

ARQCP

Introdução à Linguagem de Programação C aula TP

2024/2025

André Andrade

lao@isep.ipp.pt

Aviso

- A documentação oficial de ARQCP são os slides disponibilizados no Moodle do ISEP de ARQCP
- Embora estes slides sejam criados com base na informação oficial podem existir algumas diferenças
- Aconselha-se a consulta da informação oficial

AntiX Linux Live CD

- Antes da primeira aula PL, descarregue o ficheiro “**antiX-Dev-YYYYMMDD.iso**” que está disponível:
 - No Moodle de ARQCP:
 - Práticas Laboratoriais → “antiX Linux LiveCD ISO”
 - Nos endereços:
 - <https://ax.ttmby.org>
 - <https://lx.ttmby.org>

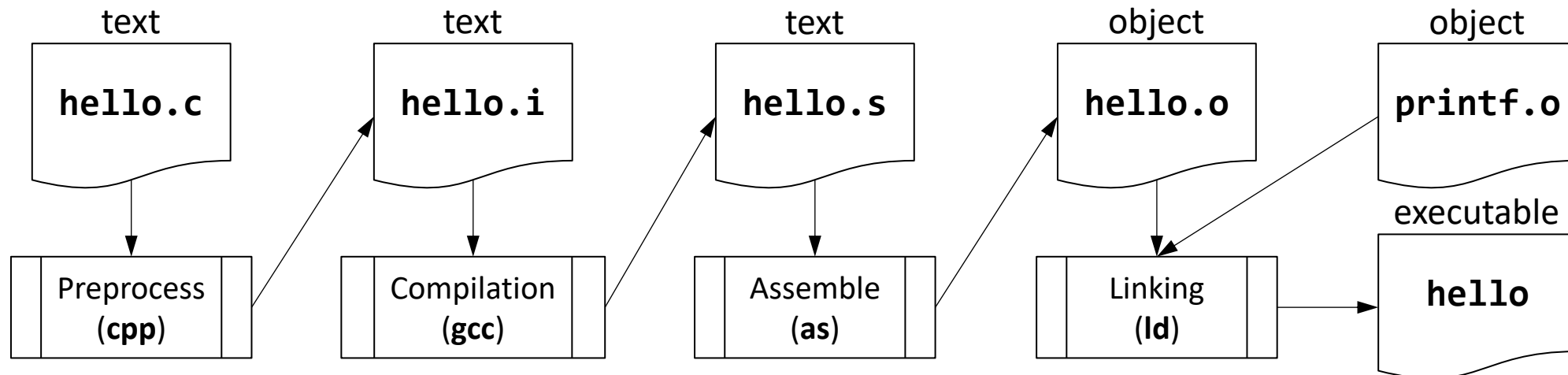
YYYYMMDD representa a data de criação do ISO, onde:
YYYY – Ano
MM – Mês
DD – Dia

“Hello World” em C

```
/*  
O programa utiliza a função printf(), que está definida na C Standard Library “stdio”  
linhas que comecem com ‘#’ são denominadas de diretivas de pré-processamento, estas  
instruções não são terminadas com ‘;’  
(Este é um comentário multi-linha)  
*/  
#include <stdio.h>  
  
/*  
A função main() devolve um inteiro e, neste caso, não recebe argumentos (void)  
A função main() é a primeira função que é invocada quando um programa é executado  
*/  
int main(void) {  
    printf("Hello World!\n"); /* Imprime informação formatada, '\n' é a mudança de linha */  
    return 0; /* A função main devolve o valor 0 */  
}
```

Compilação de programas em C

- Os programas em **C** têm de ser transformados em código máquina de forma a poderem ser executados, este processo é denominado de **compilação**
- O processo de compilação envolve também outros passos:
 - Pré-processamento, compilação, montagem e linkagem
- Vamos utilizar o compilador **GNU C Compiler (gcc)**
 - Para compilar o programa "Hello World": **gcc -Wall -Wextra -fanalyzer hello.c -o hello**
 - Se adicionarmos a opção "**-save-temps**" do **gcc** iremos obter todos os ficheiros gerados durante o processo de compilação
 - Exemplo: **gcc -save-temps -Wall -Wextra -fanalyzer hello.c -o hello**



