

# Algoritmos e Programação

**1º Ano - 1º Semestre**

***2. Programação numa linguagem de alto nível (C) – Parte 4***

**Escola Superior de Tecnologia e Gestão de Viseu 2020-21**

**Agradecimentos a Francisco Morgado e Carlos Simões**

## **2. Programação numa linguagem de alto nível (C) – Part 4**

**2.1 Introdução**

**2.2 Estrutura de um programa em linguagem C**

**2.3 Tipos de dados básicos**

**2.4 Testes e condições - expressões e operadores. Precedências**

**2.5 Estruturas de repetição**

**2.6 Funções**

**2.7 Tipos de dados estruturados: vectores e strings**

**2.8 Apontadores (Pointers)**

**2.9 Passagem de parâmetros de tipos estruturados**

**2.10 Estruturas**

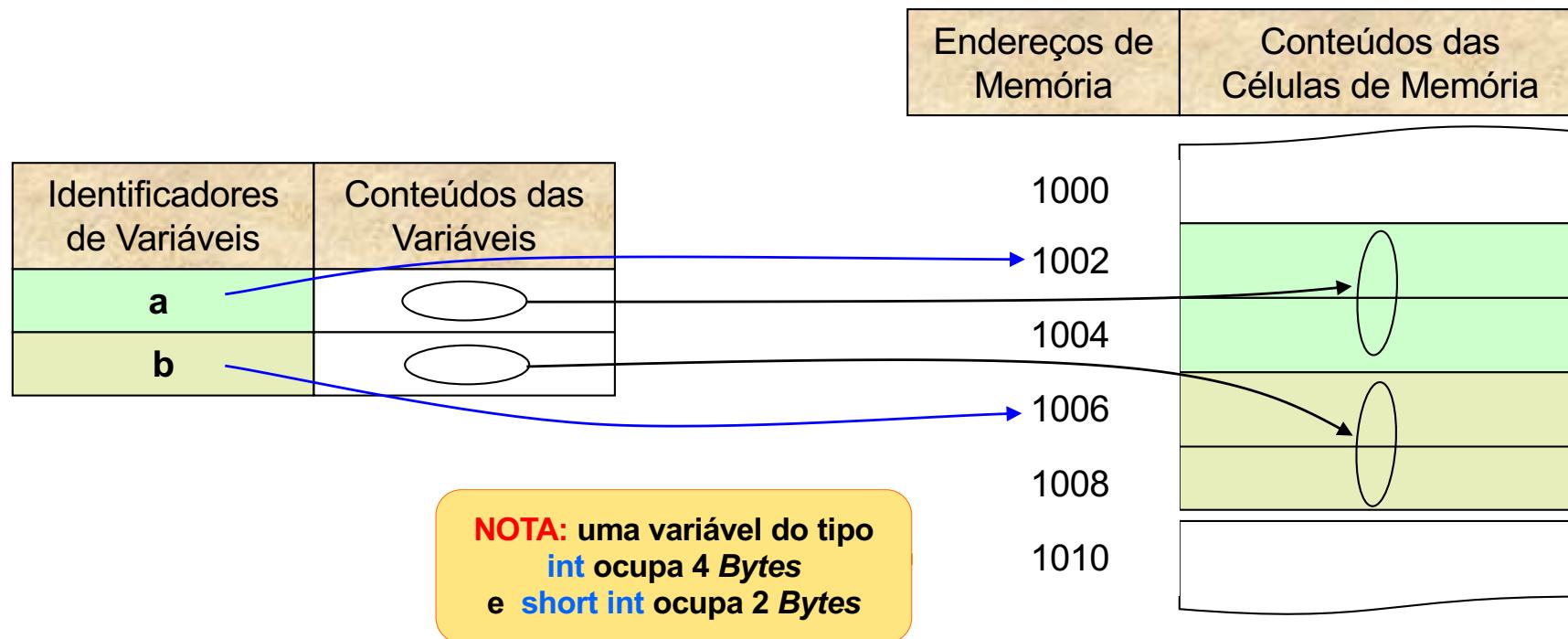
**2.11 Memória dinâmica**

## 2.8 Apontadores (Pointers)

- Noção de Variável, Endereço e Apontador
- Operadores \*(apontado por) e &(endereço de)
- Noção de *NULL*
- Declaração e inicialização de variáveis do tipo Apontador
- Aritmética de apontadores
- Apontadores e vectores
- Passagem de vectores para funções
- Apontadores de apontadores

## Noção de variável, Endereço e Apontador

Como vimos na primeira aula de Algoritmos e Programação, quando declaramos uma variável estamos a alocar-lhe espaço na memória, em função do seu tipo.



Em C existe o operador `&` que permite saber qual o endereço de uma variável

- O compilador associa a cada variável uma posição ÚNICA em memória, capaz de suportar os dados do tipo dessa variável. Sempre que num programa fazemos referência a uma variável, estamos na realidade a referir o conteúdo do endereço ou conjunto de endereços que essa variável ocupa.

**Um apontador é uma variável que aponta sempre para outra variável, de determinado tipo.**

Para indicar que uma variável é um **apontador**, coloca-se antes dela um asterisco (\*).

