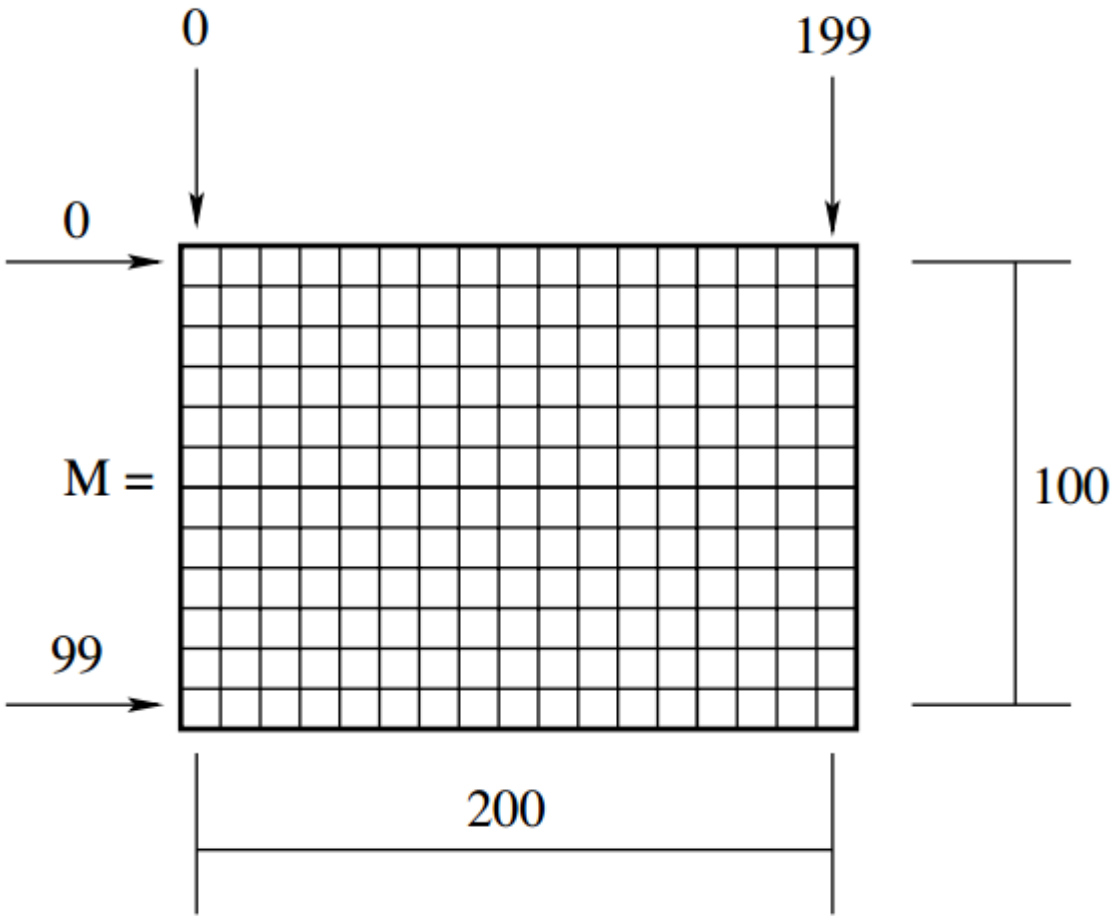
```
6
1
2
3
4
5
6
6 5 4 3 2 1
```

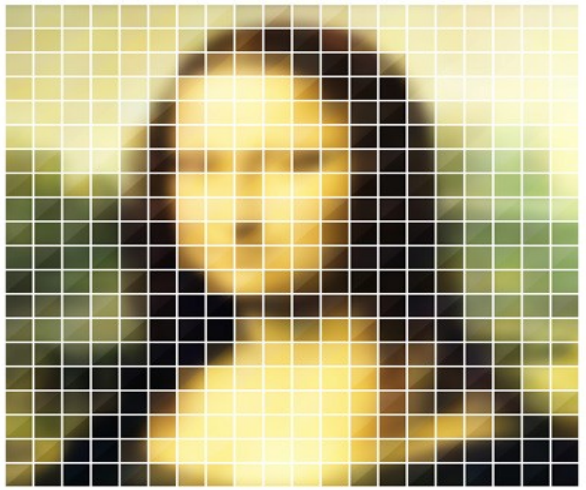
Matrizes

Material adaptado da aula de Matrizes de Ronaldo F. Hashimoto e Carlos H. Morimoto (IME/USP)

```
int M[100][200];
```

