Aula 09:
- Ponteiros (parte 2)

Prof. Jesús P. Mena-Chalco
jesus.mena@ufabc.edu.br

3Q-2017

Sobre funções (“uma ideia”)

Qual função é mais “eficiente”?

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

    t1 = a;
    t2 = b;

    a = t2;
    b = t1;

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

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

    t = a;

    a = b;
    b = t;

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

1995

2015

Qual função é mais "eficiente"?

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

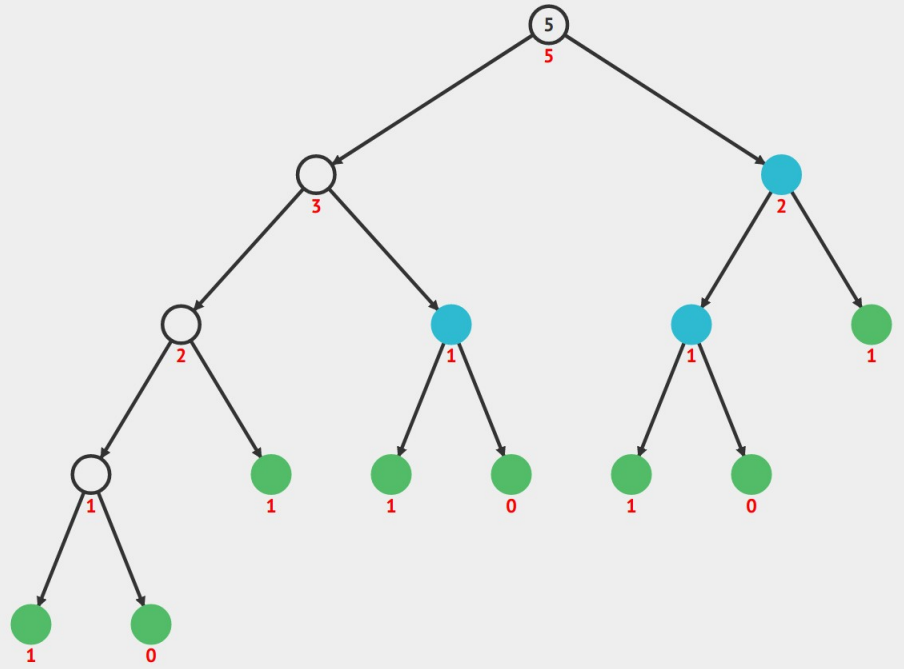
Número de multiplicações?
Proporcional a k

