



# UNIDADE I

---

## Estrutura de Dados

Prof. MSc. Olavo Ito

# Linguagem C

- Desenvolvida na Bell Laboratory, por Dennis Ritchie, em 1972, e liberada para as universidades.
- Estrutura sequencial.
- Na lógica da programação, as sintaxes foram herdadas por várias linguagens como o C++, C#, Java, PHP.
  - Diversas IDEs (Integrated Development Environment), locais (Visual Studio, VS code, Codeblocks, Eclipse) e online: [https://www.onlinegdb.com/online\\_c\\_compiler](https://www.onlinegdb.com/online_c_compiler), [colab.research.google.com](https://colab.research.google.com)

# Linguagem C

- Não é um curso de linguagem C.
- O curso ensinará técnicas de programação para além das variáveis.

# Algoritmos e solução de problemas

- “Algoritmo é uma sequência finita de instruções ou operações cuja execução, em tempo finito, resolve um problema computacional, qualquer que seja sua instância.
- Algoritmo é uma sequência de passos que visam atingir um objetivo bem-definido” (FORBELLONE, 1993, p. 3).
- “Algoritmos são regras formais para a obtenção de um resultado ou da solução de um problema, englobando fórmulas de expressões aritméticas” (MANZANO *et al.*, 1996, p. 6).
  - “Algoritmo é a descrição de uma sequência de passos que deve ser seguida para a realização de uma tarefa” (ASCENCIO; CAMPOS, 2003, p. 1).

# Conceito de análise de algoritmos

- Algoritmos diferentes criados para resolver o mesmo problema muitas vezes são muito diferentes em termos de eficiência. Essas diferenças podem ser muito mais significativas que as diferenças relativas a *hardware* e *software* (CORMEN *et al.*, 2012, p. 14).
- Analisar um algoritmo significa prever os recursos de que o algoritmo necessita.
- Analisando algoritmos candidatos para a solução de um problema, pode-se identificar aqueles que sejam os mais eficientes e apontar também aqueles descartáveis devido à qualidade inferior no processo.

# Conceito de análise de algoritmos

Na análise do desempenho de um algoritmo, precisam ser observados os seguintes parâmetros:

- Tempo de execução – quanto tempo um código levou para ser executado;
  - É representado por uma função de **custo**  $T$ , onde  $T(n)$  é a medida do tempo total necessário para executar um algoritmo para um problema de tamanho  $n$ .
- Uso de memória volátil – a quantidade de espaço ocupado na memória principal do computador;
  - $T_{\text{espaço}}$  é o custo de ocupação de memória.

$$T(n) = T_{\text{tempo}} + T_{\text{espaço}}$$

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
1		
i	v[i]	maior
		1

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
2		
i	v[i]	maior
0		1



# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
3		
i	v[i]	maior
0		1

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
4		
i	v[i]	maior
0	1	1

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
5		
i	v[i]	maior
0	1	
		1

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
6		
i	v[i]	maior
	2	
		1
1		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
7		
i	v[i]	maior
	2	
		1
1		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
8		
i	v[i]	maior
	2	
		1
1		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
9		
i	v[i]	maior
	2	
		2
1		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
10		
i	v[i]	maior
	3	
		2
2		



# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
11		
i	v[i]	maior
	3	
		2
2		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
12		
i	v[i]	maior
	3	
		2
2		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
13		
i	v[i]	maior
	3	
		3
2		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
14		
i	v[i]	maior
	4	
		3
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
15		
i	v[i]	maior
	4	
		3
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
16		
i	v[i]	maior
	4	
		3
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
17		
i	v[i]	maior
	4	
		4
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={1,2,3,4},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
18		
i	v[i]	maior
	4	
		4
4		



# Conceito de análise de algoritmos

Passo a Passo				n vezes
Linha	Instruções	Operações	Quantidade	
5	maior = v[0];	maior = v[0]	1	
6	i=0;	i=0	1	
7	while (i<4){	(i<4)	1	
8	if (v[i]>=maior)	(v[i]>=maior)	1	
9	maior=v[i];	maior=v[i]	1	
10	i++;	i++	1	
7	while (i<4){	(i<4) Laço	1	

$$T(n) = 4n + 3$$

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
1		
i	v[i]	maior
		4

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
2		
i	v[i]	maior
0		4

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
3		
i	v[i]	maior
0		4

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
4		
i	v[i]	maior
0	4	4

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
5		
i	v[i]	maior
	3	4
1		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
6		
i	v[i]	maior
	3	4
1		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
7		
i	v[i]	maior
		4
	3	
1		



# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
8		
i	v[i]	maior
		4
	2	
2		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
9		
i	v[i]	maior
		4
	2	
2		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
10		
i	v[i]	maior
		4
	2	
2		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
11		
i	v[i]	maior
		4
	1	
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
12		
i	v[i]	maior
		4
	1	
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>=maior)
            maior=v[i];
        i++;
    }
}
```

n		
13		
i	v[i]	maior
		4
	1	
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
13		
i	v[i]	maior
		4
	1	
3		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
14		
i	v[i]	maior
		4
	1	
4		



# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
15		
i	v[i]	maior
		4
	4	
4		

# Conceito de análise de algoritmos

