



Departamento de
Informática

Estruturas de Dados

Introdução à Linguagem C

Francisco Morgado

Carlos Simões

Jorge Loureiro

Miguel Ferreira

Raquel Sebastião



Cofinanciado por



Cofinanciado pela
União Europeia

Introdução

Linguagem C

- Foi criada e implementada para o SO Unix por *Dennis Ritchie* (dec. 1970)
 - Brian W. Kernighan and Dennis M. Ritchie. 1988. The C Programming Language. Prentice Hall Professional Technical Reference

Estrutura de um programa

Elementos de um programa

Declarações:

- Declaração de ficheiros a incluir pelo pré-processador
- Declaração de variáveis para armazenamento das estruturas de dados

Instruções: indicam ao computador o que fazer

- As instruções são separadas por ponto e vírgula;
- O código de instruções a executar é colocado entre chavetas { }
- Um bloco é formado por um conjunto de instruções entre { }

Comentários: são ignorados pelo computador

- // uma linha ignorada
- /* */ tudo entre estes símbolos é ignorado
- Não podem existir comentários dentro de comentários

Estrutura simples de um programa

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

```
...
```

```
tipo main()
```

```
{
```

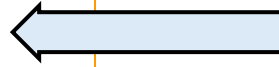
```
    int a, b, c;
```

```
    ...
```

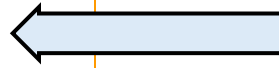
```
    printf("Hello World");
```

```
    ...
```

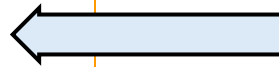
```
}
```



declaração de ficheiros a incluir na fase de compilação (diretivas do pré-processador)



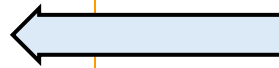
declarações de funções, constantes e variáveis globais



função principal do programa (bloco de instruções delimitado por chavetas)



declarações de constantes e variáveis locais



corpo instruções do programa

Estrutura de um programa

- Um programa em **C** é constituído por uma ou mais "*funções*"
- Identificação das funções pode ser qualquer à exceção de palavras reservadas da linguagem. Porém tem de existir uma com a identificação *main*
- Estrutura de um programa

#include "livraria" (e/ou ficheiros)

#define

tipos

variáveis globais

funções

Programa principal

```
void main()  
{  
    variáveis locais;  
    instruções;  
}
```

Funções

tipo nomeFuncao (parâmet)

```
{  
  variáveis locais;  
  
  instruções;  
}
```

- Sem tipo: **void**
- A ordem das funções é irrelevante
- Cada parâmetro é separado do seguinte através de uma vírgula
- Parâmetros declaram-se de igual forma que as variáveis locais
- Bloco { }
- { início de bloco
 - } fim de bloco

Sintaxe

- As linguagens de programação têm regras sintáticas que indicam como criar declarações, instruções e comentários corretamente
- A sintaxe do C inclui:
 - Diretivas do pré-processador
 - Palavras reservadas
 - Identificadores
 - Símbolos
 - Literais

Diretivas do pré-processador

- Instruções que se incluem num programa em linguagem C, e que são substituídas pelas respetivas tarefas antes da fase de compilação
- As diretivas são colocadas individualmente em cada linha de código, e são antecedidas do símbolo **#**
- A diretiva **#include** permite a leitura de outros ficheiros (bibliotecas de código fonte), cujas instruções são adicionadas ao ficheiro que contém essa diretiva

Exemplos:

- **#include <stdio.h>** //Biblioteca com funções de input/output
- **#include <string.h>** //Biblioteca com funções relacionadas com *strings*
- A diretiva **#define** associa um valor a um identificador

Exemplos:

- **#define PI 3.1415297**
- **#define ERRO "Operação inválida!"**

Palavras reservadas

asm	default	float	register	switch
auto	do	for	return	typedef
break	double	goto	short	union
case	else	if	signed	unsigned
char	enum	int	sizeof	void
const	extern	long	static	volatile
continue	far	near	struct	while

Exemplos:

- **do, while, for** – instruções
- **int, double** – tipos de dados (declaração de variáveis)

Símbolos e literais

Símbolos

- Operadores: `+` `-` `*` `/` `%`
 - O operador `=` tem função de atribuição
 - O operador `==` tem a função de comparação de valores, devolvendo o valor verdadeiro se estes forem iguais
- Sinais de pontuação: `{` `}` `;` `(` `)` etc.