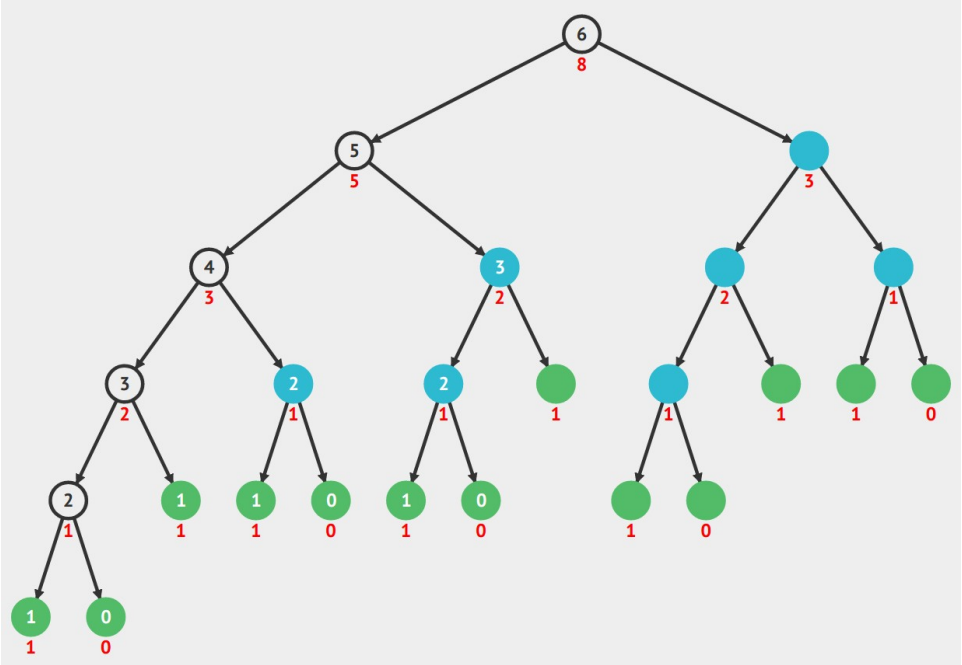
```
int f2(int a, int k) {
    int x;

    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)

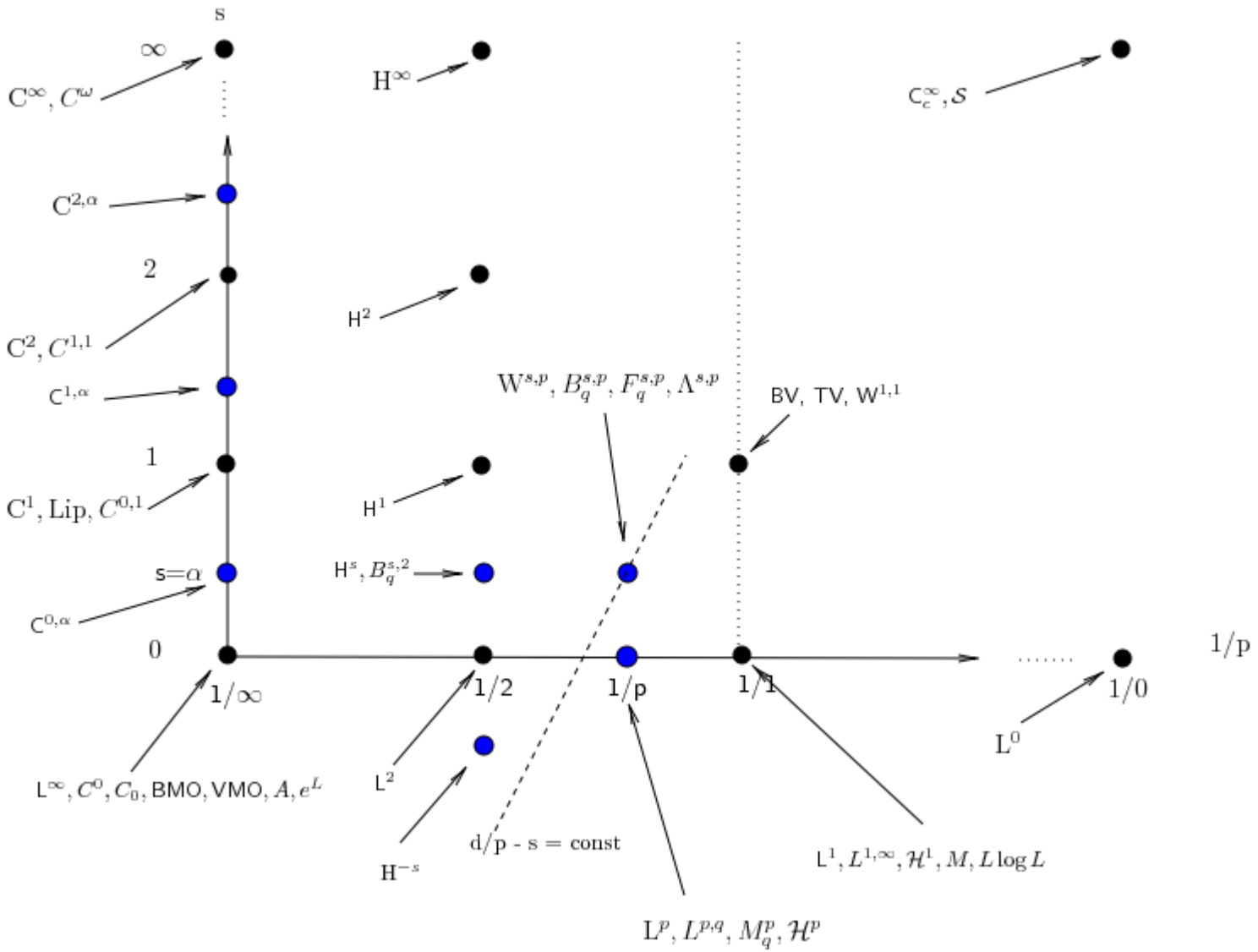
Exercício de Fibonacci



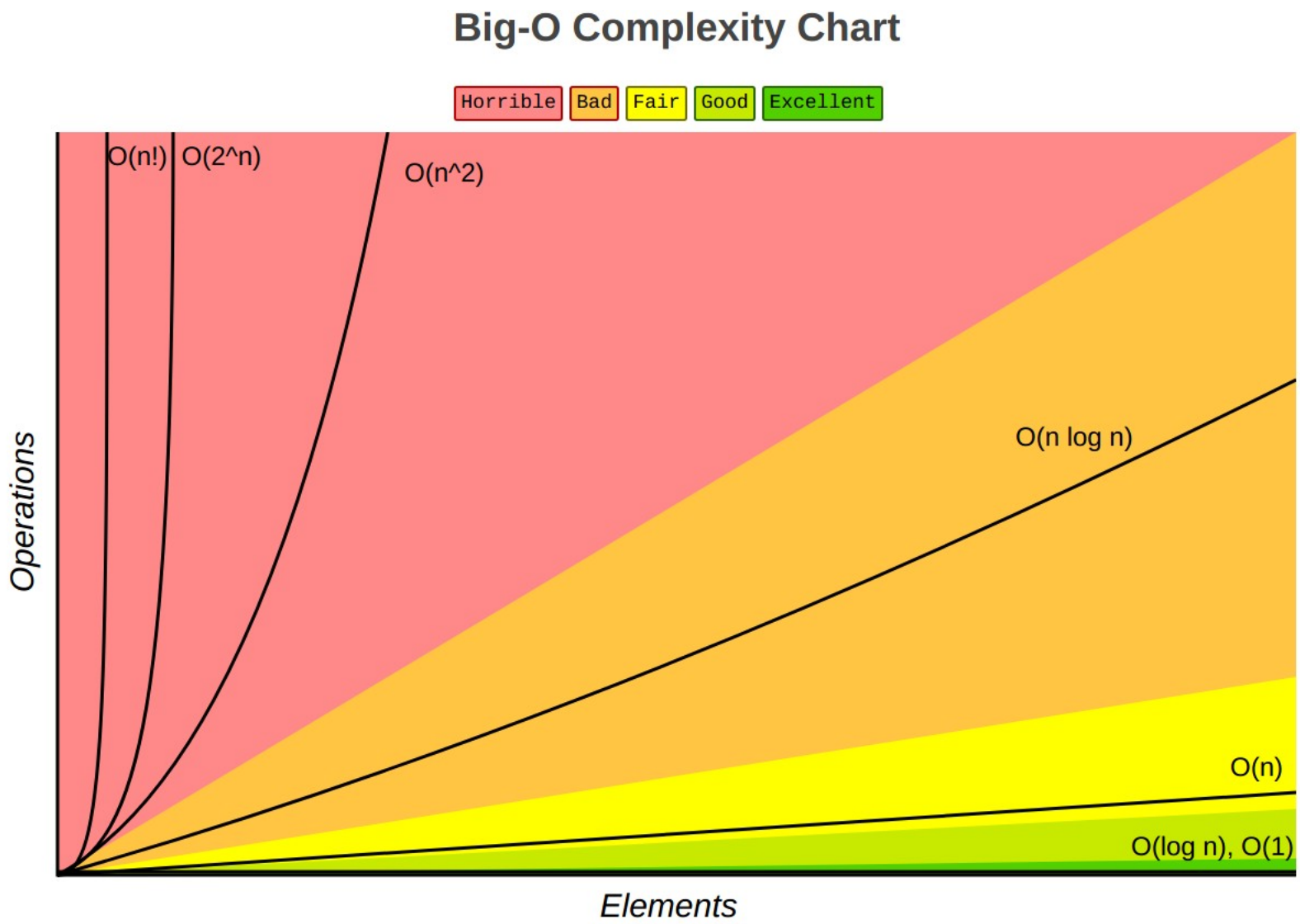


Número de chamados?
Proporcional a 2^n

https://visualgo.net/en/recursion


s
∞
C^∞, C^ω
H^∞
$C_c^\infty, \mathcal{S}$
$C^{2,\alpha}$
2
$C^2, C^{1,1}$
H^2
$C^{1,\alpha}$
$W^{s,p}, B_q^{s,p}, F_q^{s,p}, \Lambda^{s,p}$
BV, TV, $W^{1,1}$
1
C^1, Lip, $C^{0,1}$
H^1
s=α
$H^s, B_q^{s,2}$
$C^{0,\alpha}$
0
1/p
1/∞
1/2
1/p
1/1
1/0
L^0
L^∞, C^0, C_0, BMO, VMO, A, e^L
L^2
H^{-s}
d/p - s = const
$L^1, L^{1,\infty}, \mathcal{H}^1, M, L \log L$
$L^p, L^{p,q}, M_q^p, \mathcal{H}^p$

Qual função é mais “eficiente”?