Opções de comando importantes do **gcc**

Option	Description
-Wall	All warnings – use always!
-o filename	Specify output name as filename for object or executable
-c	Compile only, do not link; used to create na object file (.o) for a single (non-main) .c file (module)
-g	Insert debugging information
-E	Stop after the preprocessing stage; output goes to standard output
-v	Show information about gcc and/or compilation process
-S	Performs preprocessing and compilation only; that is, convert C source into assembly
-save-temps	Keep temporary files created (.i , .s , .o , ...)
-llibrary-name	Link with library called library-name
-ldir	Add dir to the list of dirs to be searched for header files
-Ldir	Add dir to the list of dirs to be searched for the libraries specified with -l ;

Para mais opções de comando: <http://aeno-refs.com/qr-linux/programming.html#gcc>

Notas relativas à informação do **gcc**

- Leia sempre com atenção a informação que o **gcc** produz
 - Habitualmente inclui boa informação acerca da origem do erro
 - Várias fontes de erros:
 - Preprocessor: falta de ficheiros de include
 - Parser: erros de sintaxe
 - Assembler: erros de sintaxe em código Assembly (apenas programando em Assembly)
 - Linker: Falta de bibliotecas
 - Frequentemente, um erro causa vários outros erros posteriores
 - Corrija o primeiro erro, ignore os restantes erros, tente novamente
 - Por exemplo a falta de ' ; ' irá produzir erros nas linhas de código seguintes

Compile sempre com as opções **-Wall -Wextra -fanalyzer** e **não ignore os avisos!**

Os avisos geralmente referem problemas que se vão manifestar em tempo de execução

O pré-processador de C

- O pré-processador de **C** (**C** preprocessor - cpp) permite definir macros, que são abreviações para construções mais longas
- As diretivas para o pré-processador começam com '#' no início de uma linha e são usadas para:
 - Inserir conteúdo de outro ficheiro no ficheiro que está a ser compilado:
 - `#include`
 - Definição de macros e constantes:
 - `#define`
 - Compilação condicional:
 - `#if`
 - `#ifdef`
 - `#ifndef`
 - `#else`
 - `#elif`
 - `#endif`
- Antes da compilação, o pré-processador lê o código fonte e transforma-o

O pré-processador de C

- Exemplo 1:

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

Pesquisa por `stdio.h` nas pastas definidas no sistema

Pesquisa por `mydefs.h` na pasta deste ficheiro

- Exemplo 2:

```
#define MAX 100
#define check(x) ((x) < MAX)
```

```
if(check(i)) {...}
```

resulta em:

```
if(((i) < 100)) {...}
```

Use o pré-processador de C com precaução

- É muito fácil de introduzir-se erro subtil
- Não é visível na depuração
- O código torna-se de leitura mais difícil

Por exemplo, os parêntesis em torno da variável podem parecer desnecessários, mas podem conduzir a problemas difíceis de diagnosticar!

Tipos de dados inteiros em x86-64

Tipo	Espaço de armazenamento	Gama de valores representados
char	1 Byte	-128 até 127
unsigned char		0 até 255
short	2 Bytes	-32768 até 32767
unsigned short		0 até 65535
int	4 Bytes	-2147483648 até 2147483647
unsigned int		0 até 4294967295
long	8 Bytes	-9223372036854775808 até 9223372036854775807
unsigned long		0 até 18446744073709551615

Tipos de dados de vírgula flutuante em x86-64

Tipo	Espaço de armazenamento	Gama de valores representados	Número mais próximo de 0	Precisão
float	4 Bytes	-3.40282E+38 até +3.40282E+38	$\pm 1.17549 \times 10^{-38}$	Garante 6 algarismos significativos
double	8 Bytes	-1.79769E+308 até +1.79769E+308	$\pm 2.22507 \times 10^{-308}$	Garante 15 algarismos significativos
long double	16 Bytes	-1.18973E+4932 até +1.18973E+4932	$\pm 3.3621 \times 10^{-4932}$	Garante 33 algarismos significativos

Mais informação:

- https://en.wikipedia.org/wiki/Single-precision_floating-point_format
- https://en.wikipedia.org/wiki/Double-precision_floating-point_format
- https://en.wikipedia.org/wiki/Quadruple-precision_floating-point_format