Literais

- Dados explícitos que podem ser manipulados pelo programa
 - Números inteiros
 - Reais
 - Caracteres
 - Cadeias de caracteres (*strings*)

Símbolos e literais

Carácter `\` retira o significado especial que um caracter tem

<code>\7</code>	Sinal sonoro	<code>\v</code>	Tabulação vertical
<code>\a</code>	Sinal sonoro	<code>\\</code>	(<code>\</code>)
<code>\b</code>	Backspace	<code>\'</code>	(')
<code>\n</code>	New line	<code>\"</code>	(")
<code>\r</code>	Carriage return	<code>\?</code>	(?)
<code>\t</code>	Tabulação horizontal		

Variáveis

- **Variável** - posição de memória onde se guarda um valor de um dado tipo, que pode variar durante a execução de um programa
- Sintaxe para declaração de variáveis:
 - **tipo** nome_da_variavel;
 - **tipo** nome_da_variavel = valor_inicial;
 - **tipo** nome_da_variavel1, nome_da_variavel2;
tipo permite definir o espaço de memória a reservar
nome permite manipulá-la sem ser necessário conhecer a localização de memória onde a mesma se encontra
- Exemplos:

```
int soma;  
int max = 1;  
int soma, conta;
```

Tipos de variáveis

- Caracter **char**

gama: 0 a 255 (1 *byte*)

- inteiro **int**

gama: -32768 a 32767 (2 *byte*)

- real **float**

gama: 3.4E-38 a 3.4E+38 (4 *byte*)

- real de dupla precisão **double**

gama: 1.7E-308 a 1.7E+308 (8 *byte*)

•*Strings* e *arrays* são declaradas colocando-se a dimensão entre parêntesis retos.

Exemplos:

```
int x;           /*variável inteira */
```

```
short int si;  
           /*variável inteira curta*/
```

```
char frase[80];  
           /*string ou array de caracteres*/
```

```
float x,y;  
           /*variáveis reais */
```

```
int i = 0;  
float eps = 1.0e-5;
```

Atribuição e Entrada/Saída

- Atribuição =
- Funções de entrada e saída
 (definidas em `stdio.h`)
 - **printf**
 - **scanf**
 - `fgets`
 - `getchar`
 - `getc`
 - `gets`
 - ...

printf

printf(str)

- **printf("Hoje está a chover")**

printf(controlo, arg1, arg2, ...)

- controlo:
 - **%esp** tipo_conversão
- **esp** pode ser (por esta ordem):

- As *flags* (por qualquer ordem):
 - **-** (sinal menos) - ajuste à esquerda
 - **+** (sinal mais) - mostra sempre o número com sinal
 - Espaço - se o primeiro carácter não for o sinal será mostrado um espaço em branco no início do número
 - **0** (zero) – preenche o com zeros antes do número
- número de largura mínima
- **.** (ponto)
- número largura máxima (precisão)
- **h** ou **l** - Se o inteiro for para mostrar como short (**h**) ou long (**l**).

printf

printf(controlo, arg1, arg2, ...)

- controlo:
 - % esp **tipo_conversão**
- **tipo_conversão** pode ser:

- **o** conversão para octal sem sinal
- **d** conversão para decimal
- **x** conversão para hexadecimal sem sinal
- **u** conversão para decimal sem sinal
- **c** conversão para caracter
- **s** conversão para *string*
- **e** conversão para o formato
[-]m.nnnnnnnE[±]xx
- **f** conversão para *float*
- **g** o mais curto entre **%e** e **%f**; zeros não significativos não são mostrados
- **%** mostra o carácter % e não um argumento

Exemplos

Valor	Especificador de conversão	Saída
360	%10d	360
360	%-10d	360
360	%10o	550
360	%010o	0000000550
360	%-10x	168
360	%010x	0000000168
3.14159265	%10f	3.141593
3.14159265	%10.3f	3.142
3.14159265	%-10.3f	3.142
3.14159265	%10.0f	3
3.14159265	%10g	3.14159
3.14159265	%10e	3.141593e+00
3.14159265	%10.2e	3.14e+00

Valor	Especificador de conversão	Saída
"string teste!"	%s	string teste!
"string teste!"	%10s	string teste!
"string teste!"	%.10s	string tes
"string teste!"	%-10s	string teste!
"string teste!"	%.15s	string teste!
"string teste!"	%-15s	string teste!
"string teste!"	%15.10s	string tes
"string teste!"	%-15.10s	string tes

scanf(controlelo, arg1, arg2, ...)

- Os argumentos **arg1**, **arg2**, ... têm de ser ponteiros (endereços).
- Controlelo: **%tipo_conversão**
 tipo_conversão pode ser:
 - **d** decimal
 - **o** octal
 - **x** hexadecimal
 - **h** short int
 - **c** character
 - **s** *string*
 - **f** float

Exemplos

- Programa 1

```
#include <stdio.h>

void main()
{
    printf("Hoje está a chover");
}
```

- Programa 2

```
#include <stdio.h>

void main()
{
    int x;
    printf("Introduza um valor");
    scanf("%d", &x);
    printf("O valor introduzido");
    printf("foi %d", x);
}
```

Operadores

Operadores aritméticos

- Adição $+$
- Subtracção $-$
- Divisão $/$
- Multiplicação $*$
- Resto da divisão inteira $\%$
- $++$ e $--$

Operadores relacionais

- Maior $>$
- Menor $<$
- Maior ou igual $>=$
- Menor ou igual $<=$
- Igualdade $==$
- Diferente $!=$

Operadores

Operadores lógicos

- E &&
- OU ||
- Negação !