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

Common Data Structure Operations

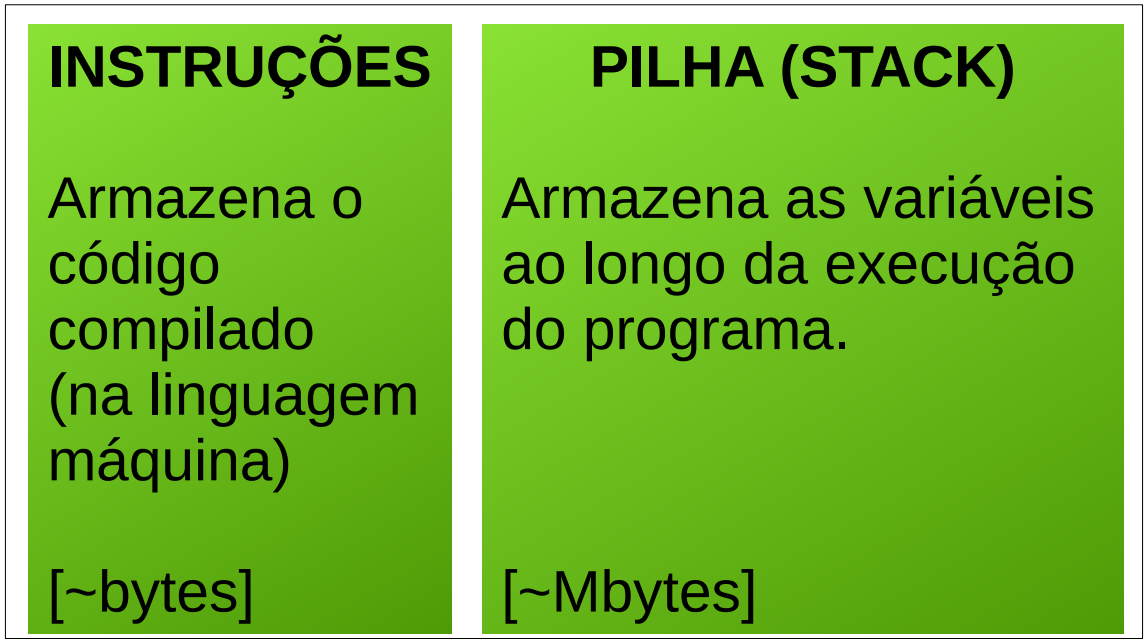
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)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

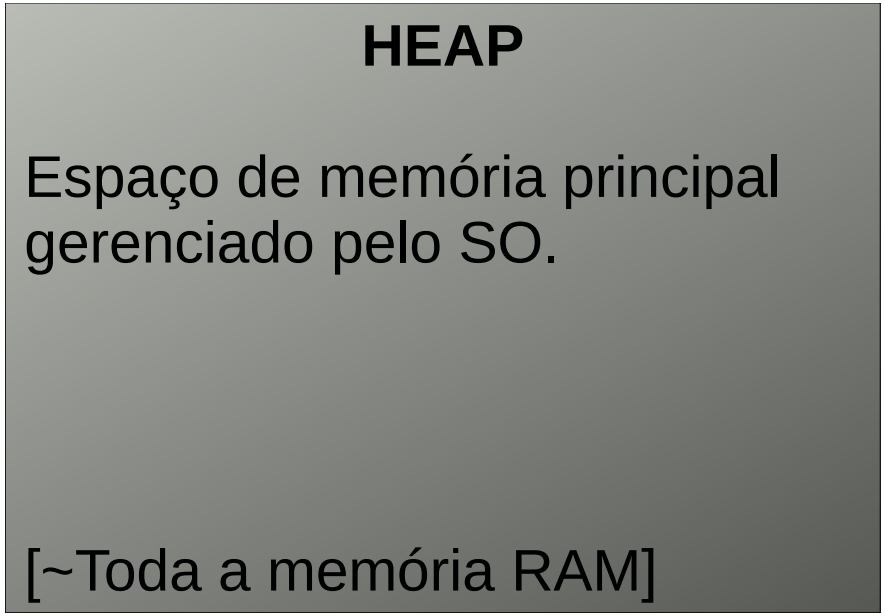
Array Sorting Algorithms

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

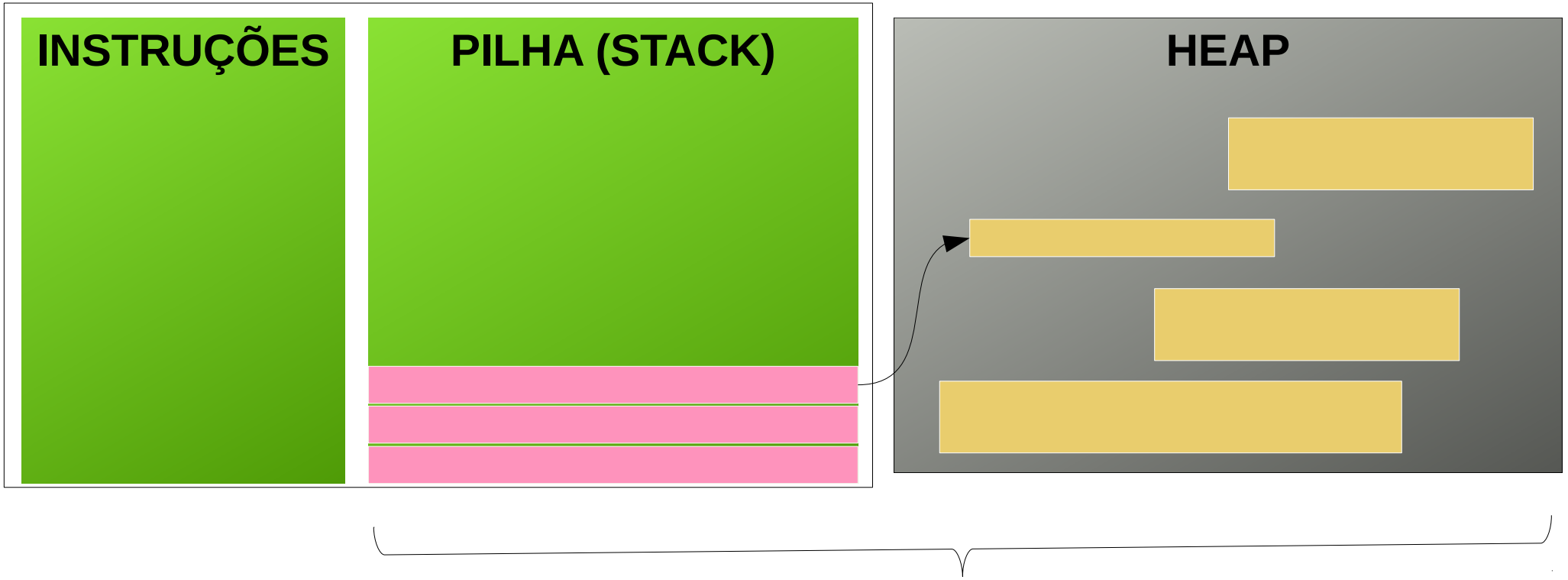
Alocação dinâmica

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

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

Ponteiros?

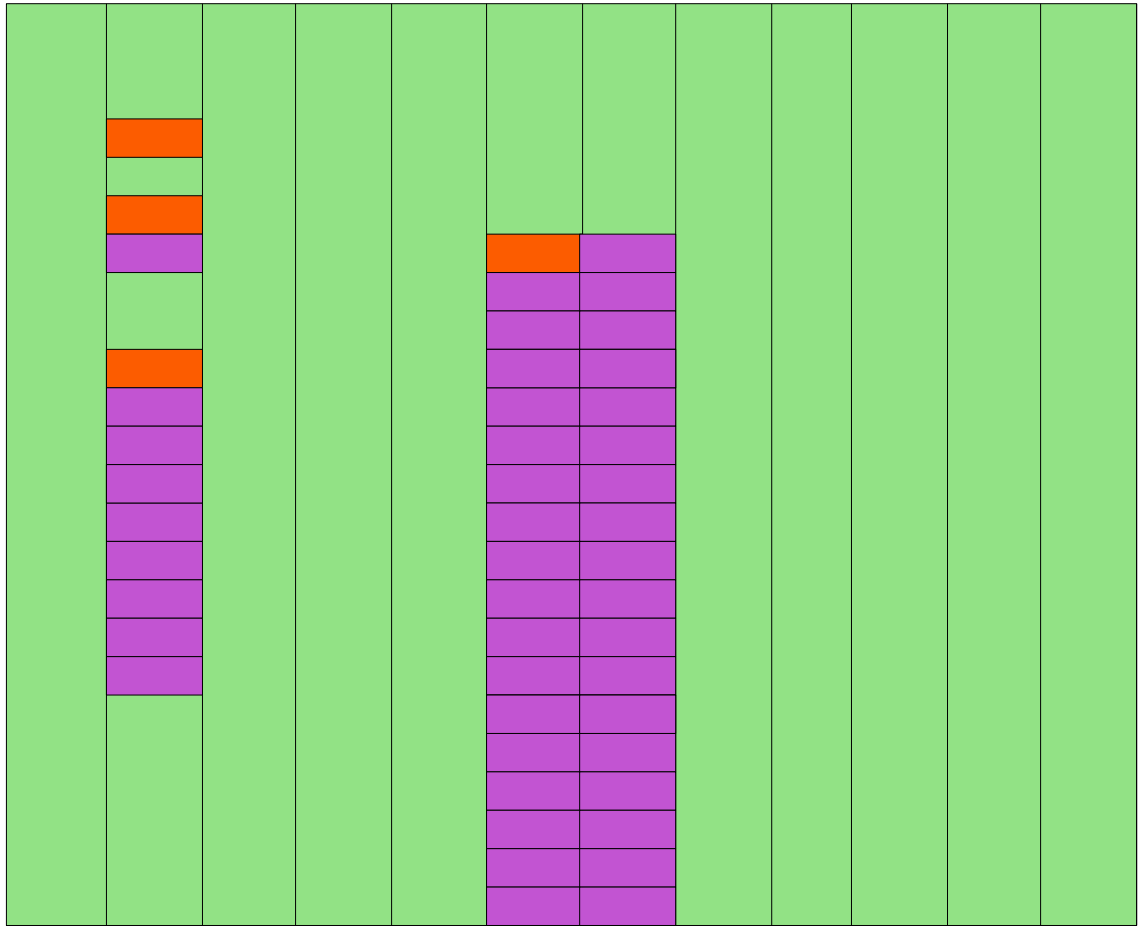
Processo na memória



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 unarios

`&` → 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

Teste interativo usando cdecl