```
int main()
{
    int v[4]={4,3,2,1},maior,i;
    maior = v[0];
    i=0;
    while (i<4){
        if (v[i]>maior)
            maior=v[i];
        i++;
    }
}
```

n		
16		
i	v[i]	maior
		4
	4	
4		

# Conceito de análise de algoritmos

Passo a Passo				n vezes
Linha	Instruções	Operações	Quantidade	
5	maior = v[0];	maior = v[0]	1	
6	i=0;	i=0	1	
7	while (i<4){	(i<4)	1	
8	if (v[i]>=maior)	(v[i]>=maior)	1	
9	maior=v[i];	maior=v[i]	0	
10	i++;	i++	1	
7	while (i<4){	(i<4) Laço	1	

$$T(n) = 3n + 3$$

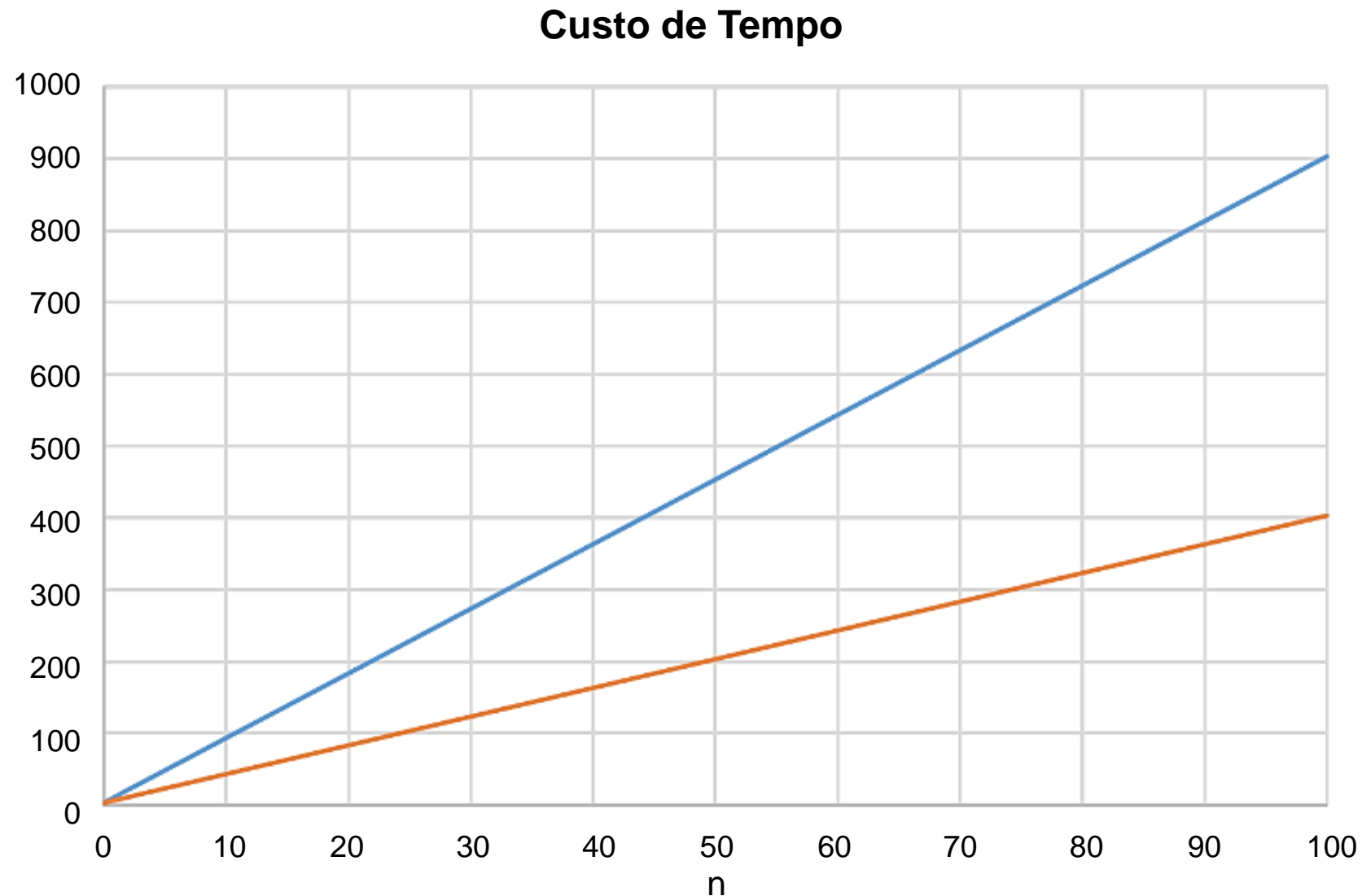
# Conceito de análise de algoritmos

Melhor caso:

- $\{4,3,2,1\}$

Pior caso:

- $\{1,2,3,4\}$



# Conceito de análise de algoritmos

## Análise assintótica

- A análise assintótica é feita tentando extrapolar o conjunto de dados de entrada tendendo-os ao infinito e tomando a liberdade de desprezar os outros. Dessa forma, se considerarmos o melhor caso temos:
- $T(n) = 4n + 3 \rightarrow T(n) = n$
- $n^3$

```
1 #include <stdio.h>
2 int main()
3 {
4     int v[4] = { 4, 3, 2, 1 }, z;
5     z = 10000;
6     for (int i = 0; i < z; i++)
7     {
8         for (int j = 0; i < z; i++)
9         {
10             for (int k = 0; i < z; i++)
11             {
12                 //laço
13             }
14         }
15     }
16     for (int i = 0; i < z; i++)
17     {
18         //laço
19     }
20     return 0;
21 }
```

# Interatividade

Assinale a alternativa correta.

- a) Somente 1 tem o comportamento assintótico  $n^2$ .
- b) Somente 2 tem o comportamento assintótico  $n^2$ .
- c) Somente 3 tem o comportamento assintótico  $n^2$ .
- d) 2 e 3 têm o comportamento assintótico  $n^2$ .
- e) Nenhum tem o comportamento assintótico  $n^2$ .

1

```
for (int i=0;i<k;i++)  
    <instruções>  
for (int i=0;i<k;i++)  
    <instruções>
```

2

```
for (int j=0;j<1;j++){  
    for (int i=0;i<0;i++)  
        <instruções>  
    <instruções>  
}
```

3

```
while (x<y){  
    for (int i=0;i<k;i++)  
        <instruções>  
    <instruções>  
}
```

# Resposta

Assinale a alternativa correta.

- a) Somente 1 tem o comportamento assintótico  $n^2$ .
- b) Somente 2 tem o comportamento assintótico  $n^2$ .
- c) Somente 3 tem o comportamento assintótico  $n^2$ .
- d) 2 e 3 têm o comportamento assintótico  $n^2$ .**
- e) Nenhum tem o comportamento assintótico  $n^2$ .

1

```
for (int i=0;i<k;i++)  
    <instruções>  
for (int i=0;i<k;i++)  
    <instruções>
```

2

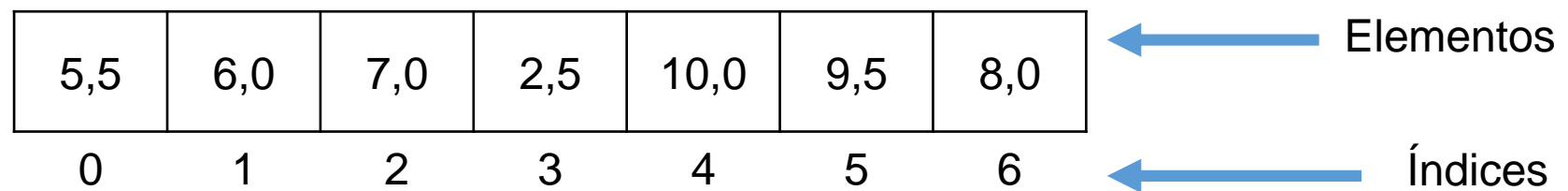
```
for (int j=0;j<1;j++){  
    for (int i=0;i<0;i++)  
        <instruções>  
    <instruções>  
}
```

3

```
while (x<y){  
    for (int i=0;i<k;i++)  
        <instruções>  
    <instruções>  
}
```

# Revisão de arranjos

- Segundo Laureano (2008, p. 2):
- O vetor é uma estrutura de dados linear que necessita de somente um índice para que seus elementos sejam endereçados.
- É utilizado para armazenar uma lista de valores do mesmo tipo, ou seja, o tipo vetor permite armazenar mais de um valor em uma mesma variável.
- Um dado vetor é definido como tendo um número fixo de células idênticas (seu conteúdo é dividido em posições).
  - Cada célula armazena um e somente um dos valores de dados do vetor. Cada uma das células de um vetor possui seu próprio endereço, ou índice, através do qual pode ser referenciada.



Vetor: v



# Representação linear de matrizes

- Matrizes.

```
float classe[3][4] = {{ 8.5, 7.0, 8.5, 10.0}  
                      ,{ 3.0, 4.0, 6.0, 5.5}  
                      ,{ 7.0, 7.5, 6.0, 5.0}};
```

```
float classe [3][4] = { 8.5, 7.0, 8.5, 10.0, 3.0, 4.0, 6.0, 5.5, 7.0, 7.5, 6.0, 5.0};
```

```
float classe[][4] = { 8.5, 7.0, 8.5, 10.0, 3.0, 4.0, 6.0, 5.5, 7.0, 7.5, 6.0, 5.0};
```

# Operações com cadeias

- As cadeias de caracteres em C (Strings) são representadas por vetores do tipo char terminadas, obrigatoriamente, pelo caractere nulo ('\0').
- Sempre que ocorre o armazenamento de uma cadeia**, é necessário reservar um elemento adicional para o caractere de fim da cadeia.