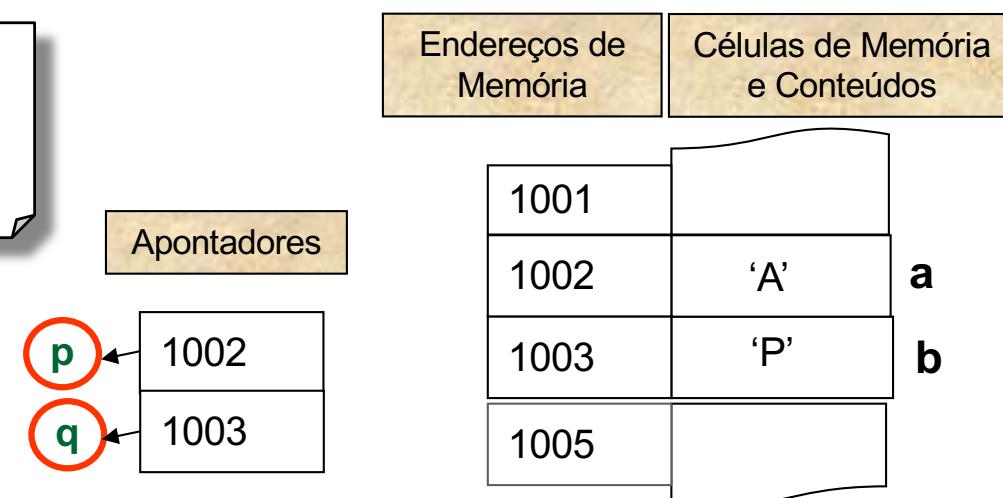
### SINTAXE DE DECLARAÇÃO:

**tipo \*ptr**

- ptr - identificador da variável
- tipo - tipo da variável para a qual apontará
- \* - indica que é uma variável do tipo apontador

### Exemplo:

```
char a, b, *p = &a, *q = &b;
```



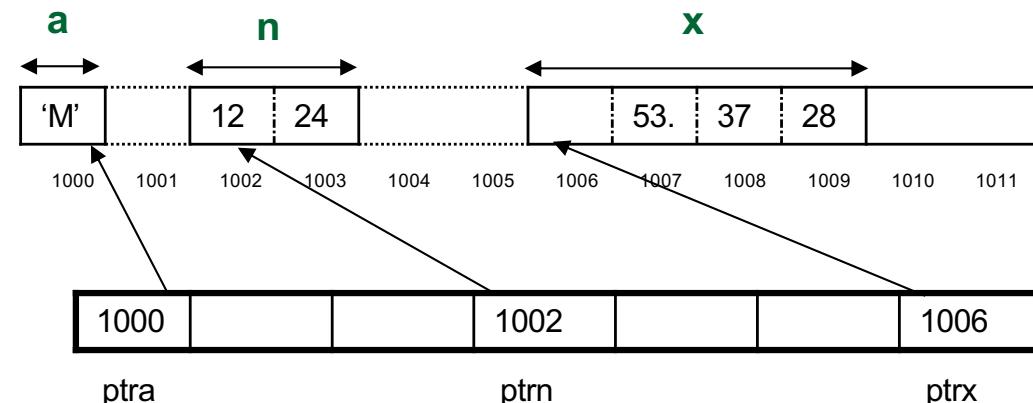
# Apontadores e tipos de dados

Um apontador tem de possuir um determinado tipo: o tipo da variável para a qual aponta

O endereço de uma variável é sempre o menor dos endereços que ela ocupa em memória

Pelo facto de as variáveis ocuparem diferentes tamanhos em memória, os apontadores para essas variáveis necessitam de saber quantos Bytes de memória terão de considerar

```
char *ptrA;  
short int *ptrN;  
float *ptrX;
```

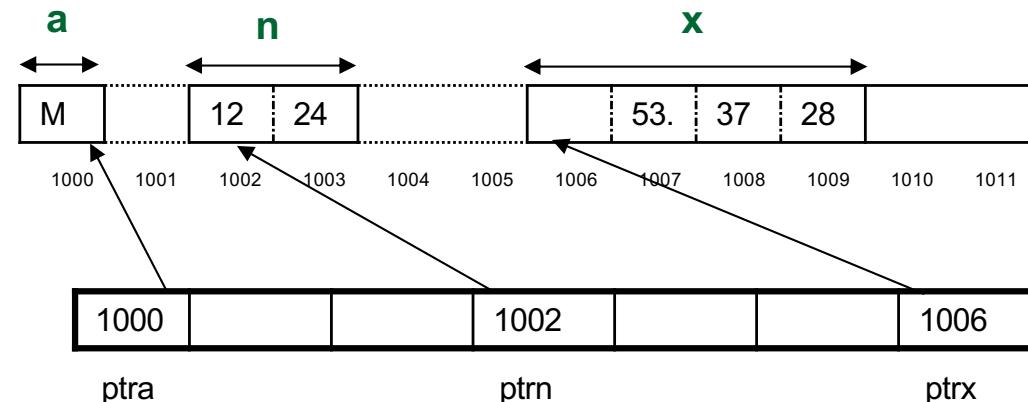


## INICIALIZAÇÃO

A inicialização de um apontador faz-se, geralmente, através do operador **&** (Endereço de).

### EXEMPLO

```
char *ptrA = &a;  
int *ptrN = &n;  
float *ptrX = &x;
```



Um apontador para o tipo xyz endereça sempre o número de bytes que esse tipo ocupa em memória (portanto, endereça **sizeof(xyz)**).

**NOTA:** o operador **sizeof** devolve o número de Bytes ocupado por uma variável ou por um determinado tipo, numa dada arquitectura.