Tipos de dados em C – Exemplos

```
char c='A'; /* armazena 65, o valor da tabela ASCII */
```

```
char n=100;
```

```
int i=-2343234;
```

```
unsigned int ui=100000000;
```

```
float pi=3.14f;
```

```
double longer_pi=3.14159265359;
```

Dec	Hx	Oct	Html	Chr
64	40	100	@	@
65	41	101	A	A
66	42	102	B	B
67	43	103	C	C
68	44	104	D	D
69	45	105	E	E
70	46	106	F	F
71	47	107	G	G
72	48	110	H	H
73	49	111	I	I
74	4A	112	J	J
75	4B	113	K	K
76	4C	114	L	L
77	4D	115	M	M
78	4E	116	N	N
79	4F	117	O	O
80	50	120	P	P

Notas sobre tipos de dados em C

- O espaço de armazenamento de alguns tipos de dados varia dependendo da arquitetura. Por exemplo, um **long** ocupa 4 Bytes em IA32 e 8 Bytes em máquinas x86-64
- **Atenção:** o tipo de dados **char**, não é o que parece... É um tipo de dados numérico, o qual, por vezes, é utilizado para armazenamento de códigos de caracteres ASCII
- O tipo de dados **void** abrange um conjunto de valores vazio; é um tipo de dados incompleto que não se pode completar
 - Não se pode definir variáveis do tipo **void**, no entanto, **void** pode ser usado para:
 - Indicar que uma função não tem parâmetros
 - Exemplo: **int** func(**void**);
 - Indicar que uma função não tem retorno
 - Exemplo: **void** func(**int** n);
 - Definir um apontador que não especifica o tipo de dados para o qual aponta
 - Exemplo: **void*** ptr;

Notas sobre tipos de dados em C

- Duas formas de conversão de tipo:

- Implícita: conversão automática de tipo por parte do compilador

- Exemplo:

```
int a=1000;
```

Em complemento para 2 com representação de 32 bits:

$1000_{10} = 000000000000000000000000111101000_2$

```
char b=a; /* b=-24 (8 bits menos significativos de a: 11101000) */
```

Em complemento para 2 com representação de 8 bits:

$-24_{10} = 11101000_2$

- Explícita: conversão explicitamente definida pelo programador

- Exemplo:

```
float f=1.2f;
```

```
int d = (int)f; /* d=1 (truncagem para inteiro) */
```

Utilização do operador **sizeof**()

- O **C** tem o operador, em tempo de compilação, **sizeof**, que pode ser usado para obter o espaço de armazenamento de variáveis e tipos de dados. O resultado, que este operador produz, deve ser interpretado como sendo o número de elementos, do tipo **char**, que ocupariam o mesmo espaço do argumento passado ao **sizeof**
 - Exemplos:
 - **sizeof(char)**: devolve o tamanho do tipo de dados **char**; obviamente retornará sempre 1
 - **sizeof(int)**: fornece o tamanho do tipo de dados **int** (quantos **chars** “cabem” num **int**)
 - **sizeof(a)**: fornece o tamanho da variável **a**

Utilização do operador **sizeof()**

- **Importante:**
 - Embora, para a maioria dos sistemas modernos, o tipo de dados **char** ocupe 8 bits, não há garantia que seja sempre verdade!
 - O número de bits do tipo de dados **char** está atribuído à constante **CHAR_BIT** definida em **<limits.h>**
- Utilize o ficheiro **<limits.h>** para obter tamanhos e limites de tipos de dados inteiros
 - Exemplos:
 - **CHAR_MIN**
 - **CHAR_MAX**
 - **INT_MIN**
 - **INT_MAX**
- Utilize o ficheiro **<float.h>** para obter tamanhos e limites de tipos de dados de vírgula flutuante
 - Exemplos:
 - **FLT_MIN**
 - **FLT_MAX**