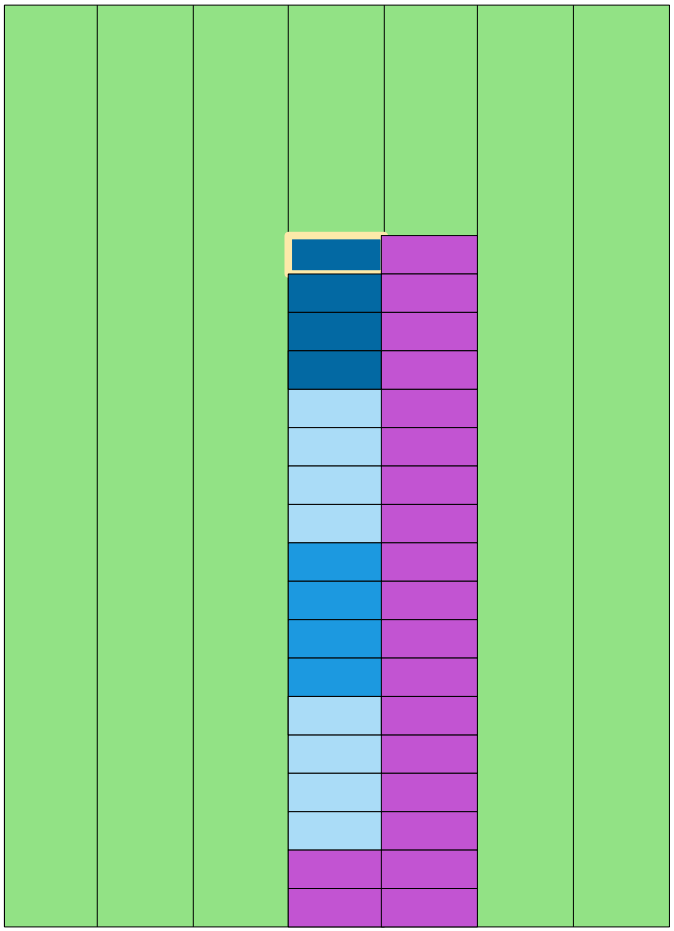
- `int *A`
- `int **A`
- `int **A[17]`
- `float *p3`
- `float &p3`
- `void (*pf)(int)`
- `double (*pf)(int)`

https://cdecl.org/

Vetores e endereços

Vetores

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

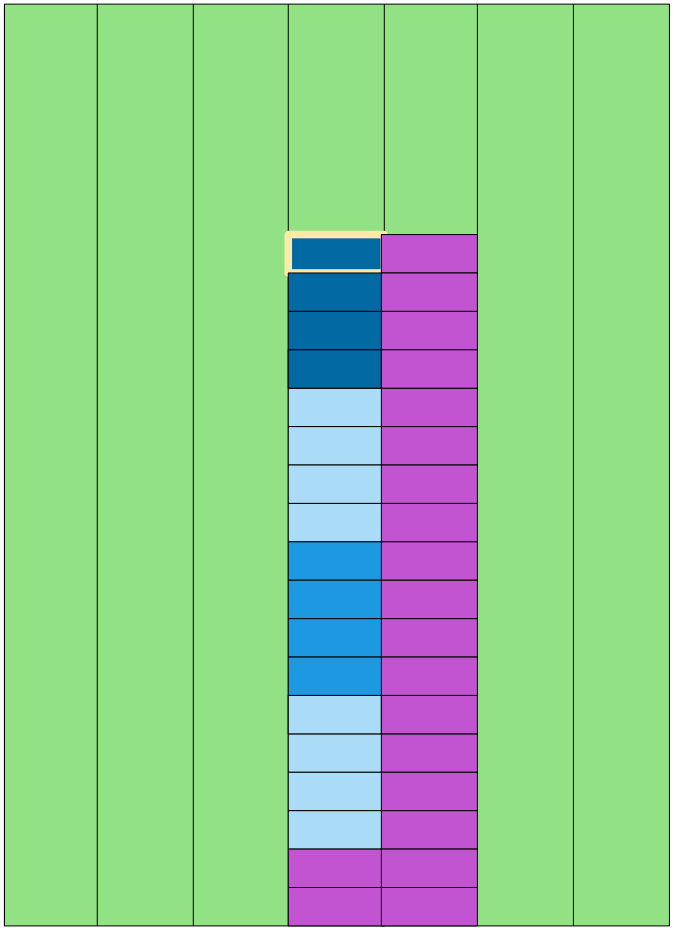


Se cada elemento ocupa **b** bytes, a diferença entre os endereços de dois elementos consecutivos será de **b**.

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

Vetores

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

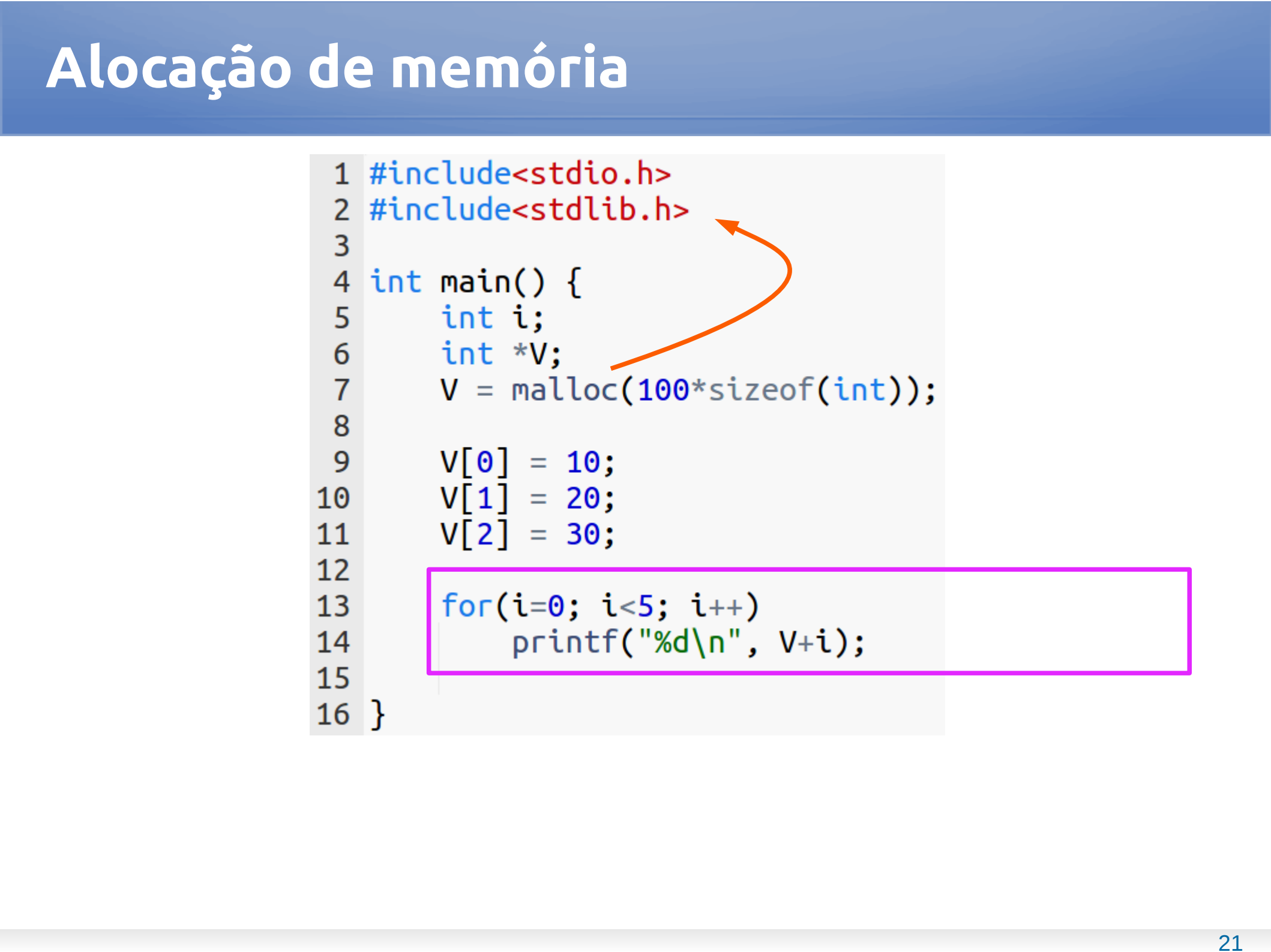


Se cada elemento ocupa **b** bytes, a diferença entre os endereços de dois elementos consecutivos será de **b**.

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

Alocação de memória