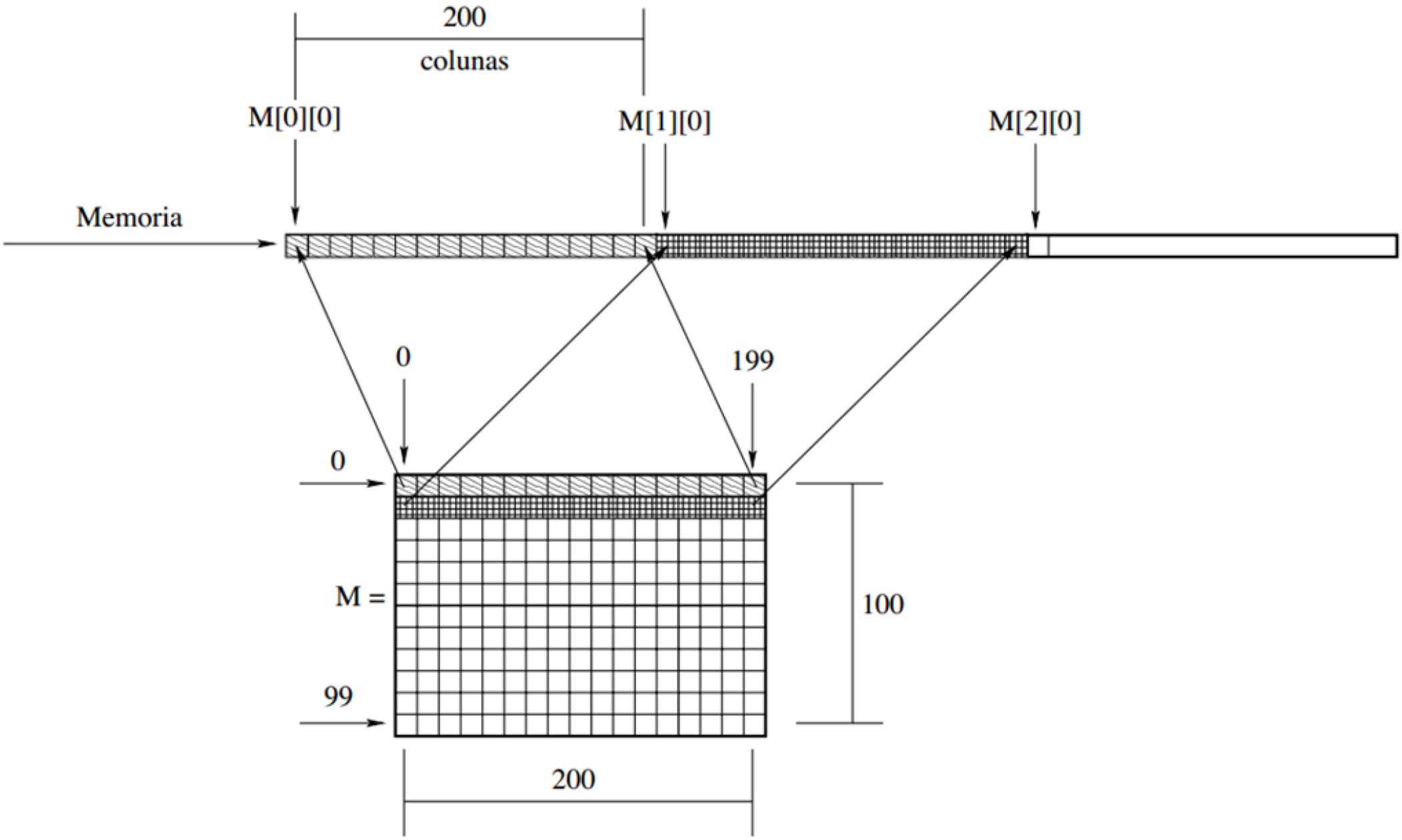
Declara uma matriz M
de 100 linhas
com 200 colunas
(20mil inteiros)