**Sintaxe:** **sizeof <expressão>** ou **sizeof(<tipo>)**

## Noção de NULL

Um apontador pode ser declarado e não ficar a apontar para qualquer variável.

A constante simbólica **NULL**, quando colocada num apontador, indica que este não aponta para nenhuma variável.

### EXEMPLO

```
...
int n = 8;
int *ptr1 = NULL; // ptr1 não aponta para nada
float y = 2.4;
float *ptr2 = &y;
printf("%.1f\n", *ptr2); // equivale a printf("%.1f", y);

ptr1= &n;
printf ("%d\n", *ptr1); // qual será o resultado desta instrução?
...
```

**NOTA:** a declaração de um apontador para variáveis de determinado tipo não as cria ...

## Aritmética de apontadores

Uma vez declarado um apontador, podem ser realizadas sobre ele praticamente todas as operações realizáveis sobre variáveis.

### Exemplos:

```
int a = 8, b = 27;  
  
int *ptr = NULL; // ptr não aponta para nada  
  
ptr = &a; // ptr aponta para a  
  
printf("Valores de a, b e *ptr: %d, %d, e %d", a,b,*ptr); // Valores de a, b e *ptr: 8, 27 e 8  
  
*ptr = 50; // a variável cujo endereço está em ptr (a) recebe o valor 50  
  
printf("a=%d, b=%d, *ptr= %d",a, b, *ptr); // é escrito a=50, b=27, *ptr=50
```

## Operações com apontadores

### Incremento

- Um apontador para o tipo xyz avança sempre **sizeof(xyz)** Bytes por cada unidade de incremento.

### Decremento

- Um apontador para o tipo xyz recua sempre **sizeof(xyz)** Bytes por cada unidade de decremento.

### Diferença

- Quando realizável, permite saber quantos elementos existem entre um endereço e outro.

### Comparação

- É possível a comparação de dois apontadores para o mesmo tipo, usando os operadores relacionais

#### NOTA

As operações de diferença e comparação só podem ser realizadas entre apontadores do mesmo tipo

## Apontadores e vectores

O nome de um vector corresponde sempre ao endereço do seu primeiro elemento:  
se **v** for um vector, utilizar **v** é o mesmo que utilizar **&v[0]**

### Exemplo

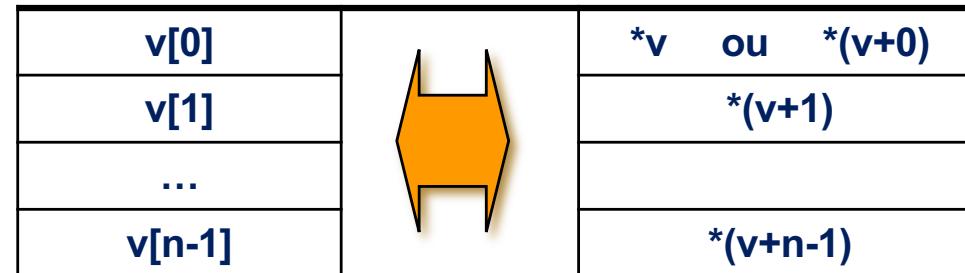
```
char s []="Viva";  
char *ptr = s;// ptr fica com o endereço de s[0] (&s[0])
```

Para acedermos ao carácter **a** presente na *string*, podemos ir por diversas vias

<b>s [3]</b>	Carácter existente na posição índice 3 do vector s
<b>* (ptr+3)</b>	Como ptr contém o endereço do primeiro carácter, ptr+3 contém o endereço do 4º
<b>* (s+3)</b>	Notar que s==&s[0]
<b>ptr [3]</b>	O endereçamento através de parêntesis rectos pode usar-se também por apontadores

## Apontadores e vectores

Se  $v$  tiver sido declarado como um vector com  $n$  componentes,  $v$  é um apontador para o primeiro elemento:



Para obtermos o elemento de índice  $i$  do vector  $v$ , pode utilizar-se:  
 $v[i]$  ou  $*(v+i)$

## Passagem de vectores para funções

Se passarmos um vector **v** como argumento a uma função, o que ela recebe na realidade não é o vector, mas o endereço do seu primeiro elemento (**v==&v[0]**).

Para que uma função conheça o número de componentes dos vectores que lhe são passados, deve ser-lhe fornecido outro parâmetro indicador do número de elementos ou um terminador de cada vector.

### OBSERVAÇÕES

Para evitar problemas de programação, é conveniente inicializar sempre os apontadores.

Quando se pretende declarar um apontador e não se quer que aponte para nenhuma variável, utiliza-se a variável simbólica **NULL**.

Numa declaração de um vector com inicialização automática (**int \*p = NULL**) é o apontador **p** que é inicializado e **não \*p**, embora a atribuição possa eventualmente sugerir o contrário.

## Apontadores de apontadores

Uma vez que os apontadores ocupam espaço em memória, é possível obter a sua posição através do operador **&**.

### EXEMPLO

Consideremos uma variável **x** do tipo **int**:

```
int x;
```

Se pretendermos armazenar o seu endereço, declaramos um apontador do tipo da variável:

```
int *prt_x;
```

Se pretendermos armazenar o endereço deste apontador, declaramos uma variável do tipo do apontador para o tipo de variável:

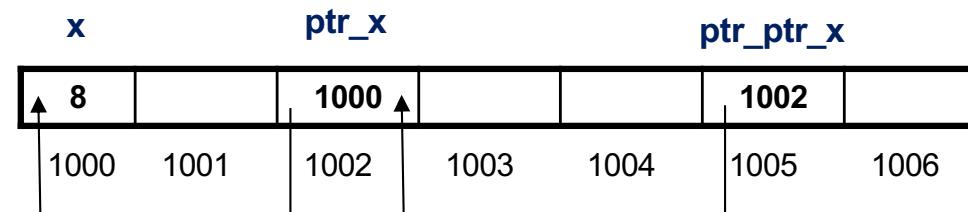
```
int **prt_ptr_x;
```

Esse processo de endereçamento podia prosseguir consecutivamente.