```
int main()
{
    char faculdade[5];
    faculdade [0] = 'U';
    faculdade [1] = 'n';
    faculdade [2] = 'i';
    faculdade [3] = 'p';
    | faculdade [4] = '\0';
    printf("%s \n", faculdade);
}
```

**String**

```
int main()
{
    char faculdade[5];
    faculdade [0] = 'U';
    faculdade [1] = 'n';
    faculdade [2] = 'i';
    faculdade [3] = 'p';
    printf("%s \n", faculdade);
}
```

**não String**

# Modularização

- No uso das funções, pode-se dividir grandes tarefas de computação em tarefas menores.
- A criação de funções evita a repetição de código.
- Quando um trecho do programa é repetido diversas vezes, deve ser transformado em uma função.

## Sem modularização

aaaaaaaaaaaaaaaaaaaaaaaaa  
bbbbbbbbbbbbbbbbbbbbbb  
cccccccccccccccccccccc  
dddddddddddddddddddddd  
eeeeeeeeeeeeeeeeeeeeee  
fffffffffffffffffffffffff  
gggggggggggggggggggggg  
hhhhhhhhhhhhhhhhhhhhh  
iiiiiiiiiiiiiiiiiiiiiii  
dddddddddddddddddddddd  
eeeeeeeeeeeeeeeeeeeeee  
fffffffffffffffffffffffff  
jjjjjjjjjjjjjjjjjjjjjj  
kkkkkkkkkkkkkkkkkkkkk  
dddddddddddddddddddddd  
eeeeeeeeeeeeeeeeeeeeee  
fffffffffffffffffffffffff

## Modularizando

aaaaaaaaaaaaaaaaaaaaaaaaa  
bbbbbbbbbbbbbbbbbbbbbb  
cccccccccccccccccccccc  
Procedimento a  
gggggggggggggggggggggg  
hhhhhhhhhhhhhhhhhhhhh  
iiiiiiiiiiiiiiiiiiiiiii  
Procedimento a  
jjjjjjjjjjjjjjjjjjjjjj  
kkkkkkkkkkkkkkkkkkkkk  
Procedimento a

## Procedimento a

dddddddddddddddddddddd  
eeeeeeeeeeeeeeeeeeeeee  
fffffffffffffffffffffffff

# Procedimentos e funções

## Função:

```
<tipo de retorno> nome( <tipo1> var1, <tipo2> var2){  
    Corpo da função  
    return retorno;  
}
```

```
#include <stdio.h>  
int fat(int n){  
    int f=1;  
    for (int i=1;i<=n;i++)  
        f*=i;  
    return f;  
}
```

## Procedimento:

- A diferença é que o procedimento tem *void* (nulo)
- É Desnecessário o *return* dentro do bloco da função.

```
Void nome( <tipo1> var1, <tipo2> var2){  
    Corpo da função  
  
}
```

# Tipo estrutura

- Uma estrutura serve basicamente para agrupar diversas variáveis dentro de um único contexto.

```
struct ponto{  
    float x;  
    float y;  
};
```

```
int main()  
{  
    struct ponto p;  
    p.x=10.0;  
    p.y=5.0;  
}
```

# Definição de “novos” tipos

- É possível criar novos tipo a partir de uma estrutura, ou mesmo renomear tipos existentes

```
typedef struct ponto{  
    float x;  
    float y;  
}Ponto;  
  
int main()  
{  
    Ponto p;  
    p.x=10.0;  
    p.y=5.0;  
}
```

```
#include <stdio.h>  
struct ponto {  
    float x;  
    float y;  
};  
typedef struct ponto Ponto;  
int main()  
{  
    Ponto a,b;  
    a.x = 10.0;  
    a.y = 5.0;  
    b.x = 1.0;  
    b.y = 2.0;  
    return 0;  
}
```

# Tipos abstratos de dados

- Conjunto de funções que operam sobre uma estrutura:
- O conceito de tipo de dado abstrato é dissociado do *hardware*.
- TAD define o que cada operação faz, mas não como faz.
- Uma boa técnica de programação é implementar os TADs em arquivos separados do programa principal. Para isso, geralmente separa-se a declaração e a implementação do TAD em dois arquivos:
  - NomeDoTAD.h: com a declaração
  - NomeDoTAD.c: com a implementação

# Conceitos de TAD cadeias

- Para manipular *strings*, a linguagem C oferece o TAD *strings.h*.

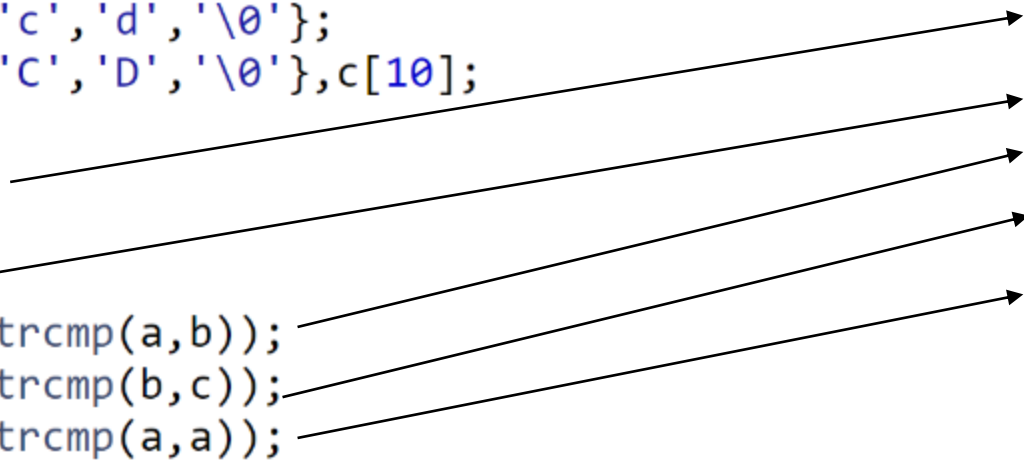
Alguns dos principais procedimentos:

- Função *strcat()*: (**string concatenate**) A função recebe duas *strings* como argumento e copia a segunda *string* no final da primeira .
- Função *strcpy()*: (**string copy**) A função recebe duas *strings* como argumento e copia na primeira *string* a segunda *string*. Na prática é a atribuição de valores.
  - Função *strcmp()*: (**string compare**) Duas cadeias são comparadas. A comparação é feita caractere a caractere, até encontrar a primeira diferença entre eles; conforme a diferença, a função devolve um valor diferente, usando o seguinte critério:
    - $< 0$ , se *cadeia1*  $<$  *cadeia2*;
    - $= 0$ , se *cadeia1*  $=$  *cadeia2*;
    - $> 0$ , se *cadeia1*  $>$  *cadeia2*.



# Conceitos de TAD cadeias

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[5]={'a','b','c','d','\0'};
    char b[5]={'A','B','C','D','\0'},c[10];
    strcpy(c,b);
    printf("c=%s\n",c);
    strcat(c,a);
    printf("c=%s\n",c);
    printf("a b=%d\n",strcmp(a,b));
    printf("b a=%d\n",strcmp(b,c));
    printf("a a=%d\n",strcmp(a,a));
}
```



The diagram shows four arrows originating from the code on the left and pointing to the output on the right. The first arrow points from the first `printf` statement to the first output line. The second arrow points from the `strcat` statement to the second output line. The third arrow points from the `printf` statement following `strcat` to the third output line. The fourth arrow points from the `printf` statement for `strcmp(a,a)` to the fourth output line.