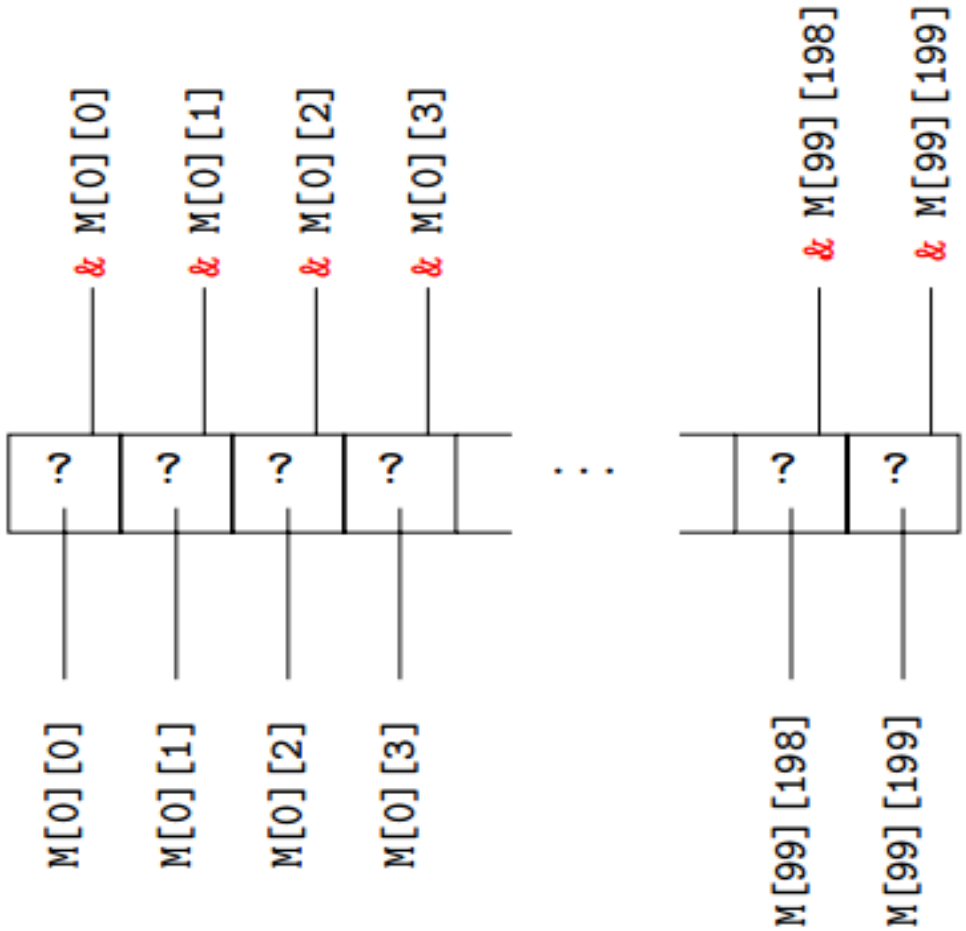
A memória do computador é linear!