## Apontadores de apontadores (cont.)

### Exemplo

Expressão	Tipo	Valor	Descrição
x	int	8	Valor da variável x
ptr_x	int *	1000	Endereço da variável x
*ptr_x	int	8	Valor apontado por ptr_x
ptr_ptr_x	int **	1002	Endereço do apontador ptr_x
*ptr_ptr_x	int *	1000	Valor apontado por ptr_ptr_x
**ptr_ptr_x	int	8	Valor apontado pelo endereço apontado por ptr_ptr_x



Para que um apontador fique a apontar para uma dada variável, esta tem que ser previamente criada ...

**EXEMPLO** - Escrever uma função que faça a concatenação de duas *strings* **x** e **y** dadas como parâmetros de entrada, devolvendo a *string* resultante em **x**. Por exemplo, se **x** for "Aula" e **y** for "Teórica" a função deverá devolver "AulaTeórica" em **x**.

```
void juntaStringVa(char *x, char *y)
{
    while (*x++ != '\0')
        ;
    x--; //para x ficar a apontar para o último caracter (o terminador)
    while ( (*(x++)=*(y++)) !='\0' )
        ;
}
```

```
void juntaStringVb(char *x, char *y)
{
    while (*x != '\0')
        x++;
    while (*y != '\0')
        { *x=*y; x++; y++; }
    *x=*y;
}
```

```
void main()
{
    char str1[21], str2[11], *s1=str1,*s2=str2; //REDUNDÂNCIA s1 e s2?
    printf("\n Escreva uma frase (max. 10 caracteres): ");
    fflush(stdin);
    gets(s1);
    printf("\n Escreva outra frase (max. 10 caracteres): ");
    fflush(stdin);
    gets(s2);
    printf("\n Primeira frase: %s", s1);
    printf("\n Segunda frase: %s", s2);
    juntaStringVa(s1,s2);
    printf("\n Concatenando, resulta: %s", str1);
}
```

**EXERCÍCIO:** Analise as diferenças entre as 2 funções.

**QUAL DELAS É PREFERÍVEL?**

## Apontadores para “arrays” - EXEMPLO

```
void juntaStringVb(char *x, char *y)
{
    while (*x!="\0")
        x++;
    while (*y!="\0")
        { *x=*y; x++; y++; }
    *x=*y;
}
```

```
void main()
{
    char s1[21], s2[11]; // 21 = 10 + 10 + 1
    printf("\n Escreva uma frase (max. 10 caracteres): ");
    fflush(stdin);
    gets(s1);
    printf("\n Escreva outra frase (max. 10 caracteres): ");
    fflush(stdin);
    gets(s2);
    printf("\n Primeira frase: %s", s1);
    printf("\n Segunda frase: %s", s2);
    juntaStringVb(s1, s2);
    printf("\n Concatenando, resulta: %s", s1);
}
```

## ***2.9 Passagem de parâmetros de tipos estruturados***

- Tipos de passagem de parâmetros
- Passagem de vectores para funções
- Envio do endereço de uma variável para uma função

## Tipos de passagem de parâmetros

**Passagem de parâmetros por valor:** são enviadas para a função **cópias** dos valores de que esta necessita.

A passagem por valor permite, assim, enviar o valor de uma variável ou expressão.

**Passagem de parâmetros por referência:** o que é enviado para a função **não são** cópias dos valores, **são as próprias variáveis**.

Em C só existe passagem de parâmetros por valor: ao colocarmos uma variável como parâmetro, o que passa é o seu valor

Para conseguirmos alterar o valor de uma variável dentro de uma função, passamos o endereço da variável e efectuamos as alterações na variável original: **USAMOS APONTADORES**.

## Envio do endereço de uma variável para uma função

### Questão

Quando é que se torna necessário passar endereços de variáveis para uma função?

Se a variável for um array, a resposta é NUNCA, já que o nome do vector corresponde ao endereço do seu primeiro elemento.

Se a variável não for um array, tem de ser precedida de & sempre que se pretender que a própria variável seja alterada dentro da função ou procedimento a que se destina.

Se passarmos o endereço de um array, a variável que o recebe tem de ser um apontador para o tipo dos elementos do array.