Operadores para manipulação de bits

- E &
- OU |
- OU exclusivo ^
- Shift à esquerda <<
- Shift à direita >>
- Complemento para um ~

Operadores

Exemplos

```
x=1; y=2;
```

```
printf("%d %d", x/y, x%y);    /* Mostra 0 1 */
```

```
x++    <=> x=x+1;
```

```
x--    <=> x=x-1;
```

```
x=10; y=++x; => x←11, y←x (11)
```

```
x=10; y=x++; => y←x (10), x←11
```

1 ⇔ Verdadeiro

0 ⇔ Falso

1 && !0 || 1 = 1 (verdadeiro)

1 && !(0 || 1) = 0 (falso)

Operadores

Exemplos

x = x+2; \Leftrightarrow **x+=2;**

exp₁ op= exp₂; \Leftrightarrow **exp₁ = (exp₁) op (exp₂);**

x *= y + 1; é igual a **x = x * (y + 1);**
e não a **x = x * y + 1;**

	1 1 0 0 0 0 0 1	x
&	<u>0 1 1 1 1 1 1 1</u>	127
	0 1 0 0 0 0 0 1	resultado

	1 0 0 0 0 0 0 0	128 em binário
	<u>0 0 0 0 0 0 1 1</u>	3 em binário
	1 0 0 0 0 0 1 1	resultado

char x;

x=7;	0 0 0 0 0 1 1 1	7 em binário
------	-----------------	--------------

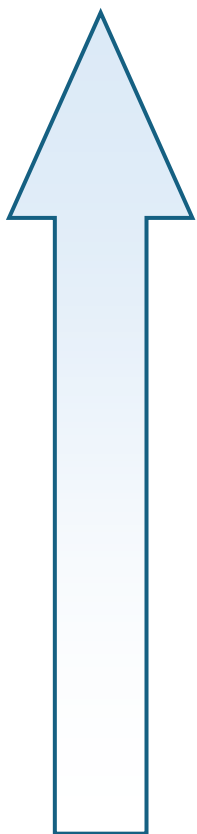
x << 1;	0 0 0 0 1 1 1 0	14 em binário (*2)
---------	-----------------	--------------------

x << 3;	0 1 1 1 0 0 0 0	112 em binário (*2 ³)
---------	-----------------	-----------------------------------

x >> 1;	0 0 1 1 1 0 0 0	56 em binário (/2)
---------	-----------------	--------------------

Prioridade dos operadores

Maior Prioridade



Menor Prioridade

Operador	Associatividade
++ -- -(menos unário) (tipo) ~ !	direita p/a a esquerda
* / %	esquerda p/a a direita
+ -	esquerda p/a a direita
<< >>	esquerda p/a a direita
< <= > >=	esquerda p/a a direita
== !=	esquerda p/a a direita
&	esquerda p/a a direita
^	esquerda p/a a direita
	esquerda p/a a direita
&&	esquerda p/a a direita
	esquerda p/a a direita
= += -= *= /=	direita p/a a esquerda

Instruções condicionais

if

```
if (condição)
    instrução;
```

```
if (condição)
{
    conjunto de instruções
}
```

if..else

```
if (condição)
    instrução1;
else
    instrução2;
```

```
if (condição)
{
    conjunto de instruções1
}
else
{
    conjunto de instruções2
}
```

Instruções condicionais

if..else..if

```
if (condição1)
    instrução1;
else if (condição2)
    instrução2;
else
    instruçãon;
```

```
if (condição1)
{
    conjunto de instruções1
}
else if (condição2)
{
    conjunto de instruções2
}
else
{
    conjunto de instruções3
}
```

Instruções condicionais

- Pode haver uma combinação entre as duas sintaxes
- O último **else** pode não existir

A que **if** pertence este **else**?

Exemplo:

```
if (condição1)
    if (condição_a)
        instrução_a;
    else if (condição_b)
        instrução_b;
else if (condição2)
    instrução2;

. . .

else
    instrução_n;
```

Instruções condicionais

- Um **else** pertence ao último **if** que ainda não tem **else**

Correção:

```
if (condição1)
{
    if (condição_a)
        instrução_a;
    else if (condição_b)
        instrução_b;
}
else if (condição2)
    instrução2;

. . .

else
    instruçãon;
```

Instruções condicionais

switch

```
switch (variável)
{
    case exp_1: instr_1; break;
    case exp_2: instr_2; break;
        . . .
    default : instr_n; break;
}
```

exp_i:

- inteiro
- caracter
- expressão de constantes

instr_i:

- Instrução ou conjunto de instruções

Instruções condicionais

A sequência de instruções:

```
switch (variável)
{
    case exp1 :
    case exp2 : instr2; break;
    .
    .
    .
    default : ... break;
} exp1
```

executa instr2 **se** variável = **ou** exp2.