Estrutura da matriz na memória do computador



Disposição dos 20 mil elementos da matriz M na memória

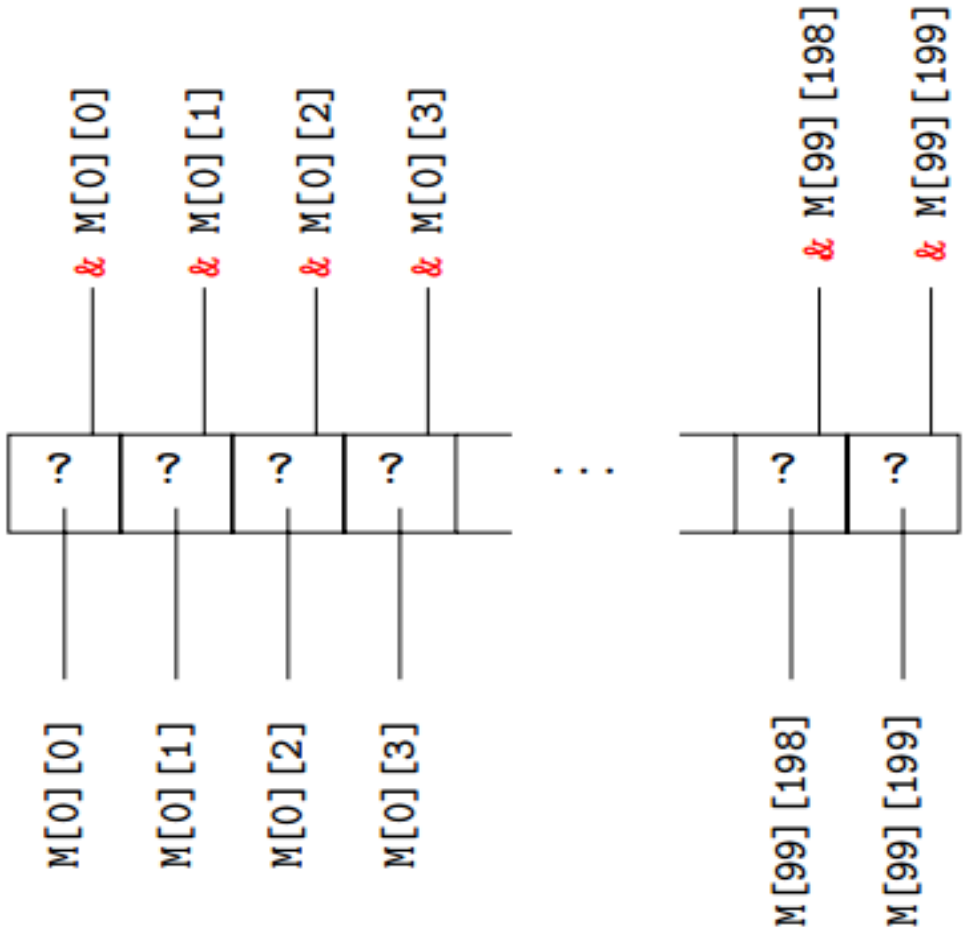
```
int M[100][200];
```



Qual o endereço de M[0][78]?
(tendo como base M[0][0])

Disposição dos 20 mil elementos da matriz M na memória

```
int M[100][200];
```

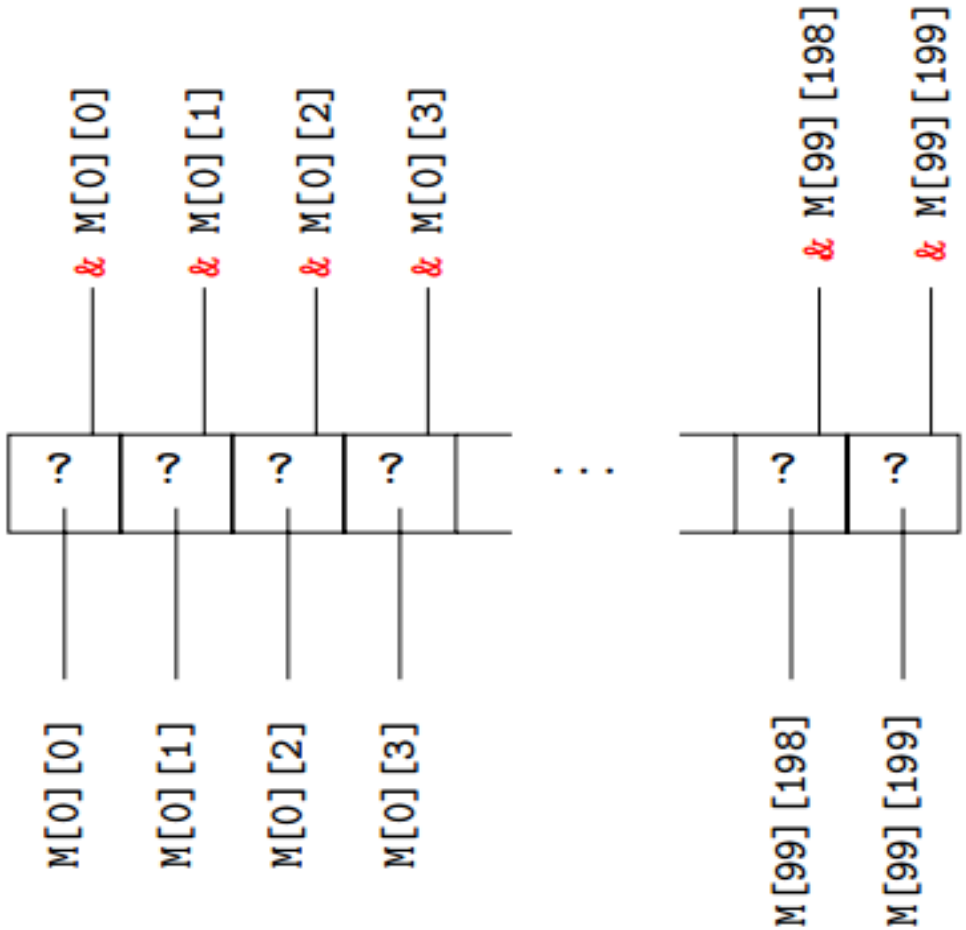


Qual o endereço de M[0][78]?
(tendo como base M[0][0])