```
 1 #include<stdio.h>
 2 #include<stdlib.h>
 3
 4 int main() {
 5     int i;
 6     int *V;
 7     V = malloc(100*sizeof(int));
 8
 9     V[0] = 10;
10     V[1] = 20;
11     V[2] = 30;
12
13     for(i=0; i<5; i++)
14         printf("%d\n", V+i);
15
16 }
```



Alocação de memória

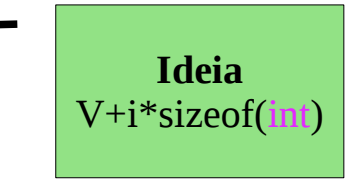
```c
#include<stdio.h>
#include<stdlib.h>

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);

}
```




```
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 memoria 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

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);
}
```

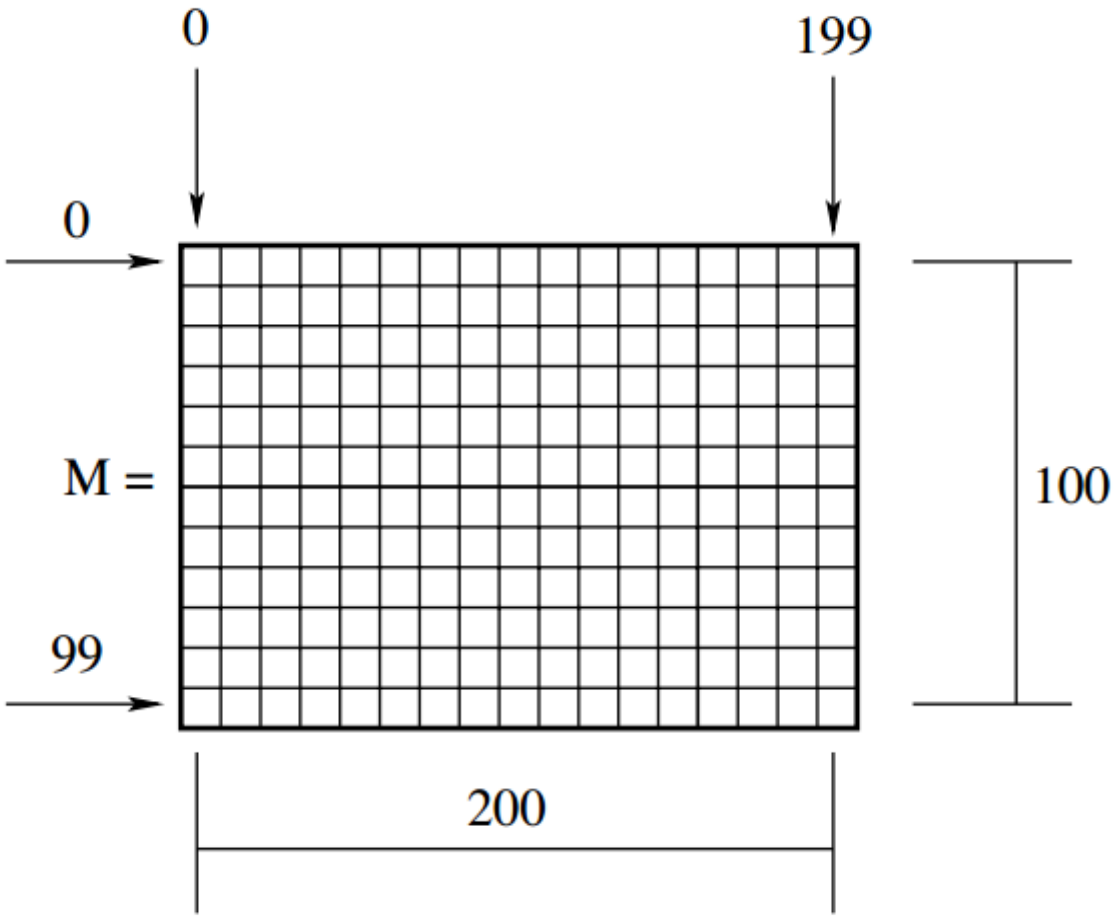
```
6
1
2
3
4
5
6
6 5 4 3 2 1
```

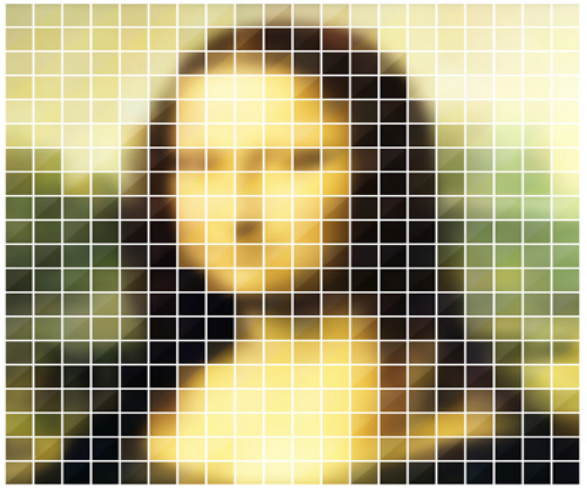
Matrizes

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

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

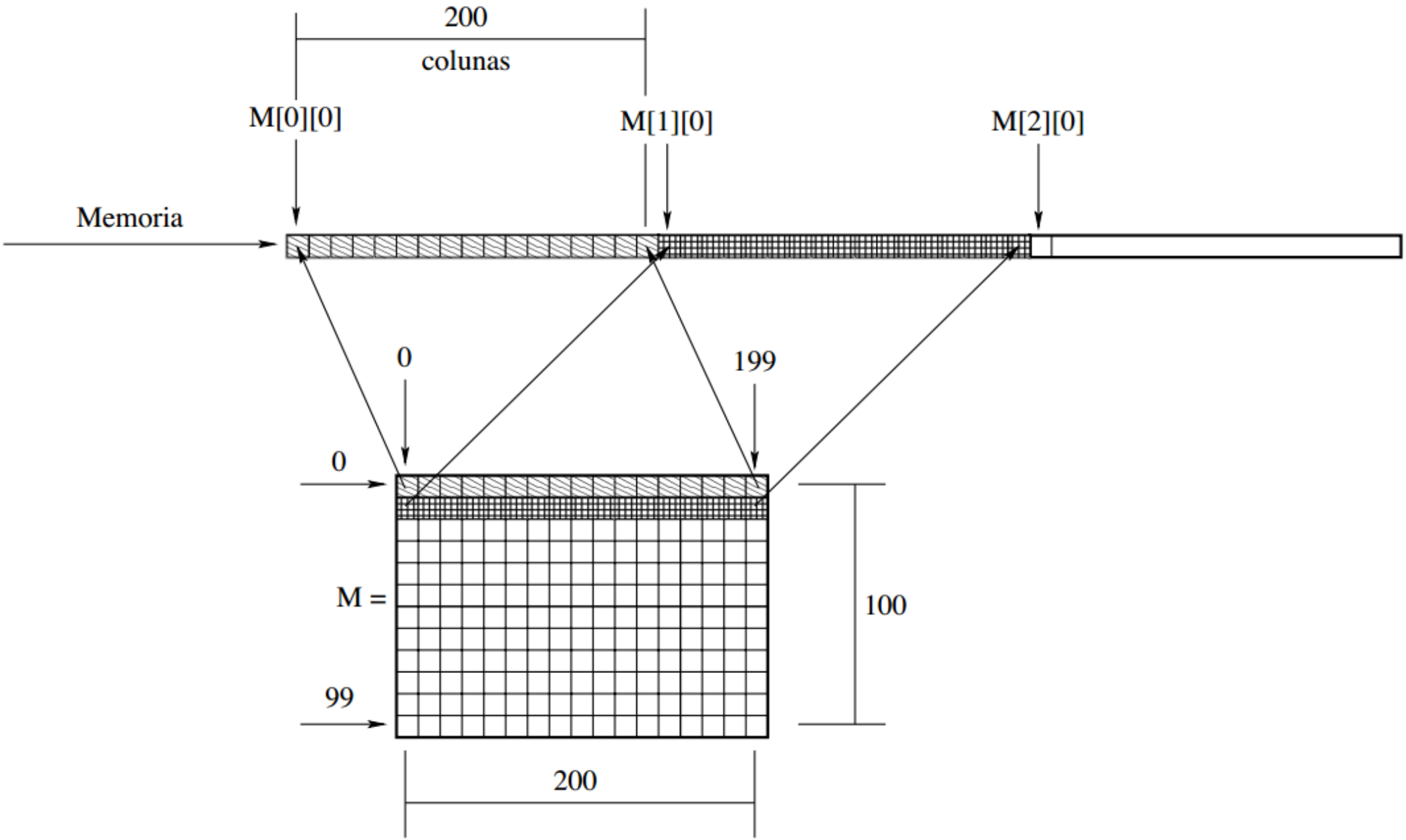
Declara uma matriz M
de 100 linhas
com 200 colunas
(20mil inteiros)





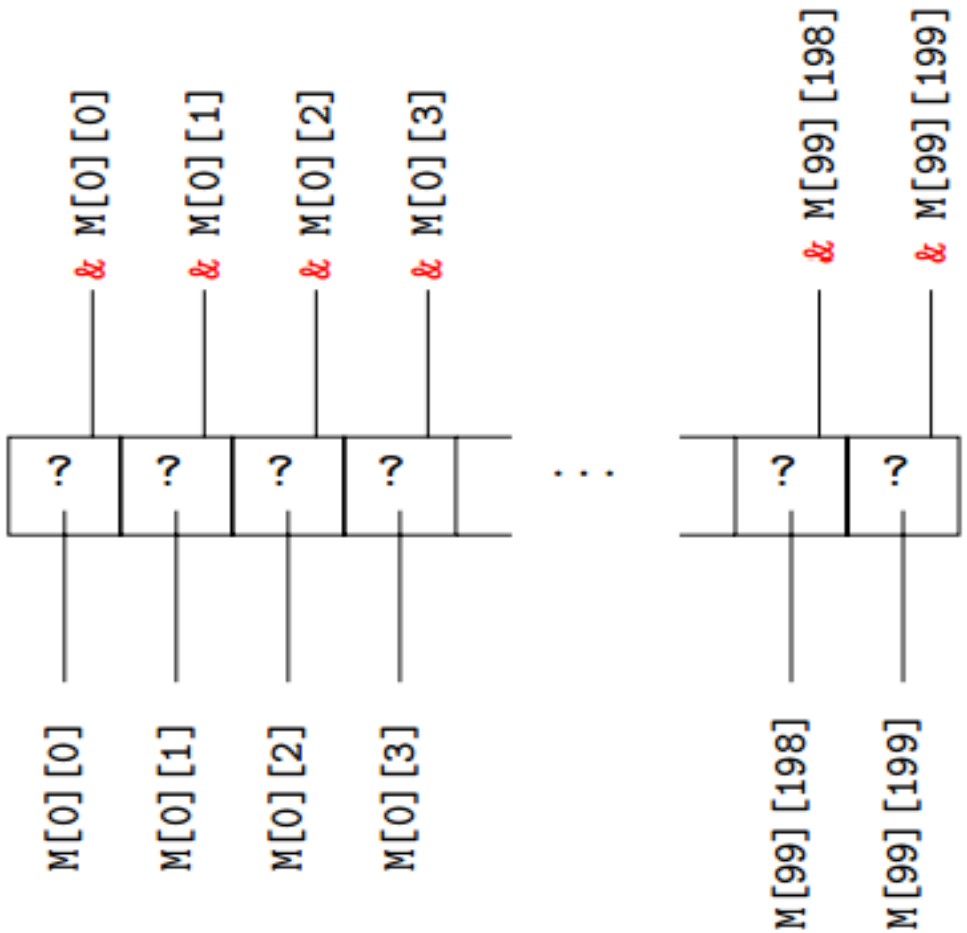
A memória do computador é linear!

Estrutura da matriz na memória do computador



Disposição dos 20mil elementos da matriz M na memória

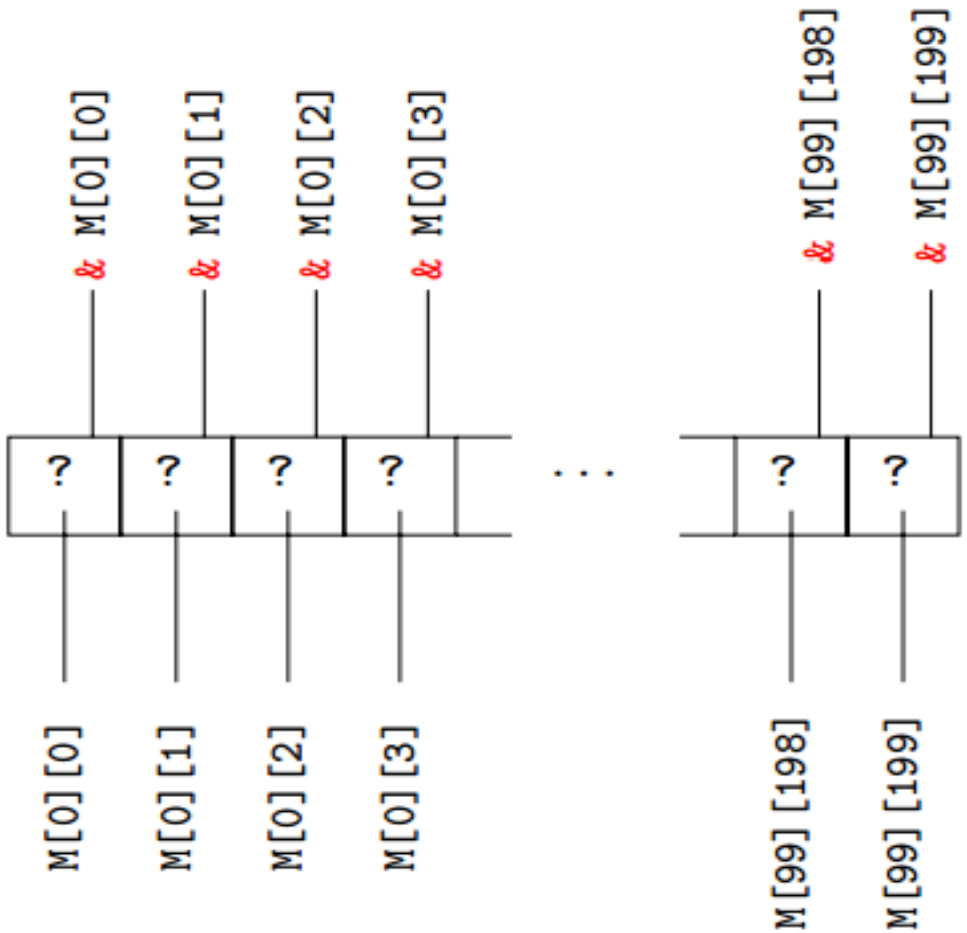
```
int M[100][200];
```