Ciclo `while` e `do .. while`

Sintaxe do `while`

```
while (condição)
    instrução;
```

```
while (condição)
{
    conjunto de instruções
}
```

(o teste da condição é feito no início)

Sintaxe do `do .. while`

```
do
{
    instrução;
}
while (condição);
```

```
do
{
    conjunto de instruções
}
while (condição);
```


Ciclo **for**

Sintaxe do **for**

```
for (exp1;exp2;exp3)
    instrução;
```

```
for (exp1;exp2;exp3)
{
    conjunto de instruções
}
```

Do ponto de vista gramatical **exp1**, **exp2** e **exp3** são expressões mas vulgarmente **exp1** e **exp3** são atribuições e **exp2** é uma expressão relacional.

Transformação de um **for** para **while**

```
exp1;
while (exp2)
{
    conjunto de instruções
    exp3;
}
```

Instruções **break** e **continue**

Exemplo 1

```
for (t=0;t<100;t++)
{
    printf("%d\n",t);
    if (t==10) break;
}
```

Exemplo 3

```
for (x=0;x<100;x++)
{
    if (x%2) continue;
    printf("%d \n",x);
}
```

Exemplo 2

```
for (t=0;t<100;++t)
{
    count=1
    for(;;)
    {
        printf("%d\n",t);
        if (t==10) break;
    }
}
```

Variáveis em blocos

- É possível fazer a declaração de variáveis num bloco:

```
i=0;
while (i < 10)
{
    char c;

    printf("Introduza um character:");
    scanf("%c", &c);
    printf("escreveu %c", c);
    i++;
}
```

Retorno de valores de funções

O retorno de valores de funções é efetuado usando a palavra reservada **return**. A sintaxe desta instrução é :

return (expressão) ;

sendo expressão:

- Um valor
- Uma expressão - neste caso a expressão evolui para um valor do tipo da função
- A invocação de outra função

Retorno de valores de funções

Exemplo

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

```
/*função que devolve a área de um quadrado*/
```

```
float area(float la)
```

```
{
```

```
    return(la*la);
```

```
}
```

```
/* função que efectua a leitura do lado de um  
rectângulo e retorna o valor lido */
```

```
float leitura()
```

```
{
```

```
    float la;
```

```
    printf("Quanto mede o lado?");
```

```
    scanf("%f",&la);
```

```
    return(la);
```

```
}
```

```
float leitura(),area();
```

```
/* programa principal */
```

```
void main()
```

```
{
```

```
    float la;
```

```
    la = leitura();
```

```
    printf("O quadrado de lado");
```

```
    printf("%f é %f",la,area(la));
```

```
}
```

atenção à ordem / declaração das funções;

Quizz informativo

Introdução à linguagem de programação C

<https://url-shortener.me/CAZ>



Array's

- Declaram-se da mesma forma que uma variável elementar só que se coloca a dimensão destes entre parêntesis retos a seguir à identificação da variável:

tipo nomeVar[dimensão] ;

- **tipo**: tipo de dados de cada um dos elementos do *array*;
- **nomeVar**: indica o nome pelo qual o *array* vai ser conhecido;
- **dimensao**: valor constante que indica quantos elementos tem o *array*;
- Um *array* pode conter elementos de qualquer tipo. No entanto, os elementos de um dado *array* têm que ser **obrigatoriamente do mesmo tipo**, o qual é definido na declaração do mesmo.

Array's

- Em C os índices de um *array* com **N** elementos variam sempre entre **0** e **N-1**:
 - O índice do primeiro elemento de qualquer *array* em C é sempre **0**;
 - O índice do último elemento de qualquer *array* em C é sempre **n - 1**;
 - Num *array*, o **i-ésimo** elemento está sempre na posição **i-1**.
 - Cada um dos elementos do *array* **nomeVar**, pode ser acedido através do respetivo **índice** colocado entre parêntesis rectos (**[]**);
- Declaração como argumento de um procedimento ou função **não necessita de dimensão**.
- Passagem para procedimentos ou funções é sempre feita por **referência**.
 - O **nome de um *array*** é, em si mesmo, um **endereço**. Por isso, os arrays são sempre passados às funções sem o &.
 - O **nome de um *array*** é um **ponteiro constante** para o primeiro elemento;