## Exemplo

### Trocar os valores de duas variáveis

```
#include <stdio.h>
void troca(int *x, int *y)
// Função que troca o valor de duas variáveis
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
void main()
{
    int a1,a2;
    printf("\nEscreva dois números inteiros, separados por ESPAÇO: ");
    scanf("%d %d",&a1, &a2);
    printf("\nAntes de troca :\ta1= %d\ta2= %d\n",a1,a2);
    troca(&a1,&a2);
    printf("\nDepois de troca :\ta1= %d\ta2= %d\n",a1,a2);
}
```

## Passagem de vectores para funções

O nome de um vector é, em si mesmo, um endereço. Por isso, os vectores são sempre passados às funções **sem o &**.

```
void leVector_va(float *u, int num)
// Função para leitura de um vector com num componentes reais
{ int i;
    for (i = 0; i < num; i++)
    {
        printf("Qual o %do elemento? ",i+1);
        scanf("%f", &u[i]);
    }
}
```

EQUIVALENTES

```
void leVector_vb(float *u, int num)
// Função para leitura de um vector com num componentes reais
{
    int i;
    for (i = 0; i < num; i++)
    {
        printf("Qual o %do elemento? ",i+1);
        scanf("%f", u+i);
    }
}
```

## Passagem de vectores para funções

```
void escreveVector_va(float *u, int num)
// Função para escrita de um vector com num componentes reais
{
    int i;
    for (i = 0; i < num; i++)
        printf("\t [%d]=%.1f    ", i+1, u[i]);
}
```

EQUIVALENTES

```
void escreveVector_vb(float *u, int num)
// Função para escrita de um vector com num reais
{
    int i;
    for (i = 0; i < num; i++)
        printf("\t [%d]=%.1f    ", i+1, *(u+i));
    printf("\n");
}
```

Elaborar um programa que determine o valor máximo de um vector com n componentes reais, bem como a respectiva posição. A leitura do vector deve ser feita através de função antes declarada.

```
...
float maximo(float *u, int dim, int *pos)
// Função que devolve o valor máximo de os elementos de um vector e disponibiliza tb. a respectiva posição {
    int i;
    *pos = 0;
    for (i=1; i<dim; i++)
        if (u[i]>u[*pos])
            *pos = i;
    return u[*pos];
}
void main()
{
    int n, pm; float v[10];
    printf("Indique o numero de componentes do vector (sem exceder %d): ", MAX);
    scanf("%d", &n);
    leVector_vb(v, n);
    printf("A maior componente do vector\n");
    escreveVector_vb(v, n);
    maximo(v, n, &pm);
    printf("\nA v[%d]= %f\n", pm+1, v[pm]);
}
```

## Determinar o produto interno de dois vectores com n componentes

```
...
float produtoInterno(float x[ ], float y[ ], int m)
//Função que calcula o produto interno de dois vectores
{
    int i;
    float soma = 0;
    for (i=0; i<m; i++)
        soma += x[i]*y[i];
    return soma;
}
```

soma += \* (x+i) \* (\* (y+i));

```
...
void main()
{
    int n; float v[10];
    puts("Escreva um inteiro");
    scanf("%d", &n);
    puts("Leitura do primeiro vector");
    leVector(v, n);
    puts(" Leitura do segundo vector");
    leVector(w, n);
    printf("\nProduto interno dos vectores lidos:");
    printf(" %f\n", produtoInterno(v, w, n));
}
```

Implementar a função

**char\* memcpy(char\* destino, char\* origem, int n)**

que copia n caracteres do vector origem para o vector destino e devolve o vector destino.

## FUNÇÃO

```
...
char* memcpy(char dest[ ], char* orig, int n)
{
    int i;
    for (i=0; i<n; i++)
        dest[i] = orig[i];
    return dest;
}
```

## Declaração

Os seguintes cabeçalhos são equivalentes:

**char\* memcpy(char dest[ ], char\* orig, int n)**

**char\* memcpy(char\* dest, char\* orig, int n)**

```
printf("\nTexto copiado: %s\n", memcpy(frase2, frase1, num));
```

**CHAMADA**

## PROCEDIMENTO

```
...
void memcpyP(char* dest, char* orig, int n)

{
    int i;
    for (i=0; i<n; i++)
        dest[i] = orig[i];
}
```

## Declaração

```
memcpyP(frase2, frase1, num);
printf("\nTexto copiado: %s\n", frase2);
```

**CHAMADA**

Elaborar uma função que determine a data da Páscoa de um dado ano, entre 1900 e 2099.

```
void dataPascoa(int ano, int *mes, int *dia)
{
    int a, b, c, d, e;
    a = resto(ano,19);
    b = resto(ano,4);
    c = resto(ano,7);
    d = resto(19*a+24,30);
    e = resto(2*b+4*c+6*d+5,7);
    if (d+e>9)
    {
        *mes = 4;
        *dia = d+e-9;
    }
    else
    {
        *mes = 3;
        *dia = 22+d+e;
    }
}
```

**SUGESTÃO:** use a função *dataPascoa* para determinar os anos em que a Páscoa ocorre em 4 de Abril, num dado intervalo de anos.

```
void main()
{
    int va, vm, vd;
    do
    {
        printf("Indique um ano entre 1900 e 2099 \n");
        scanf("%d",&va);
    }while (va <1900 || va >2099);
    dataPascoa(va, &vm, &vd);
    printf("\n\n A Pascoa de %d ocorre em %d/%d \n\n", va, vd, vm);
}
```

## MAIS SUGESTÕES

DADO UM ano ENTRE 1999 E 2099

USANDO A FUNÇÃO *dataPascoa()*

CRIAR FUNÇÕES PARA DETERMINAR

- O dia e o mês em que ocorrem
  - O Carnaval (47 dias antes da Páscoa)
  - Sexta Feira Santa (2 dias antes da Páscoa)
  - Quinta-feira de Corpo de Deus (60 dias depois da Páscoa)
- O dia de semana correspondente a uma data (dia/mês) do ano em causa

## 2.10 Estruturas

- Introdução: declaração, inicialização, acesso aos campos
- Declaração de tipos
- Passagem de estruturas para funções
- Apontadores para estruturas

## Introdução: declaração, inicialização, acesso aos campos

As estruturas em C permitem colocar, numa única entidade, elementos de diferentes tipos.