Formatadores para a função **printf()**

- Alguns formatadores para a função **printf**:
 - **%d** ou **%i**: inteiros com sinal em base 10
 - **%hhd** para **char** interpretado como número inteiro em base 10 com sinal
 - **%hd** para **short** interpretado como número inteiro em base 10 com sinal
 - **%ld** para **long** interpretado como número inteiro em base 10 com sinal
 - **%u**: inteiros sem sinal em base 10
 - **%hhu** para **char** interpretado como número em base 10 sem sinal
 - **%hu** para **short** interpretado como número inteiro em base 10 sem sinal
 - **%lu** para **long** interpretado como número inteiro em base 10 sem sinal
 - **%f**: decimais em vírgula flutuante (**float** ou **double**)
 - **%E**: números em notação científica (mantissa e expoente)
 - **%c**: character
 - **%s**: *string*
 - **%p**: permite imprimir o endereço de apontadores (apontadores serão abordados na próxima aula TP)
- Mais informação em: <http://www.cplusplus.com/reference/cstdio/printf/>
- O próximo slide apresenta um exemplo da utilização do operador **sizeof** e de várias constantes presentes em **<limits.h>** e **<float.h>**

Utilização do operador **sizeof**(): Exemplos

```
#include <stdio.h> /* devido ao printf */
#include <limits.h> /* devido às constantes CHAR_BIT, INT_MIN e INT_MAX */
#include <float.h> /* devido às constantes FLT_MIN, FLT_MAX e FLT_DIG */

int main(void) {
    char n='A';

    printf("Storage size for variable n: %lu\n\n", sizeof(n));
    printf("Storage size for char: %lu\n", sizeof(char));
    printf("Number of bits in a char: %u\n\n", CHAR_BIT);

    printf("Storage size for int: %lu\n", sizeof(int));
    printf("Minimum int value: %d\n", INT_MIN);
    printf("Maximum int value: %d\n\n", INT_MAX);

    printf("Storage size for float: %lu\n", sizeof(float));
    printf("Minimum float positive value: %E\n", FLT_MIN);
    printf("Maximum float positive value: %E\n", FLT_MAX);
    printf("Significant digits for float: %u\n\n", FLT_DIG);

    printf("Storage size for double: %lu\n", sizeof(double));

    return 0;
}
```

Output do programa:

Storage size for variable n: 1

Storage size for char: 1

Number of bits in a char: 8

Storage size for int: 4

Minimum int value: -2147483648

Maximum int value: 2147483647

Storage size for float: 4

Minimum float positive value: 1.175494E-38

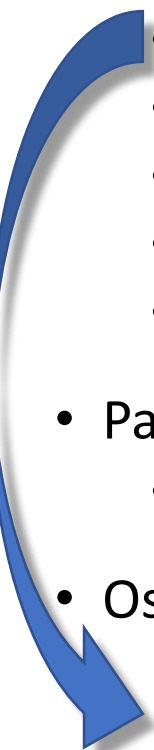
Maximum float positive value: 3.402823E+38

Significant digits for float: 6

Storage size for double: 8

Arrays em C

- O **C** permite definir *arrays* de elementos do mesmo tipo de dados
 - Historicamente, o **C** apenas suporta *arrays* onde o tamanho possa ser determinado em tempo de compilação
 - Programadores que necessitem de *arrays* de tamanho variável terão de alocar espaço para esses *arrays* usando funções como, por exemplo, o **malloc**
- Exemplos de *arrays* com tamanho definido estaticamente (tamanho fixo)
 - `int a[10]; /* array de 10 inteiros, sem inicialização */`
 - `int a1[]={1, 2, 3, 4, 5}; /* array de 5 inteiros, inicializados */`
 - `int a2[1000]={0}; /* array de 1000 inteiros, todos inicializados a 0 */`
 - `short s[100]; /* array de 100 shorts, sem inicialização */`
 - `float m[10][10]; /* matriz de 10x10 floats, sem inicialização */`
- Para um *array* **a**, de **N** elementos, os índices vão de **0** até **N-1**
 - Os elementos são acedidos através de **a[0]**, **a[1]**, ..., **a[N-1]**
- Os *arrays* são armazenados em memória de forma contínua e linear:



a:	123	-250	7	-780	45	128	-10	678	845	108
	a[0]	a[1]								a[9]

Atendendo a que o *array* **a** não foi inicializado, os elementos terão o valor que estava anteriormente nessa zona de memória. Neste contexto é considerado "lixo".