c=ABCD  
c=ABCDabcd  
a b=32  
b a=-97  
a a=0

# Interatividade

Dado o programa: Qual das alternativas resulta na figura abaixo?

```
#include <stdio.h>
#include <string.h>
typedef struct aluno{
    char nome[20];
    float nota[3];
}Aluno;
int main()
{
    Aluno Sala[5];
    /*
    numero do aluno=3;
    nome=Jeverton;
    nota da primeira prova=7
    */
}
```

Jeverton 7.000000

- a) strcpy(Aluno[3].nome, "Jeverton\0");  
Aluno[3].nota[1]=7;  
printf("%s %f",Aluno[3].nome,Aluno[3].nota[1]);
- b) strcpy(Aluno.nome, "Jeverton\0");  
Aluno.nota[1]=7;  
printf("%s %f",Aluno.nome,Aluno.nota[1]);
- c) strcpy(Aluno[1].nome, "Jeverton\0");  
Aluno[1].nota[3]=7;  
printf("%s %f",Aluno[1].nome,Aluno[1].nota[3]);
- d) strcpy(Sala.nome, "Jeverton\0");  
Sala.nota[1]=7;  
printf("%s %f",Sala.nome,Sala.nota[1]);
- e) strcpy(Sala[3].nome, "Jeverton\0");  
Sala[3].nota[1]=7;  
printf("%s %f",Sala[3].nome,Sala[3].nota[1]);

# Resposta

Dado o programa: Qual das alternativas resulta na figura abaixo?

```
#include <stdio.h>
#include <string.h>
typedef struct aluno{
    char nome[20];
    float nota[3];
}Aluno;
int main()
{
    Aluno Sala[5];
    /*
    numero do aluno=3;
    nome=Jeverton;
    nota da primeira prova=7
    */
}
```

Jeverton 7.000000

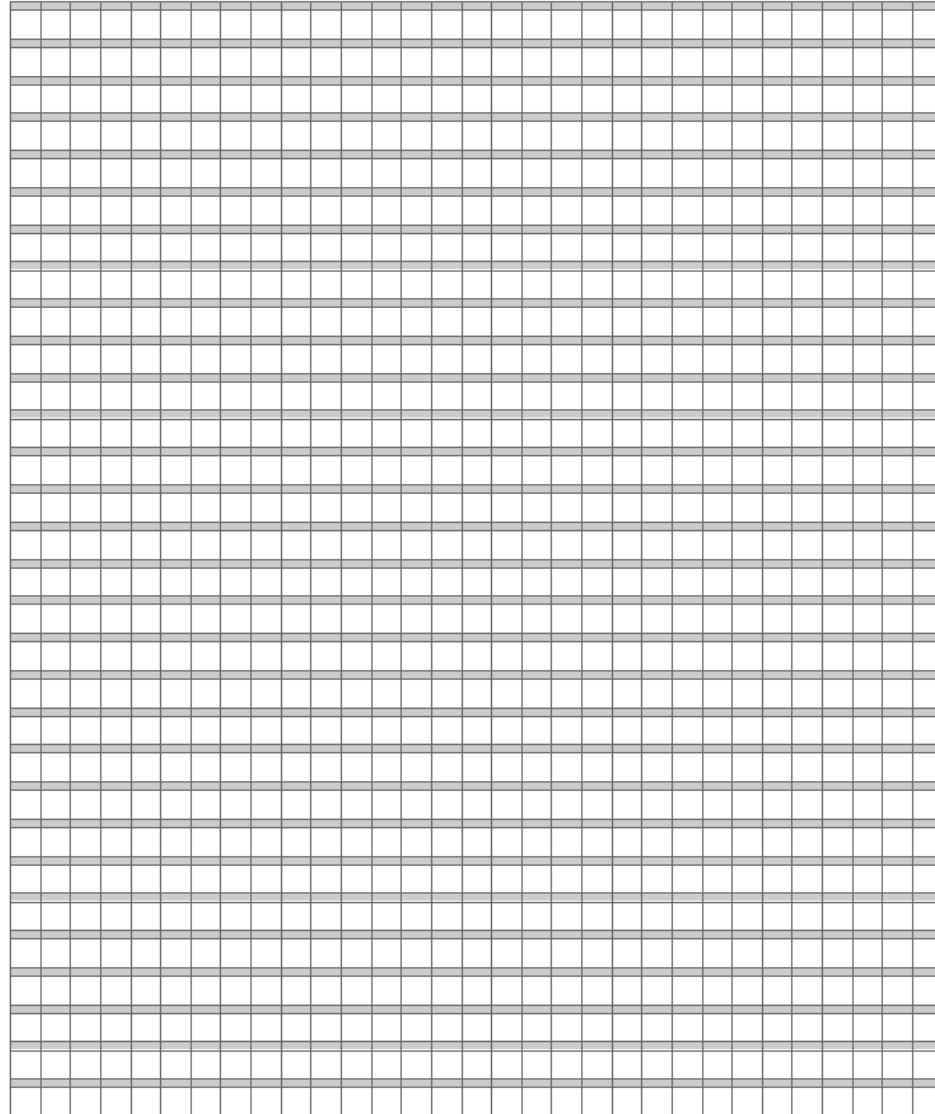
- a) strcpy(Aluno[3].nome, "Jeverton\0");  
Aluno[3].nota[1]=7;  
printf("%s %f",Aluno[3].nome,Aluno[3].nota[1]);
- b) strcpy(Aluno.nome, "Jeverton\0");  
Aluno.nota[1]=7;  
printf("%s %f",Aluno.nome,Aluno.nota[1]);
- c) strcpy(Aluno[1].nome, "Jeverton\0");  
Aluno[1].nota[3]=7;  
printf("%s %f",Aluno[1].nome,Aluno[1].nota[3]);
- d) strcpy(Sala.nome, "Jeverton\0");  
Sala.nota[1]=7;  
printf("%s %f",Sala.nome,Sala.nota[1]);
- e) strcpy(Sala[3].nome, "Jeverton\0");  
Sala[3].nota[1]=7;  
printf("%s %f",Sala[3].nome,Sala[3].nota[1]);

# Alocação dinâmica de memória: ponteiros

- O uso da memória:
- Três maneiras de reservarmos espaço de memória para o armazenamento de informações.
  - Uso de variáveis globais (e estáticas). Nesta categoria de variáveis, o espaço reservado para uma variável existirá enquanto o programa estiver sendo executado.
  - Uso de variáveis locais. Nesta categoria, o espaço na memória existe apenas no período em que a função que declarou a variável está sendo executada, sendo liberado assim que a execução da função terminar.
  - Reservar a memória é solicitar ao programa que aloque dinamicamente um espaço na memória durante sua execução.