As **componentes** armazenadas dentro de uma estrutura são vulgarmente denominadas campos ou membros da estrutura

### Declaração de estruturas

#### SINTAXE

```
struct [ nome da estrutura]
{
    tipo1     campo11, campo21, ... , campom1;
    ...
    tipon     campo1n, campo2n, ... , campokn;
};
```

A declaração de uma estrutura corresponde à declaração de um novo tipo e não à declaração de variáveis estruturadas

## Exemplos

```
struct data
{    int dia, ano;
    char mes[10];
};
```

A partir desta declaração fica disponível um novo tipo (`struct data`) , composto por dois inteiros e um vector com 10 caracteres.

### Na declaração de variáveis

```
struct data d, datas[50], *ptrData;
```

- `d` é uma variável do tipo `struct data`
- `datas` é um vector de 50 elementos, cada um deles do tipo `struct data`
- `ptrData` é um apontador para o tipo `struct data`

A declaração de variáveis pode também ser realizada quando se define a própria estrutura, através da

## Declaração de estruturas

### SINTAXE

```
struct [ nome da estrutura]
{
    tipo1    campo11, campo21, ... , campom1;
    ...
    tipon    campo1n, campo2n, ... , campokn;
} v1, v2 , ... , vk;
```

Assim, a declaração anterior poderia tomar a forma:

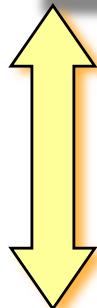
```
struct data
{
    int dia, ano;
    char mes[12];
} d, datas[50], *ptrData;
```

Mas é preferível separá-las!!

Para aceder ao membro *mmm* de uma estrutura *eee* usa-se o operador ponto (.), fazendo *eee.mmm*

## Exemplo

```
struct data{ int dia, ano; char mes[12]; }dn;  
dn.dia = 25;  
dn.ano = 2009;  
strcpy(dn.mes, "Novembro");  
printf("\nDATA: %d-%s-%d\n", dn.ano, dn.mes, dn.dia);
```



É possível inicializar uma estrutura quando é declarada, usando a **sintaxe**:



**struct estrutura variável = {valor<sub>1</sub>,..., valor<sub>n</sub>}**

```
struct data{ int dia, ano; char mes[10]; };  
struct data dn = {25, 2009, "Novembro"};  
printf("\nDATA: %d-%s-%d\n", dn.ano, dn.mes, dn.dia);
```

**NOTA:** a definição de uma estrutura pode ser feita sem indicar o seu nome. Nesse caso, todas as variáveis desse tipo têm de ser declaradas no momento da definição, pois a estrutura deixa de ser reconhecida fora do local da definição.

Tais estruturas não podem ser enviadas ou recebidas em funções, pois não possuem um nome que as identifique na definição dos parâmetros da função.

## Exemplo

```
struct { int dia, mes, ano; } Pascoa, Natal;  
  
Natal.dia = 25;  
Natal.mes = 12;  
Pascoa.dia = 4;
```

O seu uso é pouco vantajoso!!

## Declaração de tipos

Uma das desvantagens inerentes à utilização de estruturas consiste no facto de a declaração das variáveis ter de utilizar sempre a palavra reservada **struct** seguida do nome da estrutura.

A palavra reservada **typedef** permite que um determinado tipo possa ser denominado de modo diferente, de acordo com o interesse do utilizador.

TANTO PODE SER USADO COM ESTRUTURAS COMO COM UM TIPO QUALQUER DA LINGUAGEM.

### SINTAXE

**typedef** tipo existente sinónimo;

### Exemplo

```
typedef int inteiro;  
typedef float real;
```

## Exemplo

Considerando a estrutura data anteriormente definida,

```
typedef struct pessoa
{
    char nome[60];
    struct data dataNascimento;
    int idade;
    char estadoCivil;
    float salario;
} PESSOA; //PESSOA é um tipo estruturado
```

A partir desta declaração, podemos declarar variáveis com esta estrutura de forma muito simples:

```
PESSOA Tiago, Nuno, Filipe;
```

**NOTA:** na definição de um **typedef** **NÃO** podem ser declaradas variáveis (apenas tipos).

## Questão: onde definir estruturas (*struct*) e tipos (*typedef*)?

As estruturas devem ser definidas de forma a serem visíveis por todo o programa. Em geral definem-se no início do programa ou num *header file* que se junte ao mesmo.

### Exemplo

```
#include < ... >
#include < ... >

struct data { ... ; };
typedef ... ;

// protótipos das funções
...
// funções
...

main()
{
...
}
```

## Passagem de estruturas para funções

A passagem de estruturas para funções faz-se indicando no parâmetro o tipo associado à estrutura

```
#include <stdio.h>
typedef struct {int ano, mes, dia;}DATA;
typedef struct //pessoa
//poder-se-ia declarar a estrutura, independente do tipo, para uso posterior
{
    char nome[60];
    char habilitacoes[15];
    int idade;
    DATA nascimento;
    float salario;
} PESSOA;
void mostrar(PESSOA x)
{
    printf("\nNome           :%s",      x.nome);
    printf("\nIdade          :%d",      x.idade);
    printf("\nData nascimento:%d-%d-%d", x.nascimento.ano, x.nascimento.mes,
x.nascimento.dia);
    printf("\nHabilitacoes   :%s",      x.habilitacoes);
    printf("\nSalario         :%.2f\n",   x.salario);
}
void main()
{
    PESSOA p = {"Manuel Maria", "licenciado", 30, {1977,9,8}, 2000};
    mostrar (p);
}
```