Arrays em C

- O **gcc** não verifica se o programador estiver a aceder a índices inválidos

- Exemplo:

```
int a[10];
```

```
a[10]=5;
```

Ocorre *overflow* do *array*, resultará em comportamento indefinido (poderá funcionar durante algum tempo... geralmente resulta em *segmentation fault* e o programa termina)

- Um *array* não pode ser alvo de uma atribuição

- Assuma um outro *array* **int** v[5]. Depois de ter sido declarado, esta instrução não é válida:

```
v = {1, 2, 3, 4, 5};
```

- Apenas podemos atribuir valores, em bloco, a um *array* aquando da respetiva declaração
- Após a declaração, uma inicialização válida poderia ser:

```
for(i=0; i<5; ++i) {  
    v[i] = i+1;  
}
```

- Se pretender copiar *arrays*, utilize a função:

```
memcpy(dest, src, size) /* size - número de bytes a copiar */
```

Arrays em C

- O **C** não se recorda do tamanho dos *arrays* (i.e., não existe atributo **length**)
 - O operador **sizeof** funciona apenas para *arrays* definidos estaticamente, dentro do âmbito de validade do local onde foram declarados
- Exemplo:

```
{  
    int a[10];  
    printf("%lu", sizeof(a)); /* imprime 40 em x86-64 */  
}
```
- As chavetas definem o âmbito de validade das instruções
- O tamanho do *array* pode ser calculado através de:
sizeof(a) / sizeof(a[0]); /* 40/4 = 10 elementos */

Arrays em C

- Passagem do endereço de um *array* para dentro de uma função
 - Quando um *array* é passado como argumento para uma função, a informação acerca do tamanho do *array* **deixa de estar disponível!!!**

```
void func(int a[]) {  
    printf("%lu", sizeof(a)); /* imprime sempre 8 em x86-64 */  
}
```

- Mais acerca deste assunto nas próximas aulas...
- Para solucionar o problema, o programador precisa de manter o tamanho do *array* de uma destas alternativas:
 - Passando o tamanho do *array* como argumento para a função
 - Ocupando a primeira posição do vetor com o respetivo tamanho
 - Definindo uma estrutura de dados para armazenar juntamente o *array* e o seu tamanho
 - Usando uma constante definida globalmente
 - Definindo um valor que indica o fim do *array* (um *array* que termine com um valor que nunca é usado, por exemplo: **-1**)

Strings em C

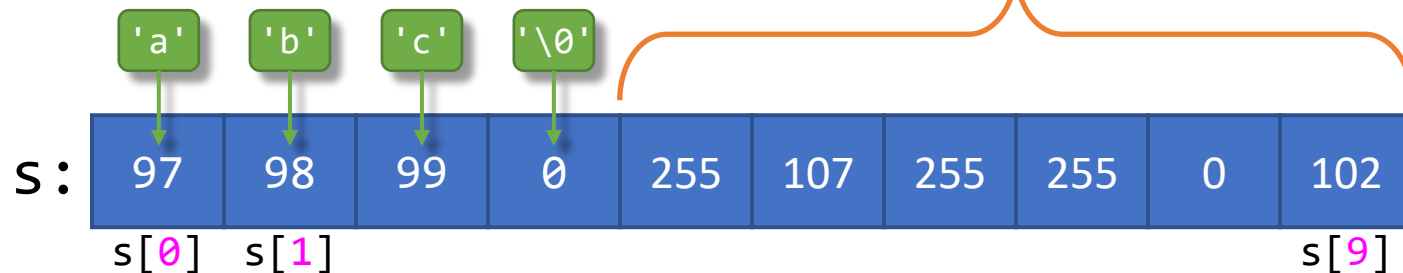
- O **C** não tem um tipo de dados específico para *strings*
 - *Strings* são apenas *arrays* do tipo de dados **char** terminadas com o caracter **NUL**
 - **NUL** representa-se em **C** através de: `'\0'`
 - O caracter **NUL** corresponde ao valor decimal zero na tabela ASCII
 - Não confundir com o caracter `'0'`, o qual corresponde ao valor decimal 48

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Strings em C

```
char s[10]="abc";
```

- O código apresentado define um *array* de 10 elementos do tipo **char**
 - Os primeiros três elementos correspondem aos códigos ASCII dos caracteres 'a', 'b' e 'c'
 - O quarto elemento será o caracter de terminação: **NUL**