Array's

Exemplos

- `int a[10];` // `a` é um *array* com 10 elementos inteiros
- `float b[100];` // `b` é um *array* com 100 elementos reais
- Considerando a seguinte declaração:

```
#define N 50
```

```
(...)
```

```
float notas[N];
```

 // `notas` é um *array* com 50 elementos reais

- **float**: tipo de cada um dos elementos do *array*
- **N**: nº de elementos do *array*
- **notas**: nome do *array*
- **notas[i]**: conteúdo da posição **i** (elemento **i-1**) do *array* **notas**

Array's

Exemplos

Declarar um *array* com 6 inteiros chamado **tabela**:

- `int tabela[6];`

tabela[0]	tabela[1]	tabela[2]	tabela[3]	tabela[4]	tabela[5]

- `tabela[0] = 5;`

5					
tabela[0]	tabela[1]	tabela[2]	tabela[3]	tabela[4]	tabela[5]

- `tabela[5] = 4*tabela[0];`

5					20
tabela[0]	tabela[1]	tabela[2]	tabela[3]	tabela[4]	tabela[5]

- `tabela[2] = 3*tabela[0] + tabela[5];`

5		35			20
tabela[0]	tabela[1]	tabela[2]	tabela[3]	tabela[4]	tabela[5]

Array's

Exemplos

Considerandos as seguintes declarações

- `int tabela_1[6] = {10, 20, 30, 40, 50, 60};`

10	20	30	40	50	60
----	----	----	----	----	----

- `int tabela_2[6] = {10, 20, 30};`

10	20	30	0	0	0
----	----	----	---	---	---

- `int tabela_3[6];`

~~`tabela_3[6] = {10, 20, 30, 40, 50, 60};`~~



Em C, a atribuição com { } só é válida aquando da declaração do array

- `int tabela_4[6];`

`tabela_4[1] = 10; tabela_4[2] = 20; (...)`

- `int tabela_5[6];`

`for (i = 0; i < 6; i++)
tabela_5[i] = 10*(i+1);`



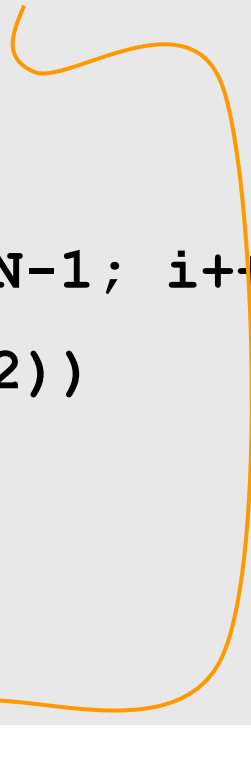
<https://tinyurl.com/exArrays>

Passagem de *array's* como parâmetro

Exemplos

```
#include <stdio.h>

int Npares(int V[],int N)
{
    int i,np=0;
    for (i=0; i<=N-1; i++)
        if (!(V[i]%2))
            np++;
    return np;
}
```



Não é necessário especificar a dimensão

```
/* programa principal */

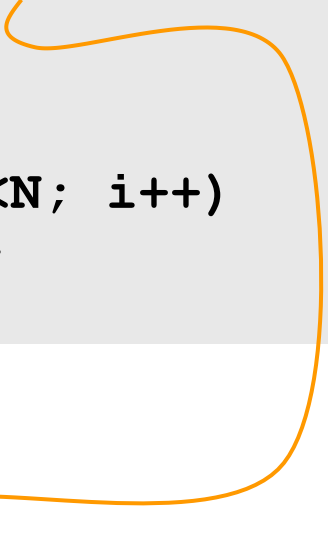
int main(){
    int Npares();
    int t[5],num;
    //preencher o array t:
    t[0]=12;t[1]=11;t[2]=15;
    t[3]=18; t[4]=20;
    num = Npares(t,5);
    printf("\nO Número de ");
    printf("pares é %d",num);
    return 0;
}
```

Passagem de *array's* como parâmetro

Exemplos

```
#include <stdio.h>
#define TAM 10

void atr(int a[], int N)
{
    int i;
    for(i=0; i<N; i++)
        a[i]=i;
}
```



Não é necessário especificar a dimensão

```
/* programa principal */

int main() {
    int Npares();
    int t[TAM], num;
    //preencher o array t:
    atr(t, TAM);
    num=Npares(t, TAM);
    printf("\nO Número de ");
    printf("pares é %d", num);
    return 0;
}
```

String's

```
char nome_var[dimensão];
```

- Verificam-se as mesma regras dos *array's*.
- Caracteres especiais:
 - tab `'\t'`
 - backspace `'\b'`
 - newline `'\n'`
 - fim de string `'\0'`
- Funções (biblioteca **string.h**):
 - **strlen(ids)**
 - **strcmp(str1, str2)**
 - **strcpy(strd, stro)**
 - **strcat(strd, stro)**

Ponteiros

- Operadores para manipulação de ponteiros:

***** apontado por ...

& endereço de ...