&M[0][0]+78

Disposição dos 20 mil elementos da matriz M na memória

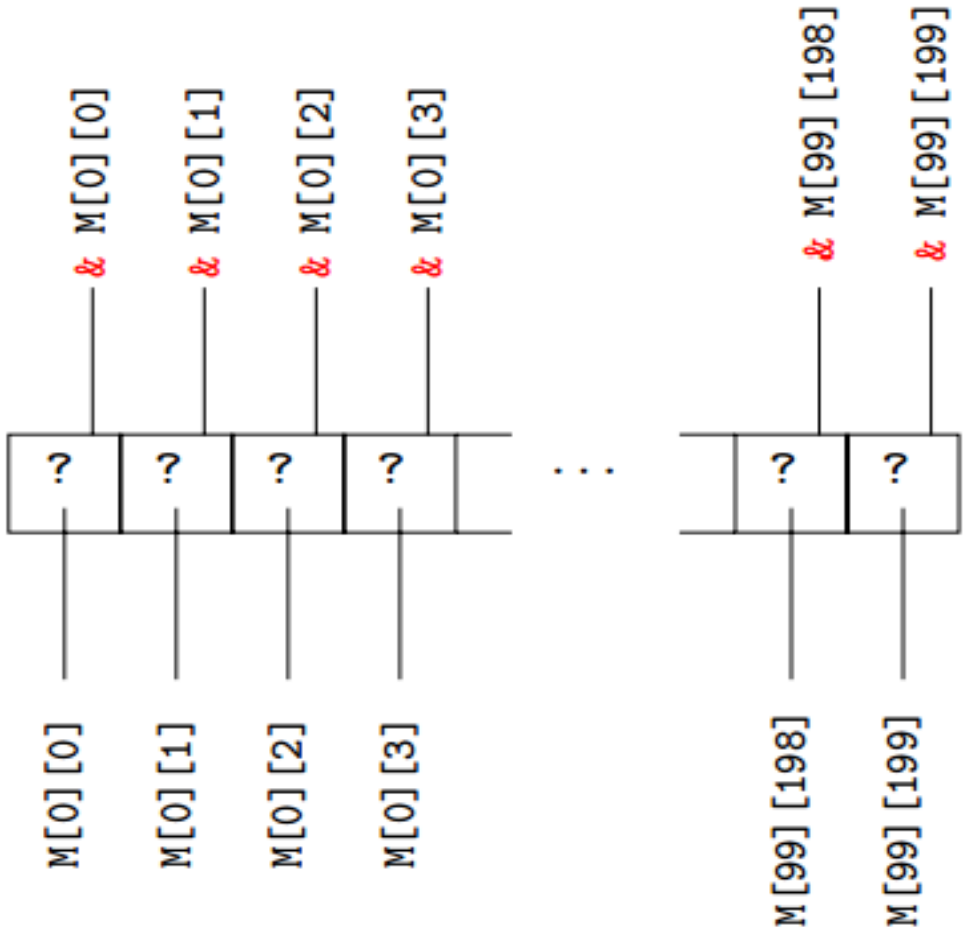
```
int M[100][200];
```



Qual o endereço de M[78][21]?
(tendo como base M[0][0])

Disposição dos 20 mil elementos da matriz M na memória

```
int M[100][200];
```



Qual o endereço de M[78][21]?
(tendo como base M[0][0])

&M[0][0] + (78***200**+21)

Índices

- Na linguagem C **não existe verificação de índices fora da matriz/vetor**.
- Quem deve controlar o uso correto dos índices é o programador.

- O acesso utilizando um índice errado pode ocasionar o acesso de outra variável na memória.
- → Se o acesso à memória é **indevido** você recebe a mensagem "segmentation fault".