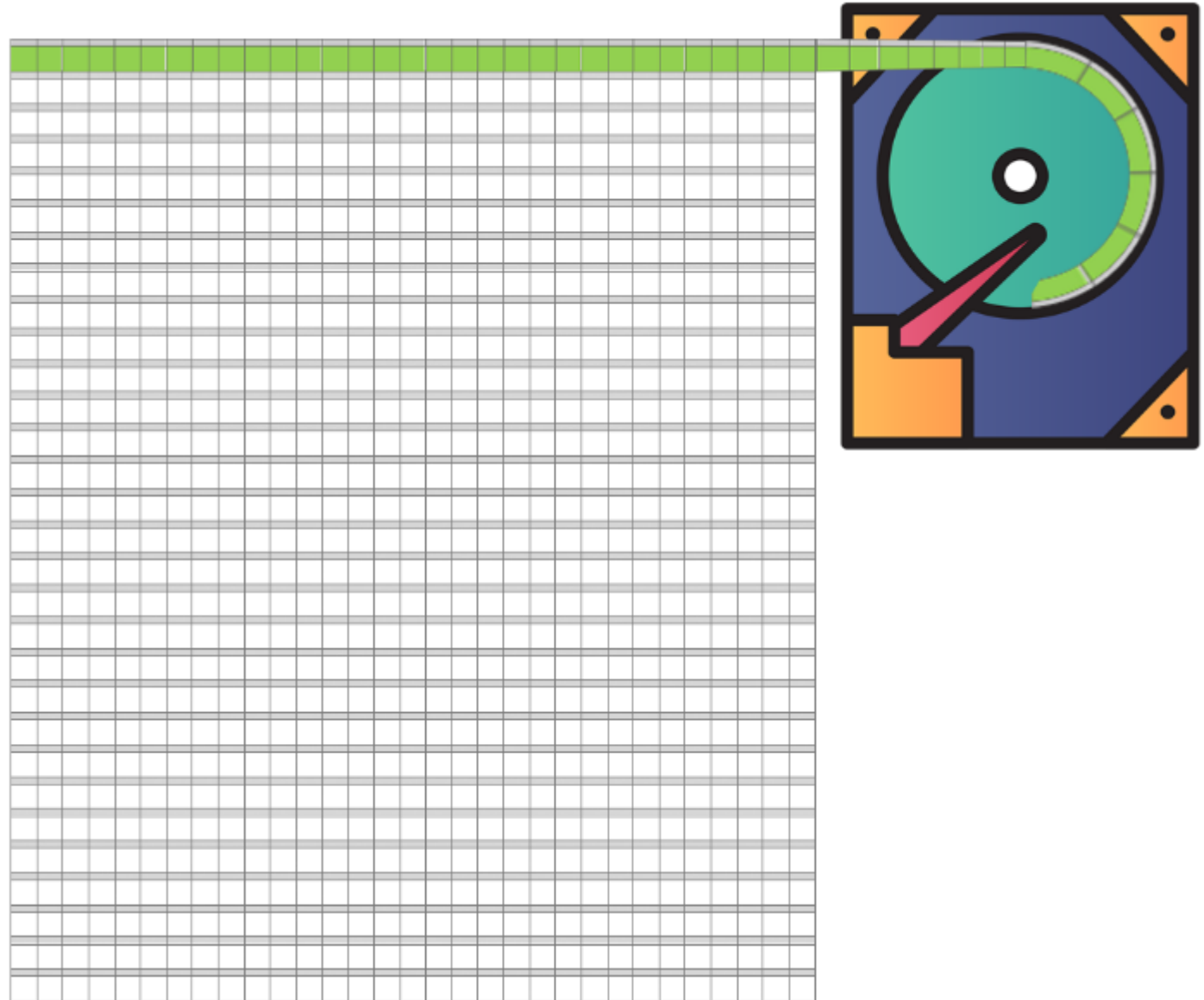
# Alocação dinâmica de memória: ponteiros

- O uso da memória:



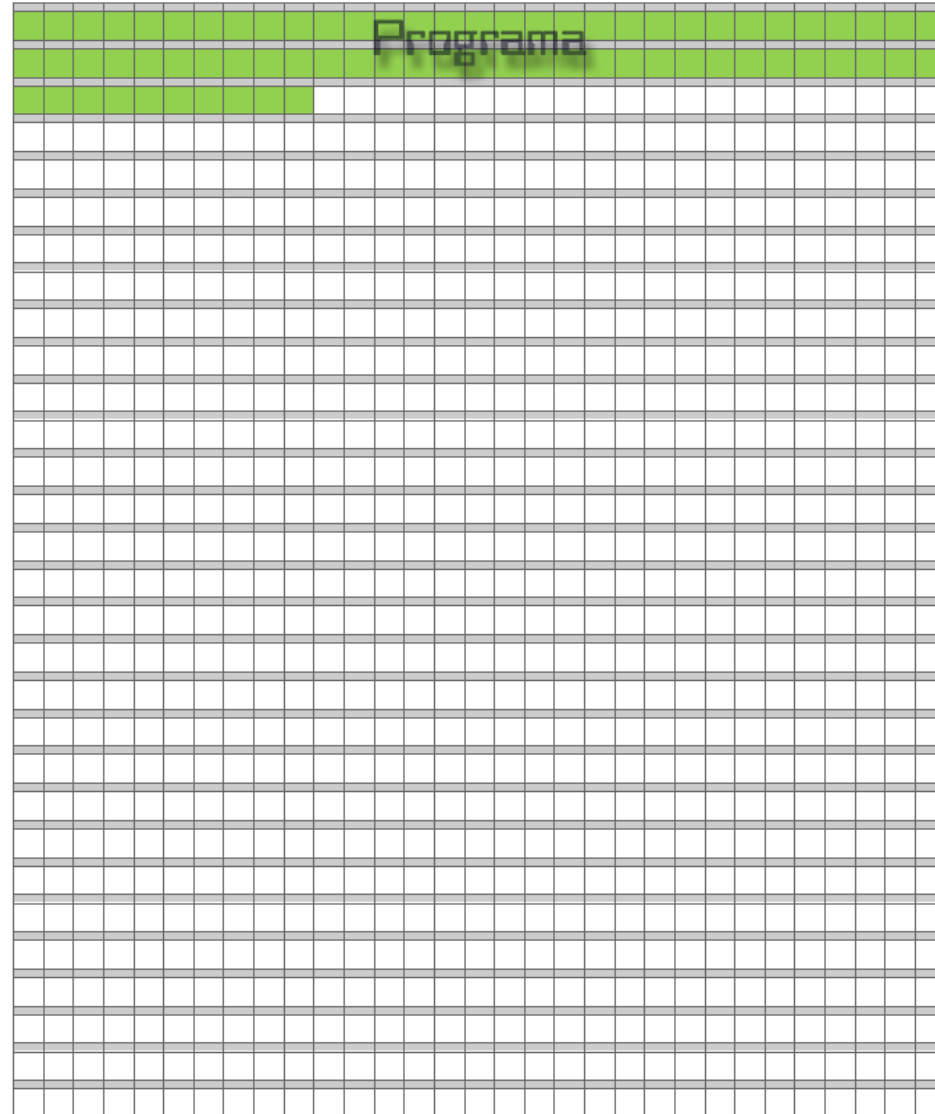
# Alocação dinâmica de memória: ponteiros

- O uso da memória:



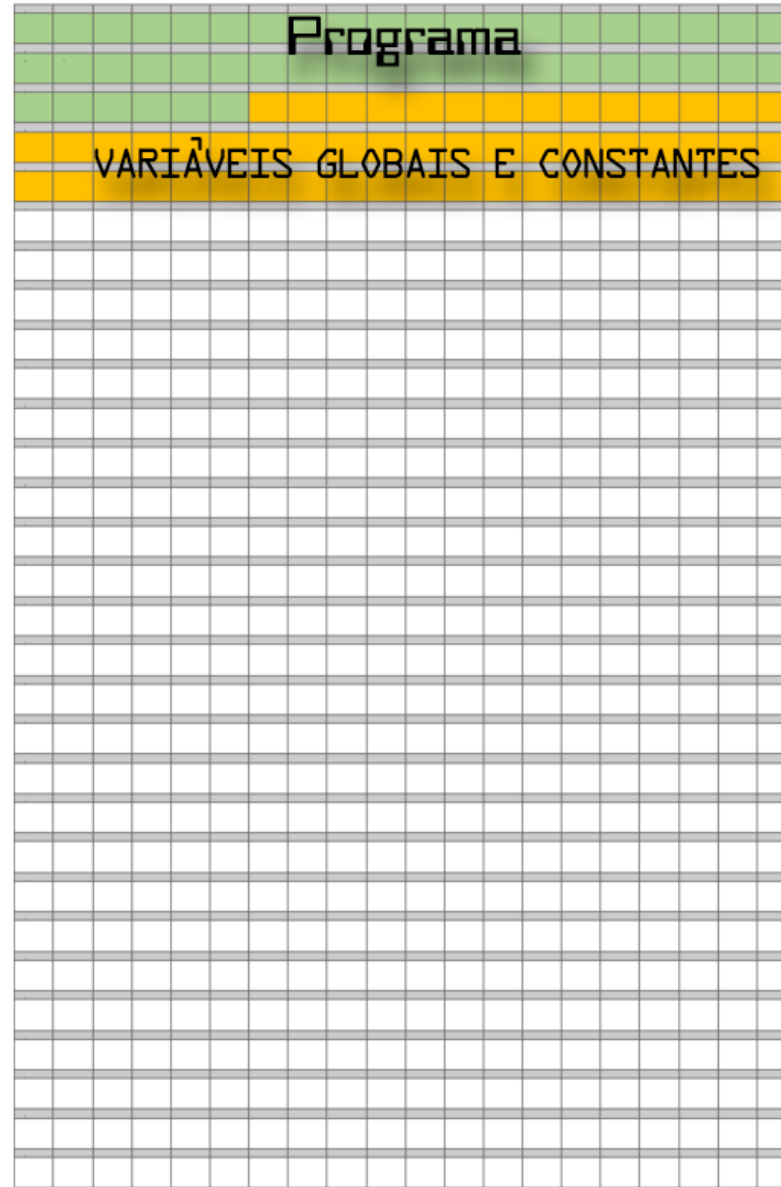
# Alocação dinâmica de memória: ponteiros

- O uso da memória:



# Alocação dinâmica de memória: ponteiros

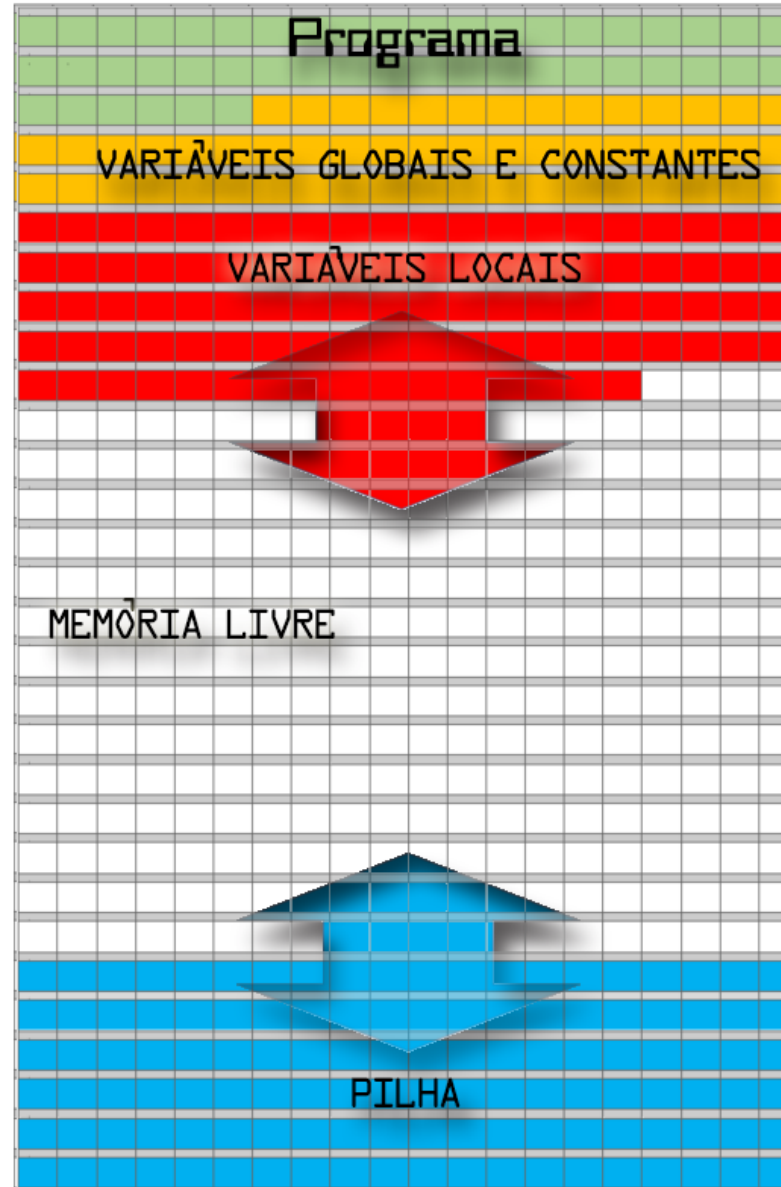
- O uso da memória:





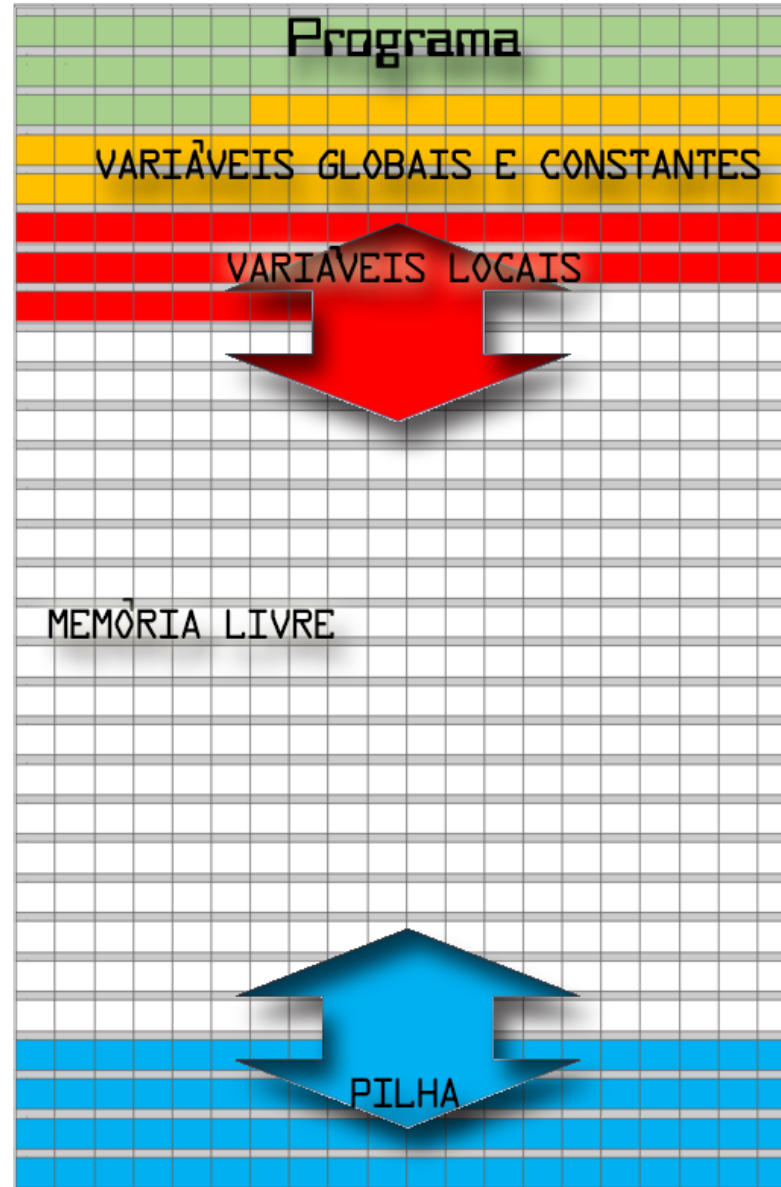
# Alocação dinâmica de memória: ponteiros