Dec	Hx	Oct	Char
0	0	000	NUL (null)
1	1	001	SOH (start of header)
2	2	002	STX (start of text)
3	3	003	ETX (end of text)

Dec	Hx	Oct	Html	Chr
96	60	140	`	`
97	61	141	a	a
98	62	142	b	b
99	63	143	c	c
100	64	144	d	d
101	65	145	e	e
102	66	146	f	f
103	67	147	g	g
104	68	150	h	h
105	69	151	i	i
106	6A	152	j	j
107	6B	153	k	k
108	6C	154	l	l
109	6D	155	m	m

Strings em C

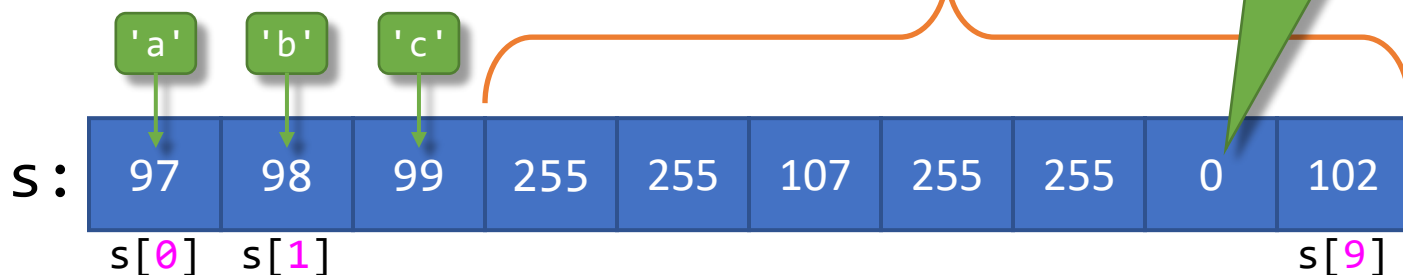
- Se no código não fecharmos convenientemente a *string* podemos ter resultados inesperados, vejamos:

```
char s[10];  
s[0]='a';  
s[1]='b';  
s[2]='c';  
printf("%s", s);
```

	Dec	Hx	Oct	Html	Chr
	96	60	140	`	`
	97	61	141	a	a
	98	62	142	b	b
	99	63	143	c	c
100	64	144	d	d	
101	65	145	e	e	
102	66	146	f	f	
103	67	147	g	g	
104	68	150	h	h	
105	69	151	i	i	
106	6A	152	j	j	
107	6B	153	k	k	
108	6C	154	l	l	
109	6D	155	m	m	

A função printf irá interpretar o primeiro zero que encontrar na memória como sendo o fecho da *string*!!!

O que estava anteriormente na memória, considerámos ser "lixo" para o nosso contexto



Output do programa:

abc??k??

Strings em C

```
#include <stdio.h> /* devido ao printf */

int main(void) {
    char str1[]="ARQCP";
    char str2[]={68, 69, 73, 0};
    char str3[]={ 'I', 'S', 'E', 'P', '\0' };
    printf("%s %s-%s", str1, str2, str3);
    return 0;
}
```

Output do programa:

ARQCP DEI-ISEP

Qual será o
output do
programa?

Dec	Hx	Oct	Html	Chr
64	40	100	@	@
65	41	101	A	A
66	42	102	B	B
67	43	103	C	C
68	44	104	D	D
69	45	105	E	E
70	46	106	F	F
71	47	107	G	G
72	48	110	H	H
73	49	111	I	I
74	4A	112	J	J
75	4B	113	K	K
76	4C	114	L	L
77	4D	115	M	M
78	4E	116	N	N
79	4F	117	O	O
80	50	120	P	P
81	51	121	Q	Q
82	52	122	R	R
83	53	123	S	S
84	54	124	T	T

Strings em C

```
unsigned int xpto(char str[]) {  
    unsigned int c=0;  
    while(str[c]!=0) {  
        ++c;  
    }  
    return c;  
}
```

- Qual é a funcionalidade da função apresentada?
 - a) A função devolve o código ASCII do último caracter
 - b) A função devolve o tamanho da *string* ✓
 - c) A função devolve o número de palavras da *string*
 - d) Nenhuma das opções apresentadas