- Invocação de um subprograma:

```
void main()  
{  
    int x;  
  
    x=10;  
    incrementa(&x) ;  
    printf("%d",x) ;  
}
```

- Declaração do subprograma:

```
void incrementa(int *z)  
{  
    *z = *z + 1;  
    // ou *z += 1  
}
```

Ponteiros

- Apontador para ...

int *a; (a variável **a** armazena o endereço de uma variável inteira e não um inteiro)

char *c; float *y;

```
...  
int x,*a;  
float w,*y;  
x = 13;  
w = 12.5;  
a = &x;  
y = &w;  
printf("%d\n",x); // ou printf("%d\n",*a);  
printf("%f\n",w); // ou ???
```

RAM		
x	13	203
y	201	202
w	12.5	201
a	203	200

Ponteiros

Exemplos

```
int a[7]={-4, 2, -3, 4, -7, 3, 2};  
  
int *p;  
  
p = &a[2];  
  
printf("&a = %p\n", a);  
  
printf("p-a = %d\n", p-a);  
  
printf("p-1 = %p\n", p-1);  
  
printf("* (p-2) = %d\n", *(p-2));  
  
printf("2+* (p+1) = %d\n", 2+*(p+1));
```



<https://tinyurl.com/exArrayPtr1>

//Output:

&a = 520

p-a = 2

p-1 = 524

*(p-2) = -4

2+*(p+1) = 6

Nota: considerando que o espaço ocupado por uma variável do tipo **int** é de **4 bytes**.

Ponteiros

Exemplos

Versão errada:

```
troca(a, b);
```

```
void troca(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Versão correta:

```
troca(&a, &b);
```

```
void troca(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```



<https://tinyurl.com/exTroca>

Ponteiros

Exemplos



<https://tinyurl.com/exPtr1>



<https://tinyurl.com/exArrayPtr2>



<https://tinyurl.com/exArrayPtr3>



<https://tinyurl.com/exDupVetor>

Estruturas

- As **estruturas** em C permitem colocar, numa única entidade, elementos de diferentes tipos de dados;
- As componentes armazenadas dentro de uma estrutura são vulgarmente denominadas campos ou membros da estrutura;
- A declaração de uma estrutura corresponde à declaração de um novo tipo de dados e não à declaração de variáveis estruturadas;
- 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.

Estruturas

- Declaração de uma estrutura:

```
struct nomeEst  
{  
    tipo1 nome_campo11, nome_campo21, nome_campo31;  
    tipo2 nome_campo21, nome_campo22;  
    (...)  
}
```

Estruturas

Definiu-se a composição da estrutura mas não se declarou a variável.

- Declaração da variável:

```
struct nomeEst nomeVar;
```

- Exemplo:

```
struct endereco end_x;
```

É possível declarar a composição da estrutura e as variáveis simultaneamente.

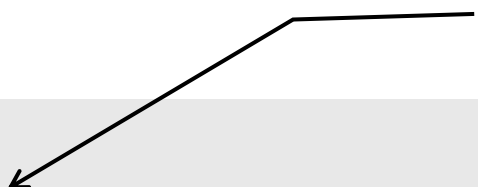
```
struct nomeEst  
{  
  
    campos  
  
} var1, var2, ...;
```

em que **var1, var2,...** são variáveis.

Estruturas

- Declaração de uma estrutura:

```
struct nomeEst
{
    tipo1 nome_campo11, nome_campo21, nome_campo31;
    tipo2 nome_campo21, nome_campo22;
    (...)
}
```



declaração de um novo tipo de dados

- Declaração de variáveis:

```
struct nomeEst d, estr[50], *ptrEstr;
```



declaração de var estruturadas

- d** é uma variável do tipo **struct nomeEst**
- estr** é um vetor de 50 elementos, cada um deles do tipo **struct nomeEst**
- ptrEstr** é um apontador para o tipo **struct nomeEst**

Estruturas

Referências aos campos de uma estrutura:

`var.campo`

`var->campo`

- Exemplos:

```
end_x.cod_postal=3510;  
printf("%s",end_x.nome);  
end_x.nome[1]='A';
```

Array's de estruturas

`struct nomeEst var[dim];`

- **var** variável
- **dim** dimensão do *array*

- Exemplo:

`struct endereco tab[100];`

- Acesso aos campos:

`tab[2].cod_postal=3510;`

Estruturas

Exemplos

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

```
struct data dNasc,
dataNascFamilia[20];
```

Inicializar **dNasc**:

```
dNasc.dia = 12;
strcpy (dNasc.mes, "Janeiro");
Natal.ano = 1975;
```

```
struct endereco
{
    char nome[31];
    char rua[51];
    char localidade[21];
    long int cod_postal;
};
```

```
struct endereco morada,
moradaFunc[40];
```