- O uso da memória:



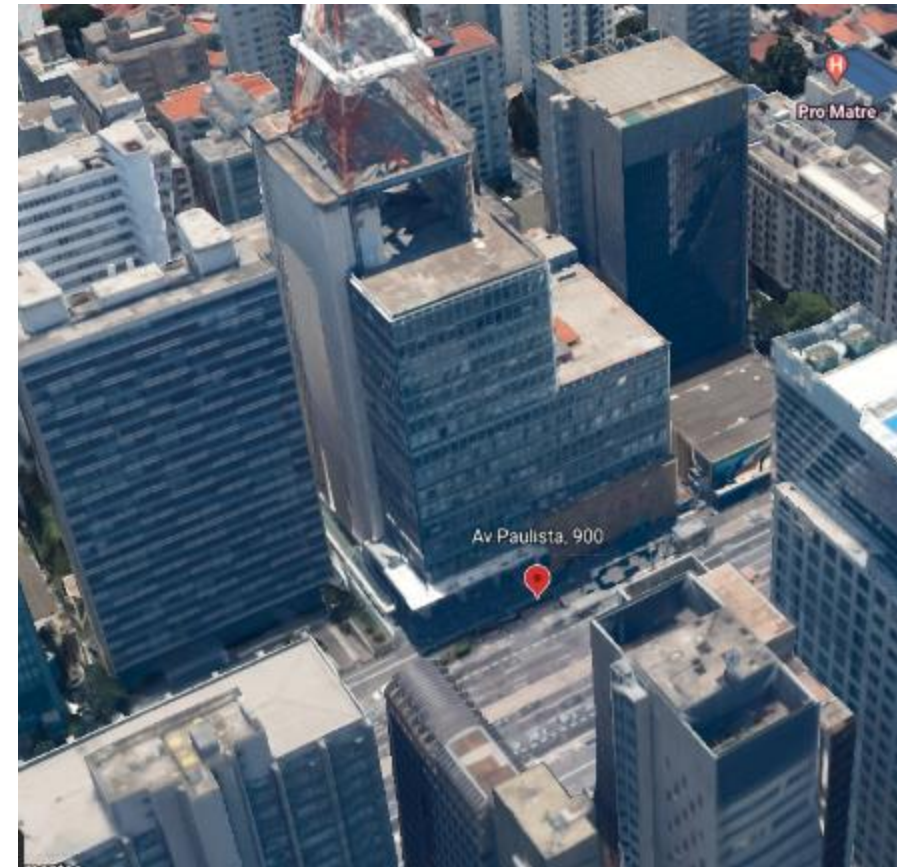
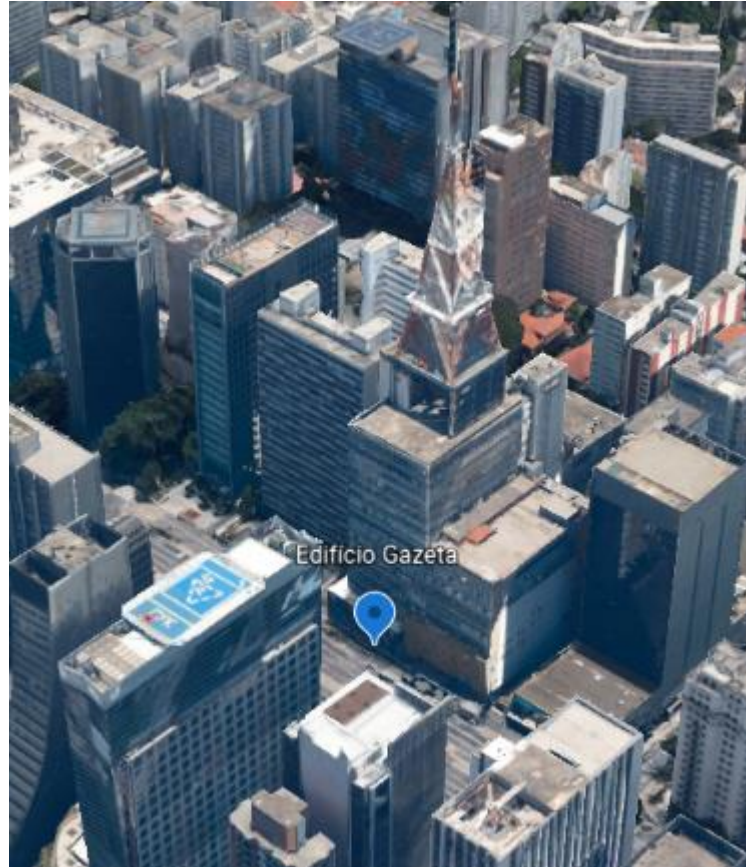
# Alocação dinâmica de memória: ponteiros

- O uso da memória:



# Ponteiro de variáveis

- Dualidade, nome e endereço:
- Edifício Gazeta;
- Av. Paulista, 900;
- Ambos chegam ao mesmo lugar.



Fonte: earth.google.com

# Ponteiro de variáveis

- Como funciona?

# Ponteiro de variáveis

- Como funciona?
- Memória limpa.
- Tabela de variáveis.

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																				
nome																				
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome																				
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis		
Nome	endereço	tipo

# Ponteiro de variáveis

■ Como funciona?

```
int idade;
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																				
nome																				
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome																				
endereço	141	142	143	144	145	146											157	158	159	160
conteúdo																				
nome																				

espaço livre para ocupar 4 bytes

Tabela das variáveis		
Nome	endereço	tipo

# Ponteiro de variáveis

- Como funciona?

```
int idade;
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																				
nome																				
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome																				
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis		
Nome	endereço	tipo
idade	125	int

# Ponteiro de variáveis

- A linguagem C tem uma maneira especial de uma variável armazenar endereços. Essa variável se chama variável ponteiro ou simplesmente ponteiro.

**<tipo> \*nome;**

Exemplo:

**int \*p;**

Para acessar os endereços de memória, a linguagem oferece dois operadores unários:

- & (“endereço de”), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável;
- \* (“conteúdo de”), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro.



# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																				
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome																				
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis		
Nome	endereço	tipo
a	117	int

# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																				
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome	p																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis		
Nome	endereço	tipo
a	117	int
* p	125	int

# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																	5			
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome	p																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis

Nome	endereço	tipo
a	117	int
* p	125	int

# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																	5			
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo					&117															
nome	p																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis

Nome	endereço	tipo
a	117	int
* p	125	int

# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																		5	6	
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo					&117															
nome	p																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis

Nome	endereço	tipo
a	117	int
* p	125	int

# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																	6			
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo					&117															
nome	p																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis

Nome	endereço	tipo
a	117	int
* p	125	int



# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																	6			
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo					&117															
nome	p b																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis		
Nome	endereço	tipo
a	117	int
* p	125	int
b	136	int

# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																	6			
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo					&117 &136															
nome	p b																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis

Nome	endereço	tipo
a	117	int
* p	125	int
b	136	int



# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																		6		
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo					&136												6			
nome	p																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome	b																			

Tabela das variáveis

Nome	endereço	tipo
a	117	int
* p	125	int
b	136	int

# Ponteiro de variáveis

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *p;
6     a=5;
7     p = &a;
8     *p = 6;
9     printf("%d",a);
10    int b;
11    p=&b;
12    *p=a;
13    printf("%d",b);
14 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																	6			
nome	a																			
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo					&136											6				
nome	p b																			
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis		
Nome	endereço	tipo
a	117	int
* p	125	int
b	136	int

6

# Interatividade

Ao se executar o programa abaixo, qual a sua saída?

- a) 1,2.
- b) 3,1.
- c) 1,3.
- d) 2,1.
- e) 2,3.

```
#include <stdio.h>
int main()
{
    int a=1, b=2, c=3;
    int *d=&a;
    int *e=&b;
    int *f;
    f=e;
    c=*d;
    a=*e;
    *f=c;
    printf("%d %d\n",a,b);
}
```

# Resposta

Ao se executar o programa abaixo, qual a sua saída?

- a) 1,2.
- b) 3,1.
- c) 1,3.
- d) 2,1.
- e) 2,3.

```
#include <stdio.h>
int main()
{
    int a=1, b=2, c=3;
    int *d=&a;
    int *e=&b;
    int *f;
    f=e;
    c=*d;
    a=*e;
    *f=c;
    printf("%d %d\n",a,b);
}
```

# Funções da biblioteca-padrão

- Muitas vezes, o uso de vetores e matrizes fica limitado pela necessidade de sabermos antecipadamente a quantidade de elementos que serão necessários.
- A biblioteca *stdlib.h* possui algumas funções que permitem criar e trabalhar dinamicamente.

```
int v[10];
```

Criando-se dinamicamente:

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

# Funções da biblioteca-padrão

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int * v; ←
6     v=(int *)malloc(10*sizeof(int));
7     v[0]=13;
8     v[1]=23;
9     printf("v[0]=%d v[1]=%d",v[0],v[1]);
10    return 0;
11 }
```

Memória RAM																				
endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																				
nome																				
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome																				
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis		
Nome	endereço	tipo
*v	101	int

# Funções da biblioteca-padrão

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int * v;
6     v=(int *)malloc(10*sizeof(int)); ←
7     v[0]=13;
8     v[1]=23;
9     printf("v[0]=%d v[1]=%d",v[0],v[1]);
10    return 0;
11 }
```

Memória RAM

endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
conteúdo																				
nome																				
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
conteúdo																				
nome																				
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
conteúdo																				
nome																				

Tabela das variáveis

Nome	endereço	tipo
*v	101	int

- `sizeof(<tipo>)`

Tamanho do tipo
- `sizeof(int)`

Tamanho do tipo int
- `10 * sizeof(int)`

Tamanho de 10 inteiros
- `malloc(10 * sizeof(int))`

Aloca o espaço de 10 inteiros

# Funções da biblioteca-padrão

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int * v;
6     v=(int *)malloc(10*sizeof(int)); ←
7     v[0]=13;
8     v[1]=23;
9     printf("v[0]=%d v[1]=%d",v[0],v[1]);
10    return 0;
11 }
```

Memória RAM																					
endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	
conteúdo	&103																				
nome																					
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	
conteúdo																					
nome																					
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	
conteúdo																					
nome																					

Tabela das variáveis		
Nome	endereço	tipo
*v	101	int

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



# Funções da biblioteca-padrão

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int * v;
6     v=(int *)malloc(10*sizeof(int));
7     v[0]=13;
8     v[1]=23;
9     printf("v[0]=%d v[1]=%d",v[0],v[1]);
10    return 0;
11 }
```

Memória RAM																									
endereço	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120					
conteúdo	&103		13				23																		
nome	*v		v[0]				v[1]				v[2]				v[3]				v[4]						
endereço	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140					
conteúdo																									
nome	v[5]					v[6]					v[7]					v[8]					v[9]				
endereço	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160					
conteúdo																									
nome																									

Tabela das variáveis		
Nome	endereço	tipo
*v	101	int

# Aritmética de ponteiros

- O ponteiro também tem uma aritmética própria.
- Ao fazermos uma soma de um número inteiro a um ponteiro, ele apontará para o endereço com o avanço de múltiplos correspondente ao tamanho do tipo definido para ele.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      int * v;
6      v=(int *)malloc(10*sizeof(int));
7      v[0]=13;
8      v[1]=23;
9      printf("v[0]=%d v[1]=%d",v[0],v[1]);
10     return 0;
11 }
```

Expressão	Ao fazer a expressão	Equivale a fazer a expressão
<code>int *x = v+0</code>	<code>printf("%d",*x)</code>	<code>printf("%d",v[0])</code>
<code>int *x = v+1</code>	<code>printf("%d",*x)</code>	<code>printf("%d",v[1])</code>
<code>int *x = v+2</code>	<code>printf("%d",*x)</code>	<code>printf("%d",v[2])</code>
<code>int *x = v+3</code>	<code>printf("%d",*x)</code>	<code>printf("%d",v[3])</code>
<code>int *x = v+4</code>	<code>printf("%d",*x)</code>	<code>printf("%d",v[4])</code>

# Interatividade

Dado o programa:

```
#include <stdio.h>
#include<stdlib.h>
int main()
{
    int v[10]={1,2,3,4,5,6,7,8,9,10};
    int *p=v;
    printf("%d",*p+4);
    return 0;
}
```

Assinale a alternativa correta.

- a) Apresenta-se o endereço do vetor v na memória RAM somado de 4 unidades.
- b) Mostra-se na tela o valor 4.
- c) Mostra-se na tela o valor 5.
- d) Apresenta-se a cadeia %d na tela.
- e) Acontece erro, pois o formato de ponteiro é %x.

# Resposta

Dado o programa:

```
#include <stdio.h>
#include<stdlib.h>
int main()
{
    int v[10]={1,2,3,4,5,6,7,8,9,10};
    int *p=v;
    printf("%d",*p+4);
    return 0;
}
```

Assinale a alternativa correta.

- a) Apresenta-se o endereço do vetor v na memória RAM somado de 4 unidades.
- b) Mostra-se na tela o valor 4.
- c) Mostra-se na tela o valor 5.**
- d) Apresenta-se a cadeia %d na tela.
- e) Acontece erro, pois o formato de ponteiro é %x.

# Referências

- ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. Rio de Janeiro: Editora Prentice Hall, 2003.
- CORMEN, T. H.; RIVEST, R. L.; LEISERSON, C. E.; STEIN, C. *Algoritmos – teoria e prática*. 3. ed. Rio de Janeiro: Editora Campus, 2012.
- FORBELLONE, A. L. V. *Lógica de programação – a construção de algoritmos e estruturas de dados*. São Paulo: Makron Books, 1993.
- LAUREANO, M. *Estrutura de dados com algoritmos e C*. Rio de Janeiro: Brasport, 2008.
- MANZANO, J. N. G.; OLIVEIRA, J. F. *Algoritmos – lógica para desenvolvimento e programação*. São Paulo: Erica, 1996.

**ATÉ A PRÓXIMA!**