Copiar *strings*

- Relembrar:
 - Os *arrays* não podem ser alvo de atribuições após a declaração, isto inclui as *strings*, pois são *arrays* de **chars**
- Pode-se declarar um *array* `s[6]`, inicializado com os caracteres `'H', 'e', 'l', 'l', 'o'` e `'\0'` através de :
`char s[]="Hello"; /* instrução válida */`
- No entanto, após a declaração de `s[]`, não é possível efetuar a atribuição de uma nova *string*:
`s="World"; /* instrução inválida */`
- Para copiar *strings* utilize a função:
 - `strncpy(dest_str, src_str, n_chars)`
 - Exemplo:
`strncpy(s, "World", 6);`
- Note-se ainda que, é responsabilidade do programador garantir a existência, na *string* de destino, de espaço suficiente para o armazenamento

Função de `string.h` que permite obter o tamanho de uma *string*

Se `n_chars > strlen(src_str)`
o `strncpy` preenche o restante espaço de `dest_str` com NUL

Exemplo:

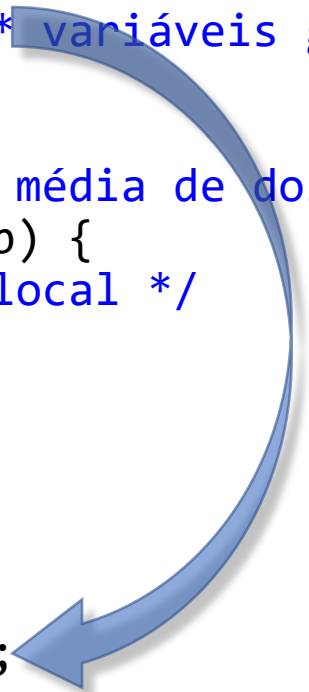
Cálculo da média de dois números inteiros

```
#include <stdio.h> /* devido ao printf */

int n1=6, n2=4, avg=0; /* variáveis globais, serão mesmo necessárias? */

/* esta função calcula a média de dois inteiros (truncada para valor inteiro) */
int calc_avg(int a, int b) {
    int c=0; /* variável local */
    c=(a+b)/2;
    return c;
}

int main(void) {
    int n1=6, n2=4, avg=0;
    avg = calc_avg(n1, n2); /* invoca a função e guarda o resultado */
    printf("avg = %d\n", avg);
    return 0;
}
```



Código C de boa qualidade

- Bom código deve ser na sua maior parte auto documentado
 - Variáveis e funções, devem ter nomes que ajudem a perceber o que se está a implementar
 - Os comentários não devem descrever o que o código faz, mas sim porque o faz
 - O que o código faz deve ser evidente (assuma que o leitor compreende a linguagem)
 - Deve comentar:
 - Cada ficheiro de código fonte
 - Os cabeçalhos das funções
 - Blocos de código longos
 - Porções de código de maior complexidade (por exemplo, manipulações de bits)
- Utilize convenção de nomenclatura ao estilo do C
 - Por exemplo, prefira `get_radius()` a `GetRadius()`
 - `i` e `j` para variáveis de ciclos
- Corps de funções, ciclos, declarações **if-else**, etc. devem indentadas

Código C de boa qualidade

- Defina constantes e use-as
 - As constantes simplificam a leitura e a manutenção do código
- Evite o uso de variáveis globais
 - Forneça variáveis como argumentos para funções
- Inicialize as variáveis antes de as usar!
- Use boa detecção e tratamento de erros
 - Verifique sempre o retorno das funções e trate os erros convenientemente
- Para mais conselhos de como usar convenientemente a linguagem C:
https://www.gnu.org/prep/standards/html_node/Writing-C.html

Pratique

- Implemente um programa em linguagem **C** que leia 10 inteiros para um *array* e calcule a respetiva média.
- A média deve ser calculada numa função à parte, mas o valor do cálculo deverá ser impresso na função `main()`.

Bibliografia

- Instituto Superior de Engenharia do Porto. Computer Architecture. Luís Nogueira. “Introduction to the C Programming Language aka C for Java Programmers”. 2024/2025. 29 Diapositivos.
<https://moodle.isep.ipp.pt/mod/resource/view.php?id=216576>