Matrizes

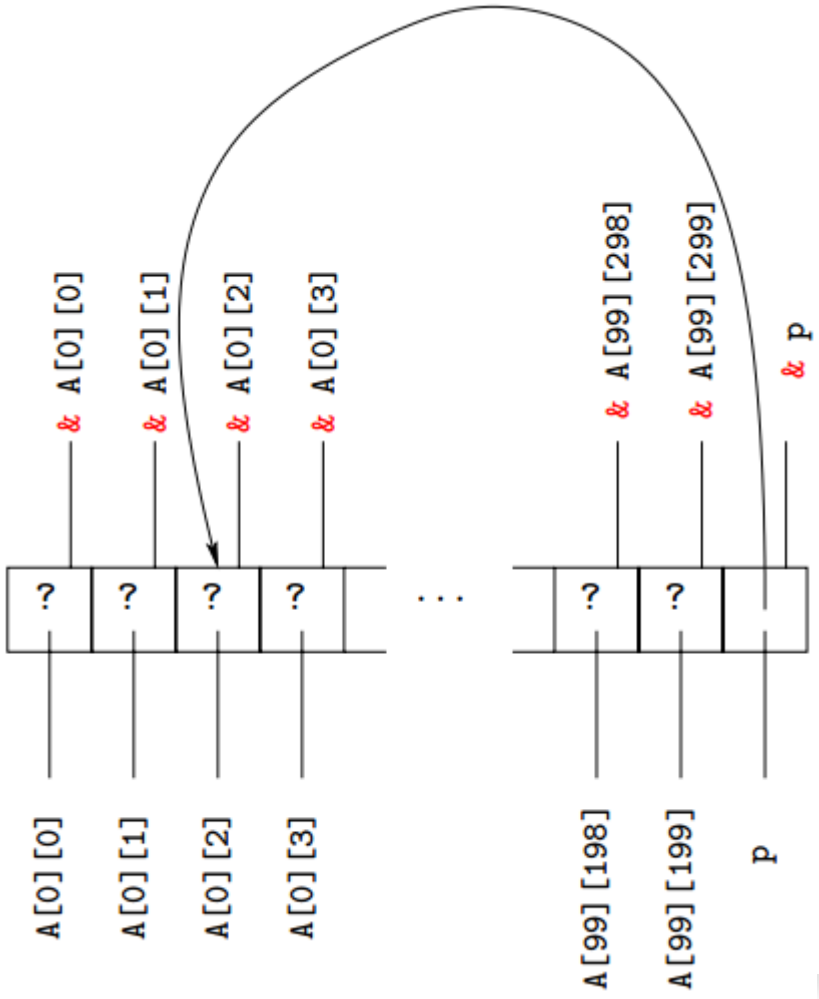
```
int A [100][300];
int *p ;              // ponteiro para inteiro

p = &A[0][2];         // p aponta para a A[0][2]
```

Matrizes

```
int A [100][300];
int *p ;                    // ponteiro para inteiro

p = &A[0][2];               // p aponta para a A[0][2]
```



```c
int A[100][300];
int *p, *q;

p = &A[0][0];
q = A[0];

printf("%p\n%p", p, q);
```

```
0x7fff68497970
0x7fff68497970
```

Teste de avaliação

Questão 1 - a

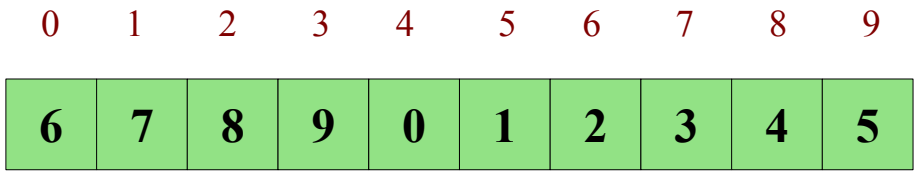
Escreva o resultado da execução do seguinte programa

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

int main() {
    int A[10] = {6,7,8,9,0,1,2,3,4,5};
    int x = -2;

    printf("Operacao 1: %d\n",    A[5] + x);
    printf("Operacao 2: %d\n", *(&A[5])+ x);
    printf("Operacao 3: %d\n", *(&A[5] + x));
    printf("Operacao 4: %d\n", *(A + 5 + x));
}
```

```
Operacao 1: -1
Operacao 2: -1
Operacao 3: 9
Operacao 4: 9
```



Questão 1 - b

Escreva o resultado da execução do seguinte programa

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

int main()
{
    int A[100][300];
    int *p = &A[0][0];
    int i, j;

    for(i=0; i<100; i++)
        for(j=0; j<300; j++)
            A[i][j] = i+j;

    printf("Elemento 1 = %d\n", p[2*300+51]     );
    printf("Elemento 2 = %d\n", p[99*300+200]   );
    printf("Elemento 3 = %d\n", *(p+99*300+200) );
}
```