Declaração de novos tipos: *TypeDef*

- A palavra reservada *typedef* permite que um determinado **tipo** possa ser denominado de modo diferente, de acordo com o interesse do utilizador;
- *typedef* pode ser usado com qualquer tipo de variável;
- Declaração de um novo tipo

typedef tipo novotipo;

Declaração de novos tipos: *TypeDef*

Exemplos

```
typedef int numero;
```

```
typedef char *string;
```

```
typedef struct pessoa  
{  
    char nome[80];  
    char bi[12];  
} PES, *PES;
```

Declaração de novos tipos: *TypeDef*

Exemplos

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

```
typedef struct pessoa
{
    char nome[80];
    char CC[12];
} PES;
```

```
void leitura()
{
    PES id; // dec estática
    printf("Insira o nome\n");
    scanf("% 79[^\n] ", id.nome);
    printf("\Nome:%s\n", id.nome);
}
```

```
/* programa principal */
```

```
void main()
{
    leitura();
}
```

Cast's

`= (tipo) valor;`

`= (tipo) expressão;`

`= (tipo) invocação de função;`

`return ((tipo) ...);`

Alocação dinâmica de memória

- Função **sizeof (arg)**
 - Retorna o tamanho do argumento **arg**
 - Exemplo:


```
#include <stdio.h>
void main()
{
    printf("Tamanho de um inteiro");
    printf(" é : %d", sizeof(int));
}
```

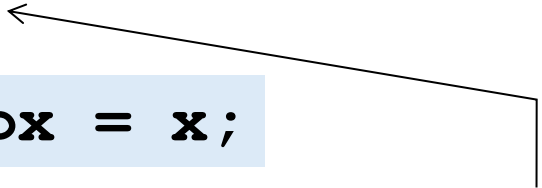
- Função **malloc (arg)**
 - Reserva memória para armazenar o tamanho de *bytes* especificado no argumento.
 - Retorna o endereço de base da memória alocada.
 - Deve libertar-se a memória quando desnecessária.

Alocação dinâmica de memória

Exemplos

Versão errada:


```
#include <stdio.h>
void main()
{
    int x, *px;
    x = 10;
    
    *px = x;
}
```



⚠ *Inicializar:*
px = &x;

Versão correcta:

```
#include <stdio.h>
void main()
{
    int x, *px;
    x = 10;
    px = (int *) malloc(sizeof(int));
    *px = x;
    free(px);
}
```



*Atenção
ao
cast*

Alocação dinâmica de memória

Exemplos

Exemplo de Utilização de *array's* declarados estaticamente

```
#include <stdio.h>
#define TAM 10
void leitura(float a[])
{ int i;
  for(i=0; i<=TAM; i++)
  {
    printf("Introduza o valor %d\n",i+1);
    scanf("%f",&(a[i]));
  }
}
void escreve(float *a)
{ int i;
  for(i=TAM-1; i>=0; i--)
    printf("O %dº valor é %f\n",i+1, a[i]);
}
void main()
{ float ar[TAM];
  void leitura(),escreve();
  leitura(ar);
  escreve(ar);
}
```

ou `float *a`

/* função que lê 10 reais para um array*/
/* atenção à omissão da dimensão do array*/

ou `float a[]`

ou

```
scanf("%f",a);
a++;
```

ou

```
scanf("%f",a++);
```

ou

```
scanf("%f", (a+i));
```

ou

```
...,*(a+i));
```

ou

```
a+=TAM-1; (antes do for)
...,*(a--));
```


Alocação dinâmica de memória

Exemplos

Exemplo anterior com alocação dinâmica

```
#include <stdio.h>
#include <stdlib.h>
#define TAM 10
void leitura(float *a)      /* ou float a[] */
{ int i;
  for(i=0; i<=TAM; i++)
  {
    printf("Introduza o valor %d°\n",i+1);
    scanf("%f",a++);
  }
}
void escreve(float *a)
{ int i;
  a+=TAM-1;
  for(i=TAM-1; i>=0; i--)
    printf("O %d° valor é %f\n",i+1, *(a--));
}
void main()
{ float *t;
  t = (float *)malloc(TAM*sizeof(float));
  leitura(t);
  escreve(t);
  free(t);
}
```

Alocação dinâmica de memória e declaração de novos tipos (*Typedef*): Exemplos

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

typedef struct pessoa
{
    char nome[80];
    char CC[12];
}*ptPES, PES;
```

```
void leitura()
{
    ptPES id; // ptr para PES
    id=(ptPES) malloc(sizeof(PES));
    printf("Insira o nome\n");
    scanf("% 79[^\n] ",id.nome);
    printf("\Nome:%s\n",id.nome);
}
```