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

Disposição dos 20mil elementos da matriz M na memória

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

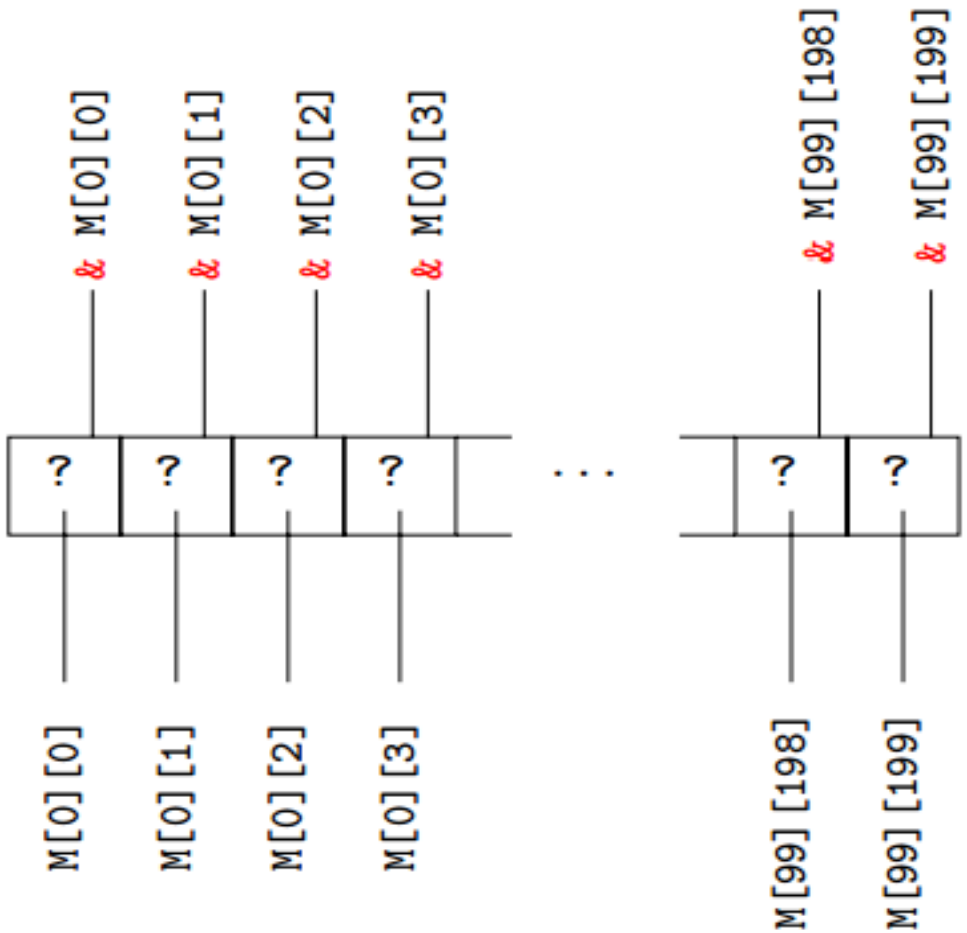


Qual o endereço de M[0][78]?
(tendo como base M[0][0])

&M[0][0]+78

Disposição dos 20mil elementos da matriz M na memória

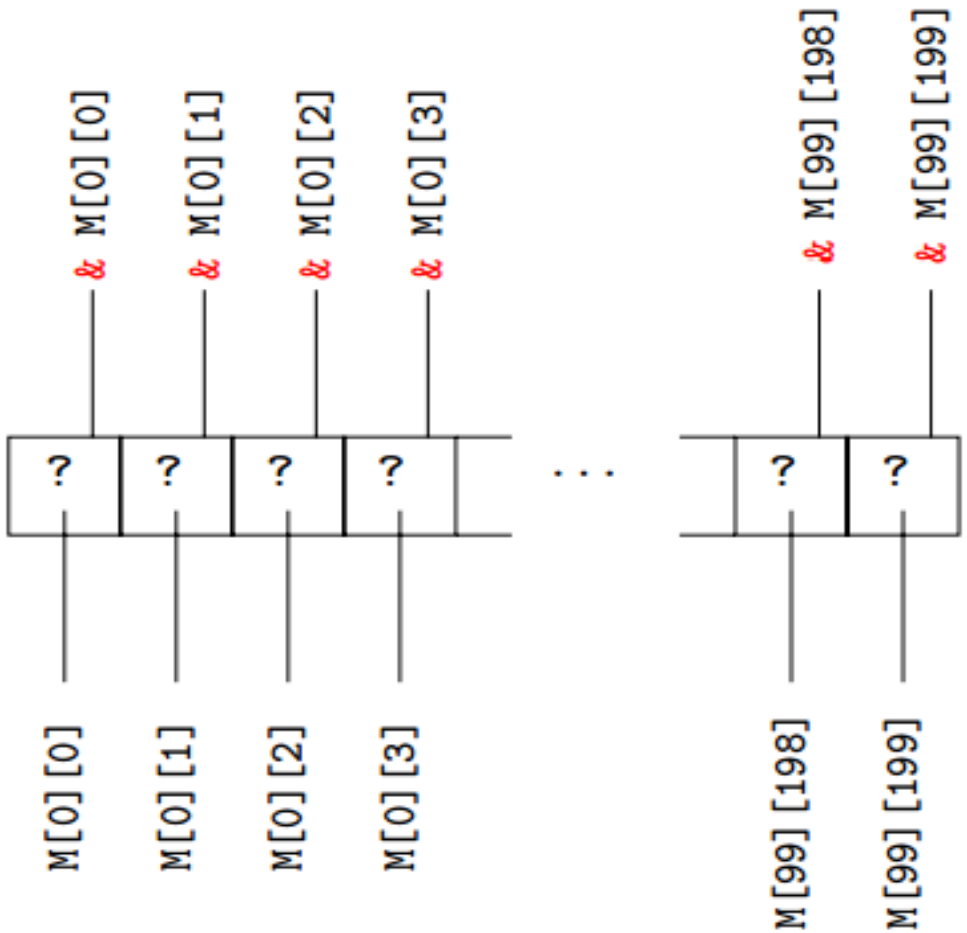
```
int M[100][200];
```



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

Disposição dos 20mil 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

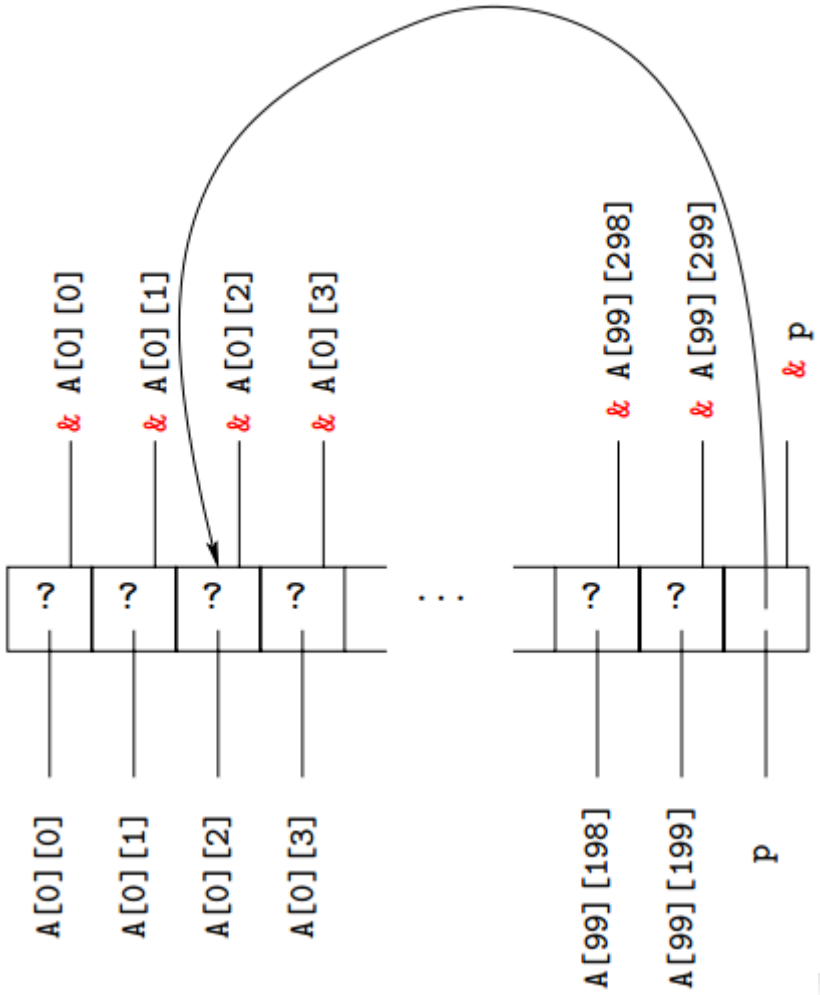
```
int A [100][300];
int *p ;              // ponteiro para inteiro