	0	1	2	3	4	...
0	0	1	2	3	4	
1	1	2	3	4	5	
2	2	3	4	5	6	
3	3	4	5	6	7	
.	4	5	6	7	8	
.						
.						

```
Elemento 1 = 53
Elemento 2 = 299
Elemento 3 = 299
```

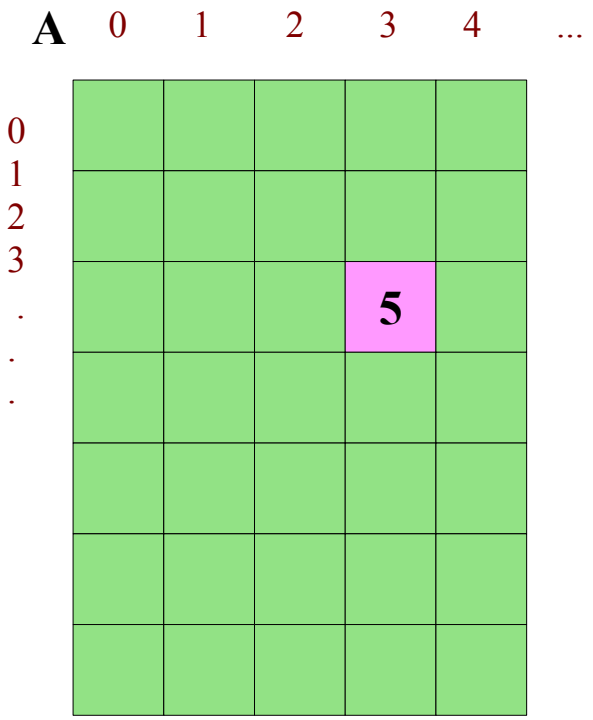
← A[2][51]
A[99][200]
A[99][200]

Questão 1 - c

Macro!

```
1  #include <stdio.h>
2  #define max 100
3
4  void funcaoX(int M[max][max]) {
5      M[2][3] = 4;
6  }
7
8  int main() {
9      int A[max][max];
10     A[2][3] = 5;
11
12     funcaoX(A);
13
14     printf("%d", A[2][3]);
15 }
```

```
4
```



Função com matriz como parâmetro

- O nome de **uma matriz** dentro do parâmetro de uma função é utilizado **como sendo um ponteiro para o primeiro elemento da matriz** que está sendo usada na hora de utilizar a função.

- → Não é alocada memória (novamente) para um vetor passado por parâmetro para uma função.

Questão 2

Escreva um programa que leia um número inteiro positivo **n** seguido de **n** números inteiros e imprima esses **n** números em ordem invertida.

Por exemplo, ao receber

5

22 33 44 55 66

o seu programa deve imprimir

66 55 44 33 22

- **Seu programa não deve impor limitações sobre o valor de n**
- **Seu programa não deve usar colchetes.**

Questão 2

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

int main() {
    int n, i;
    scanf("%d", &n);
    int *p = (int *) malloc(n*sizeof(int));

    for (i=0; i<n; i++)
        scanf("%d", p+i);

    for (i=n-1; i>=0; i--)
        printf("%d ", *(p+i));

    free(p);
}
```

```
5
11 22 33 44 55
55 44 33 22 11
```

Os ponteiros facilitam a alocação dinâmica de memória

Questão 3

3. Escreva uma função `mm` que receba um vetor de inteiros `v[0..n-1]`, um inteiro `n` que indica o comprimento do vetor, e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nestas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Sua função **não** deve usar colchetes. [4 pontos]

```
// versão 1
void mm(int *v, int n, int *min, int *max) {
    int i;
    *min = *v;
    *max = *v;

    for (i=1; i<n; i++) {
        if (*min > *(v+i))
            *min = *(v+i);
        if (*max < *(v+i))
            *max = *(v+i);
    }
}
```

Questão 3

3. Escreva uma função `mm` que receba um vetor de inteiros `v[0..n-1]`, um inteiro `n` que indica o comprimento do vetor, e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nestas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Sua função **não** deve usar colchetes. [4 pontos]

```
// versão 2
void mm2(int *v, int n, int *min, int *max) {
    int i;
    *min = *v;
    *max = *v;

    for (i=0; i<n; i++) {
        if (*min>*v)
            *min = *v;
        if (*max<*v)
            *max = *v;
        v++;
    }
}
```