```
/* programa principal */
void main()
{
    leitura();
    free(id);
}
```

Alocação dinâmica de memória e passagem de *array's* como parâmetro: Exemplos

```
#include <stdio.h>
#include <stdlib.h>
#define TAM 10
void atr(int *a, int N)
{
    int i;
    for(i=0; i<N; i++)
        *(a++)=i;
}
```

```
void escreve(int a[], int N)
{
    int i;
    for(i=0; i<N; i++)
        printf("\n%d", a[i]);
}
```

```
/* programa principal */
```

```
void main()
{
    int *ar;
    ar=(int *)
        malloc(TAM*sizeof(int));
    atr(ar, TAM);
    escreve(ar, TAM);
    free(ar);
}
```

Não é necessário especificar a dimensão

Ponteiros

Exemplos



<https://tinyurl.com/exArrayLocal1>



<https://tinyurl.com/exArrayLocal2>



<https://tinyurl.com/exArrayLocal3>



<https://tinyurl.com/exArrayDin1>

Funções passadas como parâmetro

- Passa-se um apontador para procedimento ou função:

Exemplo:

```
int incrementa(int x)
{
    return x+1;
}
void escreve(int z)
{
    printf("%d\n", z);
}
void main()
{
    escreve(incrementa(10));
}
```

Funções passadas como parâmetro

- Exemplo usando um apontador para função:

```
int incrementa(int x)
{
    return x+1;
}
void escreve(int (*func)(int))
{
    printf("%d\n", func(10));
}
void main()
{
    int incrementa();
    void escreve();
    escreve(incrementa);
}
```

Conversão de *String's*

- `#include <stdlib.h>`
- Funções de conversão *string* para valor:

`int atoi(const char *nptr);`

Converte a *string* apontada por `nptr` para inteiro

`double atof(const char *nptr);`

Converte a *string* apontada por `nptr` para double

`long atol(const char *nptr);`

Converte a *string* apontada por `nptr` para um inteiro longo

Ponteiros para ponteiros

```
#include <stdio.h>
#include <malloc.h>

void aloca(x)
int **x;
{
    int *y;
    y = (int *) malloc(sizeof(int));
    *x = y;
    **x = 10;
}
```

```
void main()
{
    int *x;

    aloca(&x);
    print("%d", *x);
}
```


Ponteiros para ponteiros

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
typedef struct elemento
{
    char nome[80];
    char bi[12];
    int nal; /* chave*/
}*ELEM;
```

```
void aloca(pe)
ELEM *pe;
{
    *pe = (ELEM)malloc(
        sizeof(struct elemento));
    strcpy((*pe)->nome, "Jose");
}
```

```
/* programa principal */
void main()
{
    ELEM pe;

    aloca(&pe);
    printf("%s", pe->nome);
}
```

Ficheiros

Abertura e fecho

- Ponteiro para ficheiro: **FILE *fp;**
- Abertura de ficheiros (através da função `fopen`):
FILE *fopen(char *caminho, char *modo);

Modo de abertura de ficheiros:

- r** - Abertura para leitura de ficheiro texto (posição no início)
 - r+** - Abertura para leitura e escrita (posição no início)
 - w** - Abertura para escrita (O ficheiro é truncado: se já existir, o seu conteúdo é apagado!)
 - w+** - Abertura para escrita e leitura (O ficheiro é truncado)
 - a** - Abertura para escrita (posição no fim)
 - a+** - Abertura para escrita e leitura (posição no fim)
- Fecho de ficheiros (através da função **fclose**):
int fclose(FILE *stream);

Ficheiros

Abertura e fecho

```
FILE *f = fopen(fileName, "r");  
if (f == NULL)  
{  
    printf("Problemas na abertura do ficheiro! \n");  
    return NULL;  
}
```

- Se não houve problemas com a abertura do ficheiro cujo nome é dado na variável **fileName**, **f** fica a apontar para esse ficheiro (para leitura);
- Se houve problemas, **f** fica a apontar para “nada”;

Ficheiros

Leitura e escrita

```
int fprintf( FILE *stream,  
            const char *format, arg1, arg2, ... );  
  
size_t fwrite(void *ptr, size_t size,  
             size_t nmemb, FILE *stream);  
  
int fscanf( FILE *stream,  
           const char *format, arg1, arg2, ... );  
  
char *fgets(char *s, int size,  
            FILE *stream);  
  
size_t fread( void *ptr, size_t size,  
             size_t nmemb, FILE *stream);
```

```
int fseek(FILE *stream,  
          long deslocamento,int pos_ini);
```

- `pos_ini` pode ser:
 - `SEEK_SET` – início de ficheiro
 - `SEEK_CUR` – posição actual
 - `SEEK_END` – fim do ficheiro

```
long int ftell (FILE *stream);
```

```
int feof (FILE *stream);
```