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

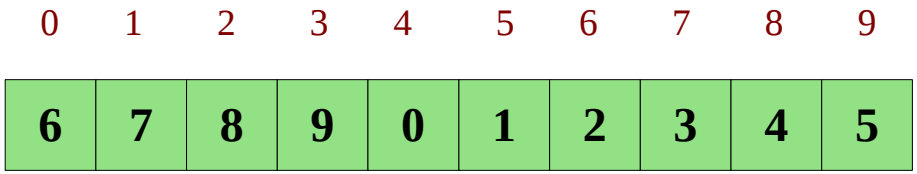
Escreva o resultado da execução do seguinte programa

```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
```



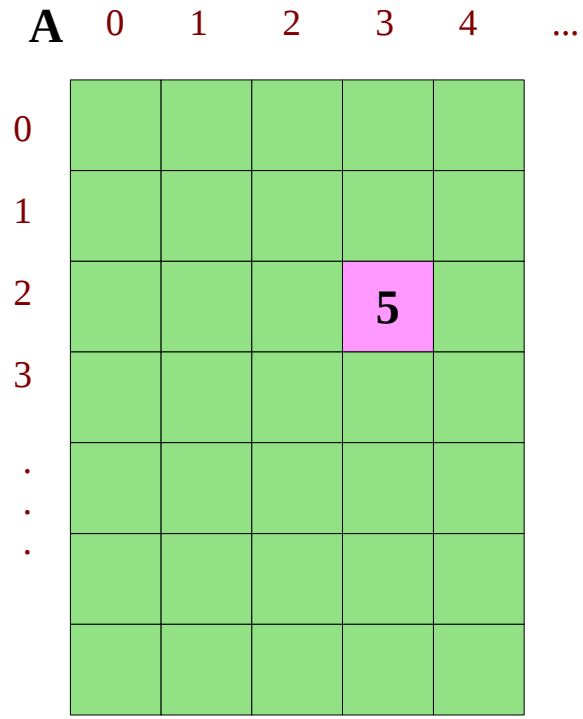
```
A[2][51]
A[99][200]
A[99][200]
```

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

Buying time promotes happiness

Ashley V. Whillans[a,1], Elizabeth W. Dunn[b], Paul Smeets[c], Rene Bekkers[d], and Michael I. Norton[a]

[a]Harvard Business School, Harvard University, Cambridge, MA 02163; [b]Department of Psychology, University of British Columbia, Vancouver, BC, Canada V6T 1Z4; [c]Department of Finance, Maastricht University, 6200 MD Maastricht, The Netherlands; and [d]Center for Philanthropic Studies, Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands

Edited by Susan T. Fiske, Princeton University, Princeton, NJ, and approved June 13, 2017 (received for review April 19, 2017)

Around the world, increases in wealth have produced an unintended consequence: a rising sense of time scarcity. We provide evidence that using money to buy time can provide a buffer against this time famine, thereby promoting happiness. Using large, diverse samples from the United States, Canada, Denmark, and The Netherlands (n = 6,271), we show that individuals who spend money on time-saving services report greater life satisfaction. A field experiment provides causal evidence that working adults report greater happiness after spending money on a time-saving purchase than on a material purchase. Together, these results suggest that using money to buy time can protect people from the detrimental effects of time pressure on life satisfaction.

time | money | happiness | well-being

In recent decades, incomes have risen in many countries (1, 2), potentially exacerbating a new form of poverty: from Germany to Korea to the United States, people with higher incomes report greater time scarcity (3). Feelings of time stress are in turn linked to lower well-being, including reduced happiness, increased anxiety, and insomnia (4–6). Time stress is also a critical factor underlying rising rates of obesity: lacking time is a primary reason that people report failing to eat healthy foods or exercise regularly (7, 8). In theory, rising incomes could offer a way out of the "time famine" of modern life (9), because wealth offers the opportunity to have more free time, such as by paying more to live closer to work. However, some evidence suggests that wealthier people spend more time engaging in stressful activities, such as shopping and commuting (10). Experimental research shows that simply leading people to feel that their time is economically valuable induces them to feel that they do not have enough of it (11).

A great deal of attention has been devoted to reducing financial scarcity, but there is relatively little rigorous research examining how to reduce feelings of time scarcity, which in fact may offer a particularly difficult challenge given that time, unlike money, is inherently finite. Could allocating discretionary income to buy free time—such as by paying to delegate common household chores, like cleaning, shopping, and cooking— reduce the negative effects of the modern time famine, thereby promoting well-being? The growth of the sharing economy has made time-saving services increasingly accessible, but no empirical research has tested whether using such services enhances happiness.

From our theoretical perspective, buying time should protect people from the negative impact of time stress on life satisfaction. This conceptualization draws on the social support literature, in which research on the "buffering hypothesis" has demonstrated that receiving social support can protect people from experiencing the negative consequences of stress (12). That is, the typical relationship between stress and reduced well-being is attenuated for individuals who are able to access social support (13–15). We suggest that buying time may provide an alternate mechanism to receiving the support needed to cope with daily demands, such that the relationship between time stress and reduced life satisfaction should be attenuated among people who use money to access more time.

Results

As an initial test of this hypothesis, we surveyed Mechanical Turk workers in the United States (n = 366), a nationally representative sample of working Americans living in the United States (n = 1,260), adults in Denmark (n = 467), and Canada (n = 326), and both a nationally representative sample (n = 1,232) and a sample of millionaires (n = 818) in The Netherlands. See Table 1 for sample demographics. In all samples, respondents completed two questions about whether—and how much—money they spent each month to increase their free time by paying someone else to complete unenjoyable daily tasks. In addition, respondents rated their satisfaction with life (SWL) and reported their annual household income, the number of hours they work each week, age, marital status, and the number of children living at home (*SI Appendix*). In the Canadian and Dutch surveys (n = 2,376), respondents also completed a measure of time stress (4), allowing us to test the prediction that buying time mitigates the negative effects of time stress on life satisfaction.

Here we report the meta-analytic effects across samples (16); results for individual samples are provided in Fig. 1 and *SI Appendix*. Across samples (n = 4,469), 28.2% of respondents spent money to buy themselves time each month [$\text{mean}_{\text{amount}}$ = $147.95 US dollars (USD) for respondents who reported buying time]. Respondents who spent money in this way reported greater life satisfaction, $d = 0.24$, $P < 0.001$, 95% CI (0.18, 0.31). This relationship was positive within each sample and reached statistical significance for the nationally representative sample of working Americans, adults in Canada and Denmark, and millionaires in The Netherlands (Fig. 1). This effect held controlling for our key set of covariates ($n = 3{,}983$), $d = 0.22$, $P < 0.001$, 95% CI (0.15, 0.29) and was not moderated by income or wealth, $Z = -0.35$, $P = 0.729$, 95% CI (−0.08, 0.06): people from across the income spectrum benefitted from buying time. These results also held when we controlled for an alternative set of covariates where we replaced household income with log income and added an age-squared variable (*SI Appendix*, Tables S6–S23b). These results provide initial evidence for a robust link between buying time and life satisfaction across diverse samples.

Significance

Despite rising incomes, people around the world are feeling increasingly pressed for time, undermining well-being. We show that the time famine of modern life can be reduced by using money to buy time. Surveys of large, diverse samples from four countries reveal that spending money on time-saving services is linked to greater life satisfaction. To establish causality, we show that working adults report greater happiness after spending money on a time-saving purchase than on a material purchase. This research reveals a previously unexamined route from wealth to well-being: spending money to buy free time.

Author contributions: A.V.W., E.W.D., and M.I.N. designed research; P.S. contributed to the design of studies 5 and 6; P.S. and R.B. collected data for studies 5 and 6; A.V.W. analyzed the data; and A.V.W., E.W.D., and M.I.N. wrote the paper.

The authors declare no conflict of interest.

This article is a PNAS Direct Submission.

Freely available online through the PNAS open access option.

[1]To whom correspondence should be addressed. Email: awhillans@hbs.edu.

This article contains supporting information online at www.pnas.org/lookup/suppl/doi:10.1073/pnas.1706541114/-/DCSupplemental.