Exemplo

Passagem por valor

```
struct pessoa p = {"Manuel Maria", "licenciado", 30, {1977,9,8}, 2000};
```

# Apontadores para estruturas

```
#include <stdio.h>
typedef struct {int ano, mes, dia;}DATA;
typedef struct pessoa
{
    char nome[60]; char habilitacoes[15];
    int idade;
    DATA nasc;
    float salario;
} PESSOA;
void ler(PESSOA *ptr)
{
    printf("\nEscreva os dados correspondentes aos itens seguintes\n");
    printf("\nNome: "); gets(*ptr.nome);
    printf("\nIdade: "); scanf("%d", &(*ptr).idade);
    printf("\nData nascimento (a m d): "); scanf("%d %d %d", &(*ptr).nasc.ano, &(*ptr).nasc.mes, &(*ptr).nasc.dia);
    fflush(stdin);
    printf("\nHabilitacoes: "); gets(ptr->habilitacoes);
    printf("\nSalario: "); scanf("%f", &(*ptr).salario);
}
void mostrar(PESSOA x)
{
    printf("\nNome\t: %s, \nIdade\t: %d ", x.nome, x.idade);
    printf("\nData nascimento: %d-%d-%d", x.nasc.ano, x.nasc.mes, x.nasc.dia);
    printf("\nHabilitacoes\t: %s",x.habilitacoes);
    printf("\nSalario\t: %.2f\n",x.salario);
}
void main()
{
    PESSOA p;
    ler(&p);
    mostrar(p);
}
```

*Passagem por referência e por valor*

O operador **Ponto(.)** tem maior precedência que o operador **Apontado(\*)**.

Para evitar confusão, o C coloca à disposição o operador **->**. Assim, \*(ptr).nome e ptr->nome são equivalentes

## **2.11 Memória dinâmica**

- Alocação de memória
- Libertação de memória
- Estruturas de dados dinâmicas

## Alocação de memória

A alocação dinâmica de memória pode realizar-se usando duas funções relativamente semelhantes: a função **malloc** e a função **calloc**.

### Função **malloc** (*Memory allocation*)

#### SINTAXE

```
void * malloc(size_t n_Bytes)
```

**size\_t** está normalmente definido no `#include <stdlib.h>` como sendo  
*typedef unsigned int size\_t;*

A função **malloc** permite alocar o conjunto de Bytes indicados pelo programador, devolvendo um apontador para o bloco de Bytes criados, ou **NULL** caso a alocação falhe.

*Repare que a função **malloc** devolve um tipo “curioso”: **void\*** . Como permite criar qualquer tipo de dados, o resultado terá que ser colocado num apontador. **Devolve um endereço de memória.***

## Alocação de memória (cont.)

### Função **calloc**

#### SINTAXE

```
void * calloc(size_t num, size_t size)
```

Esta função permite criar, dinamicamente, **num** elementos, cada um com **size** Bytes.

A função **calloc** coloca todos os bytes alocados com o valor 0 (zero).

Devolve o endereço da zona criada ou NULL.

### Função **realloc**

#### SINTAXE

```
void * realloc(void *ptr, size_t new_size)
```

A função **realloc** permite alterar o número de Bytes que estão presentemente associados a um Bloco previamente criado utilizando as funções **malloc** ou **calloc**.

## Alocação de memória (cont.)

### NOTAS RELATIVAS À FUNÇÃO REALLOC

- Se o bloco actualmente alocado puder ser aumentado para suportar a nova dimensão, a memória adicional é reservada também e é retornado ptr.
- Se não existir espaço suficiente para prolongar o bloco, é criado um novo bloco com a totalidade dos Bytes necessários. Os dados são copiados para a nova localização e é retornado o novo endereço.
- Se o parâmetro ptr for igual a NULL, a função comporta-se como malloc
- Se por algum motivo não for possível a alocação de memória ou se o número de Bytes for igual a zero, é devolvido NULL.

## Libertação de memória

A libertação de memória alocada através das funções **malloc** e a função **calloc** pode realizar-se através da função **free**.

### SINTAXE

**void free** (void \*ptr)

## Exercício

Implementar a função *strup* que cria uma nova *string* exactamente igual à que lhe foi passada por parâmetro.

```
char *strup(char *s)
// cria uma nova string idêntica à que lhe foi passada como parâmetro
{
    char *tmp = (char *) malloc(strlen(s)+1);
    //+1 para o carácter terminador ('\0')
    //tmp ficou com o endereço do bloco de bytes criado
    if (tmp!=NULL)
        strcpy(tmp, s);
    return tmp;
}
```

- Declara-se um apontador que recebe a nova zona de memória criada recorrendo à função malloc.
- Se a criação de memória for bem sucedida, copia-se o conteúdo de s para tmp e devolve-se o endereço em que a nova *string* foi criada.

***Nota:** embora esta função não faça parte da norma ANSI, a generalidade dos compiladores incorporou-a na sua biblioteca de funções (string.h)*

## Aplicação

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

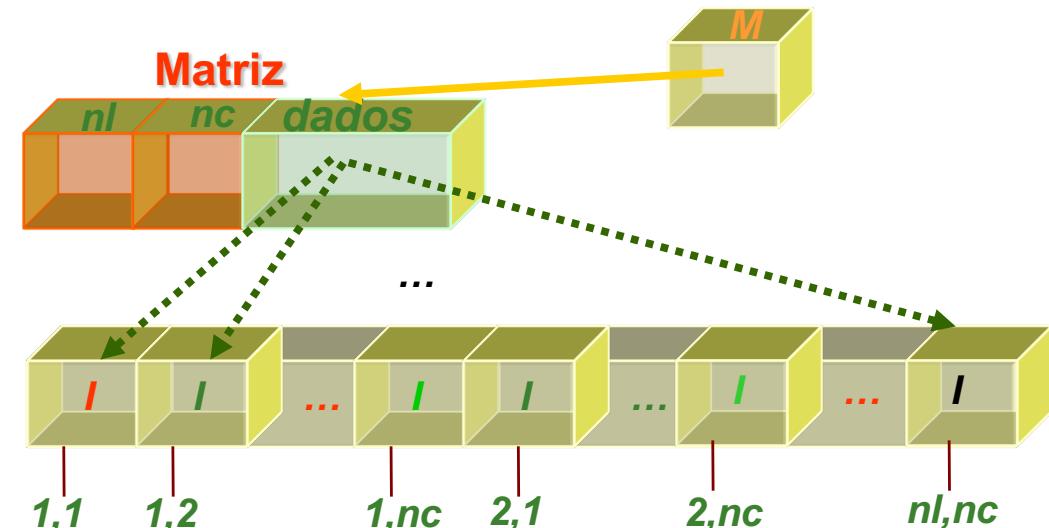
int main(void)
{
    char *str;
    str = (char *)malloc(10); //cria a string para 9 caracteres
    strcpy(str,"Manuel"); // coloca "Manuel" na string str
    printf("\nString: %s\nEndereço: %p\n", str, str);
    str = (char*) realloc(str,15); //associa 15 Bytes à string criada
    free(str); // liberta a memória
    return 0;
}
```

# Operações sobre matrizes

## Estrutura

```
typedef int Informacao;

typedef struct
{
    int nl, nc;
    Informacao *dados;
} Matriz;
```



## Alocação da matriz

```
Matriz *AlocarMatriz(int nlinhas, int ncolunas)
{
    Matriz* M; //M é um ponteiro para uma variável estruturada do tipo Matriz
    M = (Matriz *)malloc(sizeof(Matriz));
    //As 2 instruções anteriores são equivalentes à instrução
    // Matriz* M = (Matriz *)malloc(sizeof(Matriz));
    M->nl = nlinhas; // equivalente a (*M).nl = nlinhas;
    M->nc = ncolunas;
    M->dados = (int *) malloc( nlinhas*ncolunas*sizeof(Informacao) );
    if ( (M->dados) == NULL)
    {
        printf("Memória insuficiente\n");
        return(NULL); // termina o programa quando excede a memória
    }
    else
        return M;
}
```

## Operações sobre matrizes (cont.)

### Leitura de matriz a partir do teclado

```
void LerMatriz(Matriz *M)
{
    if (!M) return; // equivalente a if (M==NULL);
                  // caso M seja NULL, não faz nada
    for (int i = 0; i < M->nl; i++) // condição de paragem equivalente a j<(*M).nl
        for (int j = 0; j < M->nc; j++)
    {
        printf("[%d,%d]= ", i+1, j+1);
        int pos = i*M->nc + j;
        scanf("%d", &(M->dados[pos]));
    }
}
```

### Devolução do valor do elemento [i, j]

```
Informacao DevElemento(int i, int j, Matriz *M)
{
    return M->dados[i*M->nc + j];
}
```

### Escrita de matriz no ecrã

```
void EscreverMatriz(Matriz *M)
{
    if (!M) return; // No caso de M ser NULL não vai tentar escrever
    printf("Matriz com %d Linhas e %d Colunas\n", M->nl, M->nc);
    for (int i = 0; i < M->nl; i++)
    {
        for (int j = 0; j < M->nc; j++)
            printf("\t%d ", DevElemento(i, j, M));
        printf("\n");
    }
}
```

## Operações sobre matrizes (cont.)

```
bool MatrizSimetrica(Matriz *M)
{
    if (!M) return false; // No caso de M ser NULL nao faz nada
    if (M->nl != M->nc)
    {
        printf("A matriz nao e quadrada, pelo que não pode ser Simetrica \n");
        return false;
    }
    for (int i = 0; i < M->nl; i++)
        for (int j = i+1; j < M->nc; j++)
            if (DevElemento(i, j, M) != DevElemento(j, i, M))
                return false;
    // Se dois elementos não são "simétricos", a matriz não é simétrica!
    return true;
}
```

Matriz simétrica?

```
bool MatrizTriangularSuperior(Matriz *M)
// Uma matriz é triangular superior, quando os elementos abaixo da diagonal são nulos
{
    if (!M) return false; // No caso de M ser NULL nao faz nada
    if (M->nl != M->nc)
    {
        printf("A matriz nao e Triangular Superior, pois nao e quadrada \n");
        return false;
    }
    for (int i = 1; i < M->nl; i++)
        for (int j = 0; j < i; j++)
            if (DevElemento(i, j, M) != 0)
                return false;
    // Se algum elemento for diferente de zero, a matriz não é Triangular Superior
    return true;
}
```

Matriz triangular superior?

## EXERCÍCIO

**Escreva o programa principal para utilização das funções implementadas:**

Estrutura

Alocação

Leitura

Escrita

Simétrica?

Triangular?